The YANG 1.1 Data Modeling Language

Abstract

   YANG is a data modeling language used to model configuration data,
   state data, Remote Procedure Calls, and notifications for network
   management protocols. This document describes the syntax and
   semantics of version 1.1 of the YANG language. YANG version 1.1 is a
   maintenance release of the YANG language, addressing ambiguities and
   defects in the original specification. There are a small number of
   backward incompatibilities from YANG version 1. This document also
   specifies the YANG mappings to the Network Configuration Protocol
   (NETCONF).

Copyright Notice

Table of Contents

1. Introduction

   YANG is a data modeling language originally designed to model
   configuration and state data manipulated by the Network Configuration
   Protocol (NETCONF), NETCONF Remote Procedure Calls, and NETCONF
   notifications [RFC6241]. Since the publication of YANG version 1
   [RFC6020], YANG has been used or proposed to be used for other
   protocols (e.g., RESTCONF [RESTCONF] and the Constrained Application
   Protocol (CoAP) Management Interface (CoMI) [CoMI]). Further,
   encodings other than XML have been proposed (e.g., JSON [RFC7951]).

   This document describes the syntax and semantics of version 1.1 of
   the YANG language. It also describes how a data model defined in a
   YANG module is encoded in the Extensible Markup Language (XML) [XML]
   and how NETCONF operations are used to manipulate the data. Other
   protocols and encodings are possible but are out of scope for this
   specification.

   In terms of developing YANG data models, [YANG-Guidelines] provides
   some guidelines and recommendations.

   Note that this document does not obsolete RFC 6020 [RFC6020].

1.1. Summary of Changes from RFC 6020

   This document defines version 1.1 of the YANG language. YANG
   version 1.1 is a maintenance release of the YANG language, addressing
   ambiguities and defects in the original specification [RFC6020].

   The following changes are not backward compatible with YANG
   version 1:

   o Changed the rules for the interpretation of escaped characters in
      double-quoted strings. This is a backward-incompatible change
      from YANG version 1. When updating a YANG version 1 module to 1.1
      and the module uses a character sequence that is now illegal, the
      string must be changed to match the new rules. See Section 6.1.3
      for details.

   o An unquoted string cannot contain any single or double quote
      characters. This is a backward-incompatible change from YANG
      version 1. When updating a YANG version 1 module to 1.1 and the
      module uses such quote characters, the string must be changed to
      match the new rules. See Section 6.1.3 for details.

   o Made "when" and "if-feature" illegal on list keys. This is a
      backward-incompatible change from YANG version 1. When updating a
      YANG version 1 module to 1.1 and the module uses these constructs,
      they must be removed to match the new rules.

   o Defined the legal characters in YANG modules. When updating a
      YANG version 1 module to 1.1, any characters that are now illegal
      must be removed. See Section 6 for details.

   o Made noncharacters illegal in the built-in type "string". This
      change affects the runtime behavior of YANG-based protocols.

   The following additional changes have been done to YANG:

   o Changed the YANG version from "1" to "1.1".

   o Made the "yang-version" statement mandatory in YANG version "1.1".

   o Extended the "if-feature" syntax to be a boolean expression over
      feature names.

   o Allow "if-feature" in "bit", "enum", and "identity".

   o Allow "if-feature" in "refine".

o   Allow "choice" as a shorthand "case" statement (see
    Section 7.9.2).

o   Added a new substatement "modifier" to the "pattern" statement
    (see Section 9.4.6).

o   Allow "must" in "input", "output", and "notification".

o   Allow "require-instance" in leafref.

o   Allow "description" and "reference" in "import" and "include".

o   Allow imports of multiple revisions of a module.

o   Allow "augment" to add conditionally mandatory nodes (see
    Section 7.17).

o   Added a set of new XML Path Language (XPath) functions in
    Section 10.

o   Clarified the XPath context's tree in Section 6.4.1.

o   Defined the string value of an identityref in XPath expressions
    (see Section 9.10).

o   Clarified what unprefixed names mean in leafrefs in typedefs (see
    Sections 6.4.1 and 9.9.2).

o   Allow identities to be derived from multiple base identities (see
    Sections 7.18 and 9.10).

o   Allow enumerations and bits to be subtyped (see Sections 9.6
    and 9.7).

o   Allow leaf-lists to have default values (see Section 7.7.2).

o   Allow non-unique values in non-configuration leaf-lists (see
    Section 7.7).

o   Use syntax for case-sensitive strings (as per [RFC7405]) in the
    grammar.

o   Changed the module advertisement mechanism (see Section 5.6.4).

o   Changed the scoping rules for definitions in submodules. A
    submodule can now reference all definitions in all submodules that
    belong to the same module, without using the "include" statement.

o Added a new statement "action", which is used to define operations
   tied to data nodes.

o Allow notifications to be tied to data nodes.

o Added a new data definition statement "anydata" (see
   Section 7.10), which is RECOMMENDED to be used instead of "anyxml"
   when the data can be modeled in YANG.

o Allow types "empty" and "leafref" in unions.

o Allow type "empty" in a key.

o Removed the restriction that identifiers could not start with the
   characters "xml".

The following changes have been done to the NETCONF mapping:

o A server advertises support for YANG 1.1 modules by using
   ietf-yang-library [RFC7895] instead of listing them as
   capabilities in the <hello> message.

2. Key Words

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and
   "OPTIONAL" in this document are to be interpreted as described in
   BCP 14 [RFC2119].

3. Terminology

   The following terms are used within this document:

   o action: An operation defined for a node in the data tree.

   o anydata: A data node that can contain an unknown set of nodes that
      can be modeled by YANG, except anyxml.

   o anyxml: A data node that can contain an unknown chunk of XML data.

   o augment: Adds new schema nodes to a previously defined schema
      node.

   o base type: The type from which a derived type was derived, which
      may be either a built-in type or another derived type.

   o built-in type: A YANG data type defined in the YANG language, such
      as uint32 or string.

o choice: A schema node where only one of a number of identified
   alternatives is valid.

o client: An entity that can access YANG-defined data on a server,
   over some network management protocol.

o conformance: A measure of how accurately a server follows a data
   model.

o container: An interior data node that exists in at most one
   instance in the data tree. A container has no value, but rather a
   set of child nodes.

o data definition statement: A statement that defines new data
   nodes. One of "container", "leaf", "leaf-list", "list", "choice",
   "case", "augment", "uses", "anydata", and "anyxml".

o data model: A data model describes how data is represented and
   accessed.

o data node: A node in the schema tree that can be instantiated in a
   data tree. One of container, leaf, leaf-list, list, anydata, and
   anyxml.

o data tree: An instantiated tree of any data modeled with YANG,
   e.g., configuration data, state data, combined configuration and
   state data, RPC or action input, RPC or action output, or
   notification.

o derived type: A type that is derived from a built-in type (such as
   uint32) or another derived type.

o extension: An extension attaches non-YANG semantics to statements.
   The "extension" statement defines new statements to express these
   semantics.

o feature: A mechanism for marking a portion of the model as
   optional. Definitions can be tagged with a feature name and are
   only valid on servers that support that feature.

o grouping: A reusable set of schema nodes, which may be used
   locally in the module and by other modules that import from it.
   The "grouping" statement is not a data definition statement and,
   as such, does not define any nodes in the schema tree.

o identifier: A string used to identify different kinds of YANG
   items by name.

o identity: A globally unique, abstract, and untyped name.

o instance identifier: A mechanism for identifying a particular node
   in a data tree.

o interior node: Nodes within a hierarchy that are not leaf nodes.

o leaf: A data node that exists in at most one instance in the data
   tree. A leaf has a value but no child nodes.

o leaf-list: Like the leaf node but defines a set of uniquely
   identifiable nodes rather than a single node. Each node has a
   value but no child nodes.

o list: An interior data node that may exist in multiple instances
   in the data tree. A list has no value, but rather a set of child
   nodes.

o mandatory node: A mandatory node is one of:

   * A leaf, choice, anydata, or anyxml node with a "mandatory"
      statement with the value "true".

   * A list or leaf-list node with a "min-elements" statement with a
      value greater than zero.

   * A container node without a "presence" statement and that has at
      least one mandatory node as a child.

o module: A YANG module defines hierarchies of schema nodes. With
   its definitions and the definitions it imports or includes from
   elsewhere, a module is self-contained and "compilable".

o non-presence container: A container that has no meaning of its
   own, existing only to contain child nodes.

o presence container: A container where the presence of the
   container itself carries some meaning.

o RPC: A Remote Procedure Call.

o RPC operation: A specific Remote Procedure Call.

o schema node: A node in the schema tree. One of action, container,
   leaf, leaf-list, list, choice, case, rpc, input, output,
   notification, anydata, and anyxml.

o schema node identifier: A mechanism for identifying a particular
   node in the schema tree.

o schema tree: The definition hierarchy specified within a module.

o server: An entity that provides access to YANG-defined data to a
   client, over some network management protocol.

o server deviation: A failure of the server to implement a module
   faithfully.

o submodule: A partial module definition that contributes derived
   types, groupings, data nodes, RPCs, actions, and notifications to
   a module. A YANG module can be constructed from a number of
   submodules.

o top-level data node: A data node where there is no other data node
   between it and a "module" or "submodule" statement.

o uses: The "uses" statement is used to instantiate the set of
   schema nodes defined in a "grouping" statement. The instantiated
   nodes may be refined and augmented to tailor them to any specific
   needs.

o value space: For a data type; the set of values permitted by the
   data type. For a leaf or leaf-list instance; the value space of
   its data type.

The following terms are defined in [RFC6241]:

o configuration data

o configuration datastore

o datastore

o state data

When modeled with YANG, a datastore is realized as an instantiated
data tree.

When modeled with YANG, a configuration datastore is realized as an
instantiated data tree with configuration data.

3.1. A Note on Examples

Throughout this document, there are many examples of YANG statements.
These examples are supposed to illustrate certain features and are
not supposed to be complete, valid YANG modules.

4. YANG Overview

This non-normative section is intended to give a high-level overview
of YANG to first-time readers.

4.1. Functional Overview

YANG is a language originally designed to model data for the NETCONF
protocol. A YANG module defines hierarchies of data that can be used
for NETCONF-based operations, including configuration, state data,
RPCs, and notifications. This allows a complete description of all
data sent between a NETCONF client and server. Although out of scope
for this specification, YANG can also be used with protocols other
than NETCONF.

YANG models the hierarchical organization of data as a tree in which
each node has a name, and either a value or a set of child nodes.
YANG provides clear and concise descriptions of the nodes, as well as
the interaction between those nodes.

YANG structures data models into modules and submodules. A module
can import definitions from other external modules and can include
definitions from submodules. The hierarchy can be augmented,
allowing one module to add data nodes to the hierarchy defined in
another module. This augmentation can be conditional, with new nodes
appearing only if certain conditions are met.

YANG data models can describe constraints to be enforced on the data,
restricting the presence or value of nodes based on the presence or
value of other nodes in the hierarchy. These constraints are
enforceable by either the client or the server.

YANG defines a set of built-in types and has a type mechanism through
which additional types may be defined. Derived types can restrict
their base type's set of valid values using mechanisms like range or
pattern restrictions that can be enforced by clients or servers.
They can also define usage conventions for use of the derived type,
such as a string-based type that contains a hostname.

YANG permits the definition of reusable groupings of nodes. The
usage of these groupings can refine or augment the nodes, allowing it
to tailor the nodes to its particular needs. Derived types and
groupings can be defined in one module and used in either the same
module or another module that imports it.

YANG data hierarchy constructs include defining lists where list
entries are identified by keys that distinguish them from each other.
Such lists may be defined as either sorted by user or automatically
sorted by the system. For user-sorted lists, operations are defined
for manipulating the order of the list entries.

YANG modules can be translated into an equivalent XML syntax called
YANG Independent Notation (YIN) (Section 13), allowing applications
using XML parsers and Extensible Stylesheet Language Transformations
(XSLT) scripts to operate on the models. The conversion from YANG to
YIN is semantically lossless, so content in YIN can be round-tripped
back into YANG.

YANG is an extensible language, allowing extensions to be defined by
standards bodies, vendors, and individuals. The statement syntax
allows these extensions to coexist with standard YANG statements in a
natural way, while extensions in a YANG module stand out sufficiently
for the reader to notice them.

YANG resists the tendency to solve all possible problems, limiting
the problem space to allow expression of data models for network
management protocols such as NETCONF, not arbitrary XML documents or
arbitrary data models.

To the extent possible, YANG maintains compatibility with the Simple
Network Management Protocol's (SNMP's) SMIv2 (Structure of Management
Information version 2 [RFC2578] [RFC2579]). SMIv2-based MIB modules
can be automatically translated into YANG modules for read-only
access [RFC6643]. However, YANG is not concerned with reverse
translation from YANG to SMIv2.

4.2. Language Overview

This section introduces some important constructs used in YANG that
will aid in the understanding of the language specifics in later
sections.

4.2.1. Modules and Submodules

YANG data models are defined in modules. A module contains a
collection of related definitions.

A module contains three types of statements: module header
statements, "revision" statements, and definition statements. The
module header statements describe the module and give information
about the module itself, the "revision" statements give information
about the history of the module, and the definition statements are
the body of the module where the data model is defined.

A server may implement a number of modules, allowing multiple views
of the same data or multiple views of disjoint subsections of the
server's data. Alternatively, the server may implement only one
module that defines all available data.

A module may have portions of its definitions separated into
submodules, based on the needs of the module designer. The external
view remains that of a single module, regardless of the presence or
size of its submodules.

The "import" statement allows a module or submodule to reference
definitions defined in other modules.

The "include" statement is used in a module to identify each
submodule that belongs to it.

4.2.2. Data Modeling Basics

   YANG defines four main types of data nodes for data modeling. In
   each of the following subsections, the examples show the YANG syntax
   as well as a corresponding XML encoding. The syntax of YANG
   statements is defined in Section 6.3.

4.2.2.1. Leaf Nodes

   A leaf instance contains simple data like an integer or a string. It
   has exactly one value of a particular type and no child nodes.

   YANG Example:

     leaf host-name {
       type string;
       description
         "Hostname for this system.";
     }

   XML Encoding Example:

     <host-name>my.example.com</host-name>

   The "leaf" statement is covered in Section 7.6.

4.2.2.2. Leaf-List Nodes

   A leaf-list defines a sequence of values of a particular type.

   YANG Example:

     leaf-list domain-search {
       type string;
       description
         "List of domain names to search.";
     }

   XML Encoding Example:

     <domain-search>high.example.com</domain-search>
     <domain-search>low.example.com</domain-search>
     <domain-search>everywhere.example.com</domain-search>

   The "leaf-list" statement is covered in Section 7.7.

4.2.2.3. Container Nodes

   A container is used to group related nodes in a subtree. A container
   has only child nodes and no value. A container may contain any
   number of child nodes of any type (leafs, lists, containers,
   leaf-lists, actions, and notifications).

   YANG Example:

     container system {
       container login {
         leaf message {
           type string;
           description
             "Message given at start of login session.";
         }
       }
     }

   XML Encoding Example:

     <system>
       <login>
         <message>Good morning</message>
       </login>
     </system>

   The "container" statement is covered in Section 7.5.

4.2.2.4. List Nodes

   A list defines a sequence of list entries. Each entry is like a
   container and is uniquely identified by the values of its key leafs
   if it has any key leafs defined. A list can define multiple key
   leafs and may contain any number of child nodes of any type
   (including leafs, lists, containers, etc.).

   YANG Example:

     list user {
       key "name";
       leaf name {
         type string;
       }
       leaf full-name {
         type string;
       }
       leaf class {
         type string;
       }
     }

   XML Encoding Example:

     <user>
       <name>glocks</name>
       <full-name>Goldie Locks</full-name>
       <class>intruder</class>
     </user>
     <user>
       <name>snowey</name>
       <full-name>Snow White</full-name>
       <class>free-loader</class>
     </user>
     <user>
       <name>rzell</name>
       <full-name>Rapun Zell</full-name>
       <class>tower</class>
     </user>

   The "list" statement is covered in Section 7.8.

4.2.2.5. Example Module

   These statements are combined to define the module:

```
      // Contents of "example-system.yang"
      module example-system {
        yang-version 1.1;
        namespace "urn:example:system";
        prefix "sys";

        organization "Example Inc.";
        contact "joe@example.com";
        description
          "The module for entities implementing the Example system.";

        revision 2007-06-09 {
          description "Initial revision.";
        }

        container system {
          leaf host-name {
            type string;
            description
              "Hostname for this system.";
          }

          leaf-list domain-search {
            type string;
            description
              "List of domain names to search.";
          }

          container login {
            leaf message {
              type string;
              description
                "Message given at start of login session.";
            }
```

```
        list user {
          key "name";
          leaf name {
            type string;
          }
          leaf full-name {
            type string;
          }
          leaf class {
            type string;
          }
        }
      }
    }
  }
```

4.2.3. Configuration and State Data

   YANG can model state data, as well as configuration data, based on
   the "config" statement. When a node is tagged with "config false",
   its subhierarchy is flagged as state data. If it is tagged with
   "config true", its subhierarchy is flagged as configuration data.
   Parent containers, lists, and key leafs are reported also, giving the
   context for the state data.

   In this example, two leafs are defined for each interface, a
   configured speed and an observed speed.

```
   list interface {
     key "name";
     config true;

     leaf name {
       type string;
     }
     leaf speed {
       type enumeration {
         enum 10m;
         enum 100m;
         enum auto;
       }
     }
     leaf observed-speed {
       type uint32;
       config false;
     }
   }
```

The "config" statement is covered in Section 7.21.1.

4.2.4. Built-In Types

YANG has a set of built-in types, similar to those of many
programming languages, but with some differences due to special
requirements of network management. The following table summarizes
the built-in types discussed in Section 9:

```
+--------------------+------------------------------------+
| Name               | Description                        |
+--------------------+------------------------------------+
| binary             | Any binary data                    |
| bits               | A set of bits or flags             |
| boolean            | "true" or "false"                  |
| decimal64          | 64-bit signed decimal number       |
| empty              | A leaf that does not have any value |
| enumeration        | One of an enumerated set of strings |
| identityref        | A reference to an abstract identity |
| instance-identifier | A reference to a data tree node    |
| int8               | 8-bit signed integer               |
| int16              | 16-bit signed integer              |
| int32              | 32-bit signed integer              |
| int64              | 64-bit signed integer              |
| leafref            | A reference to a leaf instance     |
| string             | A character string                 |
| uint8              | 8-bit unsigned integer             |
| uint16             | 16-bit unsigned integer            |
| uint32             | 32-bit unsigned integer            |
| uint64             | 64-bit unsigned integer            |
| union              | Choice of member types             |
+--------------------+------------------------------------+
```

The "type" statement is covered in Section 7.4.

4.2.5. Derived Types (typedef)

YANG can define derived types from base types using the "typedef"
statement. A base type can be either a built-in type or a derived
type, allowing a hierarchy of derived types.

A derived type can be used as the argument for the "type" statement.

YANG Example:

```
typedef percent {
  type uint8 {
    range "0 .. 100";
  }
}

leaf completed {
  type percent;
}
```

XML Encoding Example:

```
<completed>20</completed>
```

The "typedef" statement is covered in Section 7.3.

4.2.6. Reusable Node Groups (grouping)

Groups of nodes can be assembled into reusable collections using the
"grouping" statement. A grouping defines a set of nodes that are
instantiated with the "uses" statement.

YANG Example:

```
grouping target {
  leaf address {
    type inet:ip-address;
    description "Target IP address.";
  }
  leaf port {
    type inet:port-number;
     description "Target port number.";
  }
}
```

```
   container peer {
     container destination {
       uses target;
     }
   }
```

   XML Encoding Example:

```
   <peer>
     <destination>
       <address>2001:db8::2</address>
       <port>830</port>
     </destination>
   </peer>
```

   The grouping can be refined as it is used, allowing certain
   statements to be overridden. In this example, the description is
   refined:

```
   container connection {
     container source {
       uses target {
         refine "address" {
           description "Source IP address.";
         }
         refine "port" {
           description "Source port number.";
         }
       }
     }
     container destination {
       uses target {
         refine "address" {
           description "Destination IP address.";
         }
         refine "port" {
           description "Destination port number.";
         }
       }
     }
   }
```

   The "grouping" statement is covered in Section 7.12.

4.2.7. Choices

YANG allows the data model to segregate incompatible nodes into
distinct choices using the "choice" and "case" statements. The
"choice" statement contains a set of "case" statements that define
sets of schema nodes that cannot appear together. Each "case" may
contain multiple nodes, but each node may appear in only one "case"
under a "choice".

The choice and case nodes appear only in the schema tree and not in
the data tree. The additional levels of hierarchy are not needed
beyond the conceptual schema. The presence of a case is indicated by
the presence of one or more of the nodes within it.

Since only one of the choice's cases can be valid at any time, when a
node from one case is created in the data tree, all nodes from all
other cases are implicitly deleted. The server handles the
enforcement of the constraint, preventing incompatibilities from
existing in the configuration.

YANG Example:

```
container food {
  choice snack {
    case sports-arena {
      leaf pretzel {
        type empty;
      }
      leaf beer {
        type empty;
      }
    }
    case late-night {
      leaf chocolate {
        type enumeration {
          enum dark;
          enum milk;
          enum first-available;
        }
      }
    }
  }
}
```

   XML Encoding Example:

```
     <food>
       <pretzel/>
       <beer/>
     </food>
```

   The "choice" statement is covered in Section 7.9.

4.2.8. Extending Data Models (augment)

   YANG allows a module to insert additional nodes into data models,
   including both the current module (and its submodules) and an
   external module. This is useful, for example, for vendors to add
   vendor-specific parameters to standard data models in an
   interoperable way.

   The "augment" statement defines the location in the data model
   hierarchy where new nodes are inserted, and the "when" statement
   defines the conditions when the new nodes are valid.

   When a server implements a module containing an "augment" statement,
   that implies that the server's implementation of the augmented module
   contains the additional nodes.

   YANG Example:

```
     augment /system/login/user {
       when "class != 'wheel'";
       leaf uid {
         type uint16 {
           range "1000 .. 30000";
         }
       }
     }
```

   This example defines a "uid" node that is valid only when the user's
   "class" is not "wheel".

If a module augments another module, the XML elements that are added
to the encoding are in the namespace of the augmenting module. For
example, if the above augmentation were in a module with prefix
"other", the XML would look like:

XML Encoding Example:

```
<user>
  <name>alicew</name>
  <full-name>Alice N. Wonderland</full-name>
  <class>drop-out</class>
  <other:uid>1024</other:uid>
</user>
```

The "augment" statement is covered in Section 7.17.

4.2.9. Operation Definitions

YANG allows the definition of operations. The operations' names,
input parameters, and output parameters are modeled using YANG data
definition statements. Operations on the top level in a module are
defined with the "rpc" statement. Operations can also be tied to a
container or list data node. Such operations are defined with the
"action" statement.

YANG Example for an operation at the top level:

```
rpc activate-software-image {
  input {
    leaf image-name {
      type string;
    }
  }
  output {
    leaf status {
      type string;
    }
  }
}
```

   NETCONF XML Example:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <activate-software-image xmlns="http://example.com/system">
    <image-name>example-fw-2.3</image-name>
  </activate-software-image>
</rpc>

<rpc-reply message-id="101"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <status xmlns="http://example.com/system">
    The image example-fw-2.3 is being installed.
  </status>
</rpc-reply>
```

   YANG Example for an operation tied to a list data node:

```
list interface {
  key "name";

  leaf name {
    type string;
  }

  action ping {
    input {
      leaf destination {
        type inet:ip-address;
      }
    }
    output {
      leaf packet-loss {
        type uint8;
      }
    }
  }
}
```

   NETCONF XML Example:

```
<rpc message-id="102"
    xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <action xmlns="urn:ietf:params:xml:ns:yang:1">
    <interface xmlns="http://example.com/system">
      <name>eth1</name>
      <ping>
        <destination>192.0.2.1</destination>
      </ping>
    </interface>
  </action>
</rpc>

<rpc-reply message-id="102"
          xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
          xmlns:sys="http://example.com/system">
  <sys:packet-loss>60</sys:packet-loss>
</rpc-reply>
```

   The "rpc" statement is covered in Section 7.14, and the "action"
   statement is covered in Section 7.15.

4.2.10. Notification Definitions

   YANG allows the definition of notifications. YANG data definition
   statements are used to model the content of the notification.

   YANG Example:

```
notification link-failure {
  description
    "A link failure has been detected.";
  leaf if-name {
    type leafref {
      path "/interface/name";
    }
  }
  leaf if-admin-status {
    type admin-status;
  }
  leaf if-oper-status {
    type oper-status;
  }
}
```

   NETCONF XML Example:

```
<notification
    xmlns="urn:ietf:params:netconf:capability:notification:1.0">
  <eventTime>2007-09-01T10:00:00Z</eventTime>
  <link-failure xmlns="urn:example:system">
    <if-name>so-1/2/3.0</if-name>
    <if-admin-status>up</if-admin-status>
    <if-oper-status>down</if-oper-status>
  </link-failure>
</notification>
```

   The "notification" statement is covered in Section 7.16.

5. Language Concepts

5.1. Modules and Submodules

   The module is the base unit of definition in YANG. A module defines
   a single data model. A module can also augment an existing data
   model with additional nodes.

   Submodules are partial modules that contribute definitions to a
   module. A module may include any number of submodules, but each
   submodule may belong to only one module.

   Developers of YANG modules and submodules are RECOMMENDED to choose
   names for their modules that will have a low probability of colliding
   with standard or other enterprise modules, e.g., by using the
   enterprise or organization name as a prefix for the module name.
   Within a server, all module names MUST be unique.

   A module uses the "include" statement to list all its submodules. A
   module, or submodule belonging to that module, can reference
   definitions in the module and all submodules included by the module.

   A module or submodule uses the "import" statement to reference
   external modules. Statements in the module or submodule can
   reference definitions in the external module using the prefix
   specified in the "import" statement.

   For backward compatibility with YANG version 1, a submodule MAY use
   the "include" statement to reference other submodules within its
   module, but this is not necessary in YANG version 1.1. A submodule
   can reference any definition in the module it belongs to and in all
   submodules included by the module. A submodule MUST NOT include
   different revisions of other submodules than the revisions that its
   module includes.

A module or submodule MUST NOT include submodules from other modules,
and a submodule MUST NOT import its own module.

The "import" and "include" statements are used to make definitions
available from other modules:

o For a module or submodule to reference definitions in an external
   module, the external module MUST be imported.

o A module MUST include all its submodules.

o A module, or submodule belonging to that module, MAY reference
   definitions in the module and all submodules included by the
   module.

There MUST NOT be any circular chains of imports. For example, if
module "a" imports module "b", "b" cannot import "a".

When a definition in an external module is referenced, a locally
defined prefix MUST be used, followed by a colon (":") and then the
external identifier. References to definitions in the local module
MAY use the prefix notation. Since built-in data types do not belong
to any module and have no prefix, references to built-in data types
(e.g., int32) cannot use the prefix notation. The syntax for a
reference to a definition is formally defined by the rule
"identifier-ref" in Section 14.

5.1.1. Import and Include by Revision

Published modules evolve independently over time. In order to allow
for this evolution, modules can be imported using specific revisions.
Initially, a module imports the revisions of other modules that are
current when the module is written. As future revisions of the
imported modules are published, the importing module is unaffected
and its contents are unchanged. When the author of the module is
prepared to move to the most recently published revision of an
imported module, the module is republished with an updated "import"
statement. By republishing with the new revision, the authors
explicitly indicate their acceptance of any changes in the imported
module.

For submodules, the issue is related but simpler. A module or
submodule that includes submodules may specify the revision of the
included submodules. If a submodule changes, any module or submodule
that includes it by revision needs to be updated to reference the new
revision.

For example, module "b" imports module "a".

```
module a {
  yang-version 1.1;
  namespace "urn:example:a";
  prefix "a";

  revision 2008-01-01 { ... }
  grouping a {
    leaf eh { .... }
  }
}

module b {
  yang-version 1.1;
  namespace "urn:example:b";
  prefix "b";

  import a {
    prefix "p";
    revision-date 2008-01-01;
  }

  container bee {
    uses p:a;
  }
}
```

When the author of "a" publishes a new revision, the changes may not
be acceptable to the author of "b". If the new revision is
acceptable, the author of "b" can republish with an updated revision
in the "import" statement.

If a module is not imported with a specific revision, it is undefined
which revision is used.

5.1.2. Module Hierarchies

YANG allows modeling of data in multiple hierarchies, where data may
have more than one top-level node. Each top-level data node in a
module defines a separate hierarchy. Models that have multiple
top-level nodes are sometimes convenient and are supported by YANG.

5.1.2.1. NETCONF XML Encoding

   NETCONF is capable of carrying any XML content as the payload in the
   <config> and <data> elements. The top-level nodes of YANG modules
   are encoded as child elements, in any order, within these elements.
   This encapsulation guarantees that the corresponding NETCONF messages
   are always well-formed XML documents.

   For example, an instance of:

      module example-config {
        yang-version 1.1;
        namespace "urn:example:config";
        prefix "co";

        container system { ... }
        container routing { ... }
      }

   could be encoded in NETCONF as:

      <rpc message-id="101"
          xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
          xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
        <edit-config>
          <target>
            <running/>
          </target>
          <config>
            <system xmlns="urn:example:config">
              <!-- system data here -->
            </system>
            <routing xmlns="urn:example:config">
              <!-- routing data here -->
            </routing>
          </config>
        </edit-config>
      </rpc>

5.2. File Layout

YANG modules and submodules are typically stored in files, one
"module" or "submodule" statement per file. The name of the file
SHOULD be of the form:

    module-or-submodule-name ['@' revision-date] ('.yang' / '.yin')

"module-or-submodule-name" is the name of the module or submodule,
and the optional "revision-date" is the latest revision of the module
or submodule, as defined by the "revision" statement (Section 7.1.9).

The file extension ".yang" denotes that the contents of the file are
written with YANG syntax (Section 6), and ".yin" denotes that the
contents of the file are written with YIN syntax (Section 13).

YANG parsers can find imported modules and included submodules via
this convention.

5.3. XML Namespaces

All YANG definitions are specified within a module. Each module is
bound to a distinct XML namespace [XML-NAMES], which is a globally
unique URI [RFC3986]. A NETCONF client or server uses the namespace
during XML encoding of data.

XML namespaces for modules published in RFC streams [RFC4844] MUST be
assigned by IANA; see Section 14 in [RFC6020].

XML namespaces for private modules are assigned by the organization
owning the module without a central registry. Namespace URIs MUST be
chosen so they cannot collide with standard or other enterprise
namespaces -- for example, by using the enterprise or organization
name in the namespace.

The "namespace" statement is covered in Section 7.1.3.

5.3.1. YANG XML Namespace

YANG defines an XML namespace for NETCONF <edit-config> operations,
<error-info> content, and the <action> element. The name of this
namespace is "urn:ietf:params:xml:ns:yang:1".

5.4. Resolving Grouping, Type, and Identity Names

   Grouping, type, and identity names are resolved in the context in
   which they are defined, rather than the context in which they are
   used. Users of groupings, typedefs, and identities are not required
   to import modules or include submodules to satisfy all references
   made by the original definition. This behaves like static scoping in
   a conventional programming language.

   For example, if a module defines a grouping in which a type is
   referenced, when the grouping is used in a second module, the type is
   resolved in the context of the original module, not the second
   module. There is no ambiguity if both modules define the type.

5.5. Nested Typedefs and Groupings

   Typedefs and groupings may appear nested under many YANG statements,
   allowing these to be lexically scoped by the statement hierarchy
   under which they appear. This allows types and groupings to be
   defined near where they are used, rather than placing them at the
   top level of the hierarchy. The close proximity increases
   readability.

   Scoping also allows types to be defined without concern for naming
   conflicts between types in different submodules. Type names can be
   specified without adding leading strings designed to prevent name
   collisions within large modules.

   Finally, scoping allows the module author to keep types and groupings
   private to their module or submodule, preventing their reuse. Since
   only top-level types and groupings (i.e., those appearing as
   substatements to a "module" or "submodule" statement) can be used
   outside the module or submodule, the developer has more control over
   what pieces of their module are presented to the outside world,
   supporting the need to hide internal information and maintaining a
   boundary between what is shared with the outside world and what is
   kept private.

   Scoped definitions MUST NOT shadow definitions at a higher scope. A
   type or grouping cannot be defined if a higher level in the statement
   hierarchy has a definition with a matching identifier.

   A reference to an unprefixed type or grouping, or one that uses the
   prefix of the current module, is resolved by locating the matching
   "typedef" or "grouping" statement among the immediate substatements
   of each ancestor statement.

5.6. Conformance

Conformance to a model is a measure of how accurately a server
follows the model. Generally speaking, servers are responsible for
implementing the model faithfully, allowing applications to treat
servers that implement the model identically. Deviations from the
model can reduce the utility of the model and increase the fragility
of applications that use it.

YANG modelers have three mechanisms for conformance:

o the basic behavior of the model

o optional features that are part of the model

o deviations from the model

We will consider each of these in sequence.

5.6.1. Basic Behavior

The model defines a contract between a YANG-based client and server;
this contract allows both parties to have faith that the other knows
the syntax and semantics behind the modeled data. The strength of
YANG lies in the strength of this contract.

5.6.2. Optional Features

In many models, the modeler will allow sections of the model to be
conditional. The server controls whether these conditional portions
of the model are supported or valid for that particular server.

For example, a syslog data model may choose to include the ability to
save logs locally, but the modeler will realize that this is only
possible if the server has local storage. If there is no local
storage, an application should not tell the server to save logs.

YANG supports this conditional mechanism using a construct called
"feature". Features give the modeler a mechanism for making portions
of the module conditional in a manner that is controlled by the
server. The model can express constructs that are not universally
present in all servers. These features are included in the model
definition, allowing a consistent view and allowing applications to
learn which features are supported and tailor their behavior to the
server.

A module may declare any number of features, identified by simple
strings, and may make portions of the module optional based on those
features. If the server supports a feature, then the corresponding
portions of the module are valid for that server. If the server
doesn't support the feature, those parts of the module are not valid,
and applications should behave accordingly.

Features are defined using the "feature" statement. Definitions in
the module that are conditional to the feature are noted by the
"if-feature" statement.

Further details are available in Section 7.20.1.

5.6.3. Deviations

In an ideal world, all servers would be required to implement the
model exactly as defined, and deviations from the model would not be
allowed. But in the real world, servers are often not able or
designed to implement the model as written. For YANG-based
automation to deal with these server deviations, a mechanism must
exist for servers to inform applications of the specifics of such
deviations.

For example, a BGP module may allow any number of BGP peers, but a
particular server may only support 16 BGP peers. Any application
configuring the 17th peer will receive an error. While an error may
suffice to let the application know it cannot add another peer, it
would be far better if the application had prior knowledge of this
limitation and could prevent the user from starting down the path
that could not succeed.

Server deviations are declared using the "deviation" statement, which
takes as its argument a string that identifies a node in the schema
tree. The contents of the statement detail the manner in which the
server implementation deviates from the contract as defined in the
module.

Further details are available in Section 7.20.3.

5.6.4. Announcing Conformance Information in NETCONF

This document defines the following mechanism for announcing
conformance information. Other mechanisms may be defined by future
specifications.

A NETCONF server MUST announce the modules it implements (see
Section 5.6.5) by implementing the YANG module "ietf-yang-library"
defined in [RFC7895] and listing all implemented modules in the
"/modules-state/module" list.

The server also MUST advertise the following capability in the
<hello> message (line breaks and whitespaces are used for formatting
reasons only):

```
urn:ietf:params:netconf:capability:yang-library:1.0?
  revision=<date>&module-set-id=<id>
```

The parameter "revision" has the same value as the revision date of
the "ietf-yang-library" module implemented by the server. This
parameter MUST be present.

The parameter "module-set-id" has the same value as the leaf
"/modules-state/module-set-id" from "ietf-yang-library". This
parameter MUST be present.

With this mechanism, a client can cache the supported modules for a
server and only update the cache if the "module-set-id" value in the
<hello> message changes.

5.6.5. Implementing a Module

A server implements a module if it implements the module's data
nodes, RPCs, actions, notifications, and deviations.

A server MUST NOT implement more than one revision of a module.

If a server implements a module A that imports a module B, and A uses
any node from B in an "augment" or "path" statement that the server
supports, then the server MUST implement a revision of module B that
has these nodes defined. This is regardless of whether module B is
imported by revision or not.

   If a server implements a module A that imports a module C without
   specifying the revision date of module C and the server does not
   implement C (e.g., if C only defines some typedefs), the server MUST
   list module C in the "/modules-state/module" list from
   "ietf-yang-library" [RFC7895], and it MUST set the leaf
   "conformance-type" to "import" for this module.

   If a server lists a module C in the "/modules-state/module" list from
   "ietf-yang-library" and there are other modules Ms listed that import
   C without specifying the revision date of module C, the server MUST
   use the definitions from the most recent revision of C listed for
   modules Ms.

   The reason for these rules is that clients need to be able to know
   the specific data model structure and types of all leafs and
   leaf-lists implemented in a server.

   For example, with these modules:

```
     module a {
       yang-version 1.1;
       namespace "urn:example:a";
       prefix "a";

       import b {
         revision-date 2015-01-01;
       }
       import c;

       revision 2015-01-01;

       feature foo;

       augment "/b:x" {
         if-feature foo;

         leaf y {
           type b:myenum;
         }
       }

       container a {
         leaf x {
           type c:bar;
         }
       }
     }
```

```
module b {
  yang-version 1.1;
  namespace "urn:example:b";
  prefix "b";

  revision 2015-01-01;

  typedef myenum {
    type enumeration {
      enum zero;
    }
  }

  container x {
  }
}

module b {
  yang-version 1.1;
  namespace "urn:example:b";
  prefix "b";

  revision 2015-04-04;
  revision 2015-01-01;

  typedef myenum {
    type enumeration {
      enum zero; // added in 2015-01-01
      enum one;  // added in 2015-04-04
    }
  }

  container x {  // added in 2015-01-01
    container y; // added in 2015-04-04
  }
}

module c {
  yang-version 1.1;
  namespace "urn:example:c";
  prefix "c";

  revision 2015-02-02;

  typedef bar {
    ...
  }
}
```

```
   module c {
     yang-version 1.1;
     namespace "urn:example:c";
     prefix "c";

     revision 2015-03-03;
     revision 2015-02-02;

     typedef bar {
       ...
     }
   }
```

A server that implements revision "2015-01-01" of module "a" and
supports feature "foo" can implement revision "2015-01-01" or
"2015-04-04" of module "b". Since "b" was imported by revision, the
type of leaf "/b:x/a:y" is the same, regardless of which revision of
"b" the server implements.

A server that implements module "a" but does not support feature
"foo" does not have to implement module "b".

A server that implements revision "2015-01-01" of module "a"
picks any revision of module "c" and lists it in the
"/modules-state/module" list from "ietf-yang-library".

The following XML encoding example shows valid data for the
"/modules-state/module" list for a server that implements module "a":

```
   <modules-state
       xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
     <module-set-id>ee1ecb017370cafd</module-set-id>
     <module>
       <name>a</name>
       <revision>2015-01-01</revision>
       <namespace>urn:example:a</namespace>
       <feature>foo</feature>
       <conformance-type>implement</conformance-type>
     </module>
     <module>
       <name>b</name>
       <revision>2015-04-04</revision>
       <namespace>urn:example:b</namespace>
       <conformance-type>implement</conformance-type>
     </module>
```

```
    <module>
      <name>c</name>
      <revision>2015-02-02</revision>
      <namespace>urn:example:c</namespace>
      <conformance-type>import</conformance-type>
    </module>
  </modules-state>
```

5.7. Datastore Modification

Data models may allow the server to alter the configuration datastore
in ways not explicitly directed via network management protocol
messages. For example, a data model may define leafs that are
assigned system-generated values when the client does not provide
one. A formal mechanism for specifying the circumstances where these
changes are allowed is out of scope for this specification.

6. YANG Syntax

The YANG syntax is similar to that of SMIng [RFC3780] and programming
languages like C and C++. This C-like syntax was chosen specifically
for its readability, since YANG values the time and effort of the
readers of models above those of modules writers and YANG tool-chain
developers. This section introduces the YANG syntax.

Legal characters in YANG modules are the Unicode and ISO/IEC 10646
[ISO.10646] characters, including tab, carriage return, and line feed
but excluding the other C0 control characters, the surrogate blocks,
and the noncharacters. The character syntax is formally defined by
the rule "yang-char" in Section 14.

YANG modules and submodules are stored in files using the UTF-8
[RFC3629] character encoding.

Lines in a YANG module end with a carriage return-line feed
combination or with a line feed alone. A carriage return that is not
followed by a line feed may only appear inside a quoted string
(Section 6.1.3). Note that carriage returns and line feeds that
appear inside quoted strings become part of the value of the string
without modification; the value of a multi-line quoted string
contains the same form of line ends as those lines of the YANG
module.

6.1. Lexical Tokenization

   YANG modules are parsed as a series of tokens. This section details
   the rules for recognizing tokens from an input stream. YANG
   tokenization rules are both simple and powerful. The simplicity is
   driven by a need to keep the parsers easy to implement, while the
   power is driven by the fact that modelers need to express their
   models in readable formats.

6.1.1. Comments

   Comments are C++ style. A single line comment starts with "//" and
   ends at the end of the line. A block comment starts with "/*" and
   ends with the nearest following "*/".

   Note that inside a quoted string (Section 6.1.3), these character
   pairs are never interpreted as the start or end of a comment.

6.1.2. Tokens

   A token in YANG is either a keyword, a string, a semicolon (";"), or
   braces ("{" or "}"). A string can be quoted or unquoted. A keyword
   is either one of the YANG keywords defined in this document, or a
   prefix identifier, followed by a colon (":"), followed by a language
   extension keyword. Keywords are case sensitive. See Section 6.2 for
   a formal definition of identifiers.

6.1.3. Quoting

   An unquoted string is any sequence of characters that does not
   contain any space, tab, carriage return, or line feed characters, a
   single or double quote character, a semicolon (";"), braces ("{" or
   "}"), or comment sequences ("//", "/*", or "*/").

   Note that any keyword can legally appear as an unquoted string.

   Within an unquoted string, every character is preserved. Note that
   this means that the backslash character does not have any special
   meaning in an unquoted string.

   If a double-quoted string contains a line break followed by space or
   tab characters that are used to indent the text according to the
   layout in the YANG file, this leading whitespace is stripped from the
   string, up to and including the column of the starting double quote
   character, or to the first non-whitespace character, whichever occurs
   first. Any tab character in a succeeding line that must be examined
   for stripping is first converted into 8 space characters.

If a double-quoted string contains space or tab characters before a
line break, this trailing whitespace is stripped from the string.

A single-quoted string (enclosed within ' ') preserves each character
within the quotes. A single quote character cannot occur in a
single-quoted string, even when preceded by a backslash.

Within a double-quoted string (enclosed within " "), a backslash
character introduces a representation of a special character, which
depends on the character that immediately follows the backslash:

     \n   newline
     \t   a tab character
     \"   a double quote
     \\   a single backslash

The backslash MUST NOT be followed by any other character.

If a quoted string is followed by a plus character ("+"), followed by
another quoted string, the two strings are concatenated into one
string, allowing multiple concatenations to build one string.
Whitespace, line breaks, and comments are allowed between the quoted
strings and the plus character.

In double-quoted strings, whitespace trimming is done before
substitution of backslash-escaped characters. Concatenation is
performed as the last step.

6.1.3.1. Quoting Examples

The following strings are equivalent:

     hello
     "hello"
     'hello'
     "hel" + "lo"
     'hel' + "lo"

The following examples show some special strings:

     "\"" - string containing a double quote
     '"'   - string containing a double quote
     "\n" - string containing a newline character
     '\n' - string containing a backslash followed
               by the character n

   The following examples show some illegal strings:

     '''' - a single-quoted string cannot contain single quotes
     """" - a double quote must be escaped in a double-quoted string

   The following strings are equivalent:

         "first line
            second line"

     "first line\n" + " second line"

6.2. Identifiers

   Identifiers are used to identify different kinds of YANG items by
   name. Each identifier starts with an uppercase or lowercase ASCII
   letter or an underscore character, followed by zero or more ASCII
   letters, digits, underscore characters, hyphens, and dots.
   Implementations MUST support identifiers up to 64 characters in
   length and MAY support longer identifiers. Identifiers are case
   sensitive. The identifier syntax is formally defined by the rule
   "identifier" in Section 14. Identifiers can be specified as quoted
   or unquoted strings.

6.2.1. Identifiers and Their Namespaces

   Each identifier is valid in a namespace that depends on the type of
   the YANG item being defined. All identifiers defined in a namespace
   MUST be unique.

   o All module and submodule names share the same global module
      identifier namespace.

   o All extension names defined in a module and its submodules share
      the same extension identifier namespace.

   o All feature names defined in a module and its submodules share the
      same feature identifier namespace.

   o All identity names defined in a module and its submodules share
      the same identity identifier namespace.

   o All derived type names defined within a parent node or at the top
      level of the module or its submodules share the same type
      identifier namespace. This namespace is scoped to all descendant
      nodes of the parent node or module. This means that any
      descendant node may use that typedef, and it MUST NOT define a
      typedef with the same name.

   o All grouping names defined within a parent node or at the top
      level of the module or its submodules share the same grouping
      identifier namespace. This namespace is scoped to all descendant
      nodes of the parent node or module. This means that any
      descendant node may use that grouping, and it MUST NOT define a
      grouping with the same name.

   o All leafs, leaf-lists, lists, containers, choices, rpcs, actions,
      notifications, anydatas, and anyxmls defined (directly or through
      a "uses" statement) within a parent node or at the top level of
      the module or its submodules share the same identifier namespace.
      This namespace is scoped to the parent node or module, unless the
      parent node is a case node. In that case, the namespace is scoped
      to the closest ancestor node that is not a case or choice node.

   o All cases within a choice share the same case identifier
      namespace. This namespace is scoped to the parent choice node.

   Forward references are allowed in YANG.

6.3. Statements

   A YANG module contains a sequence of statements. Each statement
   starts with a keyword, followed by zero or one argument, followed by
   either a semicolon (";") or a block of substatements enclosed within
   braces ("{ }"):

      statement = keyword [argument] (";" / "{" *statement "}")

   The argument is a string, as defined in Section 6.1.2.

6.3.1. Language Extensions

   A module can introduce YANG extensions by using the "extension"
   keyword (see Section 7.19). The extensions can be imported by other
   modules with the "import" statement (see Section 7.1.5). When an
   imported extension is used, the extension's keyword MUST be qualified
   using the prefix with which the extension's module was imported. If
   an extension is used in the module where it is defined, the
   extension's keyword MUST be qualified with the prefix of this module.

   The processing of extensions depends on whether support for those
   extensions is claimed for a given YANG parser or the tool set in
   which it is embedded. An unsupported extension appearing in a YANG
   module as an unknown-statement (see Section 14) MAY be ignored in its
   entirety. Any supported extension MUST be processed in accordance
   with the specification governing that extension.

Care must be taken when defining extensions so that modules that use
the extensions are meaningful also for applications that do not
support the extensions.

6.4. XPath Evaluations

YANG relies on XML Path Language (XPath) 1.0 [XPATH] as a notation
for specifying many inter-node references and dependencies. An
implementation is not required to implement an XPath interpreter but
MUST ensure that the requirements encoded in the data model are
enforced. The manner of enforcement is an implementation decision.
The XPath expressions MUST be syntactically correct, and all prefixes
used MUST be present in the XPath context (see Section 6.4.1). An
implementation may choose to implement them by hand, rather than
using the XPath expression directly.

The data model used in the XPath expressions is the same as that used
in XPath 1.0 [XPATH], with the same extension for root node children
as used by XSLT 1.0 (see Section 3.1 in [XSLT]). Specifically, it
means that the root node may have any number of element nodes as its
children.

The data tree has no concept of document order. An implementation
needs to choose some document order, but how it is done is an
implementation decision. This means that XPath expressions in YANG
modules SHOULD NOT rely on any specific document order.

Numbers in XPath 1.0 are IEEE 754 [IEEE754-2008] double-precision
floating-point values; see Section 3.5 in [XPATH]. This means that
some values of int64, uint64, and decimal64 types (see Sections 9.2
and 9.3) cannot be exactly represented in XPath expressions.
Therefore, due caution should be exercised when using nodes with
64-bit numeric values in XPath expressions. In particular, numerical
comparisons involving equality may yield unexpected results.

For example, consider the following definition:

```
leaf lxiv {
  type decimal64 {
    fraction-digits 18;
  }
  must ". <= 10";
}
```

An instance of the "lxiv" leaf having the value of
10.0000000000000001 will then successfully pass validation.

6.4.1. XPath Context

   All YANG XPath expressions share the following XPath context
   definition:

   o The set of namespace declarations is the set of all "import"
      statements' prefix and namespace pairs in the module where the
      XPath expression is specified, and the "prefix" statement's prefix
      for the "namespace" statement's URI.

   o Names without a namespace prefix belong to the same namespace as
      the identifier of the current node. Inside a grouping, that
      namespace is affected by where the grouping is used (see
      Section 7.13). Inside a typedef, that namespace is affected by
      where the typedef is referenced. If a typedef is defined and
      referenced within a grouping, the namespace is affected by where
      the grouping is used (see Section 7.13).

   o The function library is the core function library defined in
      [XPATH] and the functions defined in Section 10.

   o The set of variable bindings is empty.

   The mechanism for handling unprefixed names is adopted from XPath 2.0
   [XPATH2.0] and helps simplify XPath expressions in YANG. No
   ambiguity may ever arise, because YANG node identifiers are always
   qualified names with a non-null namespace URI.

   The accessible tree depends on where the statement with the XPath
   expression is defined:

   o If the XPath expression is defined in a substatement to a data
      node that represents configuration, the accessible tree is the
      data in the datastore where the context node exists. The root
      node has all top-level configuration data nodes in all modules as
      children.

   o If the XPath expression is defined in a substatement to a data
      node that represents state data, the accessible tree is all state
      data in the server, and the running configuration datastore. The
      root node has all top-level data nodes in all modules as children.

   o If the XPath expression is defined in a substatement to a
      "notification" statement, the accessible tree is the notification
      instance, all state data in the server, and the running
      configuration datastore. If the notification is defined on the
      top level in a module, then the root node has the node

representing the notification being defined and all top-level data
nodes in all modules as children. Otherwise, the root node has
all top-level data nodes in all modules as children.

o If the XPath expression is defined in a substatement to an "input"
   statement in an "rpc" or "action" statement, the accessible tree
   is the RPC or action operation instance, all state data in the
   server, and the running configuration datastore. The root node
   has top-level data nodes in all modules as children.
   Additionally, for an RPC, the root node also has the node
   representing the RPC operation being defined as a child. The node
   representing the operation being defined has the operation's input
   parameters as children.

o If the XPath expression is defined in a substatement to an
   "output" statement in an "rpc" or "action" statement, the
   accessible tree is the RPC or action operation instance, all state
   data in the server, and the running configuration datastore. The
   root node has top-level data nodes in all modules as children.
   Additionally, for an RPC, the root node also has the node
   representing the RPC operation being defined as a child. The node
   representing the operation being defined has the operation's
   output parameters as children.

In the accessible tree, all leafs and leaf-lists with default values
in use exist (see Sections 7.6.1 and 7.7.2).

If a node that exists in the accessible tree has a non-presence
container as a child, then the non-presence container also exists in
the accessible tree.

The context node varies with the YANG XPath expression and is
specified where the YANG statement with the XPath expression is
defined.

6.4.1.1. Examples

   Given the following module:

```
module example-a {
  yang-version 1.1;
  namespace urn:example:a;
  prefix a;

  container a {
    list b {
      key id;
      leaf id {
        type string;
      }
      notification down {
        leaf reason {
          type string;
        }
      }
      action reset {
        input {
          leaf delay {
            type uint32;
          }
        }
        output {
          leaf result {
            type string;
          }
        }
      }
    }
  }
  notification failure {
    leaf b-ref {
      type leafref {
        path "/a/b/id";
      }
    }
  }
}
```

and given the following data tree, specified in XML:

```
<a xmlns="urn:example:a">
  <b>
    <id>1</id>
  </b>
  <b>
    <id>2</id>
  </b>
</a>
```

The accessible tree for a notification "down" on /a/b[id="2"] is:

```
<a xmlns="urn:example:a">
  <b>
    <id>1</id>
  </b>
  <b>
    <id>2</id>
    <down>
      <reason>error</reason>
    </down>
  </b>
</a>
// possibly other top-level nodes here
```

The accessible tree for an action invocation of "reset" on
/a/b[id="1"] with the "when" parameter set to "10" would be:

```
<a xmlns="urn:example:a">
  <b>
    <id>1</id>
    <reset>
      <delay>10</delay>
    </reset>
  </b>
  <b>
    <id>2</id>
  </b>
</a>
// possibly other top-level nodes here
```

   The accessible tree for the action output of this action is:

```
    <a xmlns="urn:example:a">
      <b>
        <id>1</id>
        <reset>
          <result>ok</result>
        </reset>
      </b>
      <b>
        <id>2</id>
      </b>
    </a>
    // possibly other top-level nodes here
```

   The accessible tree for a notification "failure" could be:

```
    <a xmlns="urn:example:a">
      <b>
        <id>1</id>
      </b>
      <b>
        <id>2</id>
      </b>
    </a>
    <failure>
      <b-ref>2</b-ref>
    </failure>
    // possibly other top-level nodes here
```

6.5. Schema Node Identifier

   A schema node identifier is a string that identifies a node in the
   schema tree. It has two forms, "absolute" and "descendant", defined
   by the rules "absolute-schema-nodeid" and "descendant-schema-nodeid"
   in Section 14, respectively. A schema node identifier consists of a
   path of identifiers, separated by slashes ("/"). In an absolute
   schema node identifier, the first identifier after the leading slash
   is any top-level schema node in the local module or in an imported
   module.

   References to identifiers defined in external modules MUST be
   qualified with appropriate prefixes, and references to identifiers
   defined in the current module and its submodules MAY use a prefix.

   For example, to identify the child node "b" of top-level node "a",
   the string "/a/b" can be used.

7. YANG Statements

   The following sections describe all of the YANG statements.

   Note that even a statement that does not have any substatements
   defined in YANG can have vendor-specific extensions as substatements.
   For example, the "description" statement does not have any
   substatements defined in YANG, but the following is legal:

      description "Some text." {
        ex:documentation-flag 5;
      }

7.1. The "module" Statement

   The "module" statement defines the module's name and groups all
   statements that belong to the module together. The "module"
   statement's argument is the name of the module, followed by a block
   of substatements that holds detailed module information. The module
   name is an identifier (see Section 6.2).

   Names of modules published in RFC streams [RFC4844] MUST be assigned
   by IANA; see Section 14 in [RFC6020].

   Private module names are assigned by the organization owning the
   module without a central registry. See Section 5.1 for
   recommendations on how to name modules.

   A module typically has the following layout:

      module <module-name> {

        // header information
        <yang-version statement>
        <namespace statement>
        <prefix statement>

        // linkage statements
        <import statements>
        <include statements>

        // meta-information
        <organization statement>
        <contact statement>
        <description statement>
        <reference statement>

```
      // revision history
      <revision statements>

      // module definitions
      <other statements>
    }
```

7.1.1. The module's Substatements

```
        +--------------+---------+-------------+
        | substatement | section | cardinality |
        +--------------+---------+-------------+
        | anydata      | 7.10    | 0..n        |
        | anyxml       | 7.11    | 0..n        |
        | augment      | 7.17    | 0..n        |
        | choice       | 7.9     | 0..n        |
        | contact      | 7.1.8   | 0..1        |
        | container    | 7.5     | 0..n        |
        | description  | 7.21.3  | 0..1        |
        | deviation    | 7.20.3  | 0..n        |
        | extension    | 7.19    | 0..n        |
        | feature      | 7.20.1  | 0..n        |
        | grouping     | 7.12    | 0..n        |
        | identity     | 7.18    | 0..n        |
        | import       | 7.1.5   | 0..n        |
        | include      | 7.1.6   | 0..n        |
        | leaf         | 7.6     | 0..n        |
        | leaf-list    | 7.7     | 0..n        |
        | list         | 7.8     | 0..n        |
        | namespace    | 7.1.3   | 1           |
        | notification | 7.16    | 0..n        |
        | organization | 7.1.7   | 0..1        |
        | prefix       | 7.1.4   | 1           |
        | reference    | 7.21.4  | 0..1        |
        | revision     | 7.1.9   | 0..n        |
        | rpc          | 7.14    | 0..n        |
        | typedef      | 7.3     | 0..n        |
        | uses         | 7.13    | 0..n        |
        | yang-version | 7.1.2   | 1           |
        +--------------+---------+-------------+
```

7.1.2. The "yang-version" Statement

   The "yang-version" statement specifies which version of the YANG
   language was used in developing the module. The statement's argument
   is a string. It MUST contain the value "1.1" for YANG modules
   defined based on this specification.

   A module or submodule that doesn't contain the "yang-version"
   statement, or one that contains the value "1", is developed for YANG
   version 1, defined in [RFC6020].

   Handling of the "yang-version" statement for versions other than
   "1.1" (the version defined here) is out of scope for this
   specification. Any document that defines a higher version will need
   to define the backward compatibility of such a higher version.

   For compatibility between YANG versions 1 and 1.1, see Section 12.

7.1.3. The "namespace" Statement

   The "namespace" statement defines the XML namespace that all
   identifiers defined by the module are qualified by in the XML
   encoding, with the exception of identifiers for data nodes, action
   nodes, and notification nodes defined inside a grouping (see
   Section 7.13 for details). The argument to the "namespace" statement
   is the URI of the namespace.

   See also Section 5.3.

7.1.4. The "prefix" Statement

   The "prefix" statement is used to define the prefix associated with
   the module and its namespace. The "prefix" statement's argument is
   the prefix string that is used as a prefix to access a module. The
   prefix string MAY be used with the module to refer to definitions
   contained in the module, e.g., "if:ifName". A prefix is an
   identifier (see Section 6.2).

   When used inside the "module" statement, the "prefix" statement
   defines the prefix suggested to be used when this module is imported.

   To improve readability of the NETCONF XML, a NETCONF client or server
   that generates XML or XPath that uses prefixes SHOULD use the prefix
   defined by the module as the XML namespace prefix, unless there is a
   conflict.

When used inside the "import" statement, the "prefix" statement
defines the prefix to be used when accessing definitions inside the
imported module. When a reference to an identifier from the imported
module is used, the prefix string for the imported module followed by
a colon (":") and the identifier is used, e.g., "if:ifIndex". To
improve readability of YANG modules, the prefix defined by a module
SHOULD be used when the module is imported, unless there is a
conflict. If there is a conflict, i.e., two different modules that
both have defined the same prefix are imported, at least one of them
MUST be imported with a different prefix.

All prefixes, including the prefix for the module itself, MUST be
unique within the module or submodule.

7.1.5. The "import" Statement

The "import" statement makes definitions from one module available
inside another module or submodule. The argument is the name of the
module to import, and the statement is followed by a block of
substatements that holds detailed import information. When a module
is imported, the importing module may:

o   use any grouping and typedef defined at the top level in the
    imported module or its submodules.

o   use any extension, feature, and identity defined in the imported
    module or its submodules.

o   use any node in the imported module's schema tree in "must",
    "path", and "when" statements, or as the target node in "augment"
    and "deviation" statements.

The mandatory "prefix" substatement assigns a prefix for the imported
module that is scoped to the importing module or submodule. Multiple
"import" statements may be specified to import from different
modules.

When the optional "revision-date" substatement is present, any
typedef, grouping, extension, feature, and identity referenced by
definitions in the local module are taken from the specified revision
of the imported module. It is an error if the specified revision of
the imported module does not exist. If no "revision-date"
substatement is present, it is undefined from which revision of the
module they are taken.

Multiple revisions of the same module can be imported, provided that
different prefixes are used.

```
            +---------------+---------+-------------+
            | substatement  | section | cardinality |
            +---------------+---------+-------------+
            | description   | 7.21.3  | 0..1        |
            | prefix        | 7.1.4   | 1           |
            | reference     | 7.21.4  | 0..1        |
            | revision-date | 7.1.5.1 | 0..1        |
            +---------------+---------+-------------+
```

                   The import's Substatements

7.1.5.1. The import's "revision-date" Statement

   The import's "revision-date" statement is used to specify the version
   of the module to import.

7.1.6. The "include" Statement

   The "include" statement is used to make content from a submodule
   available to that submodule's parent module. The argument is an
   identifier that is the name of the submodule to include. Modules are
   only allowed to include submodules that belong to that module, as
   defined by the "belongs-to" statement (see Section 7.2.2).

   When a module includes a submodule, it incorporates the contents of
   the submodule into the node hierarchy of the module.

   For backward compatibility with YANG version 1, a submodule is
   allowed to include another submodule belonging to the same module,
   but this is not necessary in YANG version 1.1 (see Section 5.1).

   When the optional "revision-date" substatement is present, the
   specified revision of the submodule is included in the module. It is
   an error if the specified revision of the submodule does not exist.
   If no "revision-date" substatement is present, it is undefined which
   revision of the submodule is included.

   Multiple revisions of the same submodule MUST NOT be included.

```
            +---------------+---------+-------------+
            | substatement  | section | cardinality |
            +---------------+---------+-------------+
            | description   | 7.21.3  | 0..1        |
            | reference     | 7.21.4  | 0..1        |
            | revision-date | 7.1.5.1 | 0..1        |
            +---------------+---------+-------------+
```

                  The includes's Substatements

7.1.7. The "organization" Statement

   The "organization" statement defines the party responsible for this
   module. The argument is a string that is used to specify a textual
   description of the organization(s) under whose auspices this module
   was developed.

7.1.8. The "contact" Statement

   The "contact" statement provides contact information for the module.
   The argument is a string that is used to specify contact information
   for the person or persons to whom technical queries concerning this
   module should be sent, such as their name, postal address, telephone
   number, and electronic mail address.

7.1.9. The "revision" Statement

   The "revision" statement specifies the editorial revision history of
   the module, including the initial revision. A series of "revision"
   statements detail the changes in the module's definition. The
   argument is a date string in the format "YYYY-MM-DD", followed by a
   block of substatements that holds detailed revision information. A
   module SHOULD have at least one "revision" statement. For every
   published editorial change, a new one SHOULD be added in front of the
   revisions sequence so that all revisions are in reverse chronological
   order.

7.1.9.1. The revision's Substatements

```
            +--------------+---------+-------------+
            | substatement | section | cardinality |
            +--------------+---------+-------------+
            | description  | 7.21.3  | 0..1        |
            | reference    | 7.21.4  | 0..1        |
            +--------------+---------+-------------+
```

7.1.10. Usage Example

   The following example relies on [RFC6991].

```
module example-system {
  yang-version 1.1;
  namespace "urn:example:system";
  prefix "sys";

  import ietf-yang-types {
    prefix "yang";
    reference "RFC 6991: Common YANG Data Types";
  }

  include example-types;

  organization "Example Inc.";
  contact
    "Joe L. User

     Example Inc.
     42 Anywhere Drive
     Nowhere, CA 95134
     USA

     Phone: +1 800 555 0100
     Email: joe@example.com";

  description
    "The module for entities implementing the Example system.";

  revision 2007-06-09 {
    description "Initial revision.";
  }

  // definitions follow...
}
```

7.2. The "submodule" Statement

   While the primary unit in YANG is a module, a YANG module can itself
   be constructed out of several submodules. Submodules allow a module
   designer to split a complex model into several pieces where all the
   submodules contribute to a single namespace, which is defined by the
   module that includes the submodules.

   The "submodule" statement defines the submodule's name, and it groups
   all statements that belong to the submodule together. The
   "submodule" statement's argument is the name of the submodule,
   followed by a block of substatements that holds detailed submodule
   information. The submodule name is an identifier (see Section 6.2).

   Names of submodules published in RFC streams [RFC4844] MUST be
   assigned by IANA; see Section 14 in [RFC6020].

   Private submodule names are assigned by the organization owning the
   submodule without a central registry. See Section 5.1 for
   recommendations on how to name submodules.

   A submodule typically has the following layout:

   submodule <module-name> {

     <yang-version statement>

     // module identification
     <belongs-to statement>

     // linkage statements
     <import statements>

     // meta-information
     <organization statement>
     <contact statement>
     <description statement>
     <reference statement>

     // revision history
     <revision statements>

     // module definitions
     <other statements>
   }

7.2.1. The submodule's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| augment      | 7.17    | 0..n        |
| belongs-to   | 7.2.2   | 1           |
| choice       | 7.9     | 0..n        |
| contact      | 7.1.8   | 0..1        |
| container    | 7.5     | 0..n        |
| description  | 7.21.3  | 0..1        |
| deviation    | 7.20.3  | 0..n        |
| extension    | 7.19    | 0..n        |
| feature      | 7.20.1  | 0..n        |
| grouping     | 7.12    | 0..n        |
| identity     | 7.18    | 0..n        |
| import       | 7.1.5   | 0..n        |
| include      | 7.1.6   | 0..n        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| notification | 7.16    | 0..n        |
| organization | 7.1.7   | 0..1        |
| reference    | 7.21.4  | 0..1        |
| revision     | 7.1.9   | 0..n        |
| rpc          | 7.14    | 0..n        |
| typedef      | 7.3     | 0..n        |
| uses         | 7.13    | 0..n        |
| yang-version | 7.1.2   | 1           |
+--------------+---------+-------------+
```

7.2.2. The "belongs-to" Statement

The "belongs-to" statement specifies the module to which the
submodule belongs. The argument is an identifier that is the name of
the module.

A submodule MUST only be included by either the module to which it
belongs or another submodule that belongs to that module.

The mandatory "prefix" substatement assigns a prefix for the module
to which the submodule belongs. All definitions in the module that
the submodule belongs to and all its submodules can be accessed by
using the prefix.

```
                +--------------+---------+-------------+
                | substatement | section | cardinality |
                +--------------+---------+-------------+
                | prefix       | 7.1.4   | 1           |
                +--------------+---------+-------------+
```

                   The belongs-to's Substatement

7.2.3. Usage Example

```
    submodule example-types {
      yang-version 1.1;
      belongs-to "example-system" {
        prefix "sys";
      }

      import ietf-yang-types {
        prefix "yang";
      }

      organization "Example Inc.";
      contact
        "Joe L. User

         Example Inc.
         42 Anywhere Drive
         Nowhere, CA 95134
         USA

         Phone: +1 800 555 0100
         Email: joe@example.com";

      description
        "This submodule defines common Example types.";

      revision "2007-06-09" {
        description "Initial revision.";
      }

      // definitions follow...
    }
```

7.3. The "typedef" Statement

The "typedef" statement defines a new type that may be used locally
in the module or submodule, and by other modules that import from it,
according to the rules in Section 5.5. The new type is called the
"derived type", and the type from which it was derived is called the
"base type". All derived types can be traced back to a YANG
built-in type.

The "typedef" statement's argument is an identifier that is the name
of the type to be defined and MUST be followed by a block of
substatements that holds detailed typedef information.

The name of the type MUST NOT be one of the YANG built-in types. If
the typedef is defined at the top level of a YANG module or
submodule, the name of the type to be defined MUST be unique within
the module.

7.3.1. The typedef's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| default      | 7.3.4   | 0..1        |
| description  | 7.21.3  | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| type         | 7.3.2   | 1           |
| units        | 7.3.3   | 0..1        |
+--------------+---------+-------------+
```

7.3.2. The typedef's "type" Statement

The "type" statement, which MUST be present, defines the base type
from which this type is derived. See Section 7.4 for details.

7.3.3. The "units" Statement

The "units" statement, which is optional, takes as an argument a
string that contains a textual definition of the units associated
with the type.

7.3.4. The typedef's "default" Statement

The "default" statement takes as an argument a string that contains a
default value for the new type.

The value of the "default" statement MUST be valid according to the
type specified in the "type" statement.

If the base type has a default value and the new derived type does
not specify a new default value, the base type's default value is
also the default value of the new derived type.

If the type's default value is not valid according to the new
restrictions specified in a derived type or leaf definition, the
derived type or leaf definition MUST specify a new default value
compatible with the restrictions.

7.3.5. Usage Example

```
typedef listen-ipv4-address {
  type inet:ipv4-address;
  default "0.0.0.0";
}
```

7.4. The "type" Statement

The "type" statement takes as an argument a string that is the name
of a YANG built-in type (see Section 9) or a derived type (see
Section 7.3), followed by an optional block of substatements that is
used to put further restrictions on the type.

The restrictions that can be applied depend on the type being
restricted. The restriction statements for all built-in types are
described in the subsections of Section 9.

7.4.1. The type's Substatements

```
+-------------------+---------+-------------+
| substatement      | section | cardinality |
+-------------------+---------+-------------+
| base              | 7.18.2  | 0..n        |
| bit               | 9.7.4   | 0..n        |
| enum              | 9.6.4   | 0..n        |
| fraction-digits   | 9.3.4   | 0..1        |
| length            | 9.4.4   | 0..1        |
| path              | 9.9.2   | 0..1        |
| pattern           | 9.4.5   | 0..n        |
| range             | 9.2.4   | 0..1        |
| require-instance  | 9.9.3   | 0..1        |
| type              | 7.4     | 0..n        |
+-------------------+---------+-------------+
```

7.5. The "container" Statement

The "container" statement is used to define an interior data node in
the schema tree. It takes one argument, which is an identifier,
followed by a block of substatements that holds detailed container
information.

A container node does not have a value, but it has a list of child
nodes in the data tree. The child nodes are defined in the
container's substatements.

7.5.1. Containers with Presence

YANG supports two styles of containers, those that exist only for
organizing the hierarchy of data nodes and those whose presence in
the data tree has an explicit meaning.

In the first style, the container has no meaning of its own, existing
only to contain child nodes. In particular, the presence of the
container node with no child nodes is semantically equivalent to the
absence of the container node. YANG calls this style a "non-presence
container". This is the default style.

For example, the set of scrambling options for Synchronous Optical
Network (SONET) interfaces may be placed inside a "scrambling"
container to enhance the organization of the configuration hierarchy
and to keep these nodes together. The "scrambling" node itself has
no meaning, so removing the node when it becomes empty relieves the
user from performing this task.

In the second style, the presence of the container itself carries
some meaning, representing a single bit of data.

For configuration data, the container acts as both a configuration
knob and a means of organizing related configuration nodes. These
containers are explicitly created and deleted.

YANG calls this style a "presence container", and it is indicated
using the "presence" statement, which takes as its argument a text
string indicating what the presence of the node means.

For example, an "ssh" container may turn on the ability to log into
the server using Secure SHell (SSH) but can also contain any
SSH-related configuration knobs, such as connection rates or retry
limits.

The "presence" statement (see Section 7.5.5) is used to give
semantics to the existence of the container in the data tree.

7.5.2. The container's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| action       | 7.15    | 0..n        |
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| choice       | 7.9     | 0..n        |
| config       | 7.21.1  | 0..1        |
| container    | 7.5     | 0..n        |
| description  | 7.21.3  | 0..1        |
| grouping     | 7.12    | 0..n        |
| if-feature   | 7.20.2  | 0..n        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| must         | 7.5.3   | 0..n        |
| notification | 7.16    | 0..n        |
| presence     | 7.5.5   | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| typedef      | 7.3     | 0..n        |
| uses         | 7.13    | 0..n        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.5.3. The "must" Statement

The "must" statement, which is optional, takes as an argument a
string that contains an XPath expression (see Section 6.4). It is
used to formally declare a constraint on valid data. The constraint
is enforced according to the rules in Section 8.

When a datastore is validated, all "must" constraints are
conceptually evaluated once for each node in the accessible tree (see
Section 6.4.1).

All such constraints MUST evaluate to "true" for the data to be
valid.

The XPath expression is conceptually evaluated in the following
context, in addition to the definition in Section 6.4.1:

o   If the "must" statement is a substatement of a "notification"
    statement, the context node is the node representing the
    notification in the accessible tree.

o   If the "must" statement is a substatement of an "input" statement,
    the context node is the node representing the operation in the
    accessible tree.

o   If the "must" statement is a substatement of an "output"
    statement, the context node is the node representing the operation
    in the accessible tree.

o   Otherwise, the context node is the node in the accessible tree for
    which the "must" statement is defined.

The result of the XPath expression is converted to a boolean value
using the standard XPath rules.

Note that since all leaf values in the data tree are conceptually
stored in their canonical form (see Section 9.1), any XPath
comparisons are done on the canonical value.

Also note that the XPath expression is conceptually evaluated. This
means that an implementation does not have to use an XPath evaluator
in the server. How the evaluation is done in practice is an
implementation decision.

7.5.4. The must's Substatements

```
+----------------+---------+-------------+
| substatement   | section | cardinality |
+----------------+---------+-------------+
| description    | 7.21.3  | 0..1        |
| error-app-tag  | 7.5.4.2 | 0..1        |
| error-message  | 7.5.4.1 | 0..1        |
| reference      | 7.21.4  | 0..1        |
+----------------+---------+-------------+
```

7.5.4.1. The "error-message" Statement

The "error-message" statement, which is optional, takes a string as
an argument. If the constraint evaluates to "false", the string is
passed as <error-message> in the <rpc-error> in NETCONF.

7.5.4.2. The "error-app-tag" Statement

The "error-app-tag" statement, which is optional, takes a string as
an argument. If the constraint evaluates to "false", the string is
passed as <error-app-tag> in the <rpc-error> in NETCONF.

7.5.4.3. Usage Example of must and error-message

```
   container interface {
     leaf ifType {
       type enumeration {
         enum ethernet;
         enum atm;
       }
     }
     leaf ifMTU {
       type uint32;
     }
     must 'ifType != "ethernet" or ifMTU = 1500' {
       error-message "An Ethernet MTU must be 1500";
     }
     must 'ifType != "atm" or'
       + ' (ifMTU <= 17966 and ifMTU >= 64)' {
       error-message "An ATM MTU must be 64 .. 17966";
     }
   }
```

7.5.5. The "presence" Statement

The "presence" statement assigns a meaning to the presence of a
container in the data tree. It takes as an argument a string that
contains a textual description of what the node's presence means.

If a container has the "presence" statement, the container's
existence in the data tree carries some meaning. Otherwise, the
container is used to give some structure to the data, and it carries
no meaning by itself.

See Section 7.5.1 for additional information.

7.5.6. The container's Child Node Statements

Within a container, the "container", "leaf", "list", "leaf-list",
"uses", "choice", "anydata", and "anyxml" statements can be used to
define child nodes to the container.

7.5.7. XML Encoding Rules

A container node is encoded as an XML element. The element's local
name is the container's identifier, and its namespace is the module's
XML namespace (see Section 7.1.3).

The container's child nodes are encoded as subelements to the
container element. If the container defines RPC or action input or
output parameters, these subelements are encoded in the same order as
they are defined within the "container" statement. Otherwise, the
subelements are encoded in any order.

Any whitespace between the subelements to the container is
insignificant, i.e., an implementation MAY insert whitespace
characters between subelements.

If a non-presence container does not have any child nodes, the
container may or may not be present in the XML encoding.

7.5.8. NETCONF <edit-config> Operations

   Containers can be created, deleted, replaced, and modified through
   <edit-config> by using the "operation" attribute (see Section 7.2 in
   [RFC6241]) in the container's XML element.

   If a container does not have a "presence" statement and the last
   child node is deleted, the NETCONF server MAY delete the container.

   When a NETCONF server processes an <edit-config> request, the
   elements of procedure for the container node are as follows:

   o If the operation is "merge" or "replace", the node is created if
      it does not exist.

   o If the operation is "create", the node is created if it does not
      exist. If the node already exists, a "data-exists" error is
      returned.

   o If the operation is "delete", the node is deleted if it exists.
      If the node does not exist, a "data-missing" error is returned.

7.5.9. Usage Example

   Given the following container definition:

```
     container system {
       description
         "Contains various system parameters.";
       container services {
         description
           "Configure externally available services.";
         container "ssh" {
           presence "Enables SSH";
           description
             "SSH service-specific configuration.";
           // more leafs, containers, and stuff here...
         }
       }
     }
```

A corresponding XML instance example:

```
<system>
  <services>
    <ssh/>
  </services>
</system>
```

Since the <ssh> element is present, SSH is enabled.

To delete a container with an <edit-config>:

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config">
        <services>
          <ssh nc:operation="delete"/>
        </services>
      </system>
    </config>
  </edit-config>
</rpc>
```

7.6. The "leaf" Statement

The "leaf" statement is used to define a leaf node in the schema
tree. It takes one argument, which is an identifier, followed by a
block of substatements that holds detailed leaf information.

A leaf node has a value, but no child nodes, in the data tree.
Conceptually, the value in the data tree is always in the canonical
form (see Section 9.1).

A leaf node exists in zero or one instance in the data tree.

The "leaf" statement is used to define a scalar variable of a
particular built-in or derived type.

7.6.1. The leaf's Default Value

The default value of a leaf is the value that the server uses if the
leaf does not exist in the data tree. The usage of the default value
depends on the leaf's closest ancestor node in the schema tree that
is not a non-presence container (see Section 7.5.1):

o If no such ancestor exists in the schema tree, the default value
   MUST be used.

o Otherwise, if this ancestor is a case node, the default value MUST
   be used if any node from the case exists in the data tree or the
   case node is the choice's default case, and if no nodes from any
   other case exist in the data tree.

o Otherwise, the default value MUST be used if the ancestor node
   exists in the data tree.

In these cases, the default value is said to be in use.

Note that if the leaf or any of its ancestors has a "when" condition
or "if-feature" expression that evaluates to "false", then the
default value is not in use.

When the default value is in use, the server MUST operationally
behave as if the leaf was present in the data tree with the default
value as its value.

If a leaf has a "default" statement, the leaf's default value is the
value of the "default" statement. Otherwise, if the leaf's type has
a default value and the leaf is not mandatory, then the leaf's
default value is the type's default value. In all other cases, the
leaf does not have a default value.

7.6.2. The leaf's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| config       | 7.21.1  | 0..1        |
| default      | 7.6.4   | 0..1        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| mandatory    | 7.6.5   | 0..1        |
| must         | 7.5.3   | 0..n        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| type         | 7.6.3   | 1           |
| units        | 7.3.3   | 0..1        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.6.3. The leaf's "type" Statement

The "type" statement, which MUST be present, takes as an argument the
name of an existing built-in or derived type. The optional
substatements specify restrictions on this type. See Section 7.4 for
details.

7.6.4. The leaf's "default" Statement

The "default" statement, which is optional, takes as an argument a
string that contains a default value for the leaf.

The value of the "default" statement MUST be valid according to the
type specified in the leaf's "type" statement.

The "default" statement MUST NOT be present on nodes where
"mandatory" is "true".

The definition of the default value MUST NOT be marked with an
"if-feature" statement. For example, the following is illegal:

```
leaf color {
  type enumeration {
    enum blue { if-feature blue; }
    ...
  }
  default blue; // illegal - enum value is conditional
}
```

7.6.5. The leaf's "mandatory" Statement

   The "mandatory" statement, which is optional, takes as an argument
   the string "true" or "false" and puts a constraint on valid data. If
   not specified, the default is "false".

   If "mandatory" is "true", the behavior of the constraint depends on
   the type of the leaf's closest ancestor node in the schema tree that
   is not a non-presence container (see Section 7.5.1):

   o If no such ancestor exists in the schema tree, the leaf MUST
      exist.

   o Otherwise, if this ancestor is a case node, the leaf MUST exist if
      any node from the case exists in the data tree.

   o Otherwise, the leaf MUST exist if the ancestor node exists in the
      data tree.

   This constraint is enforced according to the rules in Section 8.

7.6.6. XML Encoding Rules

   A leaf node is encoded as an XML element. The element's local name
   is the leaf's identifier, and its namespace is the module's XML
   namespace (see Section 7.1.3).

   The value of the leaf node is encoded to XML according to the type
   and is sent as character data in the element.

   See Section 7.6.8 for an example.

7.6.7. NETCONF <edit-config> Operations

   When a NETCONF server processes an <edit-config> request, the
   elements of procedure for the leaf node are as follows:

   o If the operation is "merge" or "replace", the node is created if
      it does not exist, and its value is set to the value found in the
      XML RPC data.

   o If the operation is "create", the node is created if it does not
      exist. If the node already exists, a "data-exists" error is
      returned.

   o If the operation is "delete", the node is deleted if it exists.
      If the node does not exist, a "data-missing" error is returned.

7.6.8. Usage Example

   Given the following "leaf" statement, placed in the previously
   defined "ssh" container (see Section 7.5.9):

      leaf port {
        type inet:port-number;
        default 22;
        description
          "The port to which the SSH server listens.";
      }

   A corresponding XML instance example:

      <port>2022</port>

   To set the value of a leaf with an <edit-config>:

      <rpc message-id="101"
          xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
          xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
        <edit-config>
          <target>
            <running/>
          </target>
          <config>
            <system xmlns="urn:example:config">
              <services>
                <ssh>
                  <port>2022</port>
                </ssh>
              </services>
            </system>
          </config>
        </edit-config>
      </rpc>

7.7. The "leaf-list" Statement

   Where the "leaf" statement is used to define a simple scalar variable
   of a particular type, the "leaf-list" statement is used to define an
   array of a particular type. The "leaf-list" statement takes one
   argument, which is an identifier, followed by a block of
   substatements that holds detailed leaf-list information.

   In configuration data, the values in a leaf-list MUST be unique.

The definitions of the default values MUST NOT be marked with an
"if-feature" statement.

Conceptually, the values in the data tree MUST be in the canonical
form (see Section 9.1).

7.7.1. Ordering

YANG supports two styles for ordering the entries within lists and
leaf-lists. In many lists, the order of list entries does not impact
the implementation of the list's configuration, and the server is
free to sort the list entries in any reasonable order. The
"description" string for the list may suggest an order to the server
implementor. YANG calls this style of list "system ordered"; such
lists are indicated with the statement "ordered-by system".

For example, a list of valid users would typically be sorted
alphabetically, since the order in which the users appeared in the
configuration would not impact the creation of those users' accounts.

In the other style of lists, the order of list entries matters for
the implementation of the list's configuration and the user is
responsible for ordering the entries, while the server maintains that
order. YANG calls this style of list "user ordered"; such lists are
indicated with the statement "ordered-by user".

For example, the order in which packet filter entries are applied to
incoming traffic may affect how that traffic is filtered. The user
would need to decide if the filter entry that discards all TCP
traffic should be applied before or after the filter entry that
allows all traffic from trusted interfaces. The choice of order
would be crucial.

YANG provides a rich set of facilities within NETCONF's <edit-config>
operation that allows the order of list entries in user-ordered lists
to be controlled. List entries may be inserted or rearranged,
positioned as the first or last entry in the list, or positioned
before or after another specific entry.

The "ordered-by" statement is covered in Section 7.7.7.

7.7.2. The leaf-list's Default Values

   The default values of a leaf-list are the values that the server uses
   if the leaf-list does not exist in the data tree. The usage of the
   default values depends on the leaf-list's closest ancestor node in
   the schema tree that is not a non-presence container (see
   Section 7.5.1):

   o If no such ancestor exists in the schema tree, the default values
      MUST be used.

   o Otherwise, if this ancestor is a case node, the default values
      MUST be used if any node from the case exists in the data tree or
      the case node is the choice's default case, and if no nodes from
      any other case exist in the data tree.

   o Otherwise, the default values MUST be used if the ancestor node
      exists in the data tree.

   In these cases, the default values are said to be in use.

   Note that if the leaf-list or any of its ancestors has a "when"
   condition or "if-feature" expression that evaluates to "false", then
   the default values are not in use.

   When the default values are in use, the server MUST operationally
   behave as if the leaf-list was present in the data tree with the
   default values as its values.

   If a leaf-list has one or more "default" statements, the leaf-list's
   default values are the values of the "default" statements, and if the
   leaf-list is user ordered, the default values are used in the order
   of the "default" statements. Otherwise, if the leaf-list's type has
   a default value and the leaf-list does not have a "min-elements"
   statement with a value greater than or equal to one, then the
   leaf-list's default value is one instance of the type's default
   value. In all other cases, the leaf-list does not have any default
   values.

7.7.3. The leaf-list's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| config       | 7.21.1  | 0..1        |
| default      | 7.7.4   | 0..n        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| max-elements | 7.7.6   | 0..1        |
| min-elements | 7.7.5   | 0..1        |
| must         | 7.5.3   | 0..n        |
| ordered-by   | 7.7.7   | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| type         | 7.4     | 1           |
| units        | 7.3.3   | 0..1        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.7.4. The leaf-list's "default" Statement

The "default" statement, which is optional, takes as an argument a
string that contains a default value for the leaf-list.

The value of the "default" statement MUST be valid according to the
type specified in the leaf-list's "type" statement.

The "default" statement MUST NOT be present on nodes where
"min-elements" has a value greater than or equal to one.

7.7.5. The "min-elements" Statement

The "min-elements" statement, which is optional, takes as an argument
a non-negative integer that puts a constraint on valid list entries.
A valid leaf-list or list MUST have at least min-elements entries.

If no "min-elements" statement is present, it defaults to zero.

The behavior of the constraint depends on the type of the leaf-list's
or list's closest ancestor node in the schema tree that is not a
non-presence container (see Section 7.5.1):

o If no such ancestor exists in the schema tree, the constraint is
   enforced.

o Otherwise, if this ancestor is a case node, the constraint is
   enforced if any other node from the case exists.

   o Otherwise, it is enforced if the ancestor node exists.

   The constraint is further enforced according to the rules in
   Section 8.

7.7.6. The "max-elements" Statement

   The "max-elements" statement, which is optional, takes as an argument
   a positive integer or the string "unbounded", which puts a constraint
   on valid list entries. A valid leaf-list or list always has at most
   max-elements entries.

   If no "max-elements" statement is present, it defaults to
   "unbounded".

   The "max-elements" constraint is enforced according to the rules in
   Section 8.

7.7.7. The "ordered-by" Statement

   The "ordered-by" statement defines whether the order of entries
   within a list are determined by the user or the system. The argument
   is one of the strings "system" or "user". If not present, ordering
   defaults to "system".

   This statement is ignored if the list represents state data, RPC
   output parameters, or notification content.

   See Section 7.7.1 for additional information.

7.7.7.1. ordered-by system

   The entries in the list are ordered according to an order determined
   by the system. The "description" string for the list may suggest an
   order to the server implementor. If not, an implementation is free
   to order the entries in any order. An implementation SHOULD use the
   same order for the same data, regardless of how the data were
   created. Using a deterministic order will make comparisons possible
   using simple tools like "diff".

   This is the default order.

7.7.7.2. ordered-by user

   The entries in the list are ordered according to an order defined by
   the user. In NETCONF, this order is controlled by using special XML
   attributes in the <edit-config> request. See Section 7.7.9 for
   details.

7.7.8. XML Encoding Rules

A leaf-list node is encoded as a series of XML elements. Each
element's local name is the leaf-list's identifier, and its namespace
is the module's XML namespace (see Section 7.1.3).

The value of each leaf-list entry is encoded to XML according to the
type and is sent as character data in the element.

The XML elements representing leaf-list entries MUST appear in the
order specified by the user if the leaf-list is "ordered-by user";
otherwise, the order is implementation dependent. The XML elements
representing leaf-list entries MAY be interleaved with elements for
siblings of the leaf-list, unless the leaf-list defines RPC or action
input or output parameters.

See Section 7.7.10 for an example.

7.7.9. NETCONF <edit-config> Operations

Leaf-list entries can be created and deleted, but not modified,
through <edit-config>, by using the "operation" attribute in the
leaf-list entry's XML element.

In an "ordered-by user" leaf-list, the attributes "insert" and
"value" in the YANG XML namespace (Section 5.3.1) can be used to
control where in the leaf-list the entry is inserted. These can be
used during "create" operations to insert a new leaf-list entry, or
during "merge" or "replace" operations to insert a new leaf-list
entry or move an existing one.

The "insert" attribute can take the values "first", "last", "before",
and "after". If the value is "before" or "after", the "value"
attribute MUST also be used to specify an existing entry in the
leaf-list.

If no "insert" attribute is present in the "create" operation, it
defaults to "last".

If several entries in an "ordered-by user" leaf-list are modified in
the same <edit-config> request, the entries are modified one at a
time, in the order of the XML elements in the request.

In a <copy-config> or in an <edit-config> with a "replace" operation
that covers the entire leaf-list, the leaf-list order is the same as
the order of the XML elements in the request.

When a NETCONF server processes an <edit-config> request, the
elements of procedure for a leaf-list node are as follows:

o If the operation is "merge" or "replace", the leaf-list entry is
   created if it does not exist.

o If the operation is "create", the leaf-list entry is created if it
   does not exist. If the leaf-list entry already exists, a
   "data-exists" error is returned.

o If the operation is "delete", the entry is deleted from the
   leaf-list if it exists. If the leaf-list entry does not exist, a
   "data-missing" error is returned.

7.7.10. Usage Example

```
leaf-list allow-user {
  type string;
  description
    "A list of user name patterns to allow.";
}
```

A corresponding XML instance example:

```
<allow-user>alice</allow-user>
<allow-user>bob</allow-user>
```

To create a new element in this list, using the default <edit-config>
operation "merge":

```
<rpc message-id="101"
    xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
    xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config">
        <services>
          <ssh>
            <allow-user>eric</allow-user>
          </ssh>
        </services>
      </system>
    </config>
  </edit-config>
</rpc>
```

Given the following ordered-by user leaf-list:

```
leaf-list cipher {
  type string;
  ordered-by user;
  description
    "A list of ciphers.";
}
```

The following would be used to insert a new cipher "blowfish-cbc"
after "3des-cbc":

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:yang="urn:ietf:params:xml:ns:yang:1">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config">
        <services>
          <ssh>
            <cipher nc:operation="create"
                    yang:insert="after"
                    yang:value="3des-cbc">blowfish-cbc</cipher>
          </ssh>
        </services>
      </system>
    </config>
  </edit-config>
</rpc>
```

7.8. The "list" Statement

The "list" statement is used to define an interior data node in the
schema tree. A list node may exist in multiple instances in the data
tree. Each such instance is known as a list entry. The "list"
statement takes one argument, which is an identifier, followed by a
block of substatements that holds detailed list information.

A list entry is uniquely identified by the values of the list's keys,
if defined.

7.8.1. The list's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| action       | 7.15    | 0..n        |
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| choice       | 7.9     | 0..n        |
| config       | 7.21.1  | 0..1        |
| container    | 7.5     | 0..n        |
| description  | 7.21.3  | 0..1        |
| grouping     | 7.12    | 0..n        |
| if-feature   | 7.20.2  | 0..n        |
| key          | 7.8.2   | 0..1        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| max-elements | 7.7.6   | 0..1        |
| min-elements | 7.7.5   | 0..1        |
| must         | 7.5.3   | 0..n        |
| notification | 7.16    | 0..n        |
| ordered-by   | 7.7.7   | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| typedef      | 7.3     | 0..n        |
| unique       | 7.8.3   | 0..n        |
| uses         | 7.13    | 0..n        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.8.2. The list's "key" Statement

The "key" statement, which MUST be present if the list represents
configuration and MAY be present otherwise, takes as an argument a
string that specifies a space-separated list of one or more leaf
identifiers of this list. A leaf identifier MUST NOT appear more
than once in the key. Each such leaf identifier MUST refer to a
child leaf of the list. The leafs can be defined directly in
substatements to the list or in groupings used in the list.

The combined values of all the leafs specified in the key are used to
uniquely identify a list entry. All key leafs MUST be given values
when a list entry is created. Thus, any default values in the key
leafs or their types are ignored. Any "mandatory" statements in the
key leafs are ignored.

A leaf that is part of the key can be of any built-in or
derived type.

All key leafs in a list MUST have the same value for their "config"
as the list itself.

The key string syntax is formally defined by the rule "key-arg" in
Section 14.

7.8.3. The list's "unique" Statement

The "unique" statement is used to put constraints on valid list
entries. It takes as an argument a string that contains a space-
separated list of schema node identifiers, which MUST be given in the
descendant form (see the rule "descendant-schema-nodeid" in
Section 14). Each such schema node identifier MUST refer to a leaf.

If one of the referenced leafs represents configuration data, then
all of the referenced leafs MUST represent configuration data.

The "unique" constraint specifies that the combined values of all the
leaf instances specified in the argument string, including leafs with
default values, MUST be unique within all list entry instances in
which all referenced leafs exist or have default values. The
constraint is enforced according to the rules in Section 8.

The unique string syntax is formally defined by the rule "unique-arg"
in Section 14.

7.8.3.1. Usage Example

With the following list:

```
list server {
  key "name";
  unique "ip port";
  leaf name {
    type string;
  }
  leaf ip {
    type inet:ip-address;
  }
  leaf port {
    type inet:port-number;
  }
}
```

   the following configuration is not valid:

      <server>
        <name>smtp</name>
        <ip>192.0.2.1</ip>
        <port>25</port>
      </server>

      <server>
        <name>http</name>
        <ip>192.0.2.1</ip>
        <port>25</port>
      </server>

   The following configuration is valid, since the "http" and "ftp" list
   entries do not have a value for all referenced leafs and are thus not
   taken into account when the "unique" constraint is enforced:

      <server>
        <name>smtp</name>
        <ip>192.0.2.1</ip>
        <port>25</port>
      </server>

      <server>
        <name>http</name>
        <ip>192.0.2.1</ip>
      </server>

      <server>
        <name>ftp</name>
        <ip>192.0.2.1</ip>
      </server>

7.8.4. The list's Child Node Statements

   Within a list, the "container", "leaf", "list", "leaf-list", "uses",
   "choice", "anydata", and "anyxml" statements can be used to define
   child nodes to the list.

7.8.5. XML Encoding Rules

A list is encoded as a series of XML elements, one for each entry in
the list. Each element's local name is the list's identifier, and
its namespace is the module's XML namespace (see Section 7.1.3).
There is no XML element surrounding the list as a whole.

The list's key nodes are encoded as subelements to the list's
identifier element, in the same order as they are defined within the
"key" statement.

The rest of the list's child nodes are encoded as subelements to the
list element, after the keys. If the list defines RPC or action
input or output parameters, the subelements are encoded in the same
order as they are defined within the "list" statement. Otherwise,
the subelements are encoded in any order.

Any whitespace between the subelements to the list entry is
insignificant, i.e., an implementation MAY insert whitespace
characters between subelements.

The XML elements representing list entries MUST appear in the order
specified by the user if the list is "ordered-by user"; otherwise,
the order is implementation dependent. The XML elements representing
list entries MAY be interleaved with elements for siblings of the
list, unless the list defines RPC or action input or output
parameters.

7.8.6. NETCONF <edit-config> Operations

List entries can be created, deleted, replaced, and modified through
<edit-config> by using the "operation" attribute in the list's XML
element. In each case, the values of all keys are used to uniquely
identify a list entry. If all keys are not specified for a list
entry, a "missing-element" error is returned.

In an "ordered-by user" list, the attributes "insert" and "key" in
the YANG XML namespace (Section 5.3.1) can be used to control where
in the list the entry is inserted. These can be used during "create"
operations to insert a new list entry, or during "merge" or "replace"
operations to insert a new list entry or move an existing one.

The "insert" attribute can take the values "first", "last", "before",
and "after". If the value is "before" or "after", the "key"
attribute MUST also be used, to specify an existing element in the
list. The value of the "key" attribute is the key predicates of the
full instance identifier (see Section 9.13) for the list entry.

If no "insert" attribute is present in the "create" operation, it
defaults to "last".

If several entries in an "ordered-by user" list are modified in the
same <edit-config> request, the entries are modified one at a time,
in the order of the XML elements in the request.

In a <copy-config> or in an <edit-config> with a "replace" operation
that covers the entire list, the list entry order is the same as the
order of the XML elements in the request.

When a NETCONF server processes an <edit-config> request, the
elements of procedure for a list node are as follows:

o If the operation is "merge" or "replace", the list entry is
   created if it does not exist. If the list entry already exists
   and the "insert" and "key" attributes are present, the list entry
   is moved according to the values of the "insert" and "key"
   attributes. If the list entry exists and the "insert" and "key"
   attributes are not present, the list entry is not moved.

o If the operation is "create", the list entry is created if it does
   not exist. If the list entry already exists, a "data-exists"
   error is returned.

o If the operation is "delete", the entry is deleted from the list
   if it exists. If the list entry does not exist, a "data-missing"
   error is returned.

7.8.7. Usage Example

   Given the following list:

     list user {
       key "name";
       config true;
       description
         "This is a list of users in the system.";

       leaf name {
         type string;
       }
       leaf type {
         type string;
       }
       leaf full-name {
         type string;
       }
     }

   A corresponding XML instance example:

     <user>
       <name>fred</name>
       <type>admin</type>
       <full-name>Fred Flintstone</full-name>
     </user>

To create a new user "barney":

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config">
        <user nc:operation="create">
          <name>barney</name>
          <type>admin</type>
          <full-name>Barney Rubble</full-name>
        </user>
      </system>
    </config>
  </edit-config>
</rpc>
```

To change the type of "fred" to "superuser":

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config">
        <user>
          <name>fred</name>
          <type>superuser</type>
        </user>
      </system>
    </config>
  </edit-config>
</rpc>
```

   Given the following ordered-by user list:

```
list user {
  description
    "This is a list of users in the system.";
  ordered-by user;
  config true;

  key "first-name surname";

  leaf first-name {
    type string;
  }
  leaf surname {
    type string;
  }
  leaf type {
    type string;
  }
}
```

   The following would be used to insert a new user "barney rubble"
   after the user "fred flintstone":

```
<rpc message-id="101"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:yang="urn:ietf:params:xml:ns:yang:1">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config"
              xmlns:ex="urn:example:config">
        <user nc:operation="create"
              yang:insert="after"
              yang:key="[ex:first-name='fred']
                        [ex:surname='flintstone']">
          <first-name>barney</first-name>
          <surname>rubble</surname>
          <type>admin</type>
        </user>
      </system>
    </config>
  </edit-config>
</rpc>
```

The following would be used to move the user "barney rubble" before
the user "fred flintstone":

```
<rpc message-id="102"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
     xmlns:yang="urn:ietf:params:xml:ns:yang:1">
  <edit-config>
    <target>
      <running/>
    </target>
    <config>
      <system xmlns="urn:example:config"
          xmlns:ex="urn:example:config">
        <user nc:operation="merge"
              yang:insert="before"
              yang:key="[ex:name='fred']
                        [ex:surname='flintstone']">
          <first-name>barney</first-name>
          <surname>rubble</surname>
        </user>
      </system>
    </config>
  </edit-config>
</rpc>
```

7.9. The "choice" Statement

The "choice" statement defines a set of alternatives, only one of
which may be present in any one data tree. The argument is an
identifier, followed by a block of substatements that holds detailed
choice information. The identifier is used to identify the choice
node in the schema tree. A choice node does not exist in the data
tree.

A choice consists of a number of branches, each defined with a "case"
substatement. Each branch contains a number of child nodes. The
nodes from at most one of the choice's branches exist at the same
time.

Since only one of the choice's cases can be valid at any time in the
data tree, the creation of a node from one case implicitly deletes
all nodes from all other cases. If a request creates a node from a
case, the server will delete any existing nodes that are defined in
other cases inside the choice.

7.9.1. The choice's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| case         | 7.9.2   | 0..n        |
| choice       | 7.9     | 0..n        |
| config       | 7.21.1  | 0..1        |
| container    | 7.5     | 0..n        |
| default      | 7.9.3   | 0..1        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| mandatory    | 7.9.4   | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.9.2. The choice's "case" Statement

The "case" statement is used to define branches of the choice. It
takes as an argument an identifier, followed by a block of
substatements that holds detailed case information.

The identifier is used to identify the case node in the schema tree.
A case node does not exist in the data tree.

Within a "case" statement, the "anydata", "anyxml", "choice",
"container", "leaf", "list", "leaf-list", and "uses" statements can
be used to define child nodes to the case node. The identifiers of
all these child nodes MUST be unique within all cases in a choice.
For example, the following is illegal:

```
  choice interface-type {     // This example is illegal YANG
    case a {
      leaf ethernet { ... }
    }
    case b {
      container ethernet { ...}
    }
  }
```

As a shorthand, the "case" statement can be omitted if the branch
contains a single "anydata", "anyxml", "choice", "container", "leaf",
"list", or "leaf-list" statement. In this case, the case node still
exists in the schema tree, and its identifier is the same as the
identifier of the child node. Schema node identifiers (Section 6.5)
MUST always explicitly include case node identifiers. The following
example:

```
choice interface-type {
  container ethernet { ... }
}
```

is equivalent to:

```
choice interface-type {
  case ethernet {
    container ethernet { ... }
  }
}
```

The case identifier MUST be unique within a choice.

7.9.2.1. The case's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| choice       | 7.9     | 0..n        |
| container    | 7.5     | 0..n        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| uses         | 7.13    | 0..n        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.9.3. The choice's "default" Statement

The "default" statement indicates if a case should be considered as
the default if no child nodes from any of the choice's cases exist.
The argument is the identifier of the default "case" statement. If
the "default" statement is missing, there is no default case.

The "default" statement MUST NOT be present on choices where
"mandatory" is "true".

The default case is only important when considering the "default"
statements of nodes under the cases (i.e., default values of leafs
and leaf-lists, and default cases of nested choices). The default
values and nested default cases under the default case are used if
none of the nodes under any of the cases are present.

There MUST NOT be any mandatory nodes (Section 3) directly under the
default case.

Default values for child nodes under a case are only used if one of
the nodes under that case is present or if that case is the default
case. If none of the nodes under a case are present and the case is
not the default case, the default values of the cases' child nodes
are ignored.

   In this example, the choice defaults to "interval", and the default
   value will be used if none of "daily", "time-of-day", or "manual" are
   present. If "daily" is present, the default value for "time-of-day"
   will be used.

```
    container transfer {
      choice how {
        default interval;
        case interval {
          leaf interval {
            type uint16;
            units minutes;
            default 30;
          }
        }
        case daily {
          leaf daily {
            type empty;
          }
          leaf time-of-day {
            type string;
            units 24-hour-clock;
            default "01.00";
          }
        }
        case manual {
          leaf manual {
            type empty;
          }
        }
      }
    }
```

7.9.4. The choice's "mandatory" Statement

The "mandatory" statement, which is optional, takes as an argument
the string "true" or "false" and puts a constraint on valid data. If
"mandatory" is "true", at least one node from exactly one of the
choice's case branches MUST exist.

If not specified, the default is "false".

The behavior of the constraint depends on the type of the choice's
closest ancestor node in the schema tree that is not a non-presence
container (see Section 7.5.1):

o If no such ancestor exists in the schema tree, the constraint is
   enforced.

o Otherwise, if this ancestor is a case node, the constraint is
   enforced if any other node from the case exists.

o Otherwise, it is enforced if the ancestor node exists.

The constraint is further enforced according to the rules in
Section 8.

7.9.5. XML Encoding Rules

The choice and case nodes are not visible in XML.

The child nodes of the selected "case" statement MUST be encoded in
the same order as they are defined in the "case" statement if they
are part of an RPC or action input or output parameter definition.
Otherwise, the subelements are encoded in any order.

7.9.6. Usage Example

   Given the following choice:

       container protocol {
         choice name {
           case a {
             leaf udp {
               type empty;
             }
           }
           case b {
             leaf tcp {
               type empty;
             }
           }
         }
       }

   A corresponding XML instance example:

       <protocol>
         <tcp/>
       </protocol>

   To change the protocol from TCP to UDP:

       <rpc message-id="101"
            xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
            xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
         <edit-config>
           <target>
             <running/>
           </target>
           <config>
             <system xmlns="urn:example:config">
               <protocol>
                 <udp nc:operation="create"/>
               </protocol>
             </system>
           </config>
         </edit-config>
       </rpc>

7.10. The "anydata" Statement

   The "anydata" statement defines an interior node in the schema tree.
   It takes one argument, which is an identifier, followed by a block of
   substatements that holds detailed anydata information.

   The "anydata" statement is used to represent an unknown set of nodes
   that can be modeled with YANG, except anyxml, but for which the data
   model is not known at module design time. It is possible, though not
   required, for the data model for anydata content to become known
   through protocol signaling or other means that are outside the scope
   of this document.

   An example of where anydata can be useful is a list of received
   notifications where the specific notifications are not known at
   design time.

   An anydata node cannot be augmented (see Section 7.17).

   An anydata node exists in zero or one instance in the data tree.

   An implementation may or may not know the data model used to model a
   specific instance of an anydata node.

   Since the use of anydata limits the manipulation of the content, the
   "anydata" statement SHOULD NOT be used to define configuration data.

7.10.1. The anydata's Substatements

```
         +--------------+---------+-------------+
         | substatement | section | cardinality |
         +--------------+---------+-------------+
         | config       | 7.21.1  | 0..1        |
         | description  | 7.21.3  | 0..1        |
         | if-feature   | 7.20.2  | 0..n        |
         | mandatory    | 7.6.5   | 0..1        |
         | must         | 7.5.3   | 0..n        |
         | reference    | 7.21.4  | 0..1        |
         | status       | 7.21.2  | 0..1        |
         | when         | 7.21.5  | 0..1        |
         +--------------+---------+-------------+
```

7.10.2. XML Encoding Rules

   An anydata node is encoded as an XML element. The element's local
   name is the anydata's identifier, and its namespace is the module's
   XML namespace (see Section 7.1.3). The value of the anydata node is
   a set of nodes, which are encoded as XML subelements to the anydata
   element.

7.10.3. NETCONF <edit-config> Operations

   An anydata node is treated as an opaque chunk of data. This data can
   be modified in its entirety only.

   Any "operation" attributes present on subelements of an anydata node
   are ignored by the NETCONF server.

   When a NETCONF server processes an <edit-config> request, the
   elements of procedure for the anydata node are as follows:

   o If the operation is "merge" or "replace", the node is created if
      it does not exist, and its value is set to the subelements of the
      anydata node found in the XML RPC data.

   o If the operation is "create", the node is created if it does not
      exist, and its value is set to the subelements of the anydata node
      found in the XML RPC data. If the node already exists, a
      "data-exists" error is returned.

   o If the operation is "delete", the node is deleted if it exists.
      If the node does not exist, a "data-missing" error is returned.

7.10.4. Usage Example

   Given the following "anydata" statement:

     list logged-notification {
       key time;
       leaf time {
         type yang:date-and-time;
       }
       anydata data;
     }

The following is a valid encoding of such a list instance:

```
<logged-notification>
  <time>2014-07-29T13:43:12Z</time>
  <data>
    <notification
      xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
      <eventTime>2014-07-29T13:43:01Z</eventTime>
      <event xmlns="urn:example:event">
        <event-class>fault</event-class>
        <reporting-entity>
          <card>Ethernet0</card>
        </reporting-entity>
        <severity>major</severity>
      </event>
    </notification>
  </data>
</logged-notification>
```

7.11. The "anyxml" Statement

The "anyxml" statement defines an interior node in the schema tree.
It takes one argument, which is an identifier, followed by a block of
substatements that holds detailed anyxml information.

The "anyxml" statement is used to represent an unknown chunk of XML.
No restrictions are placed on the XML. This can be useful, for
example, in RPC replies. An example is the <filter> parameter in the
<get-config> operation in NETCONF.

An anyxml node cannot be augmented (see Section 7.17).

An anyxml node exists in zero or one instance in the data tree.

Since the use of anyxml limits the manipulation of the content, the
"anyxml" statement SHOULD NOT be used to define configuration data.

It should be noted that in YANG version 1, "anyxml" was the only
statement that could model an unknown hierarchy of data. In many
cases, this unknown hierarchy of data is actually modeled in YANG,
but the specific YANG data model is not known at design time. In
these situations, it is RECOMMENDED to use "anydata" (Section 7.10)
instead of "anyxml".

7.11.1. The anyxml's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| config       | 7.21.1  | 0..1        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| mandatory    | 7.6.5   | 0..1        |
| must         | 7.5.3   | 0..n        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.11.2. XML Encoding Rules

An anyxml node is encoded as an XML element. The element's local
name is the anyxml's identifier, and its namespace is the module's
XML namespace (see Section 7.1.3). The value of the anyxml node is
encoded as XML content of this element.

Note that any XML prefixes used in the encoding are local to each
instance encoding. This means that the same XML may be encoded
differently by different implementations.

7.11.3. NETCONF <edit-config> Operations

An anyxml node is treated as an opaque chunk of data. This data can
be modified in its entirety only.

Any "operation" attributes present on subelements of an anyxml node
are ignored by the NETCONF server.

When a NETCONF server processes an <edit-config> request, the
elements of procedure for the anyxml node are as follows:

o If the operation is "merge" or "replace", the node is created if
   it does not exist, and its value is set to the XML content of the
   anyxml node found in the XML RPC data.

o If the operation is "create", the node is created if it does not
   exist, and its value is set to the XML content of the anyxml node
   found in the XML RPC data. If the node already exists, a
   "data-exists" error is returned.

o If the operation is "delete", the node is deleted if it exists.
   If the node does not exist, a "data-missing" error is returned.

7.11.4. Usage Example

Given the following "anyxml" statement:

    anyxml html-info;

The following are two valid encodings of the same anyxml value:

    <html-info>
      <p xmlns="http://www.w3.org/1999/xhtml">
        This is very cool.
      </p>
    </html-info>

    <html-info>
      <x:p xmlns:x="http://www.w3.org/1999/xhtml">
        This is <x:em>very</x:em> cool.
      </x:p>
    </html-info>

7.12. The "grouping" Statement

The "grouping" statement is used to define a reusable block of nodes,
which may be used locally in the module or submodule, and by other
modules that import from it, according to the rules in Section 5.5.
It takes one argument, which is an identifier, followed by a block of
substatements that holds detailed grouping information.

The "grouping" statement is not a data definition statement and, as
such, does not define any nodes in the schema tree.

A grouping is like a "structure" or a "record" in conventional
programming languages.

Once a grouping is defined, it can be referenced in a "uses"
statement (see Section 7.13). A grouping MUST NOT reference itself,
neither directly nor indirectly through a chain of other groupings.

If the grouping is defined at the top level of a YANG module or
submodule, the grouping's identifier MUST be unique within the
module.

A grouping is more than just a mechanism for textual substitution;
it also defines a collection of nodes. Identifiers appearing inside
the grouping are resolved relative to the scope in which the grouping
is defined, not where it is used. Prefix mappings, type names,
grouping names, and extension usage are evaluated in the hierarchy

where the "grouping" statement appears. For extensions, this means
that extensions defined as direct children to a "grouping" statement
are applied to the grouping itself.

Note that if a grouping defines an action or a notification node in
its hierarchy, then it cannot be used in all contexts. For example,
it cannot be used in an rpc definition. See Sections 7.15 and 7.16.

7.12.1. The grouping's Substatements

substatement	section	cardinality
action	7.15	0..n
anydata	7.10	0..n
anyxml	7.11	0..n
choice	7.9	0..n
container	7.5	0..n
description	7.21.3	0..1
grouping	7.12	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
notification	7.16	0..n
reference	7.21.4	0..1
status	7.21.2	0..1
typedef	7.3	0..n
uses	7.13	0..n

7.12.2. Usage Example

```
import ietf-inet-types {
  prefix "inet";
}

grouping endpoint {
  description "A reusable endpoint group.";
  leaf ip {
    type inet:ip-address;
  }
  leaf port {
    type inet:port-number;
  }
}
```

7.13. The "uses" Statement

   The "uses" statement is used to reference a "grouping" definition.
   It takes one argument, which is the name of the grouping.

   The effect of a "uses" reference to a grouping is that the nodes
   defined by the grouping are copied into the current schema tree and
   are then updated according to the "refine" and "augment" statements.

   The identifiers defined in the grouping are not bound to a namespace
   until the contents of the grouping are added to the schema tree via a
   "uses" statement that does not appear inside a "grouping" statement,
   at which point they are bound to the namespace of the current module.

7.13.1. The uses's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| augment      | 7.17    | 0..n        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| reference    | 7.21.4  | 0..1        |
| refine       | 7.13.2  | 0..n        |
| status       | 7.21.2  | 0..1        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.13.2. The "refine" Statement

   Some of the properties of each node in the grouping can be refined
   with the "refine" statement. The argument is a string that
   identifies a node in the grouping. This node is called the refine's
   target node. If a node in the grouping is not present as a target
   node of a "refine" statement, it is not refined and thus is used
   exactly as it was defined in the grouping.

   The argument string is a descendant schema node identifier (see
   Section 6.5).

   The following refinements can be done:

   o A leaf or choice node may get a default value, or a new default
      value if it already had one.

   o A leaf-list node may get a set of default values, or a new set of
      default values if it already had defaults; i.e., the set of
      refined default values replaces the defaults already given.

o Any node may get a specialized "description" string.

o Any node may get a specialized "reference" string.

o Any node may get a different "config" statement.

o A leaf, anydata, anyxml, or choice node may get a different
   "mandatory" statement.

o A container node may get a "presence" statement.

o A leaf, leaf-list, list, container, anydata, or anyxml node may
   get additional "must" expressions.

o A leaf-list or list node may get a different "min-elements" or
   "max-elements" statement.

o A leaf, leaf-list, list, container, choice, case, anydata, or
   anyxml node may get additional "if-feature" expressions.

o Any node can get refined extensions, if the extension allows
   refinement. See Section 7.19 for details.

7.13.3. XML Encoding Rules

Each node in the grouping is encoded as if it was defined inline,
even if it is imported from another module with another XML
namespace.

7.13.4. Usage Example

To use the "endpoint" grouping defined in Section 7.12.2 in a
definition of an HTTP server in some other module, we can do:

```
import example-system {
  prefix "sys";
}

container http-server {
  leaf name {
    type string;
  }
  uses sys:endpoint;
}
```

A corresponding XML instance example:

```
<http-server>
  <name>extern-web</name>
  <ip>192.0.2.1</ip>
  <port>80</port>
</http-server>
```

If port 80 should be the default for the HTTP server, a default can
be added:

```
container http-server {
  leaf name {
    type string;
  }
  uses sys:endpoint {
    refine port {
      default 80;
    }
  }
}
```

If we want to define a list of servers and each server has "ip" and
"port" as keys, we can do:

```
list server {
  key "ip port";
  leaf name {
    type string;
  }
  uses sys:endpoint;
}
```

The following is an error:

```
container http-server {
  uses sys:endpoint;
  leaf ip {          // illegal - same identifier "ip" used twice
    type string;
  }
}
```

7.14. The "rpc" Statement

The "rpc" statement is used to define an RPC operation. It takes one
argument, which is an identifier, followed by a block of
substatements that holds detailed rpc information. This argument is
the name of the RPC.

The "rpc" statement defines an rpc node in the schema tree. Under
the rpc node, a schema node with the name "input" and a schema node
with the name "output" are also defined. The nodes "input" and
"output" are defined in the module's namespace.

7.14.1. The rpc's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| description  | 7.21.3  | 0..1        |
| grouping     | 7.12    | 0..n        |
| if-feature   | 7.20.2  | 0..n        |
| input        | 7.14.2  | 0..1        |
| output       | 7.14.3  | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| typedef      | 7.3     | 0..n        |
+--------------+---------+-------------+
```

7.14.2. The "input" Statement

The "input" statement, which is optional, is used to define input
parameters to the operation. It does not take an argument. The
substatements to "input" define nodes under the operation's input
node.

If a leaf in the input tree has a "mandatory" statement with the
value "true", the leaf MUST be present in an RPC invocation.

If a leaf in the input tree has a default value, the server MUST use
this value in the same cases as those described in Section 7.6.1. In
these cases, the server MUST operationally behave as if the leaf was
present in the RPC invocation with the default value as its value.

If a leaf-list in the input tree has one or more default values, the
server MUST use these values in the same cases as those described in
Section 7.7.2. In these cases, the server MUST operationally behave
as if the leaf-list was present in the RPC invocation with the
default values as its values.

Since the input tree is not part of any datastore, all "config"
statements for nodes in the input tree are ignored.

If any node has a "when" statement that would evaluate to "false",
then this node MUST NOT be present in the input tree.

7.14.2.1. The input's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| choice       | 7.9     | 0..n        |
| container    | 7.5     | 0..n        |
| grouping     | 7.12    | 0..n        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| must         | 7.5.3   | 0..n        |
| typedef      | 7.3     | 0..n        |
| uses         | 7.13    | 0..n        |
+--------------+---------+-------------+
```

7.14.3. The "output" Statement

   The "output" statement, which is optional, is used to define output
   parameters to the RPC operation. It does not take an argument. The
   substatements to "output" define nodes under the operation's output
   node.

   If a leaf in the output tree has a "mandatory" statement with the
   value "true", the leaf MUST be present in an RPC reply.

   If a leaf in the output tree has a default value, the client MUST use
   this value in the same cases as those described in Section 7.6.1. In
   these cases, the client MUST operationally behave as if the leaf was
   present in the RPC reply with the default value as its value.

   If a leaf-list in the output tree has one or more default values, the
   client MUST use these values in the same cases as those described in
   Section 7.7.2. In these cases, the client MUST operationally behave
   as if the leaf-list was present in the RPC reply with the default
   values as its values.

   Since the output tree is not part of any datastore, all "config"
   statements for nodes in the output tree are ignored.

   If any node has a "when" statement that would evaluate to "false",
   then this node MUST NOT be present in the output tree.

7.14.3.1. The output's Substatements

```
+-------------+---------+-------------+
| substatement | section | cardinality |
+-------------+---------+-------------+
| anydata     | 7.10    | 0..n        |
| anyxml      | 7.11    | 0..n        |
| choice      | 7.9     | 0..n        |
| container   | 7.5     | 0..n        |
| grouping    | 7.12    | 0..n        |
| leaf        | 7.6     | 0..n        |
| leaf-list   | 7.7     | 0..n        |
| list        | 7.8     | 0..n        |
| must        | 7.5.3   | 0..n        |
| typedef     | 7.3     | 0..n        |
| uses        | 7.13    | 0..n        |
+-------------+---------+-------------+
```

7.14.4. NETCONF XML Encoding Rules

An rpc node is encoded as a child XML element to the <rpc> element,
as designated by the substitution group "rpcOperation" in [RFC6241].
The element's local name is the rpc's identifier, and its namespace
is the module's XML namespace (see Section 7.1.3).

Input parameters are encoded as child XML elements to the rpc node's
XML element, in the same order as they are defined within the "input"
statement.

If the RPC operation invocation succeeded and no output parameters
are returned, the <rpc-reply> contains a single <ok/> element defined
in [RFC6241]. If output parameters are returned, they are encoded as
child elements to the <rpc-reply> element defined in [RFC6241], in
the same order as they are defined within the "output" statement.

7.14.5. Usage Example

   The following example defines an RPC operation:

      module example-rock {
        yang-version 1.1;
        namespace "urn:example:rock";
        prefix "rock";

        rpc rock-the-house {
          input {
            leaf zip-code {
              type string;
            }
          }
        }
      }

   A corresponding XML instance example of the complete rpc and
   rpc-reply:

      <rpc message-id="101"
           xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
        <rock-the-house xmlns="urn:example:rock">
          <zip-code>27606-0100</zip-code>
        </rock-the-house>
      </rpc>

      <rpc-reply message-id="101"
                 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
        <ok/>
      </rpc-reply>

7.15. The "action" Statement

   The "action" statement is used to define an operation connected to a
   specific container or list data node. It takes one argument, which
   is an identifier, followed by a block of substatements that holds
   detailed action information. The argument is the name of the action.

   The "action" statement defines an action node in the schema tree.
   Under the action node, a schema node with the name "input" and a
   schema node with the name "output" are also defined. The nodes
   "input" and "output" are defined in the module's namespace.

   An action MUST NOT be defined within an rpc, another action, or a
   notification, i.e., an action node MUST NOT have an rpc, action, or a
   notification node as one of its ancestors in the schema tree. For
   example, this means that it is an error if a grouping that contains
   an action somewhere in its node hierarchy is used in a notification
   definition.

   An action MUST NOT have any ancestor node that is a list node without
   a "key" statement.

   Since an action cannot be defined at the top level of a module or in
   a "case" statement, it is an error if a grouping that contains an
   action at the top of its node hierarchy is used at the top level of a
   module or in a case definition.

   The difference between an action and an rpc is that an action is tied
   to a node in the datastore, whereas an rpc is not. When an action is
   invoked, the node in the datastore is specified along with the name
   of the action and the input parameters.

7.15.1. The action's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| description  | 7.21.3  | 0..1        |
| grouping     | 7.12    | 0..n        |
| if-feature   | 7.20.2  | 0..n        |
| input        | 7.14.2  | 0..1        |
| output       | 7.14.3  | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| typedef      | 7.3     | 0..n        |
+--------------+---------+-------------+
```

7.15.2. NETCONF XML Encoding Rules

When an action is invoked, an element with the local name "action" in
the namespace "urn:ietf:params:xml:ns:yang:1" (see Section 5.3.1) is
encoded as a child XML element to the <rpc> element defined in
[RFC6241], as designated by the substitution group "rpcOperation" in
[RFC6241].

The <action> element contains a hierarchy of nodes that identifies
the node in the datastore. It MUST contain all containers and list
nodes in the direct path from the top level down to the list or
container containing the action. For lists, all key leafs MUST also
be included. The innermost container or list contains an XML element
that carries the name of the defined action. Within this element,
the input parameters are encoded as child XML elements, in the same
order as they are defined within the "input" statement.

Only one action can be invoked in one <rpc>. If more than one action
is present in the <rpc>, the server MUST reply with a "bad-element"
<error-tag> in the <rpc-error>.

If the action operation invocation succeeded and no output parameters
are returned, the <rpc-reply> contains a single <ok/> element defined
in [RFC6241]. If output parameters are returned, they are encoded as
child elements to the <rpc-reply> element defined in [RFC6241], in
the same order as they are defined within the "output" statement.

7.15.3. Usage Example

   The following example defines an action to reset one server at a
   server farm:

```
module example-server-farm {
  yang-version 1.1;
  namespace "urn:example:server-farm";
  prefix "sfarm";

  import ietf-yang-types {
    prefix "yang";
  }

  list server {
    key name;
    leaf name {
      type string;
    }
    action reset {
      input {
        leaf reset-at {
          type yang:date-and-time;
          mandatory true;
        }
      }
      output {
        leaf reset-finished-at {
          type yang:date-and-time;
          mandatory true;
        }
      }
    }
  }
}
```

A corresponding XML instance example of the complete rpc and
rpc-reply:

```
<rpc message-id="101"
       xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <action xmlns="urn:ietf:params:xml:ns:yang:1">
      <server xmlns="urn:example:server-farm">
        <name>apache-1</name>
        <reset>
          <reset-at>2014-07-29T13:42:00Z</reset-at>
        </reset>
      </server>
    </action>
  </rpc>

  <rpc-reply message-id="101"
             xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <reset-finished-at xmlns="urn:example:server-farm">
      2014-07-29T13:42:12Z
    </reset-finished-at>
  </rpc-reply>
```

7.16. The "notification" Statement

The "notification" statement is used to define a notification. It
takes one argument, which is an identifier, followed by a block of
substatements that holds detailed notification information. The
"notification" statement defines a notification node in the schema
tree.

A notification can be defined at the top level of a module, or
connected to a specific container or list data node in the schema
tree.

A notification MUST NOT be defined within an rpc, action, or another
notification, i.e., a notification node MUST NOT have an rpc, action,
or a notification node as one of its ancestors in the schema tree.
For example, this means that it is an error if a grouping that
contains a notification somewhere in its node hierarchy is used in an
rpc definition.

A notification MUST NOT have any ancestor node that is a list node
without a "key" statement.

Since a notification cannot be defined in a "case" statement, it is
an error if a grouping that contains a notification at the top of its
node hierarchy is used in a case definition.

If a leaf in the notification tree has a "mandatory" statement with
the value "true", the leaf MUST be present in a notification
instance.

If a leaf in the notification tree has a default value, the client
MUST use this value in the same cases as those described in
Section 7.6.1. In these cases, the client MUST operationally behave
as if the leaf was present in the notification instance with the
default value as its value.

If a leaf-list in the notification tree has one or more default
values, the client MUST use these values in the same cases as those
described in Section 7.7.2. In these cases, the client MUST
operationally behave as if the leaf-list was present in the
notification instance with the default values as its values.

Since the notification tree is not part of any datastore, all
"config" statements for nodes in the notification tree are ignored.

7.16.1. The notification's Substatements

substatement	section	cardinality
anydata	7.10	0..n
anyxml	7.11	0..n
choice	7.9	0..n
container	7.5	0..n
description	7.21.3	0..1
grouping	7.12	0..n
if-feature	7.20.2	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
must	7.5.3	0..n
reference	7.21.4	0..1
status	7.21.2	0..1
typedef	7.3	0..n
uses	7.13	0..n

7.16.2. NETCONF XML Encoding Rules

A notification node that is defined on the top level of a module is
encoded as a child XML element to the <notification> element defined
in "NETCONF Event Notifications" [RFC5277]. The element's local name
is the notification's identifier, and its namespace is the module's
XML namespace (see Section 7.1.3).

When a notification node is defined as a child to a data node, the
<notification> element defined in [RFC5277] contains a hierarchy of
nodes that identifies the node in the datastore. It MUST contain all
containers and list nodes from the top level down to the list or
container containing the notification. For lists, all key leafs MUST
also be included. The innermost container or list contains an XML
element that carries the name of the defined notification.

The notification's child nodes are encoded as subelements to the
notification node's XML element, in any order.

7.16.3. Usage Example

The following example defines a notification at the top level of a
module:

```
module example-event {
  yang-version 1.1;
  namespace "urn:example:event";
  prefix "ev";

  notification event {
    leaf event-class {
      type string;
    }
    leaf reporting-entity {
      type instance-identifier;
    }
    leaf severity {
      type string;
    }
  }
}
```

A corresponding XML instance example of the complete notification:

```
<notification
  xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2008-07-08T00:01:00Z</eventTime>
  <event xmlns="urn:example:event">
    <event-class>fault</event-class>
    <reporting-entity>
      /ex:interface[ex:name='Ethernet0']
    </reporting-entity>
    <severity>major</severity>
  </event>
</notification>
```

The following example defines a notification in a data node:

```
module example-interface-module {
  yang-version 1.1;
  namespace "urn:example:interface-module";
  prefix "if";

  container interfaces {
    list interface {
      key "name";
      leaf name {
        type string;
      }
      notification interface-enabled {
        leaf by-user {
          type string;
        }
      }
    }
  }
}
```

A corresponding XML instance example of the complete notification:

```
<notification
  xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2008-07-08T00:01:00Z</eventTime>
  <interfaces xmlns="urn:example:interface-module">
    <interface>
      <name>eth1</name>
      <interface-enabled>
        <by-user>fred</by-user>
      </interface-enabled>
    </interface>
  </interfaces>
</notification>
```

7.17. The "augment" Statement

The "augment" statement allows a module or submodule to add to a
schema tree defined in an external module, or in the current module
and its submodules, and to add to the nodes from a grouping in a
"uses" statement. The argument is a string that identifies a node in
the schema tree. This node is called the augment's target node. The
target node MUST be either a container, list, choice, case, input,
output, or notification node. It is augmented with the nodes defined
in the substatements that follow the "augment" statement.

The argument string is a schema node identifier (see Section 6.5).
If the "augment" statement is on the top level in a module or
submodule, the absolute form (defined by the rule
"absolute-schema-nodeid" in Section 14) of a schema node identifier
MUST be used. If the "augment" statement is a substatement to the
"uses" statement, the descendant form (defined by the rule
"descendant-schema-nodeid" in Section 14) MUST be used.

If the target node is a container, list, case, input, output, or
notification node, the "container", "leaf", "list", "leaf-list",
"uses", and "choice" statements can be used within the "augment"
statement.

If the target node is a container or list node, the "action" and
"notification" statements can be used within the "augment" statement.

If the target node is a choice node, the "case" statement or a
shorthand "case" statement (see Section 7.9.2) can be used within the
"augment" statement.

The "augment" statement MUST NOT add multiple nodes with the same
name from the same module to the target node.

If the augmentation adds mandatory nodes (see Section 3) that
represent configuration to a target node in another module, the
augmentation MUST be made conditional with a "when" statement. Care
must be taken when defining the "when" expression so that clients
that do not know about the augmenting module do not break.

In the following example, it is OK to augment the "interface" entry
with "mandatory-leaf" because the augmentation depends on support for
"some-new-iftype". The old client does not know about this type, so
it would never select this type and would therefore not be adding a
mandatory data node.

```
module example-augment {
  yang-version 1.1;
  namespace "urn:example:augment";
  prefix mymod;

  import ietf-interfaces {
    prefix if;
  }

  identity some-new-iftype {
    base if:interface-type;
  }
```

```
   augment "/if:interfaces/if:interface" {
      when 'derived-from-or-self(if:type, "mymod:some-new-iftype")';

      leaf mandatory-leaf {
         mandatory true;
         type string;
      }
   }
}
```

7.17.1. The augment's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| action       | 7.15    | 0..n        |
| anydata      | 7.10    | 0..n        |
| anyxml       | 7.11    | 0..n        |
| case         | 7.9.2   | 0..n        |
| choice       | 7.9     | 0..n        |
| container    | 7.5     | 0..n        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| leaf         | 7.6     | 0..n        |
| leaf-list    | 7.7     | 0..n        |
| list         | 7.8     | 0..n        |
| notification | 7.16    | 0..n        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
| uses         | 7.13    | 0..n        |
| when         | 7.21.5  | 0..1        |
+--------------+---------+-------------+
```

7.17.2. XML Encoding Rules

All data nodes defined in the "augment" statement are defined as XML elements in the XML namespace of the module where the "augment" is specified.

When a node is augmented, the augmenting child nodes are encoded as subelements to the augmented node, in any order.

7.17.3. Usage Example

   In namespace urn:example:interface-module, we have:

```
   container interfaces {
     list ifEntry {
       key "ifIndex";

       leaf ifIndex {
         type uint32;
       }
       leaf ifDescr {
         type string;
       }
       leaf ifType {
         type iana:IfType;
       }
       leaf ifMtu {
         type int32;
       }
     }
   }
```

   Then, in namespace urn:example:ds0, we have:

```
   import example-interface-module {
     prefix "if";
   }
   augment "/if:interfaces/if:ifEntry" {
     when "if:ifType='ds0'";
     leaf ds0ChannelNumber {
       type ChannelNumber;
     }
   }
```

A corresponding XML instance example:

```
<interfaces xmlns="urn:example:interface-module"
            xmlns:ds0="urn:example:ds0">
  <ifEntry>
    <ifIndex>1</ifIndex>
    <ifDescr>Flintstone Inc Ethernet A562</ifDescr>
    <ifType>ethernetCsmacd</ifType>
    <ifMtu>1500</ifMtu>
  </ifEntry>
  <ifEntry>
    <ifIndex>2</ifIndex>
    <ifDescr>Flintstone Inc DS0</ifDescr>
    <ifType>ds0</ifType>
    <ds0:ds0ChannelNumber>1</ds0:ds0ChannelNumber>
  </ifEntry>
</interfaces>
```

As another example, suppose we have the choice defined in
Section 7.9.6. The following construct can be used to extend the
protocol definition:

```
augment /ex:system/ex:protocol/ex:name {
  case c {
    leaf smtp {
      type empty;
    }
  }
}
```

A corresponding XML instance example:

```
<ex:system>
  <ex:protocol>
    <ex:tcp/>
  </ex:protocol>
</ex:system>
```

or

```
<ex:system>
  <ex:protocol>
    <other:smtp/>
  </ex:protocol>
</ex:system>
```

7.18. The "identity" Statement

The "identity" statement is used to define a new globally unique,
abstract, and untyped identity. The identity's only purpose is to
denote its name, semantics, and existence. An identity can be either
defined from scratch or derived from one or more base identities.
The identity's argument is an identifier that is the name of the
identity. It is followed by a block of substatements that holds
detailed identity information.

The built-in datatype "identityref" (see Section 9.10) can be used to
reference identities within a data model.

7.18.1. The identity's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| base         | 7.18.2  | 0..n        |
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
+--------------+---------+-------------+
```

7.18.2. The "base" Statement

The "base" statement, which is optional, takes as an argument a
string that is the name of an existing identity, from which the new
identity is derived. If no "base" statement is present, the identity
is defined from scratch. If multiple "base" statements are present,
the identity is derived from all of them.

If a prefix is present on the base name, it refers to an identity
defined in the module that was imported with that prefix, or the
local module if the prefix matches the local module's prefix.
Otherwise, an identity with the matching name MUST be defined in the
current module or an included submodule.

An identity MUST NOT reference itself, neither directly nor
indirectly through a chain of other identities.

The derivation of identities has the following properties:

o It is irreflexive, which means that an identity is not derived
   from itself.

o It is transitive, which means that if identity B is derived from A
   and C is derived from B, then C is also derived from A.

7.18.3. Usage Example

```
module example-crypto-base {
  yang-version 1.1;
  namespace "urn:example:crypto-base";
  prefix "crypto";

  identity crypto-alg {
    description
      "Base identity from which all crypto algorithms
       are derived.";
  }

  identity symmetric-key {
    description
      "Base identity used to identify symmetric-key crypto
       algorithms.";
  }

  identity public-key {
    description
      "Base identity used to identify public-key crypto
       algorithms.";
  }
}

module example-des {
  yang-version 1.1;
  namespace "urn:example:des";
  prefix "des";

  import "example-crypto-base" {
    prefix "crypto";
  }

  identity des {
    base "crypto:crypto-alg";
    base "crypto:symmetric-key";
    description "DES crypto algorithm.";
  }
```

```
      identity des3 {
        base "crypto:crypto-alg";
        base "crypto:symmetric-key";
        description "Triple DES crypto algorithm.";
      }
    }
```

7.19. The "extension" Statement

The "extension" statement allows the definition of new statements
within the YANG language. This new statement definition can be
imported and used by other modules.

The "extension" statement's argument is an identifier that is the new
keyword for the extension and must be followed by a block of
substatements that holds detailed extension information. The purpose
of the "extension" statement is to define a keyword so that it can be
imported and used by other modules.

The extension can be used like a normal YANG statement, with the
statement name followed by an argument if one is defined by the
"extension" statement, and an optional block of substatements. The
statement's name is created by combining the prefix of the module in
which the extension was defined, a colon (":"), and the extension's
keyword, with no interleaving whitespace. The substatements of an
extension are defined by the "extension" statement, using some
mechanism outside the scope of this specification. Syntactically,
the substatements MUST be YANG statements, including extensions
defined using "extension" statements. YANG statements in extensions
MUST follow the syntactical rules in Section 14.

An extension can allow refinement (see Section 7.13.2) and deviations
(Section 7.20.3.2), but the mechanism for how this is defined is
outside the scope of this specification.

7.19.1. The extension's Substatements

substatement	section	cardinality
argument	7.19.2	0..1
description	7.21.3	0..1
reference	7.21.4	0..1
status	7.21.2	0..1

7.19.2. The "argument" Statement

   The "argument" statement, which is optional, takes as an argument a
   string that is the name of the argument to the keyword. If no
   "argument" statement is present, the keyword expects no argument when
   it is used.

   The argument's name is used in the YIN mapping, where it is used as
   an XML attribute or element name, depending on the argument's
   "yin-element" statement.

7.19.2.1. The argument's Substatement

```
           +--------------+----------+-------------+
           | substatement | section  | cardinality |
           +--------------+----------+-------------+
           | yin-element  | 7.19.2.2 | 0..1        |
           +--------------+----------+-------------+
```

7.19.2.2. The "yin-element" Statement

   The "yin-element" statement, which is optional, takes as an argument
   the string "true" or "false". This statement indicates whether the
   argument is mapped to an XML element in YIN or to an XML attribute
   (see Section 13).

   If no "yin-element" statement is present, it defaults to "false".

7.19.3. Usage Example

   To define an extension:

```
     module example-extensions {
       yang-version 1.1;
       ...

       extension c-define {
         description
           "Takes as an argument a name string.
            Makes the code generator use the given name
            in the #define.";
         argument "name";
       }
     }
```

   To use the extension:

```
   module example-interfaces {
     yang-version 1.1;

     ...
     import example-extensions {
       prefix "myext";
     }
     ...

     container interfaces {
       ...
       myext:c-define "MY_INTERFACES";
     }
   }
```

7.20. Conformance-Related Statements

   This section defines statements related to conformance, as described
   in Section 5.6.

7.20.1. The "feature" Statement

   The "feature" statement is used to define a mechanism by which
   portions of the schema are marked as conditional. A feature name is
   defined that can later be referenced using the "if-feature" statement
   (see Section 7.20.2). Schema nodes tagged with an "if-feature"
   statement are ignored by the server unless the server supports the
   given feature expression. This allows portions of the YANG module to
   be conditional based on conditions in the server. The model can
   represent the abilities of the server within the model, giving a
   richer model that allows for differing server abilities and roles.

   The argument to the "feature" statement is the name of the new
   feature and follows the rules for identifiers in Section 6.2. This
   name is used by the "if-feature" statement to tie the schema nodes to
   the feature.

In this example, a feature called "local-storage" represents the
ability for a server to store syslog messages on local storage of
some sort. This feature is used to make the "local-storage-limit"
leaf conditional on the presence of some sort of local storage. If
the server does not report that it supports this feature, the
"local-storage-limit" node is not supported.

```
module example-syslog {
  yang-version 1.1;

  ...
  feature local-storage {
    description
      "This feature means that the server supports local
       storage (memory, flash, or disk) that can be used to
       store syslog messages.";
  }

  container syslog {
    leaf local-storage-limit {
      if-feature local-storage;
      type uint64;
      units "kilobyte";
      config false;
      description
        "The amount of local storage that can be
         used to hold syslog messages.";
    }
  }
}
```

The "if-feature" statement can be used in many places within the YANG
syntax. Definitions tagged with "if-feature" are ignored when the
server does not support that feature.

A feature MUST NOT reference itself, neither directly nor indirectly
through a chain of other features.

In order for a server to support a feature that is dependent on any
other features (i.e., the feature has one or more "if-feature"
substatements), the server MUST also support all the dependent
features.

7.20.1.1. The feature's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
+--------------+---------+-------------+
```

7.20.2. The "if-feature" Statement

   The "if-feature" statement makes its parent statement conditional.
   The argument is a boolean expression over feature names. In this
   expression, a feature name evaluates to "true" if and only if the
   feature is supported by the server. The parent statement is
   implemented by servers where the boolean expression evaluates to
   "true".

   The if-feature boolean expression syntax is formally defined by the
   rule "if-feature-expr" in Section 14. Parentheses are used to group
   expressions. When the expression is evaluated, the order of
   precedence is (highest precedence first): grouping (parentheses),
   "not", "and", "or".

   If a prefix is present on a feature name in the boolean expression,
   the prefixed name refers to a feature defined in the module that was
   imported with that prefix, or the local module if the prefix matches
   the local module's prefix. Otherwise, a feature with the matching
   name MUST be defined in the current module or an included submodule.

   A leaf that is a list key MUST NOT have any "if-feature" statements.

7.20.2.1. Usage Example

   In this example, the container "target" is implemented if either the
   "outbound-tls" or "outbound-ssh" feature is supported by the server.

```
     container target {
       if-feature "outbound-tls or outbound-ssh";
       ...
     }
```

   The following examples are equivalent:

```
     if-feature "not foo or bar and baz";
```

```
     if-feature "(not foo) or (bar and baz)";
```

7.20.3. The "deviation" Statement

   The "deviation" statement defines a hierarchy of a module that the
   server does not implement faithfully. The argument is a string that
   identifies the node in the schema tree where a deviation from the
   module occurs. This node is called the deviation's target node. The
   contents of the "deviation" statement give details about the
   deviation.

   The argument string is an absolute schema node identifier (see
   Section 6.5).

   Deviations define the way a server or class of servers deviate from a
   standard. This means that deviations MUST never be part of a
   published standard, since they are the mechanism for learning how
   implementations vary from the standards.

   Server deviations are strongly discouraged and MUST only be used as a
   last resort. Telling the application how a server fails to follow a
   standard is no substitute for implementing the standard correctly. A
   server that deviates from a module is not fully compliant with the
   module.

   However, in some cases, a particular device may not have the hardware
   or software ability to support parts of a standard module. When this
   occurs, the server makes a choice to either treat attempts to
   configure unsupported parts of the module as an error that is
   reported back to the unsuspecting application or ignore those
   incoming requests. Neither choice is acceptable.

Instead, YANG allows servers to document portions of a base module
that are not supported, or that are supported but with different
syntax, by using the "deviation" statement.

After applying all deviations announced by a server, in any order,
the resulting data model MUST still be valid.

7.20.3.1. The deviation's Substatements

```
+--------------+----------+-------------+
| substatement | section  | cardinality |
+--------------+----------+-------------+
| description  | 7.21.3   | 0..1        |
| deviate      | 7.20.3.2 | 1..n        |
| reference    | 7.21.4   | 0..1        |
+--------------+----------+-------------+
```

7.20.3.2. The "deviate" Statement

The "deviate" statement defines how the server's implementation of
the target node deviates from its original definition. The argument
is one of the strings "not-supported", "add", "replace", or "delete".

The argument "not-supported" indicates that the target node is not
implemented by this server.

The argument "add" adds properties to the target node. The
properties to add are identified by substatements to the "deviate"
statement. If a property can only appear once, the property MUST NOT
exist in the target node.

The argument "replace" replaces properties of the target node. The
properties to replace are identified by substatements to the
"deviate" statement. The properties to replace MUST exist in the
target node.

The argument "delete" deletes properties from the target node. The
properties to delete are identified by substatements to the "delete"
statement. The substatement's keyword MUST match a corresponding
keyword in the target node, and the argument's string MUST be equal
to the corresponding keyword's argument string in the target node.

```
+--------------+---------------+-------------+
| substatement | section       | cardinality |
+--------------+---------------+-------------+
| config       | 7.21.1        | 0..1        |
| default      | 7.6.4, 7.7.4  | 0..n        |
| mandatory    | 7.6.5         | 0..1        |
| max-elements | 7.7.6         | 0..1        |
| min-elements | 7.7.5         | 0..1        |
| must         | 7.5.3         | 0..n        |
| type         | 7.4           | 0..1        |
| unique       | 7.8.3         | 0..n        |
| units        | 7.3.3         | 0..1        |
+--------------+---------------+-------------+
```

               The deviate's Substatements

   If the target node has a property defined by an extension, this
   property can be deviated if the extension allows deviations. See
   Section 7.19 for details.

7.20.3.3. Usage Example

   In this example, the server is informing client applications that it
   does not support the "daytime" service in the style of RFC 867.

```
   module example-deviations {
     yang-version 1.1;
     namespace "urn:example:deviations";
     prefix md;

     import example-base {
       prefix base;
     }

     deviation /base:system/base:daytime {
       deviate not-supported;
     }
     ...
   }
```

   A server would advertise both modules "example-base" and
   "example-deviations".

   The following example sets a server-specific default value to a leaf
   that does not have a default value defined:

```
deviation /base:system/base:user/base:type {
  deviate add {
    default "admin"; // new users are 'admin' by default
  }
}
```

   In this example, the server limits the number of name servers to 3:

```
deviation /base:system/base:name-server {
  deviate replace {
    max-elements 3;
  }
}
```

   If the original definition is:

```
container system {
  must "daytime or time";
  ...
}
```

   a server might remove this "must" constraint by doing:

```
deviation /base:system {
  deviate delete {
    must "daytime or time";
  }
}
```

7.21. Common Statements

   This section defines substatements common to several other
   statements.

7.21.1. The "config" Statement

   The "config" statement takes as an argument the string "true" or
   "false". If "config" is "true", the definition represents
   configuration. Data nodes representing configuration are part of
   configuration datastores.

   If "config" is "false", the definition represents state data. Data
   nodes representing state data are not part of configuration
   datastores.

   If "config" is not specified, the default is the same as the parent
   schema node's "config" value. If the parent node is a case node, the
   value is the same as the case node's parent choice node.

   If the top node does not specify a "config" statement, the default is
   "true".

   If a node has "config" set to "false", no node underneath it can have
   "config" set to "true".

7.21.2. The "status" Statement

   The "status" statement takes as an argument one of the strings
   "current", "deprecated", or "obsolete".

   o "current" means that the definition is current and valid.

   o "deprecated" indicates an obsolete definition, but it permits
      new/continued implementation in order to foster interoperability
      with older/existing implementations.

   o "obsolete" means that the definition is obsolete and SHOULD NOT be
      implemented and/or can be removed from implementations.

   If no status is specified, the default is "current".

   If a definition is "current", it MUST NOT reference a "deprecated" or
   "obsolete" definition within the same module.

   If a definition is "deprecated", it MUST NOT reference an "obsolete"
   definition within the same module.

   For example, the following is illegal:

```
     typedef my-type {
       status deprecated;
       type int32;
     }

     leaf my-leaf {
       status current;
       type my-type; // illegal, since my-type is deprecated
     }
```

7.21.3. The "description" Statement

   The "description" statement takes as an argument a string that
   contains a human-readable textual description of this definition.
   The text is provided in a language (or languages) chosen by the
   module developer; for the sake of interoperability, it is RECOMMENDED
   to choose a language that is widely understood among the community of
   network administrators who will use the module.

7.21.4. The "reference" Statement

   The "reference" statement takes as an argument a string that is a
   human-readable cross-reference to an external document -- either
   another module that defines related management information or a
   document that provides additional information relevant to this
   definition.

   For example, a typedef for a "uri" data type could look like:

      typedef uri {
        type string;
        reference
          "RFC 3986: Uniform Resource Identifier (URI): Generic Syntax";
        ...
      }

7.21.5. The "when" Statement

   The "when" statement makes its parent data definition statement
   conditional. The node defined by the parent data definition
   statement is only valid when the condition specified by the "when"
   statement is satisfied. The statement's argument is an XPath
   expression (see Section 6.4), which is used to formally specify this
   condition. If the XPath expression conceptually evaluates to "true"
   for a particular instance, then the node defined by the parent data
   definition statement is valid; otherwise, it is not.

   A leaf that is a list key MUST NOT have a "when" statement.

   If a key leaf is defined in a grouping that is used in a list, the
   "uses" statement MUST NOT have a "when" statement.

   See Section 8.3.2 for additional information.

The XPath expression is conceptually evaluated in the following
context, in addition to the definition in Section 6.4.1:

o If the "when" statement is a child of an "augment" statement, then
   the context node is the augment's target node in the data tree, if
   the target node is a data node. Otherwise, the context node is
   the closest ancestor node to the target node that is also a data
   node. If no such node exists, the context node is the root node.
   The accessible tree is tentatively altered during the processing
   of the XPath expression by removing all instances (if any) of the
   nodes added by the "augment" statement.

o If the "when" statement is a child of a "uses", "choice", or
   "case" statement, then the context node is the closest ancestor
   node to the node with the "when" statement that is also a data
   node. If no such node exists, the context node is the root node.
   The accessible tree is tentatively altered during the processing
   of the XPath expression by removing all instances (if any) of the
   nodes added by the "uses", "choice", or "case" statement.

o If the "when" statement is a child of any other data definition
   statement, the accessible tree is tentatively altered during the
   processing of the XPath expression by replacing all instances of
   the data node for which the "when" statement is defined with a
   single dummy node with the same name, but with no value and no
   children. If no such instance exists, the dummy node is
   tentatively created. The context node is this dummy node.

The result of the XPath expression is converted to a boolean value
using the standard XPath rules.

If the XPath expression references any node that also has associated
"when" statements, those "when" expressions MUST be evaluated first.
There MUST NOT be any circular dependencies among "when" expressions.

Note that the XPath expression is conceptually evaluated. This means
that an implementation does not have to use an XPath evaluator in the
server. The "when" statement can very well be implemented with
specially written code.

8. Constraints

8.1. Constraints on Data

Several YANG statements define constraints on valid data. These constraints are enforced in different ways, depending on what type of data the statement defines.

o If the constraint is defined on configuration data, it MUST be true in a valid configuration data tree.

o If the constraint is defined on state data, it MUST be true in a valid state data tree.

o If the constraint is defined on notification content, it MUST be true in any notification data tree.

o If the constraint is defined on RPC or action input parameters, it MUST be true in an invocation of the RPC or action operation.

o If the constraint is defined on RPC or action output parameters, it MUST be true in the RPC or action reply.

The following properties are true in all data trees:

o All leaf data values MUST match the type constraints for the leaf, including those defined in the type's "range", "length", and "pattern" properties.

o All key leafs MUST be present for all list entries.

o Nodes MUST be present for at most one case branch in all choices.

o There MUST be no nodes tagged with "if-feature" present if the "if-feature" expression evaluates to "false" in the server.

o There MUST be no nodes tagged with "when" present if the "when" condition evaluates to "false" in the data tree.

The following properties are true in a valid data tree:

o All "must" constraints MUST evaluate to "true".

o All referential integrity constraints defined via the "path" statement MUST be satisfied.

o All "unique" constraints on lists MUST be satisfied.

o The "mandatory" constraint is enforced for leafs and choices,
   unless the node or any of its ancestors has a "when" condition or
   "if-feature" expression that evaluates to "false".

o The "min-elements" and "max-elements" constraints are enforced for
   lists and leaf-lists, unless the node or any of its ancestors has
   a "when" condition or "if-feature" expression that evaluates to
   "false".

The running configuration datastore MUST always be valid.

8.2. Configuration Data Modifications

o If a request creates configuration data nodes under a choice, any
   existing nodes from other case branches in the data tree are
   deleted by the server.

o If a request modifies a configuration data node such that any
   node's "when" expression becomes false, then the node in the data
   tree with the "when" expression is deleted by the server.

8.3. NETCONF Constraint Enforcement Model

For configuration data, there are three windows when constraints MUST
be enforced:

o during parsing of RPC payloads

o during processing of the <edit-config> operation

o during validation

Each of these scenarios is considered in the following sections.

8.3.1. Payload Parsing

When content arrives in RPC payloads, it MUST be well-formed XML,
following the hierarchy and content rules defined by the set of
models the server implements.

o If a leaf data value does not match the type constraints for the
   leaf, including those defined in the type's "range", "length", and
   "pattern" properties, the server MUST reply with an
   "invalid-value" <error-tag> in the <rpc-error>, and with the
   error-app-tag (Section 7.5.4.2) and error-message
   (Section 7.5.4.1) associated with the constraint, if any exist.

o   If all keys of a list entry are not present, the server MUST reply
    with a "missing-element" <error-tag> in the <rpc-error>.

o   If data for more than one case branch of a choice is present, the
    server MUST reply with a "bad-element" <error-tag> in the
    <rpc-error>.

o   If data for a node tagged with "if-feature" is present and the
    "if-feature" expression evaluates to "false" in the server, the
    server MUST reply with an "unknown-element" <error-tag> in the
    <rpc-error>.

o   If data for a node tagged with "when" is present and the "when"
    condition evaluates to "false", the server MUST reply with an
    "unknown-element" <error-tag> in the <rpc-error>.

o   For insert handling, if the values for the attributes "before" and
    "after" are not valid for the type of the appropriate key leafs,
    the server MUST reply with a "bad-attribute" <error-tag> in the
    <rpc-error>.

o   If the attributes "before" and "after" appear in any element that
    is not a list whose "ordered-by" property is "user", the server
    MUST reply with an "unknown-attribute" <error-tag> in the
    <rpc-error>.

8.3.2. NETCONF <edit-config> Processing

After the incoming data is parsed, the NETCONF server performs the
<edit-config> operation by applying the data to the configuration
datastore. During this processing, the following errors MUST be
detected:

o   Delete requests for non-existent data.

o   Create requests for existent data.

o   Insert requests with "before" or "after" parameters that do not
    exist.

o   Modification requests for nodes tagged with "when", and the "when"
    condition evaluates to "false". In this case, the server MUST
    reply with an "unknown-element" <error-tag> in the <rpc-error>.

8.3.3. Validation

When datastore processing is complete, the final contents MUST obey
all validation constraints. This validation processing is performed
at differing times according to the datastore. If the datastore is
"running" or "startup", these constraints MUST be enforced at the end
of the <edit-config> or <copy-config> operation. If the datastore is
"candidate", the constraint enforcement is delayed until a <commit>
or <validate> operation takes place.

9. Built-In Types

YANG has a set of built-in types, similar to those of many
programming languages, but with some differences due to special
requirements from the management information model.

Additional types may be defined that are derived from those built-in
types or from other derived types. Derived types may use subtyping
to formally restrict the set of possible values.

The different built-in types and their derived types allow different
kinds of subtyping, namely length and regular expression restrictions
of strings (Sections 9.4.4 and 9.4.5) and range restrictions of
numeric types (Section 9.2.4).

The lexical representation of a value of a certain type is used in
the XML encoding and when specifying default values and numerical
ranges in YANG modules.

9.1. Canonical Representation

For most types, there is a single canonical representation of the
type's values. Some types allow multiple lexical representations of
the same value; for example, the positive integer "17" can be
represented as "+17" or "17". Implementations MUST support all
lexical representations specified in this document.

When a server sends XML-encoded data, it MUST use the canonical form
defined in this section. Other encodings may introduce alternate
representations. Note, however, that values in the data tree are
conceptually stored in the canonical representation as defined in
this section. In particular, any XPath expression evaluations are
done using the canonical form if the data type has a canonical form.
If the data type does not have a canonical form, the format of the
value MUST match the data type's lexical representation, but the
exact format is implementation dependent.

Some types have a lexical representation that depends on the
encoding, e.g., the XML context in which they occur. These types do
not have a canonical form.

9.2. The Integer Built-In Types

The integer built-in types are int8, int16, int32, int64, uint8,
uint16, uint32, and uint64. They represent signed and unsigned
integers of different sizes:

int8   represents integer values between -128 and 127, inclusively.

int16   represents integer values between -32768 and 32767,
    inclusively.

int32   represents integer values between -2147483648 and 2147483647,
    inclusively.

int64   represents integer values between -9223372036854775808 and
    9223372036854775807, inclusively.

uint8   represents integer values between 0 and 255, inclusively.

uint16   represents integer values between 0 and 65535, inclusively.

uint32   represents integer values between 0 and 4294967295,
    inclusively.

uint64   represents integer values between 0 and 18446744073709551615,
    inclusively.

9.2.1. Lexical Representation

An integer value is lexically represented as an optional sign ("+" or
"-"), followed by a sequence of decimal digits. If no sign is
specified, "+" is assumed.

For convenience, when specifying a default value for an integer in a
YANG module, an alternative lexical representation can be used that
represents the value in a hexadecimal or octal notation. The
hexadecimal notation consists of an optional sign ("+" or "-"),
followed by the characters "0x", followed by a number of hexadecimal
digits where letters may be uppercase or lowercase. The octal
notation consists of an optional sign ("+" or "-"), followed by the
character "0", followed by a number of octal digits.

Note that if a default value in a YANG module has a leading zero
("0"), it is interpreted as an octal number. In the XML encoding, an
integer is always interpreted as a decimal number, and leading zeros
are allowed.

Examples:

```
  // legal values
  +4711                          // legal positive value
  4711                           // legal positive value
  -123                           // legal negative value
  0xf00f                         // legal positive hexadecimal value
  -0xf                           // legal negative hexadecimal value
  052                            // legal positive octal value

  // illegal values
  - 1                            // illegal intermediate space
```

9.2.2. Canonical Form

The canonical form of a positive integer does not include the sign
"+". Leading zeros are prohibited. The value zero is represented
as "0".

9.2.3. Restrictions

All integer types can be restricted with the "range" statement
(Section 9.2.4).

9.2.4. The "range" Statement

The "range" statement, which is an optional substatement to the
"type" statement, takes as an argument a range expression string. It
is used to restrict integer and decimal built-in types, or types
derived from them.

A range consists of an explicit value, or a lower-inclusive bound,
two consecutive dots "..", and an upper-inclusive bound. Multiple
values or ranges can be given, separated by "|". If multiple values
or ranges are given, they all MUST be disjoint and MUST be in
ascending order. If a range restriction is applied to a type that is
already range-restricted, the new restriction MUST be equally
limiting or more limiting, i.e., raising the lower bounds, reducing
the upper bounds, removing explicit values or ranges, or splitting
ranges into multiple ranges with intermediate gaps. Each explicit
value and range boundary value given in the range expression MUST

match the type being restricted or be one of the special values "min"
or "max". "min" and "max" mean the minimum and maximum values
accepted for the type being restricted, respectively.

The range expression syntax is formally defined by the rule
"range-arg" in Section 14.

9.2.4.1. The range's Substatements

```
+----------------+---------+-------------+
| substatement   | section | cardinality |
+----------------+---------+-------------+
| description    | 7.21.3  | 0..1        |
| error-app-tag  | 7.5.4.2 | 0..1        |
| error-message  | 7.5.4.1 | 0..1        |
| reference      | 7.21.4  | 0..1        |
+----------------+---------+-------------+
```

9.2.5. Usage Example

```
typedef my-base-int32-type {
  type int32 {
    range "1..4 | 10..20";
  }
}

typedef my-type1 {
  type my-base-int32-type {
    // legal range restriction
    range "11..max"; // 11..20
  }
}

typedef my-type2 {
  type my-base-int32-type {
    // illegal range restriction
    range "11..100";
  }
}
```

9.3. The decimal64 Built-In Type

The decimal64 built-in type represents a subset of the real numbers,
which can be represented by decimal numerals. The value space of
decimal64 is the set of numbers that can be obtained by multiplying a
64-bit signed integer by a negative power of ten, i.e., expressible
as "$i \times 10^{-n}$" where i is an integer64 and n is an integer between 1
and 18, inclusively.

9.3.1. Lexical Representation

A decimal64 value is lexically represented as an optional sign ("+" or "-"), followed by a sequence of decimal digits, optionally followed by a period ('.') as a decimal indicator and a sequence of decimal digits. If no sign is specified, "+" is assumed.

9.3.2. Canonical Form

The canonical form of a positive decimal64 value does not include the sign "+". The decimal point is required. Leading and trailing zeros are prohibited, subject to the rule that there MUST be at least one digit before and after the decimal point. The value zero is represented as "0.0".

9.3.3. Restrictions

A decimal64 type can be restricted with the "range" statement (Section 9.2.4).

9.3.4. The "fraction-digits" Statement

The "fraction-digits" statement, which is a substatement to the "type" statement, MUST be present if the type is "decimal64". It takes as an argument an integer between 1 and 18, inclusively. It controls the size of the minimum difference between values of a decimal64 type by restricting the value space to numbers that are expressible as "$i \times 10^{-n}$" where n is the fraction-digits argument.

The following table lists the minimum and maximum values for each
fraction-digit value:

```
+---------------+----------------------+----------------------+
| fraction-digit | min                 | max                  |
+---------------+----------------------+----------------------+
| 1             | -922337203685477580.8 | 922337203685477580.7 |
| 2             | -9223372036854775808  | 92233720368547758.07 |
| 3             | -9223372036854775808  | 9223372036854775.807 |
| 4             | -922337203685477.5808 | 922337203685477.5807 |
| 5             | -9223372036854.75808  | 9223372036854.75807  |
| 6             | -922337203685.4775808 | 9223372036854.775807 |
| 7             | -92233720368.54775808 | 922337203685.4775807 |
| 8             | -9223372036.854775808 | 92233720368.54775807 |
| 9             | -922337203.6854775808 | 9223372036.854775807 |
| 10            | -92233720.36854775808 | 922337203.6854775807 |
| 11            | -9223372.036854775808 | 92233720.36854775807 |
| 12            | -922337.2036854775808 | 9223372.036854775807 |
| 13            | -92233.72036854775808 | 922337.2036854775807 |
| 14            | -9223.372036854775808 | 92233.72036854775807 |
| 15            | -922.3372036854775808 | 9223.372036854775807 |
| 16            | -92.23372036854775808 | 922.3372036854775807 |
| 17            | -9.223372036854775808 | 92.23372036854775807 |
| 18            | -9.223372036854775808 | 9.223372036854775807 |
+---------------+----------------------+----------------------+
```

9.3.5. Usage Example

```
typedef my-decimal {
  type decimal64 {
    fraction-digits 2;
    range "1 .. 3.14 | 10 | 20..max";
  }
}
```

9.4. The string Built-In Type

The string built-in type represents human-readable strings in YANG.
Legal characters are the Unicode and ISO/IEC 10646 [ISO.10646]
characters, including tab, carriage return, and line feed but
excluding the other C0 control characters, the surrogate blocks, and
the noncharacters. The string syntax is formally defined by the rule
"yang-string" in Section 14.

9.4.1. Lexical Representation

A string value is lexically represented as character data in the XML
encoding.

9.4.2. Canonical Form

The canonical form is the same as the lexical representation. No
Unicode normalization of string values is performed.

9.4.3. Restrictions

A string can be restricted with the "length" (Section 9.4.4) and
"pattern" (Section 9.4.5) statements.

9.4.4. The "length" Statement

The "length" statement, which is an optional substatement to the
"type" statement, takes as an argument a length expression string.
It is used to restrict the built-in types "string" and "binary" or
types derived from them.

A "length" statement restricts the number of Unicode characters in
the string.

A length range consists of an explicit value, or a lower bound, two
consecutive dots "..", and an upper bound. Multiple values or ranges
can be given, separated by "|". Length-restricting values MUST NOT
be negative. If multiple values or ranges are given, they all MUST
be disjoint and MUST be in ascending order. If a length restriction
is applied to a type that is already length-restricted, the new
restriction MUST be equally limiting or more limiting, i.e., raising
the lower bounds, reducing the upper bounds, removing explicit length
values or ranges, or splitting ranges into multiple ranges with
intermediate gaps. A length value is a non-negative integer or one
of the special values "min" or "max". "min" and "max" mean the
minimum and maximum lengths accepted for the type being restricted,
respectively. An implementation is not required to support a length
value larger than 18446744073709551615.

The length expression syntax is formally defined by the rule
"length-arg" in Section 14.

9.4.4.1. The length's Substatements

substatement	section	cardinality
description	7.21.3	0..1
error-app-tag	7.5.4.2	0..1
error-message	7.5.4.1	0..1
reference	7.21.4	0..1

9.4.5. The "pattern" Statement

   The "pattern" statement, which is an optional substatement to the
   "type" statement, takes as an argument a regular expression string,
   as defined in [XSD-TYPES]. It is used to restrict the built-in type
   "string", or types derived from "string", to values that match the
   pattern.

   If the type has multiple "pattern" statements, the expressions are
   ANDed together, i.e., all such expressions have to match.

   If a pattern restriction is applied to a type that is already
   pattern-restricted, values must match all patterns in the base type,
   in addition to the new patterns.

9.4.5.1. The pattern's Substatements

```
            +---------------+---------+-------------+
            | substatement  | section | cardinality |
            +---------------+---------+-------------+
            | description   | 7.21.3  | 0..1        |
            | error-app-tag | 7.5.4.2 | 0..1        |
            | error-message | 7.5.4.1 | 0..1        |
            | modifier      | 9.4.6   | 0..1        |
            | reference     | 7.21.4  | 0..1        |
            +---------------+---------+-------------+
```

9.4.6. The "modifier" Statement

   The "modifier" statement, which is an optional substatement to the
   "pattern" statement, takes as an argument the string "invert-match".

   If a pattern has the "invert-match" modifier present, the type is
   restricted to values that do not match the pattern.

9.4.7. Usage Example

   With the following typedef:

      typedef my-base-str-type {
        type string {
          length "1..255";
        }
      }

   the following refinement is legal:

      type my-base-str-type {
        // legal length refinement
        length "11 | 42..max"; // 11 | 42..255
      }

   and the following refinement is illegal:

      type my-base-str-type {
        // illegal length refinement
        length "1..999";
      }

   With the following type:

      type string {
        length "0..4";
        pattern "[0-9a-fA-F]*";
      }

   the following strings match:

      AB   // legal
      9A00   // legal

   and the following strings do not match:

      00ABAB   // illegal, too long
      xx00   // illegal, bad characters

With the following type:

```
type string {
  length "1..max";
  pattern '[a-zA-Z_][a-zA-Z0-9\-_.]*';
  pattern '[xX][mM][lL].*' {
    modifier invert-match;
  }
}
```

the following string matches:

```
enabled     // legal
```

and the following strings do not match:

```
10-mbit     // illegal, starts with a number
xml-element // illegal, starts with illegal sequence
```

9.5. The boolean Built-In Type

The boolean built-in type represents a boolean value.

9.5.1. Lexical Representation

The lexical representation of a boolean value is a string with a value of "true" or "false". These values MUST be in lowercase.

9.5.2. Canonical Form

The canonical form is the same as the lexical representation.

9.5.3. Restrictions

A boolean cannot be restricted.

9.6. The enumeration Built-In Type

The enumeration built-in type represents values from a set of assigned names.

9.6.1. Lexical Representation

The lexical representation of an enumeration value is the assigned name string.

9.6.2. Canonical Form

The canonical form is the assigned name string.

9.6.3. Restrictions

An enumeration can be restricted with one or more "enum"
(Section 9.6.4) statements, which enumerate a subset of the values
for the base type.

9.6.4. The "enum" Statement

The "enum" statement, which is a substatement to the "type"
statement, MUST be present if the type is "enumeration". It is
repeatedly used to specify each assigned name of an enumeration type.
It takes as an argument a string that is the assigned name. The
string MUST NOT be zero-length and MUST NOT have any leading or
trailing whitespace characters (any Unicode character with the
"White_Space" property). The use of Unicode control codes SHOULD be
avoided.

The statement is optionally followed by a block of substatements that
holds detailed enum information.

All assigned names in an enumeration MUST be unique.

When an existing enumeration type is restricted, the set of assigned
names in the new type MUST be a subset of the base type's set of
assigned names. The value of such an assigned name MUST NOT be
changed.

9.6.4.1. The enum's Substatements

substatement	section	cardinality
description	7.21.3	0..1
if-feature	7.20.2	0..n
reference	7.21.4	0..1
status	7.21.2	0..1
value	9.6.4.2	0..1

9.6.4.2. The "value" Statement

   The "value" statement, which is optional, is used to associate an
   integer value with the assigned name for the enum. This integer
   value MUST be in the range -2147483648 to 2147483647, and it MUST be
   unique within the enumeration type.

   If a value is not specified, then one will be automatically assigned.
   If the "enum" substatement is the first one defined, the assigned
   value is zero (0); otherwise, the assigned value is one greater than
   the current highest enum value (i.e., the highest enum value,
   implicit or explicit, prior to the current "enum" substatement in the
   parent "type" statement).

   Note that the presence of an "if-feature" statement in an "enum"
   statement does not affect the automatically assigned value.

   If the current highest value is equal to 2147483647, then an enum
   value MUST be specified for "enum" substatements following the one
   with the current highest value.

   When an existing enumeration type is restricted, the "value"
   statement MUST either have the same value as in the base type or not
   be present, in which case the value is the same as in the base type.

9.6.5. Usage Example

```
   leaf myenum {
     type enumeration {
       enum zero;
       enum one;
       enum seven {
         value 7;
       }
     }
   }
```

   The lexical representation of the leaf "myenum" with
   value "seven" is:

      <myenum>seven</myenum>

With the following typedef:

```
typedef my-base-enumeration-type {
  type enumeration {
    enum white {
      value 1;
    }
    enum yellow {
      value 2;
    }
    enum red {
      value 3;
    }
  }
}
```

the following refinement is legal:

```
type my-base-enumeration-type {
  // legal enum refinement
  enum yellow;
  enum red {
    value 3;
  }
}
```

and the following refinement is illegal:

```
type my-base-enumeration-type {
  // illegal enum refinement
  enum yellow {
    value 4; // illegal value change
  }
  enum black; // illegal addition of new name
}
```

The following example shows how an "enum" can be tagged with
"if-feature", making the value legal only on servers that advertise
the corresponding feature:

```
type enumeration {
  enum tcp;
  enum ssh {
    if-feature ssh;
  }
  enum tls {
    if-feature tls;
  }
}
```

9.7. The bits Built-In Type

The bits built-in type represents a bit set. That is, a bits value
is a set of flags identified by small integer position numbers
starting at 0. Each bit number has an assigned name.

When an existing bits type is restricted, the set of assigned names
in the new type MUST be a subset of the base type's set of assigned
names. The bit position of such an assigned name MUST NOT be
changed.

9.7.1. Restrictions

A bits type can be restricted with the "bit" (Section 9.7.4)
statement.

9.7.2. Lexical Representation

The lexical representation of the bits type is a space-separated list
of the names of the bits that are set. A zero-length string thus
represents a value where no bits are set.

9.7.3. Canonical Form

In the canonical form, the bit values are separated by a single space
character and they appear ordered by their position (see
Section 9.7.4.2).

9.7.4. The "bit" Statement

The "bit" statement, which is a substatement to the "type" statement,
MUST be present if the type is "bits". It is repeatedly used to
specify each assigned named bit of a bits type. It takes as an
argument a string that is the assigned name of the bit. It is
followed by a block of substatements that holds detailed bit
information. The assigned name follows the same syntax rules as an
identifier (see Section 6.2).

All assigned names in a bits type MUST be unique.

9.7.4.1. The bit's Substatements

```
+--------------+---------+-------------+
| substatement | section | cardinality |
+--------------+---------+-------------+
| description  | 7.21.3  | 0..1        |
| if-feature   | 7.20.2  | 0..n        |
| position     | 9.7.4.2 | 0..1        |
| reference    | 7.21.4  | 0..1        |
| status       | 7.21.2  | 0..1        |
+--------------+---------+-------------+
```

9.7.4.2. The "position" Statement

The "position" statement, which is optional, takes as an argument a
non-negative integer value that specifies the bit's position within a
hypothetical bit field. The position value MUST be in the range 0 to
4294967295, and it MUST be unique within the bits type.

If a bit position is not specified, then one will be automatically
assigned. If the "bit" substatement is the first one defined, the
assigned value is zero (0); otherwise, the assigned value is one
greater than the current highest bit position (i.e., the highest bit
position, implicit or explicit, prior to the current "bit"
substatement in the parent "type" statement).

Note that the presence of an "if-feature" statement in a "bit"
statement does not affect the automatically assigned position.

If the current highest bit position value is equal to 4294967295,
then a position value MUST be specified for "bit" substatements
following the one with the current highest position value.

When an existing bits type is restricted, the "position" statement
MUST either have the same value as in the base type or not be
present, in which case the value is the same as in the base type.

9.7.5. Usage Example

   Given the following typedef and leaf:

```
typedef mybits-type {
  type bits {
    bit disable-nagle {
      position 0;
    }
    bit auto-sense-speed {
      position 1;
    }
    bit ten-mb-only {
      position 2;
    }
  }
}

leaf mybits {
  type mybits-type;
  default "auto-sense-speed";
}
```

   The lexical representation of this leaf with bit values disable-nagle
   and ten-mb-only set would be:

```
<mybits>disable-nagle ten-mb-only</mybits>
```

   The following example shows a legal refinement of this type:

```
type mybits-type {
  // legal bit refinement
  bit disable-nagle {
    position 0;
  }
  bit auto-sense-speed {
    position 1;
  }
}
```

and the following refinement is illegal:

```
type mybits-type {
  // illegal bit refinement
  bit disable-nagle {
    position 2; // illegal position change
  }
  bit hundred-mb-only; // illegal addition of new name
}
```

9.8. The binary Built-In Type

The binary built-in type represents any binary data, i.e., a sequence of octets.

9.8.1. Restrictions

A binary type can be restricted with the "length" (Section 9.4.4) statement. The length of a binary value is the number of octets it contains.

9.8.2. Lexical Representation

Binary values are encoded with the base64 encoding scheme (see Section 4 in [RFC4648]).

9.8.3. Canonical Form

The canonical form of a binary value follows the rules of "Base 64 Encoding" in [RFC4648].

9.9. The leafref Built-In Type

The leafref built-in type is restricted to the value space of some leaf or leaf-list node in the schema tree and optionally further restricted by corresponding instance nodes in the data tree. The "path" substatement (Section 9.9.2) is used to identify the referred leaf or leaf-list node in the schema tree. The value space of the referring node is the value space of the referred node.

If the "require-instance" property (Section 9.9.3) is "true", there MUST exist a node in the data tree, or a node with a default value in use (see Sections 7.6.1 and 7.7.2), of the referred schema tree leaf or leaf-list node with the same value as the leafref value in a valid data tree.

If the referring node represents configuration data and the
"require-instance" property (Section 9.9.3) is "true", the referred
node MUST also represent configuration.

There MUST NOT be any circular chains of leafrefs.

If the leaf that the leafref refers to is conditional based on one or
more features (see Section 7.20.2), then the leaf with the leafref
type MUST also be conditional based on at least the same set of
features.

9.9.1. Restrictions

A leafref can be restricted with the "require-instance" statement
(Section 9.9.3).

9.9.2. The "path" Statement

The "path" statement, which is a substatement to the "type"
statement, MUST be present if the type is "leafref". It takes as an
argument a string that MUST refer to a leaf or leaf-list node.

The syntax for a path argument is a subset of the XPath abbreviated
syntax. Predicates are used only for constraining the values for the
key nodes for list entries. Each predicate consists of exactly one
equality test per key, and multiple adjacent predicates MAY be
present if a list has multiple keys. The syntax is formally defined
by the rule "path-arg" in Section 14.

The predicates are only used when more than one key reference is
needed to uniquely identify a leaf instance. This occurs if a list
has multiple keys or a reference to a leaf other than the key in a
list is needed. In these cases, multiple leafrefs are typically
specified, and predicates are used to tie them together.

The "path" expression evaluates to a node set consisting of zero,
one, or more nodes. If the "require-instance" property is "true",
this node set MUST be non-empty.

The "path" XPath expression is conceptually evaluated in the
following context, in addition to the definition in Section 6.4.1:

o If the "path" statement is defined within a typedef, the context
   node is the leaf or leaf-list node in the data tree that
   references the typedef.

o Otherwise, the context node is the node in the data tree for which
   the "path" statement is defined.

9.9.3. The "require-instance" Statement

The "require-instance" statement, which is a substatement to the
"type" statement, MAY be present if the type is "instance-identifier"
or "leafref". It takes as an argument the string "true" or "false".
If this statement is not present, it defaults to "true".

If "require-instance" is "true", it means that the instance being
referred to MUST exist for the data to be valid. This constraint is
enforced according to the rules in Section 8.

If "require-instance" is "false", it means that the instance being
referred to MAY exist in valid data.

9.9.4. Lexical Representation

A leafref value is lexically represented the same way as the leaf it
references represents its value.

9.9.5. Canonical Form

The canonical form of a leafref is the same as the canonical form of
the leaf it references.

9.9.6. Usage Example

With the following list:

```
list interface {
  key "name";
  leaf name {
    type string;
  }
  leaf admin-status {
    type admin-status;
  }
  list address {
    key "ip";
    leaf ip {
      type yang:ip-address;
    }
  }
}
```

the following leafref refers to an existing interface:

```
leaf mgmt-interface {
  type leafref {
    path "../interface/name";
  }
}
```

An example of a corresponding XML snippet:

```
<interface>
  <name>eth0</name>
</interface>
<interface>
  <name>lo</name>
</interface>

<mgmt-interface>eth0</mgmt-interface>
```

The following leafrefs refer to an existing address of an interface:

```
container default-address {
  leaf ifname {
    type leafref {
      path "../../interface/name";
    }
  }
  leaf address {
    type leafref {
      path "../../interface[name = current()/../ifname]"
          + "/address/ip";
    }
  }
}
```

   An example of a corresponding XML snippet:

```
<interface>
  <name>eth0</name>
  <admin-status>up</admin-status>
  <address>
    <ip>192.0.2.1</ip>
  </address>
  <address>
    <ip>192.0.2.2</ip>
  </address>
</interface>
<interface>
  <name>lo</name>
  <admin-status>up</admin-status>
  <address>
    <ip>127.0.0.1</ip>
  </address>
</interface>

<default-address>
  <ifname>eth0</ifname>
  <address>192.0.2.2</address>
</default-address>
```

   The following list uses a leafref for one of its keys. This is
   similar to a foreign key in a relational database.

```
list packet-filter {
  key "if-name filter-id";
  leaf if-name {
    type leafref {
      path "/interface/name";
    }
  }
  leaf filter-id {
    type uint32;
  }
  ...
}
```

An example of a corresponding XML snippet:

```
<interface>
  <name>eth0</name>
  <admin-status>up</admin-status>
  <address>
    <ip>192.0.2.1</ip>
  </address>
  <address>
    <ip>192.0.2.2</ip>
  </address>
</interface>

<packet-filter>
  <if-name>eth0</if-name>
  <filter-id>1</filter-id>
  ...
</packet-filter>
<packet-filter>
  <if-name>eth0</if-name>
  <filter-id>2</filter-id>
  ...
</packet-filter>
```

The following notification defines two leafrefs to refer to an existing admin-status:

```
notification link-failure {
  leaf if-name {
    type leafref {
      path "/interface/name";
    }
  }
  leaf admin-status {
    type leafref {
      path "/interface[name = current()/../if-name]"
        + "/admin-status";
    }
  }
}
```

An example of a corresponding XML notification:

```
<notification
  xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2008-04-01T00:01:00Z</eventTime>
  <link-failure xmlns="urn:example:system">
    <if-name>eth0</if-name>
    <admin-status>up</admin-status>
  </link-failure>
</notification>
```

9.10. The identityref Built-In Type

The identityref built-in type is used to reference an existing
identity (see Section 7.18).

9.10.1. Restrictions

An identityref cannot be restricted.

9.10.2. The identityref's "base" Statement

The "base" statement, which is a substatement to the "type"
statement, MUST be present at least once if the type is
"identityref". The argument is the name of an identity, as defined
by an "identity" statement. If a prefix is present on the identity
name, it refers to an identity defined in the module that was
imported with that prefix. Otherwise, an identity with the matching
name MUST be defined in the current module or an included submodule.

Valid values for an identityref are any identities derived from all
the identityref's base identities. On a particular server, the valid
values are further restricted to the set of identities defined in the
modules implemented by the server.

9.10.3. Lexical Representation

An identityref is lexically represented as the referred identity's
qualified name as defined in [XML-NAMES]. If the prefix is not
present, the namespace of the identityref is the default namespace
in effect on the element that contains the identityref value.

When an identityref is given a default value using the "default"
statement, the identity name in the default value MAY have a prefix.
If a prefix is present on the identity name, it refers to an identity
defined in the module that was imported with that prefix, or the
prefix for the current module if the identity is defined in the

current module or one of its submodules. Otherwise, an identity with
the matching name MUST be defined in the current module or one of its
submodules.

The string value of a node of type "identityref" in a "must" or
"when" XPath expression is the referred identity's qualified name
with the prefix present. If the referred identity is defined in an
imported module, the prefix in the string value is the prefix defined
in the corresponding "import" statement. Otherwise, the prefix in
the string value is the prefix for the current module.

9.10.4. Canonical Form

Since the lexical form depends on the XML context in which the value
occurs, this type does not have a canonical form.

9.10.5. Usage Example

With the identity definitions in Section 7.18.3 and the following
module:

```
module example-my-crypto {
  yang-version 1.1;
  namespace "urn:example:my-crypto";
  prefix mc;

  import "example-crypto-base" {
    prefix "crypto";
  }

  identity aes {
    base "crypto:crypto-alg";
  }

  leaf crypto {
    type identityref {
      base "crypto:crypto-alg";
    }
  }

  container aes-parameters {
    when "../crypto = 'mc:aes'";
    ...
  }
}
```

the following is an example of how the leaf "crypto" can be encoded,
if the value is the "des3" identity defined in the "des" module:

```
<crypto xmlns:des="urn:example:des">des:des3</crypto>
```

Any prefixes used in the encoding are local to each instance
encoding. This means that the same identityref may be encoded
differently by different implementations. For example, the following
example encodes the same leaf as above:

```
<crypto xmlns:x="urn:example:des">x:des3</crypto>
```

If the "crypto" leaf's value is instead "aes", defined in the
"example-my-crypto" module, it can be encoded as:

```
<crypto xmlns:mc="urn:example:my-crypto">mc:aes</crypto>
```

or, using the default namespace:

```
<crypto>aes</crypto>
```

9.11. The empty Built-In Type

The empty built-in type represents a leaf that does not have any
value; it conveys information by its presence or absence.

An empty type cannot have a default value.

9.11.1. Restrictions

An empty type cannot be restricted.

9.11.2. Lexical Representation

Not applicable.

9.11.3. Canonical Form

Not applicable.

9.11.4. Usage Example

With the following leaf:

```
leaf enable-qos {
  type empty;
}
```

the following is an example of a valid encoding if the leaf exists:

```
<enable-qos/>
```

9.12. The union Built-In Type

The union built-in type represents a value that corresponds to one of its member types.

When the type is "union", the "type" statement (Section 7.4) MUST be present. It is repeatedly used to specify each member type of the union. It takes as an argument a string that is the name of a member type.

A member type can be of any built-in or derived type.

When generating an XML encoding, a value is encoded according to the rules of the member type to which the value belongs. When interpreting an XML encoding, a value is validated consecutively against each member type, in the order they are specified in the "type" statement, until a match is found. The type that matched will be the type of the value for the node that was validated, and the encoding is interpreted according to the rules for that type.

Any default value or "units" property defined in the member types is not inherited by the union type.

9.12.1. Restrictions

A union cannot be restricted. However, each member type can be restricted, based on the rules defined in Section 9.

9.12.2. Lexical Representation

The lexical representation of a union is a value that corresponds to the representation of any one of the member types.

9.12.3. Canonical Form

   The canonical form of a union value is the same as the canonical form
   of the member type of the value.

9.12.4. Usage Example

   The following is a union of an int32 and an enumeration:

      type union {
        type int32;
        type enumeration {
          enum "unbounded";
        }
      }

   Care must be taken when a member type is a leafref where the
   "require-instance" property (Section 9.9.3) is "true". If a leaf of
   such a type refers to an existing instance, the leaf's value must be
   revalidated if the target instance is deleted. For example, with the
   following definitions:

      list filter {
        key name;
        leaf name {
          type string;
        }
        ...
      }

      leaf outbound-filter {
        type union {
          type leafref {
            path "/filter/name";
          }
          type enumeration {
            enum default-filter;
          }
        }
      }

   assume that there exists an entry in the filter list with the name
   "http" and that the outbound-filter leaf has this value:

      <filter>
        <name>http</name>
      </filter>
      <outbound-filter>http</outbound-filter>

If the filter entry "http" is removed, the outbound-filter leaf's
value doesn't match the leafref, and the next member type is checked.
The current value ("http") doesn't match the enumeration, so the
resulting configuration is invalid.

If the second member type in the union had been of type "string"
instead of an enumeration, the current value would have matched, and
the resulting configuration would have been valid.

9.13. The instance-identifier Built-In Type

The instance-identifier built-in type is used to uniquely identify a
particular instance node in the data tree.

The syntax for an instance-identifier is a subset of the XPath
abbreviated syntax, formally defined by the rule
"instance-identifier" in Section 14. It is used to uniquely identify
a node in the data tree. Predicates are used only for specifying the
values for the key nodes for list entries, a value of a leaf-list
entry, or a positional index for a list without keys. For
identifying list entries with keys, each predicate consists of one
equality test per key, and each key MUST have a corresponding
predicate. If a key is of type "empty", it is represented as a
zero-length string ("").

If the leaf with the instance-identifier type represents
configuration data and the "require-instance" property
(Section 9.9.3) is "true", the node it refers to MUST also represent
configuration. Such a leaf puts a constraint on valid data. All
such leaf nodes MUST reference existing nodes or leaf or leaf-list
nodes with their default value in use (see Sections 7.6.1 and 7.7.2)
for the data to be valid. This constraint is enforced according to
the rules in Section 8.

The "instance-identifier" XPath expression is conceptually evaluated
in the following context, in addition to the definition in
Section 6.4.1:

o The context node is the root node in the accessible tree.

9.13.1. Restrictions

An instance-identifier can be restricted with the "require-instance"
statement (Section 9.9.3).

9.13.2. Lexical Representation

An instance-identifier value is lexically represented as a string.
All node names in an instance-identifier value MUST be qualified with
explicit namespace prefixes, and these prefixes MUST be declared in
the XML namespace scope in the instance-identifier's XML element.

Any prefixes used in the encoding are local to each instance
encoding. This means that the same instance-identifier may be
encoded differently by different implementations.

9.13.3. Canonical Form

Since the lexical form depends on the XML context in which the value
occurs, this type does not have a canonical form.

9.13.4. Usage Example

The following are examples of instance identifiers:

```
/* instance-identifier for a container */
/ex:system/ex:services/ex:ssh

/* instance-identifier for a leaf */
/ex:system/ex:services/ex:ssh/ex:port

/* instance-identifier for a list entry */
/ex:system/ex:user[ex:name='fred']

/* instance-identifier for a leaf in a list entry */
/ex:system/ex:user[ex:name='fred']/ex:type

/* instance-identifier for a list entry with two keys */
/ex:system/ex:server[ex:ip='192.0.2.1'][ex:port='80']

/* instance-identifier for a list entry where the second
   key ("enabled") is of type "empty" */
/ex:system/ex:service[ex:name='foo'][ex:enabled='']

/* instance-identifier for a leaf-list entry */
/ex:system/ex:services/ex:ssh/ex:cipher[.='blowfish-cbc']

/* instance-identifier for a list entry without keys */
/ex:stats/ex:port[3]
```

10. XPath Functions

This document defines two generic XPath functions and five YANG
type-specific XPath functions. The function signatures are specified
with the syntax used in [XPATH].

10.1. Function for Node Sets

10.1.1. current()

```
node-set current()
```

The current() function takes no input parameters and returns a node
set with the initial context node as its only member.

10.1.1.1. Usage Example

With this list:

```
list interface {
  key "name";
  ...
  leaf enabled {
    type boolean;
  }
  ...
}
```

the following leaf defines a "must" expression that ensures that the
referred interface is enabled:

```
leaf outgoing-interface {
  type leafref {
    path "/interface/name";
  }
  must '/interface[name=current()]/enabled = "true"';
}
```

10.2. Function for Strings

10.2.1. re-match()

```
boolean re-match(string subject, string pattern)
```

The re-match() function returns "true" if the "subject" string
matches the regular expression "pattern"; otherwise, it returns
"false".

The re-match() function checks to see if a string matches a given
regular expression. The regular expressions used are the XML Schema
regular expressions [XSD-TYPES]. Note that this includes implicit
anchoring of the regular expression at the head and tail.

10.2.1.1. Usage Example

The expression:

   re-match("1.22.333", "\d{1,3}\.\d{1,3}\.\d{1,3}")

returns "true".

To count all logical interfaces called eth0.<number>, do:

   count(/interface[re-match(name, "eth0\.\d+")])

10.3. Function for the YANG Types "leafref" and "instance-identifier"

10.3.1. deref()

   node-set deref(node-set nodes)

The deref() function follows the reference defined by the first node
in document order in the argument "nodes" and returns the nodes it
refers to.

If the first argument node is of type "instance-identifier", the
function returns a node set that contains the single node that the
instance identifier refers to, if it exists. If no such node exists,
an empty node set is returned.

If the first argument node is of type "leafref", the function returns
a node set that contains the nodes that the leafref refers to.
Specifically, this set contains the nodes selected by the leafref's
"path" statement (Section 9.9.2) that have the same value as the
first argument node.

If the first argument node is of any other type, an empty node set is
returned.

10.3.1.1. Usage Example

```
    list interface {
      key "name type";
      leaf name { ... }
      leaf type { ... }
      leaf enabled {
        type boolean;
      }
      ...
    }

    container mgmt-interface {
      leaf name {
        type leafref {
          path "/interface/name";
        }
      }
      leaf type {
        type leafref {
          path "/interface[name=current()/../name]/type";
        }
        must 'deref(.)/../enabled = "true"' {
          error-message
            "The management interface cannot be disabled.";
        }
      }
    }
```

10.4. Functions for the YANG Type "identityref"

10.4.1. derived-from()

       boolean derived-from(node-set nodes, string identity)

   The derived-from() function returns "true" if any node in the
   argument "nodes" is a node of type "identityref" and its value is an
   identity that is derived from (see Section 7.18.2) the identity
   "identity"; otherwise, it returns "false".

   The parameter "identity" is a string matching the rule
   "identifier-ref" in Section 14. If a prefix is present on the
   identity, it refers to an identity defined in the module that was
   imported with that prefix, or the local module if the prefix matches
   the local module's prefix. If no prefix is present, the identity
   refers to an identity defined in the current module or an included
   submodule.

10.4.1.1. Usage Example

```
    module example-interface {
      yang-version 1.1;

      ...
      identity interface-type;

      identity ethernet {
        base interface-type;
      }

      identity fast-ethernet {
        base ethernet;
      }

      identity gigabit-ethernet {
        base ethernet;
      }

      list interface {
        key name;
        ...
        leaf type {
          type identityref {
            base interface-type;
          }
        }
        ...
      }

      augment "/interface" {
        when 'derived-from(type, "exif:ethernet")';
        // generic Ethernet definitions here
      }
      ...
    }
```

10.4.2. derived-from-or-self()

   boolean derived-from-or-self(node-set nodes, string identity)

   The derived-from-or-self() function returns "true" if any node in the
   argument "nodes" is a node of type "identityref" and its value is an
   identity that is equal to or derived from (see Section 7.18.2) the
   identity "identity"; otherwise, it returns "false".

   The parameter "identity" is a string matching the rule
   "identifier-ref" in Section 14. If a prefix is present on the
   identity, it refers to an identity defined in the module that was
   imported with that prefix, or the local module if the prefix matches
   the local module's prefix. If no prefix is present, the identity
   refers to an identity defined in the current module or an included
   submodule.

10.4.2.1. Usage Example

   The module defined in Section 10.4.1.1 might also have:

       augment "/interface" {
         when 'derived-from-or-self(type, "exif:fast-ethernet");
         // Fast-Ethernet-specific definitions here
       }

10.5. Function for the YANG Type "enumeration"

10.5.1. enum-value()

   number enum-value(node-set nodes)

   The enum-value() function checks to see if the first node in document
   order in the argument "nodes" is a node of type "enumeration" and
   returns the enum's integer value. If the "nodes" node set is empty
   or if the first node in "nodes" is not of type "enumeration", it
   returns NaN (not a number).

10.5.1.1. Usage Example

With this data model:

```
list alarm {
  ...
  leaf severity {
    type enumeration {
      enum cleared {
        value 1;
      }
      enum indeterminate {
        value 2;
      }
      enum minor {
        value 3;
      }
      enum warning {
        value 4;
      }
      enum major {
        value 5;
      }
      enum critical {
        value 6;
      }
    }
  }
}
```

the following XPath expression selects only alarms that are of
severity "major" or higher:

```
/alarm[enum-value(severity) >= 5]
```

10.6. Function for the YANG Type "bits"

10.6.1. bit-is-set()

```
boolean bit-is-set(node-set nodes, string bit-name)
```

The bit-is-set() function returns "true" if the first node in
document order in the argument "nodes" is a node of type "bits" and
its value has the bit "bit-name" set; otherwise, it returns "false".

10.6.1.1. Usage Example

   If an interface has this leaf:

      leaf flags {
        type bits {
          bit UP;
          bit PROMISCUOUS
          bit DISABLED;
        }
       }

   the following XPath expression can be used to select all interfaces
   with the UP flag set:

      /interface[bit-is-set(flags, "UP")]

11. Updating a Module

   As experience is gained with a module, it may be desirable to revise
   that module. However, changes to published modules are not allowed
   if they have any potential to cause interoperability problems between
   a client using an original specification and a server using an
   updated specification.

   For any published change, a new "revision" statement (Section 7.1.9)
   MUST be included in front of the existing "revision" statements. If
   there are no existing "revision" statements, then one MUST be added
   to identify the new revision. Furthermore, any necessary changes
   MUST be applied to any metadata statements, including the
   "organization" and "contact" statements (Sections 7.1.7 and 7.1.8).

   Note that definitions contained in a module are available to be
   imported by any other module and are referenced in "import"
   statements via the module name. Thus, a module name MUST NOT be
   changed. Furthermore, the "namespace" statement MUST NOT be changed,
   since all XML elements are qualified by the namespace.

   Obsolete definitions MUST NOT be removed from published modules,
   since their identifiers may still be referenced by other modules.

A definition in a published module may be revised in any of the
following ways:

o An "enumeration" type may have new enums added, provided the old
   enums's values do not change. Note that inserting a new enum
   before an existing enum or reordering existing enums will result
   in new values for the existing enums, unless they have explicit
   values assigned to them.

o A "bits" type may have new bits added, provided the old bit
   positions do not change. Note that inserting a new bit before an
   existing bit or reordering existing bits will result in new
   positions for the existing bits, unless they have explicit
   positions assigned to them.

o A "range", "length", or "pattern" statement may expand the allowed
   value space.

o A "default" statement may be added to a leaf that does not have a
   default value (either directly or indirectly through its type).

o A "units" statement may be added.

o A "reference" statement may be added or updated.

o A "must" statement may be removed or its constraint relaxed.

o A "when" statement may be removed or its constraint relaxed.

o A "mandatory" statement may be removed or changed from "true" to
   "false".

o A "min-elements" statement may be removed, or changed to require
   fewer elements.

o A "max-elements" statement may be removed, or changed to allow
   more elements.

o A "description" statement may be added or changed without changing
   the semantics of the definition.

o A "base" statement may be added to an "identity" statement.

o A "base" statement may be removed from an "identityref" type,
   provided there is at least one "base" statement left.

o New typedefs, groupings, rpcs, notifications, extensions,
   features, and identities may be added.

o New data definition statements may be added if they do not add
   mandatory nodes (Section 3) to existing nodes or at the top level
   in a module or submodule, or if they are conditionally dependent
   on a new feature (i.e., have an "if-feature" statement that refers
   to a new feature).

o A new "case" statement may be added.

o A node that represented state data may be changed to represent
   configuration, provided it is not mandatory (Section 3).

o An "if-feature" statement may be removed, provided its node is not
   mandatory (Section 3).

o A "status" statement may be added, or changed from "current" to
   "deprecated" or "obsolete", or changed from "deprecated" to
   "obsolete".

o A "type" statement may be replaced with another "type" statement
   that does not change the syntax or semantics of the type. For
   example, an inline type definition may be replaced with a typedef,
   but an int8 type cannot be replaced by an int16, since the syntax
   would change.

o Any set of data definition nodes may be replaced with another set
   of syntactically and semantically equivalent nodes. For example,
   a set of leafs may be replaced by a "uses" statement of a grouping
   with the same leafs.

o A module may be split into a set of submodules or a submodule may
   be removed, provided the definitions in the module do not change
   in any way other than those allowed here.

o The "prefix" statement may be changed, provided all local uses of
   the prefix are also changed.

Otherwise, if the semantics of any previous definition are changed
(i.e., if a non-editorial change is made to any definition other than
those specifically allowed above), then this MUST be achieved by a
new definition with a new identifier.

In statements that have any data definition statements as
substatements, those data definition substatements MUST NOT be
reordered. If new data definition statements are added, they can be
added anywhere in the sequence of existing substatements.

12. Coexistence with YANG Version 1

A YANG version 1.1 module MUST NOT include a YANG version 1
submodule, and a YANG version 1 module MUST NOT include a YANG
version 1.1 submodule.

A YANG version 1 module or submodule MUST NOT import a YANG
version 1.1 module by revision.

A YANG version 1.1 module or submodule MAY import a YANG version 1
module by revision.

If a YANG version 1 module A imports module B without revision and
module B is updated to YANG version 1.1, a server MAY implement both
of these modules (A and B) at the same time. In such cases, a
NETCONF server MUST advertise both modules using the rules defined in
Section 5.6.4, and SHOULD advertise module A and the latest revision
of module B that is specified with YANG version 1 according to the
rules defined in [RFC6020].

This rule exists in order to allow implementations of existing YANG
version 1 modules together with YANG version 1.1 modules. Without
this rule, updating a single module to YANG version 1.1 would have a
cascading effect on modules that import it, requiring all of them to
also be updated to YANG version 1.1, and so on.

13. YIN

A YANG module can be translated into an alternative XML-based syntax
called YIN. The translated module is called a YIN module. This
section describes bidirectional mapping rules between the two
formats.

The YANG and YIN formats contain equivalent information using
different notations. The YIN notation enables developers to
represent YANG data models in XML and therefore use the rich set of
XML-based tools for data filtering and validation, automated
generation of code and documentation, and other tasks. Tools like
XSLT or XML validators can be utilized.

The mapping between YANG and YIN does not modify the information
content of the model. Comments and whitespace are not preserved.

13.1. Formal YIN Definition

There is a one-to-one correspondence between YANG keywords and YIN
elements. The local name of a YIN element is identical to the
corresponding YANG keyword. This means, in particular, that the
document element (root) of a YIN document is always <module> or
<submodule>.

YIN elements corresponding to the YANG keywords belong to the
namespace whose associated URI is
"urn:ietf:params:xml:ns:yang:yin:1".

YIN elements corresponding to extension keywords belong to the
namespace of the YANG module where the extension keyword is declared
via the "extension" statement.

The names of all YIN elements MUST be properly qualified with their
namespaces (as specified above) using the standard mechanisms of
[XML-NAMES], i.e., "xmlns" and "xmlns:xxx" attributes.

The argument of a YANG statement is represented in YIN as either an
XML attribute or a subelement of the keyword element. Table 1
defines the mapping for the set of YANG keywords. For extensions,
the argument mapping is specified within the "extension" statement
(see Section 7.19). The following rules hold for arguments:

o If the argument is represented as an attribute, this attribute has
   no namespace.

o If the argument is represented as an element, it is qualified by
   the same namespace as its parent keyword element.

o If the argument is represented as an element, it MUST be the first
   child of the keyword element.

Substatements of a YANG statement are represented as (additional)
children of the keyword element, and their relative order MUST be the
same as the order of substatements in YANG.

Comments in YANG MAY be mapped to XML comments.

keyword	argument name	yin-element
action	name	false
anydata	name	false
anyxml	name	false
argument	name	false
augment	target-node	false
base	name	false
belongs-to	module	false
bit	name	false
case	name	false
choice	name	false
config	value	false
contact	text	true
container	name	false
default	value	false
description	text	true
deviate	value	false
deviation	target-node	false
enum	name	false
error-app-tag	value	false
error-message	value	true
extension	name	false
feature	name	false
fraction-digits	value	false
grouping	name	false
identity	name	false
if-feature	name	false
import	module	false
include	module	false
input	<no argument>	n/a
key	value	false
leaf	name	false
leaf-list	name	false
length	value	false
list	name	false
mandatory	value	false
max-elements	value	false
min-elements	value	false
modifier	value	false
module	name	false
must	condition	false
namespace	uri	false
notification	name	false
ordered-by	value	false
organization	text	true
output	<no argument>	n/a

```
        │ path              │ value          │ false       │
        │ pattern           │ value          │ false       │
        │ position          │ value          │ false       │
        │ prefix            │ value          │ false       │
        │ presence          │ value          │ false       │
        │ range             │ value          │ false       │
        │ reference         │ text           │ true        │
        │ refine            │ target-node    │ false       │
        │ require-instance  │ value          │ false       │
        │ revision          │ date           │ false       │
        │ revision-date     │ date           │ false       │
        │ rpc               │ name           │ false       │
        │ status            │ value          │ false       │
        │ submodule         │ name           │ false       │
        │ type              │ name           │ false       │
        │ typedef           │ name           │ false       │
        │ unique            │ tag            │ false       │
        │ units             │ name           │ false       │
        │ uses              │ name           │ false       │
        │ value             │ value          │ false       │
        │ when              │ condition      │ false       │
        │ yang-version      │ value          │ false       │
        │ yin-element       │ value          │ false       │
        +-------------------+----------------+-------------+
```

          Table 1: Mapping of Arguments of the YANG Statements

13.1.1. Usage Example

   The following YANG module:

```
     module example-foo {
       yang-version 1.1;
       namespace "urn:example:foo";
       prefix "foo";

       import example-extensions {
         prefix "myext";
       }

       list interface {
         key "name";
         leaf name {
           type string;
         }
```

```
    leaf mtu {
      type uint32;
      description "The MTU of the interface.";
      myext:c-define "MY_MTU";
    }
  }
}
```

where the extension "c-define" is defined in Section 7.19.3, is
translated into the following YIN:

```
<module name="example-foo"
        xmlns="urn:ietf:params:xml:ns:yang:yin:1"
        xmlns:foo="urn:example:foo"
        xmlns:myext="urn:example:extensions">

  <namespace uri="urn:example:foo"/>
  <prefix value="foo"/>

  <import module="example-extensions">
    <prefix value="myext"/>
  </import>

  <list name="interface">
    <key value="name"/>
    <leaf name="name">
      <type name="string"/>
    </leaf>
    <leaf name="mtu">
      <type name="uint32"/>
      <description>
        <text>The MTU of the interface.</text>
      </description>
      <myext:c-define name="MY_MTU"/>
    </leaf>
  </list>
</module>
```

14. YANG ABNF Grammar

   In YANG, almost all statements are unordered. The ABNF grammar
   [RFC5234] [RFC7405] defines the canonical order. To improve module
   readability, it is RECOMMENDED that clauses be entered in this order.

   Within the ABNF grammar, unordered statements are marked with
   comments.

   This grammar assumes that the scanner replaces YANG comments with a
   single space character.

   <CODE BEGINS> file "yang.abnf"

```
module-stmt          = optsep module-keyword sep identifier-arg-str
                         optsep
                         "{" stmtsep
                             module-header-stmts
                             linkage-stmts
                             meta-stmts
                             revision-stmts
                             body-stmts
                         "}" optsep

submodule-stmt       = optsep submodule-keyword sep identifier-arg-str
                         optsep
                         "{" stmtsep
                             submodule-header-stmts
                             linkage-stmts
                             meta-stmts
                             revision-stmts
                             body-stmts
                         "}" optsep

module-header-stmts = ;; these stmts can appear in any order
                         yang-version-stmt
                         namespace-stmt
                         prefix-stmt

submodule-header-stmts =
                         ;; these stmts can appear in any order
                         yang-version-stmt
                         belongs-to-stmt
```

```
   meta-stmts          = ;; these stmts can appear in any order
                         [organization-stmt]
                         [contact-stmt]
                         [description-stmt]
                         [reference-stmt]

   linkage-stmts       = ;; these stmts can appear in any order
                         *import-stmt
                         *include-stmt

   revision-stmts      = *revision-stmt

   body-stmts          = *(extension-stmt /
                           feature-stmt /
                           identity-stmt /
                           typedef-stmt /
                           grouping-stmt /
                           data-def-stmt /
                           augment-stmt /
                           rpc-stmt /
                           notification-stmt /
                           deviation-stmt)

   data-def-stmt       = container-stmt /
                         leaf-stmt /
                         leaf-list-stmt /
                         list-stmt /
                         choice-stmt /
                         anydata-stmt /
                         anyxml-stmt /
                         uses-stmt

   yang-version-stmt   = yang-version-keyword sep yang-version-arg-str
                         stmtend

   yang-version-arg-str = < a string that matches the rule >
                           < yang-version-arg >

   yang-version-arg    = "1.1"

   import-stmt         = import-keyword sep identifier-arg-str optsep
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             prefix-stmt
                             [revision-date-stmt]
                             [description-stmt]
                             [reference-stmt]
                         "}" stmtsep
```

```
include-stmt         = include-keyword sep identifier-arg-str optsep
                       (";" /
                        "{" stmtsep
                            ;; these stmts can appear in any order
                            [revision-date-stmt]
                            [description-stmt]
                            [reference-stmt]
                         "}") stmtsep

namespace-stmt       = namespace-keyword sep uri-str stmtend

uri-str              = < a string that matches the rule >
                       < URI in RFC 3986 >

prefix-stmt          = prefix-keyword sep prefix-arg-str stmtend

belongs-to-stmt      = belongs-to-keyword sep identifier-arg-str
                       optsep
                       "{" stmtsep
                           prefix-stmt
                       "}" stmtsep

organization-stmt    = organization-keyword sep string stmtend

contact-stmt         = contact-keyword sep string stmtend

description-stmt     = description-keyword sep string stmtend

reference-stmt       = reference-keyword sep string stmtend

units-stmt           = units-keyword sep string stmtend

revision-stmt        = revision-keyword sep revision-date optsep
                       (";" /
                        "{" stmtsep
                            ;; these stmts can appear in any order
                            [description-stmt]
                            [reference-stmt]
                         "}") stmtsep

revision-date        = date-arg-str

revision-date-stmt = revision-date-keyword sep revision-date stmtend
```

```
extension-stmt        = extension-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            [argument-stmt]
                            [status-stmt]
                            [description-stmt]
                            [reference-stmt]
                         "}") stmtsep

argument-stmt         = argument-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                            [yin-element-stmt]
                         "}") stmtsep

yin-element-stmt      = yin-element-keyword sep yin-element-arg-str
                        stmtend

yin-element-arg-str   = < a string that matches the rule >
                        < yin-element-arg >

yin-element-arg       = true-keyword / false-keyword

identity-stmt         = identity-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            *if-feature-stmt
                            *base-stmt
                            [status-stmt]
                            [description-stmt]
                            [reference-stmt]
                         "}") stmtsep

base-stmt             = base-keyword sep identifier-ref-arg-str
                        stmtend

feature-stmt          = feature-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            *if-feature-stmt
                            [status-stmt]
                            [description-stmt]
                            [reference-stmt]
                         "}") stmtsep
```

```
if-feature-stmt       = if-feature-keyword sep if-feature-expr-str
                        stmtend

if-feature-expr-str = < a string that matches the rule >
                      < if-feature-expr >

if-feature-expr       = if-feature-term
                          [sep or-keyword sep if-feature-expr]

if-feature-term       = if-feature-factor
                          [sep and-keyword sep if-feature-term]

if-feature-factor     = not-keyword sep if-feature-factor /
                        "(" optsep if-feature-expr optsep ")" /
                        identifier-ref-arg

typedef-stmt          = typedef-keyword sep identifier-arg-str optsep
                        "{" stmtsep
                          ;; these stmts can appear in any order
                          type-stmt
                          [units-stmt]
                          [default-stmt]
                          [status-stmt]
                          [description-stmt]
                          [reference-stmt]
                        "}" stmtsep

type-stmt             = type-keyword sep identifier-ref-arg-str optsep
                        (";" /
                         "{" stmtsep
                            [type-body-stmts]
                         "}") stmtsep

type-body-stmts       = numerical-restrictions /
                        decimal64-specification /
                        string-restrictions /
                        enum-specification /
                        leafref-specification /
                        identityref-specification /
                        instance-identifier-specification /
                        bits-specification /
                        union-specification /
                        binary-specification
```

```
   numerical-restrictions = [range-stmt]

   range-stmt          = range-keyword sep range-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             [error-message-stmt]
                             [error-app-tag-stmt]
                             [description-stmt]
                             [reference-stmt]
                          "}") stmtsep

   decimal64-specification = ;; these stmts can appear in any order
                         fraction-digits-stmt
                         [range-stmt]

   fraction-digits-stmt = fraction-digits-keyword sep
                         fraction-digits-arg-str stmtend

   fraction-digits-arg-str = < a string that matches the rule >
                         < fraction-digits-arg >

   fraction-digits-arg = ("1" ["0" / "1" / "2" / "3" / "4" /
                             "5" / "6" / "7" / "8"])
                         / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"

   string-restrictions = ;; these stmts can appear in any order
                         [length-stmt]
                         *pattern-stmt

   length-stmt         = length-keyword sep length-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             [error-message-stmt]
                             [error-app-tag-stmt]
                             [description-stmt]
                             [reference-stmt]
                          "}") stmtsep
```

```
   pattern-stmt           = pattern-keyword sep string optsep
                            (";" /
                             "{" stmtsep
                                 ;; these stmts can appear in any order
                                 [modifier-stmt]
                                 [error-message-stmt]
                                 [error-app-tag-stmt]
                                 [description-stmt]
                                 [reference-stmt]
                              "}") stmtsep

   modifier-stmt          = modifier-keyword sep modifier-arg-str stmtend

   modifier-arg-str       = < a string that matches the rule >
                            < modifier-arg >

   modifier-arg           = invert-match-keyword

   default-stmt           = default-keyword sep string stmtend

   enum-specification     = 1*enum-stmt

   enum-stmt              = enum-keyword sep string optsep
                            (";" /
                             "{" stmtsep
                                 ;; these stmts can appear in any order
                                 *if-feature-stmt
                                 [value-stmt]
                                 [status-stmt]
                                 [description-stmt]
                                 [reference-stmt]
                              "}") stmtsep

   leafref-specification =
                            ;; these stmts can appear in any order
                            path-stmt
                            [require-instance-stmt]

   path-stmt              = path-keyword sep path-arg-str stmtend

   require-instance-stmt = require-instance-keyword sep
                              require-instance-arg-str stmtend

   require-instance-arg-str = < a string that matches the rule >
                                < require-instance-arg >

   require-instance-arg = true-keyword / false-keyword
```

```
   instance-identifier-specification =
                    [require-instance-stmt]

   identityref-specification =
                    1*base-stmt

   union-specification = 1*type-stmt

   binary-specification = [length-stmt]

   bits-specification  = 1*bit-stmt

   bit-stmt            = bit-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              *if-feature-stmt
                              [position-stmt]
                              [status-stmt]
                              [description-stmt]
                              [reference-stmt]
                           "}") stmtsep

   position-stmt       = position-keyword sep
                         position-value-arg-str stmtend

   position-value-arg-str = < a string that matches the rule >
                            < position-value-arg >

   position-value-arg  = non-negative-integer-value

   status-stmt         = status-keyword sep status-arg-str stmtend

   status-arg-str      = < a string that matches the rule >
                         < status-arg >

   status-arg          = current-keyword /
                         obsolete-keyword /
                         deprecated-keyword

   config-stmt         = config-keyword sep
                         config-arg-str stmtend

   config-arg-str      = < a string that matches the rule >
                         < config-arg >

   config-arg          = true-keyword / false-keyword
```

```
   mandatory-stmt       = mandatory-keyword sep
                          mandatory-arg-str stmtend

   mandatory-arg-str    = < a string that matches the rule >
                          < mandatory-arg >

   mandatory-arg        = true-keyword / false-keyword

   presence-stmt        = presence-keyword sep string stmtend

   ordered-by-stmt      = ordered-by-keyword sep
                          ordered-by-arg-str stmtend

   ordered-by-arg-str   = < a string that matches the rule >
                          < ordered-by-arg >

   ordered-by-arg       = user-keyword / system-keyword

   must-stmt            = must-keyword sep string optsep
                          (";" /
                           "{" stmtsep
                               ;; these stmts can appear in any order
                               [error-message-stmt]
                               [error-app-tag-stmt]
                               [description-stmt]
                               [reference-stmt]
                           "}") stmtsep

   error-message-stmt   = error-message-keyword sep string stmtend

   error-app-tag-stmt   = error-app-tag-keyword sep string stmtend

   min-elements-stmt    = min-elements-keyword sep
                          min-value-arg-str stmtend

   min-value-arg-str    = < a string that matches the rule >
                          < min-value-arg >

   min-value-arg        = non-negative-integer-value

   max-elements-stmt    = max-elements-keyword sep
                          max-value-arg-str stmtend

   max-value-arg-str    = < a string that matches the rule >
                          < max-value-arg >

   max-value-arg        = unbounded-keyword /
                          positive-integer-value
```

```
   value-stmt          = value-keyword sep integer-value-str stmtend

   integer-value-str   = < a string that matches the rule >
                         < integer-value >

   grouping-stmt       = grouping-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [status-stmt]
                              [description-stmt]
                              [reference-stmt]
                              *(typedef-stmt / grouping-stmt)
                              *data-def-stmt
                              *action-stmt
                              *notification-stmt
                          "}") stmtsep

   container-stmt      = container-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [when-stmt]
                              *if-feature-stmt
                              *must-stmt
                              [presence-stmt]
                              [config-stmt]
                              [status-stmt]
                              [description-stmt]
                              [reference-stmt]
                              *(typedef-stmt / grouping-stmt)
                              *data-def-stmt
                              *action-stmt
                              *notification-stmt
                          "}") stmtsep
```

```
leaf-stmt              = leaf-keyword sep identifier-arg-str optsep
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             [when-stmt]
                             *if-feature-stmt
                             type-stmt
                             [units-stmt]
                             *must-stmt
                             [default-stmt]
                             [config-stmt]
                             [mandatory-stmt]
                             [status-stmt]
                             [description-stmt]
                             [reference-stmt]
                          "}" stmtsep

leaf-list-stmt         = leaf-list-keyword sep identifier-arg-str optsep
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             [when-stmt]
                             *if-feature-stmt
                             type-stmt stmtsep
                             [units-stmt]
                             *must-stmt
                             *default-stmt
                             [config-stmt]
                             [min-elements-stmt]
                             [max-elements-stmt]
                             [ordered-by-stmt]
                             [status-stmt]
                             [description-stmt]
                             [reference-stmt]
                          "}" stmtsep
```

```
list-stmt              = list-keyword sep identifier-arg-str optsep
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             [when-stmt]
                             *if-feature-stmt
                             *must-stmt
                             [key-stmt]
                             *unique-stmt
                             [config-stmt]
                             [min-elements-stmt]
                             [max-elements-stmt]
                             [ordered-by-stmt]
                             [status-stmt]
                             [description-stmt]
                             [reference-stmt]
                             *(typedef-stmt / grouping-stmt)
                             1*data-def-stmt
                             *action-stmt
                             *notification-stmt
                          "}" stmtsep

key-stmt               = key-keyword sep key-arg-str stmtend

key-arg-str            = < a string that matches the rule >
                         < key-arg >

key-arg                = node-identifier *(sep node-identifier)

unique-stmt            = unique-keyword sep unique-arg-str stmtend

unique-arg-str         = < a string that matches the rule >
                         < unique-arg >

unique-arg             = descendant-schema-nodeid
                         *(sep descendant-schema-nodeid)
```

```
choice-stmt         = choice-keyword sep identifier-arg-str optsep
                      (";" /
                       "{" stmtsep
                          ;; these stmts can appear in any order
                          [when-stmt]
                          *if-feature-stmt
                          [default-stmt]
                          [config-stmt]
                          [mandatory-stmt]
                          [status-stmt]
                          [description-stmt]
                          [reference-stmt]
                          *(short-case-stmt / case-stmt)
                       "}") stmtsep

short-case-stmt     = choice-stmt /
                      container-stmt /
                      leaf-stmt /
                      leaf-list-stmt /
                      list-stmt /
                      anydata-stmt /
                      anyxml-stmt

case-stmt           = case-keyword sep identifier-arg-str optsep
                      (";" /
                       "{" stmtsep
                          ;; these stmts can appear in any order
                          [when-stmt]
                          *if-feature-stmt
                          [status-stmt]
                          [description-stmt]
                          [reference-stmt]
                          *data-def-stmt
                       "}") stmtsep
```

```
anydata-stmt          = anydata-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            [when-stmt]
                            *if-feature-stmt
                            *must-stmt
                            [config-stmt]
                            [mandatory-stmt]
                            [status-stmt]
                            [description-stmt]
                            [reference-stmt]
                          "}") stmtsep

anyxml-stmt           = anyxml-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            [when-stmt]
                            *if-feature-stmt
                            *must-stmt
                            [config-stmt]
                            [mandatory-stmt]
                            [status-stmt]
                            [description-stmt]
                            [reference-stmt]
                          "}") stmtsep

uses-stmt             = uses-keyword sep identifier-ref-arg-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            [when-stmt]
                            *if-feature-stmt
                            [status-stmt]
                            [description-stmt]
                            [reference-stmt]
                            *refine-stmt
                            *uses-augment-stmt
                          "}") stmtsep
```

```
   refine-stmt          = refine-keyword sep refine-arg-str optsep
                           "{" stmtsep
                               ;; these stmts can appear in any order
                               *if-feature-stmt
                               *must-stmt
                               [presence-stmt]
                               *default-stmt
                               [config-stmt]
                               [mandatory-stmt]
                               [min-elements-stmt]
                               [max-elements-stmt]
                               [description-stmt]
                               [reference-stmt]
                           "}" stmtsep

   refine-arg-str       = < a string that matches the rule >
                           < refine-arg >

   refine-arg           = descendant-schema-nodeid

   uses-augment-stmt    = augment-keyword sep uses-augment-arg-str optsep
                           "{" stmtsep
                               ;; these stmts can appear in any order
                               [when-stmt]
                               *if-feature-stmt
                               [status-stmt]
                               [description-stmt]
                               [reference-stmt]
                               1*(data-def-stmt / case-stmt /
                                  action-stmt / notification-stmt)
                           "}" stmtsep

   uses-augment-arg-str = < a string that matches the rule >
                           < uses-augment-arg >

   uses-augment-arg     = descendant-schema-nodeid
```

```
augment-stmt            = augment-keyword sep augment-arg-str optsep
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [when-stmt]
                              *if-feature-stmt
                              [status-stmt]
                              [description-stmt]
                              [reference-stmt]
                              1*(data-def-stmt / case-stmt /
                                 action-stmt / notification-stmt)
                           "}" stmtsep

augment-arg-str         = < a string that matches the rule >
                          < augment-arg >

augment-arg             = absolute-schema-nodeid

when-stmt               = when-keyword sep string optsep
                          (";" /
                           "{" stmtsep
                               ;; these stmts can appear in any order
                               [description-stmt]
                               [reference-stmt]
                            "}") stmtsep

rpc-stmt                = rpc-keyword sep identifier-arg-str optsep
                          (";" /
                           "{" stmtsep
                               ;; these stmts can appear in any order
                               *if-feature-stmt
                               [status-stmt]
                               [description-stmt]
                               [reference-stmt]
                               *(typedef-stmt / grouping-stmt)
                               [input-stmt]
                               [output-stmt]
                            "}") stmtsep
```

```
    action-stmt          = action-keyword sep identifier-arg-str optsep
                          (";" /
                           "{" stmtsep
                              ;; these stmts can appear in any order
                              *if-feature-stmt
                              [status-stmt]
                              [description-stmt]
                              [reference-stmt]
                              *(typedef-stmt / grouping-stmt)
                              [input-stmt]
                              [output-stmt]
                           "}") stmtsep

    input-stmt           = input-keyword optsep
                           "{" stmtsep
                              ;; these stmts can appear in any order
                              *must-stmt
                              *(typedef-stmt / grouping-stmt)
                              1*data-def-stmt
                           "}" stmtsep

    output-stmt          = output-keyword optsep
                           "{" stmtsep
                              ;; these stmts can appear in any order
                              *must-stmt
                              *(typedef-stmt / grouping-stmt)
                              1*data-def-stmt
                           "}" stmtsep

    notification-stmt    = notification-keyword sep
                           identifier-arg-str optsep
                          (";" /
                           "{" stmtsep
                              ;; these stmts can appear in any order
                              *if-feature-stmt
                              *must-stmt
                              [status-stmt]
                              [description-stmt]
                              [reference-stmt]
                              *(typedef-stmt / grouping-stmt)
                              *data-def-stmt
                           "}") stmtsep
```

```
deviation-stmt        = deviation-keyword sep
                        deviation-arg-str optsep
                        "{" stmtsep
                            ;; these stmts can appear in any order
                            [description-stmt]
                            [reference-stmt]
                            (deviate-not-supported-stmt /
                              1*(deviate-add-stmt /
                                 deviate-replace-stmt /
                                 deviate-delete-stmt))
                        "}" stmtsep

deviation-arg-str     = < a string that matches the rule >
                        < deviation-arg >

deviation-arg         = absolute-schema-nodeid

deviate-not-supported-stmt =
                        deviate-keyword sep
                        not-supported-keyword-str stmtend

deviate-add-stmt      = deviate-keyword sep add-keyword-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            [units-stmt]
                            *must-stmt
                            *unique-stmt
                            *default-stmt
                            [config-stmt]
                            [mandatory-stmt]
                            [min-elements-stmt]
                            [max-elements-stmt]
                          "}") stmtsep

deviate-delete-stmt = deviate-keyword sep delete-keyword-str optsep
                        (";" /
                         "{" stmtsep
                            ;; these stmts can appear in any order
                            [units-stmt]
                            *must-stmt
                            *unique-stmt
                            *default-stmt
                          "}") stmtsep
```

```
   deviate-replace-stmt = deviate-keyword sep replace-keyword-str optsep
                          (";" /
                           "{" stmtsep
                               ;; these stmts can appear in any order
                               [type-stmt]
                               [units-stmt]
                               [default-stmt]
                               [config-stmt]
                               [mandatory-stmt]
                               [min-elements-stmt]
                               [max-elements-stmt]
                            "}") stmtsep

   not-supported-keyword-str = < a string that matches the rule >
                               < not-supported-keyword >

   add-keyword-str     = < a string that matches the rule >
                         < add-keyword >

   delete-keyword-str  = < a string that matches the rule >
                         < delete-keyword >

   replace-keyword-str = < a string that matches the rule >
                         < replace-keyword >

   ;; represents the usage of an extension
   unknown-statement   = prefix ":" identifier [sep string] optsep
                         (";" /
                          "{" optsep
                             *((yang-stmt / unknown-statement) optsep)
                           "}") stmtsep

   yang-stmt           = action-stmt /
                         anydata-stmt /
                         anyxml-stmt /
                         argument-stmt /
                         augment-stmt /
                         base-stmt /
                         belongs-to-stmt /
                         bit-stmt /
                         case-stmt /
                         choice-stmt /
                         config-stmt /
                         contact-stmt /
                         container-stmt /
                         default-stmt /
                         description-stmt /
                         deviate-add-stmt /
```

```
                    deviate-delete-stmt /
                    deviate-not-supported-stmt /
                    deviate-replace-stmt /
                    deviation-stmt /
                    enum-stmt /
                    error-app-tag-stmt /
                    error-message-stmt /
                    extension-stmt /
                    feature-stmt /
                    fraction-digits-stmt /
                    grouping-stmt /
                    identity-stmt /
                    if-feature-stmt /
                    import-stmt /
                    include-stmt /
                    input-stmt /
                    key-stmt /
                    leaf-list-stmt /
                    leaf-stmt /
                    length-stmt /
                    list-stmt /
                    mandatory-stmt /
                    max-elements-stmt /
                    min-elements-stmt /
                    modifier-stmt /
                    module-stmt /
                    must-stmt /
                    namespace-stmt /
                    notification-stmt /
                    ordered-by-stmt /
                    organization-stmt /
                    output-stmt /
                    path-stmt /
                    pattern-stmt /
                    position-stmt /
                    prefix-stmt /
                    presence-stmt /
                    range-stmt /
                    reference-stmt /
                    refine-stmt /
                    require-instance-stmt /
                    revision-date-stmt /
                    revision-stmt /
                    rpc-stmt /
                    status-stmt /
                    submodule-stmt /
                    typedef-stmt /
                    type-stmt /
```

```
                        unique-stmt /
                        units-stmt /
                        uses-augment-stmt /
                        uses-stmt /
                        value-stmt /
                        when-stmt /
                        yang-version-stmt /
                        yin-element-stmt

   ;; Ranges

   range-arg-str       = < a string that matches the rule >
                         < range-arg >

   range-arg           = range-part *(optsep "|" optsep range-part)

   range-part          = range-boundary
                         [optsep ".." optsep range-boundary]

   range-boundary      = min-keyword / max-keyword /
                         integer-value / decimal-value

   ;; Lengths

   length-arg-str      = < a string that matches the rule >
                         < length-arg >

   length-arg          = length-part *(optsep "|" optsep length-part)

   length-part         = length-boundary
                         [optsep ".." optsep length-boundary]

   length-boundary     = min-keyword / max-keyword /
                         non-negative-integer-value

   ;; Date

   date-arg-str        = < a string that matches the rule >
                         < date-arg >

   date-arg            = 4DIGIT "-" 2DIGIT "-" 2DIGIT

   ;; Schema Node Identifiers

   schema-nodeid       = absolute-schema-nodeid /
                         descendant-schema-nodeid

   absolute-schema-nodeid = 1*("/" node-identifier)
```

```
descendant-schema-nodeid =
                        node-identifier
                        [absolute-schema-nodeid]

node-identifier    = [prefix ":"] identifier

;; Instance Identifiers

instance-identifier = 1*("/" (node-identifier
                       [1*key-predicate /
                        leaf-list-predicate /
                        pos]))

key-predicate      = "[" *WSP key-predicate-expr *WSP "]"

key-predicate-expr = node-identifier *WSP "=" *WSP quoted-string

leaf-list-predicate = "[" *WSP leaf-list-predicate-expr *WSP "]"

leaf-list-predicate-expr = "." *WSP "=" *WSP quoted-string

pos                = "[" *WSP positive-integer-value *WSP "]"

quoted-string      = (DQUOTE string DQUOTE) / (SQUOTE string SQUOTE)

;; leafref path

path-arg-str       = < a string that matches the rule >
                     < path-arg >

path-arg           = absolute-path / relative-path

absolute-path      = 1*("/" (node-identifier *path-predicate))

relative-path      = 1*("../") descendant-path

descendant-path    = node-identifier
                     [*path-predicate absolute-path]

path-predicate     = "[" *WSP path-equality-expr *WSP "]"

path-equality-expr = node-identifier *WSP "=" *WSP path-key-expr

path-key-expr      = current-function-invocation *WSP "/" *WSP
                     rel-path-keyexpr
```

```
rel-path-keyexpr    = 1*(".." *WSP "/" *WSP)
                      *(node-identifier *WSP "/" *WSP)
                      node-identifier

;;; Keywords, using the syntax for case-sensitive strings (RFC 7405)

;; statement keywords
action-keyword             = %s"action"
anydata-keyword            = %s"anydata"
anyxml-keyword             = %s"anyxml"
argument-keyword           = %s"argument"
augment-keyword            = %s"augment"
base-keyword               = %s"base"
belongs-to-keyword         = %s"belongs-to"
bit-keyword                = %s"bit"
case-keyword               = %s"case"
choice-keyword             = %s"choice"
config-keyword             = %s"config"
contact-keyword            = %s"contact"
container-keyword          = %s"container"
default-keyword            = %s"default"
description-keyword        = %s"description"
deviate-keyword            = %s"deviate"
deviation-keyword          = %s"deviation"
enum-keyword               = %s"enum"
error-app-tag-keyword      = %s"error-app-tag"
error-message-keyword      = %s"error-message"
extension-keyword          = %s"extension"
feature-keyword            = %s"feature"
fraction-digits-keyword    = %s"fraction-digits"
grouping-keyword           = %s"grouping"
identity-keyword           = %s"identity"
if-feature-keyword         = %s"if-feature"
import-keyword             = %s"import"
include-keyword            = %s"include"
input-keyword              = %s"input"
key-keyword                = %s"key"
leaf-keyword               = %s"leaf"
leaf-list-keyword          = %s"leaf-list"
length-keyword             = %s"length"
list-keyword               = %s"list"
mandatory-keyword          = %s"mandatory"
max-elements-keyword       = %s"max-elements"
min-elements-keyword       = %s"min-elements"
modifier-keyword           = %s"modifier"
module-keyword             = %s"module"
must-keyword               = %s"must"
namespace-keyword          = %s"namespace"
```

```
   notification-keyword     = %s"notification"
   ordered-by-keyword       = %s"ordered-by"
   organization-keyword     = %s"organization"
   output-keyword           = %s"output"
   path-keyword             = %s"path"
   pattern-keyword          = %s"pattern"
   position-keyword         = %s"position"
   prefix-keyword           = %s"prefix"
   presence-keyword         = %s"presence"
   range-keyword            = %s"range"
   reference-keyword        = %s"reference"
   refine-keyword           = %s"refine"
   require-instance-keyword = %s"require-instance"
   revision-keyword         = %s"revision"
   revision-date-keyword    = %s"revision-date"
   rpc-keyword              = %s"rpc"
   status-keyword           = %s"status"
   submodule-keyword        = %s"submodule"
   type-keyword             = %s"type"
   typedef-keyword          = %s"typedef"
   unique-keyword           = %s"unique"
   units-keyword            = %s"units"
   uses-keyword             = %s"uses"
   value-keyword            = %s"value"
   when-keyword             = %s"when"
   yang-version-keyword     = %s"yang-version"
   yin-element-keyword      = %s"yin-element"

   ;; other keywords

   add-keyword              = %s"add"
   current-keyword          = %s"current"
   delete-keyword           = %s"delete"
   deprecated-keyword       = %s"deprecated"
   false-keyword            = %s"false"
   invert-match-keyword     = %s"invert-match"
   max-keyword              = %s"max"
   min-keyword              = %s"min"
   not-supported-keyword    = %s"not-supported"
   obsolete-keyword         = %s"obsolete"
   replace-keyword          = %s"replace"
   system-keyword           = %s"system"
   true-keyword             = %s"true"
   unbounded-keyword        = %s"unbounded"
   user-keyword             = %s"user"
```

```
and-keyword             = %s"and"
or-keyword              = %s"or"
not-keyword             = %s"not"

current-function-invocation = current-keyword *WSP "(" *WSP ")"

;;; Basic Rules

prefix-arg-str       = < a string that matches the rule >
                         < prefix-arg >

prefix-arg           = prefix

prefix               = identifier

identifier-arg-str   = < a string that matches the rule >
                         < identifier-arg >

identifier-arg       = identifier

identifier           = (ALPHA / "_")
                         *(ALPHA / DIGIT / "_" / "-" / ".")

identifier-ref-arg-str = < a string that matches the rule >
                           < identifier-ref-arg >

identifier-ref-arg   = identifier-ref

identifier-ref       = [prefix ":"] identifier

string               = < an unquoted string, as returned by >
                         < the scanner, that matches the rule >
                         < yang-string >

yang-string          = *yang-char
```

```
   ;; any Unicode or ISO/IEC 10646 character, including tab, carriage
   ;; return, and line feed but excluding the other C0 control
   ;; characters, the surrogate blocks, and the noncharacters
yang-char = %x09 / %x0A / %x0D / %x20-D7FF /
                                     ; exclude surrogate blocks %xD800-DFFF
            %xE000-FDCF /     ; exclude noncharacters %xFDD0-FDEF
            %xFDF0-FFFD /     ; exclude noncharacters %xFFFE-FFFF
            %x10000-1FFFD /   ; exclude noncharacters %x1FFFE-1FFFF
            %x20000-2FFFD /   ; exclude noncharacters %x2FFFE-2FFFF
            %x30000-3FFFD /   ; exclude noncharacters %x3FFFE-3FFFF
            %x40000-4FFFD /   ; exclude noncharacters %x4FFFE-4FFFF
            %x50000-5FFFD /   ; exclude noncharacters %x5FFFE-5FFFF
            %x60000-6FFFD /   ; exclude noncharacters %x6FFFE-6FFFF
            %x70000-7FFFD /   ; exclude noncharacters %x7FFFE-7FFFF
            %x80000-8FFFD /   ; exclude noncharacters %x8FFFE-8FFFF
            %x90000-9FFFD /   ; exclude noncharacters %x9FFFE-9FFFF
            %xA0000-AFFFD /   ; exclude noncharacters %xAFFFE-AFFFF
            %xB0000-BFFFD /   ; exclude noncharacters %xBFFFE-BFFFF
            %xC0000-CFFFD /   ; exclude noncharacters %xCFFFE-CFFFF
            %xD0000-DFFFD /   ; exclude noncharacters %xDFFFE-DFFFF
            %xE0000-EFFFD /   ; exclude noncharacters %xEFFFE-EFFFF
            %xF0000-FFFFD /   ; exclude noncharacters %xFFFFE-FFFFF
            %x100000-10FFFD  ; exclude noncharacters %x10FFFE-10FFFF

   integer-value       = ("-" non-negative-integer-value) /
                         non-negative-integer-value

   non-negative-integer-value = "0" / positive-integer-value

   positive-integer-value = (non-zero-digit *DIGIT)

   zero-integer-value  = 1*DIGIT

   stmtend             = optsep (";" / "{" stmtsep "}") stmtsep

   sep                 = 1*(WSP / line-break)
                         ; unconditional separator

   optsep              = *(WSP / line-break)

   stmtsep             = *(WSP / line-break / unknown-statement)

   line-break          = CRLF / LF

   non-zero-digit      = %x31-39

   decimal-value       = integer-value ("." zero-integer-value)
```

```
SQUOTE              = %x27
                      ; single quote

;;; core rules from RFC 5234

ALPHA               = %x41-5A / %x61-7A
                      ; A-Z / a-z

CR                  = %x0D
                      ; carriage return

CRLF                = CR LF
                      ; Internet standard newline

DIGIT               = %x30-39
                      ; 0-9

DQUOTE              = %x22
                      ; double quote

HTAB                = %x09
                      ; horizontal tab

LF                  = %x0A
                      ; line feed

SP                  = %x20
                      ; space

WSP                 = SP / HTAB
                      ; whitespace

<CODE ENDS>
```

15. NETCONF Error Responses for YANG-Related Errors

A number of NETCONF error responses are defined for error cases related to the data model handling. If the relevant YANG statement has an "error-app-tag" substatement, that overrides the default value specified below.

15.1. Error Message for Data That Violates a "unique" Statement

If a NETCONF operation would result in configuration data where a "unique" constraint is invalidated, the following error MUST be returned:

```
error-tag:       operation-failed
error-app-tag:   data-not-unique
error-info:      <non-unique>: Contains an instance identifier that
                 points to a leaf that invalidates the "unique"
                 constraint.  This element is present once for each
                 non-unique leaf.

                 The <non-unique> element is in the YANG
                 namespace ("urn:ietf:params:xml:ns:yang:1").
```

15.2. Error Message for Data That Violates a "max-elements" Statement

If a NETCONF operation would result in configuration data where a list or a leaf-list would have too many entries, the following error MUST be returned:

```
error-tag:       operation-failed
error-app-tag:   too-many-elements
```

This error is returned once, with the error-path identifying the list node, even if there is more than one extra child present.

15.3. Error Message for Data That Violates a "min-elements" Statement

If a NETCONF operation would result in configuration data where a list or a leaf-list would have too few entries, the following error MUST be returned:

```
error-tag:       operation-failed
error-app-tag:   too-few-elements
```

This error is returned once, with the error-path identifying the list node, even if there is more than one child missing.

15.4. Error Message for Data That Violates a "must" Statement

If a NETCONF operation would result in configuration data where the
restrictions imposed by a "must" statement are violated, the
following error MUST be returned, unless a specific "error-app-tag"
substatement is present for the "must" statement.

```
error-tag:       operation-failed
error-app-tag:  must-violation
```

15.5. Error Message for Data That Violates a "require-instance" Statement

If a NETCONF operation would result in configuration data where a
leaf of type "instance-identifier" or "leafref" marked with
require-instance "true" refers to an instance that does not exist,
the following error MUST be returned:

```
error-tag:       data-missing
error-app-tag:  instance-required
error-path:     Path to the instance-identifier or leafref leaf.
```

15.6. Error Message for Data That Violates a Mandatory "choice" Statement

If a NETCONF operation would result in configuration data where no
nodes exists in a mandatory choice, the following error MUST be
returned:

```
error-tag:       data-missing
error-app-tag:  missing-choice
error-path:     Path to the element with the missing choice.
error-info:     <missing-choice>: Contains the name of the missing
                mandatory choice.

                The <missing-choice> element is in the YANG
                namespace ("urn:ietf:params:xml:ns:yang:1").
```

15.7. Error Message for the "insert" Operation

If the "insert" and "key" or "value" attributes are used in an
<edit-config> for a list or leaf-list node and the "key" or "value"
refers to an instance that does not exist, the following error MUST
be returned:

```
error-tag:       bad-attribute
error-app-tag:  missing-instance
```

16. IANA Considerations

   This document registers one capability identifier URN from the
   "Network Configuration Protocol (NETCONF) Capability URNs" registry:

      Index   Capability Identifier
      -------------   --
      :yang-library   urn:ietf:params:netconf:capability:yang-library:1.0

17. Security Considerations

   This document defines a language with which to write and read
   descriptions of management information. The language itself has no
   security impact on the Internet.

   The same considerations are relevant as those for the base NETCONF
   protocol (see Section 9 in [RFC6241]).

   Data modeled in YANG might contain sensitive information. RPCs or
   notifications defined in YANG might transfer sensitive information.

   Security issues are related to the usage of data modeled in YANG.
   Such issues shall be dealt with in documents describing the data
   models and documents about the interfaces used to manipulate the
   data, e.g., the NETCONF documents.

   Data modeled in YANG is dependent upon:

   o the security of the transmission infrastructure used to send
      sensitive information.

   o the security of applications that store or release such sensitive
      information.

   o adequate authentication and access control mechanisms to restrict
      the usage of sensitive data.

   YANG parsers need to be robust with respect to malformed documents.
   Reading malformed documents from unknown or untrusted sources could
   result in an attacker gaining the privileges of the user running the
   YANG parser. In an extreme situation, the entire machine could be
   compromised.

18. References

18.1. Normative References

[ISO.10646]
          International Organization for Standardization,
          "Information Technology - Universal Multiple-Octet Coded
          Character Set (UCS)", ISO Standard 10646:2014, 2014.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of
          ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629,
          November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
          Resource Identifier (URI): Generic Syntax", STD 66,
          RFC 3986, DOI 10.17487/RFC3986, January 2005,
          <http://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data
          Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006,
          <http://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed., and P. Overell, "Augmented BNF for
          Syntax Specifications: ABNF", STD 68, RFC 5234,
          DOI 10.17487/RFC5234, January 2008,
          <http://www.rfc-editor.org/info/rfc5234>.

[RFC5277] Chisholm, S. and H. Trevino, "NETCONF Event
          Notifications", RFC 5277, DOI 10.17487/RFC5277, July 2008,
          <http://www.rfc-editor.org/info/rfc5277>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed.,
          and A. Bierman, Ed., "Network Configuration Protocol
          (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011,
          <http://www.rfc-editor.org/info/rfc6241>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF",
          RFC 7405, DOI 10.17487/RFC7405, December 2014,
          <http://www.rfc-editor.org/info/rfc7405>.

[RFC7895] Bierman, A., Bjorklund, M., and K. Watsen, "YANG Module
          Library", RFC 7895, DOI 10.17487/RFC7895, June 2016,
          <http://www.rfc-editor.org/info/rfc7895>.

   [XML]   Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and
                F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth
                Edition)", W3C Recommendation REC-xml-20081126,
                November 2008,
                <https://www.w3.org/TR/2008/REC-xml-20081126/>.

   [XML-NAMES]
                Bray, T., Hollander, D., Layman, A., Tobin, R., and H.
                Thompson, "Namespaces in XML 1.0 (Third Edition)", World
                Wide Web Consortium Recommendation REC-xml-names-20091208,
                December 2009,
                <http://www.w3.org/TR/2009/REC-xml-names-20091208>.

   [XPATH]   Clark, J. and S. DeRose, "XML Path Language (XPath)
                Version 1.0", World Wide Web Consortium Recommendation
                REC-xpath-19991116, November 1999,
                <http://www.w3.org/TR/1999/REC-xpath-19991116>.

   [XSD-TYPES]
                Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes
                Second Edition", World Wide Web Consortium Recommendation
                REC-xmlschema-2-20041028, October 2004,
                <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028>.

18.2. Informative References

   [CoMI]   van der Stok, P. and A. Bierman, "CoAP Management
                Interface", Work in Progress,
                draft-vanderstok-core-comi-09, March 2016.

   [IEEE754-2008]
                IEEE, "IEEE Standard for Floating-Point Arithmetic",
                IEEE 754-2008, DOI 10.1109/IEEESTD.2008.4610935, 2008,
                <http://standards.ieee.org/findstds/
                standard/754-2008.html>.

   [RESTCONF]
                Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF
                Protocol", Work in Progress,
                draft-ietf-netconf-restconf-16, August 2016.

   [RFC2578]   McCloghrie, K., Ed., Perkins, D., Ed., and J.
                Schoenwaelder, Ed., "Structure of Management Information
                Version 2 (SMIv2)", STD 58, RFC 2578,
                DOI 10.17487/RFC2578, April 1999,
                <http://www.rfc-editor.org/info/rfc2578>.

[RFC2579]   McCloghrie, K., Ed., Perkins, D., Ed., and J.
            Schoenwaelder, Ed., "Textual Conventions for SMIv2",
            STD 58, RFC 2579, DOI 10.17487/RFC2579, April 1999,
            <http://www.rfc-editor.org/info/rfc2579>.

[RFC3780]   Strauss, F. and J. Schoenwaelder, "SMIng - Next Generation
            Structure of Management Information", RFC 3780,
            DOI 10.17487/RFC3780, May 2004,
            <http://www.rfc-editor.org/info/rfc3780>.

[RFC4844]   Daigle, L., Ed., and Internet Architecture Board, "The RFC
            Series and RFC Editor", RFC 4844, DOI 10.17487/RFC4844,
            July 2007, <http://www.rfc-editor.org/info/rfc4844>.

[RFC6020]   Bjorklund, M., Ed., "YANG - A Data Modeling Language for
            the Network Configuration Protocol (NETCONF)", RFC 6020,
            DOI 10.17487/RFC6020, October 2010,
            <http://www.rfc-editor.org/info/rfc6020>.

[RFC6643]   Schoenwaelder, J., "Translation of Structure of Management
            Information Version 2 (SMIv2) MIB Modules to YANG
            Modules", RFC 6643, DOI 10.17487/RFC6643, July 2012,
            <http://www.rfc-editor.org/info/rfc6643>.

[RFC6991]   Schoenwaelder, J., Ed., "Common YANG Data Types",
            RFC 6991, DOI 10.17487/RFC6991, July 2013,
            <http://www.rfc-editor.org/info/rfc6991>.

[RFC7951]   Lhotka, L., "JSON Encoding of Data Modeled with YANG",
            RFC 7951, DOI 10.17487/RFC7951, August 2016,
            <http://www.rfc-editor.org/info/rfc7951>.

[XPATH2.0]
            Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.,
            Kay, M., Robie, J., and J. Simeon, "XML Path Language
            (XPath) 2.0 (Second Edition)", World Wide Web Consortium
            Recommendation REC-xpath20-20101214, December 2010,
            <http://www.w3.org/TR/2010/REC-xpath20-20101214>.

[XSLT]   Clark, J., "XSL Transformations (XSLT) Version 1.0", World
            Wide Web Consortium Recommendation REC-xslt-19991116,
            November 1999,
            <http://www.w3.org/TR/1999/REC-xslt-19991116>.

[YANG-Guidelines]
            Bierman, A., "Guidelines for Authors and Reviewers of YANG
            Data Model Documents", Work in Progress,
            draft-ietf-netmod-rfc6087bis-07, July 2016.

Author's Address

   Martin Bjorklund (editor)
   Tail-f Systems

   Email: mbj@tail-f.com