Network Working Group M. Bjorklund, Ed.
Internet-Draft Tail-f Systems
Intended status: Standards Track August 29, 2008
Expires: March 2, 2009

YANG - A data modeling language for NETCONF
draft-ietf-netmod-yang-01

Status of this Memo

Copyright Notice

Abstract

YANG is a data modeling language used to model configuration and state data manipulated by the NETCONF protocol, NETCONF remote procedure calls, and NETCONF notifications.

Table of Contents

1. Introduction

Today, the NETCONF protocol [RFC4741] lacks a standardized way to create data models. Instead, vendors are forced to use proprietary solutions. In order for NETCONF to be a interoperable protocol, models must be defined in a vendor-neutral way. YANG provides the language and rules for defining such models for use with NETCONF.

YANG is a data modeling language used to model configuration and state data manipulated by the NETCONF protocol, NETCONF remote procedure calls, and NETCONF notifications. This document describes the syntax and semantics of the YANG language, how the data model defined in a YANG module is represented in XML, and how NETCONF operations are used to manipulate the data.

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

- anyxml: A node which can contain an unknown chunk of XML data.

- augment: Adds new nodes to a previously defined node.

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

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

- choice: A node where only one of a number of identified alternative values is valid.

- configuration data: The set of writable data that is required to transform a system from its initial default state into its current state [RFC4741].

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

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

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

- data model object: A definition within a module that represents a construct which can be accessed via a network management protocol. Also called an object.

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

- data tree: The instantiated tree of configuration and state data on a device.

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

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

- grouping: A reusable set of nodes, which may be used locally in the module, in modules which include it, and by other modules which import from it.

- identifier: Used to identify different kinds of YANG items by name.

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

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

- leaf: A node in the data tree with a value but no child nodes.

- 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.

- list: Interior nodes in the data tree which may exist in multiple instances. A list node has no value, but rather a set of child nodes.

- MIB: A Management Information Base, traditionally referring to a management information defined using SNMP's SMI.

- module: A YANG module defines a hierarchy of nodes which can be used for NETCONF-based operations. With its definitions and the definitions it imports or includes from elsewhere, a module is self-contained and "compilable".

- RPC: A Remote Procedure Call, as used within the NETCONF protocol.

- RPC method: A specific Remote Procedure Call, as used within the NETCONF protocol. Also called a protocol operation.

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

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

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

- state data: The additional data on a system that is not configuration data such as read-only status information and collected statistics [RFC4741].

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

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

3.1. Mandatory nodes

A mandatory node is one of:

- A leaf or choice 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, which has has at least one mandatory node as a child.

4. YANG Overview

4.1. Functional Overview

YANG is a language used to model data for the NETCONF protocol. A YANG module defines a hierarchy of nodes which can be used for NETCONF-based operations, including configuration, state data, remote procedure calls (RPCs), and notifications. This allows a complete description of all data sent between a NETCONF client and server.

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

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

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 host name.

YANG permits the definition of complex types using reusable grouping of nodes. The instantiation 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 or submodule and used in either that location or in another module or submodule that imports or includes it.

YANG organizational constructs include defining lists of nodes with the same names and identifying the keys which distinguish list members 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 nodes.

YANG modules can be translated into an XML format called YIN (Section 10), allowing applications using XML parsers and XSLT scripts to operate on the models.

YANG strikes a balance between high-level object-oriented modeling and low-level bits-on-the-wire encoding. The reader of a YANG module can easily see the high-level view of the data model while seeing how the object will be encoded in NETCONF operations.

YANG is an extensible language, allowing extension statements 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 making extensions 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 NETCONF data models, not arbitrary XML documents or arbitrary data models. The data models described by YANG are designed to be easily operated upon by NETCONF operations.

To the extent possible, YANG maintains compatibility with 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. However YANG is not concerned with reverse translation from YANG to SMIv2.

Like NETCONF, YANG targets smooth integration with device's native management infrastructure. This allows implementations to leverage their existing access control mechanisms to protect or expose elements of the data model.

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 defines modules using the "module" statement. This statement defines the name of the module, which is typically used as the base name of the file containing the module. The file suffix ".yang" is typically used for YANG files. 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.

Submodule are partial modules that contribute derived types, groupings, data nodes, RPCs and notifications to a module. A module may include a number of submodules, but each submodule may belong to only one module. The "include" statement allows a module or submodule to reference material in submodules, and the "import" statement allows references to material defined in other modules.

To reference an item that is defined in an external module it MUST be imported. Identifiers that are neither defined nor imported MUST NOT be visible in the local module.

To reference an item that is defined in one of its submodules, the module MUST include the submodule.

A submodule that needs to reference an item defined in another submodule of the same module, MUST include this submodule.

There MUST NOT be any circular chains of imports or includes. For example, if submodule "a" includes submodule "b", "b" cannot include "a".

When a definition in an external module is referenced, a locally defined prefix MUST be used, followed by ":", and then the external identifier. References to definitions in the local module MAY use the prefix notation. References to built-in data types (e.g., int32) MUST NOT use the prefix notation.

Forward references are allowed in YANG.

4.2.2. Data Modeling Basics

YANG defines four types of nodes for data modeling. In each of the following subsections, the example shows the YANG syntax as well as a corresponding NETCONF XML representation.

4.2.2.1. Leaf Nodes

A leaf node 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";
```

```
       }
```

NETCONF XML Encoding:

```
       <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 is a sequence of leaf nodes with exactly one value of a particular type per leaf.

YANG Example:

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

NETCONF XML Encoding:

```
       <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 node 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 (including leafs, lists, containers, and leaf-lists).

YANG Example:

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

NETCONF XML Encoding:

```
       <system>
           <login>
               <message>Good morning, Dave</message>
           </login>
       </system>
```

The "container" statement is covered in Section 7.5.

4.2.2.4. List Nodes

A list is a sequence of list entries. An entry is like a structure or a record. A list entry is uniquely identified by the values of its key leafs. A list entry can have multiple keys. A list entry 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;
         }
     }
```

NETCONF XML Encoding:

```
     <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>rzull</name>
       <full-name>Repun 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 "acme-system.yang"
     module acme-system {
         namespace "http://acme.example.com/system";
         prefix "acme";

         organization "ACME Inc.";
         contact "joe@acme.example.com";
         description
             "The module for entities implementing the ACME 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. Operational Data

YANG can model operational data, as well as configuration data, based on the "config" statement. When a node is tagged with "config false", its subhierarchy is flagged as operational data, to be reported using NETCONF's <get> operation, not the <get-config> operation. Parent containers, lists, and key leafs are reported also, giving the context for the operational data.

In this example, two leafs are defined for each interface, a configured speed and an observed speed. The observed speed is not configuration, so it can be returned with NETCONF <get> operations, but not with <get-config> operations. The observed speed is not configuration data, and cannot be manipulated using <edit-config>.

```
     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;
         }
     }
```

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 from the management domain. The following table summarizes the built-in types discussed in Section 8:

Name	Type	Description
int8	Number	8-bit signed integer
int16	Number	16-bit signed integer
int32	Number	32-bit signed integer
int64	Number	64-bit signed integer
uint8	Number	8-bit unsigned integer
uint16	Number	16-bit unsigned integer
uint32	Number	32-bit unsigned integer
uint64	Number	64-bit unsigned integer
float32	Number	32-bit IEEE floating point real number
float64	Number	64-bit IEEE floating point real number
string	Text	Human readable string
boolean	Text	"true" or "false"
enumeration	Text/Number	Enumerated strings with associated numeric values
bits	Text/Number	A set of bits or flags
binary	Text	Any binary data
keyref	Text/Number	A reference to a list's key value
empty	Empty	A leaf that does not have any value
union	Text/Number	Choice of member types
instance-identifier	Text	References a data tree node

The "type" statement is covered in Section 8.

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 uint16 {
             range "0 .. 100";
         }
         description "Percentage";
     }

     leaf completed {
         type percent;
     }
```

NETCONF XML Encoding:

```
     <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 the equivalent of complex types using the "grouping" statement. "grouping" defines a set of nodes that are instantiated with the "uses" statement:

```
     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;
         }
     }
```

NETCONF XML Encoding:

```
     <peer>
       <destination>
         <address>192.0.2.1</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 {
              leaf address {
                  description "Source IP address";
              }
              leaf port {
                  description "Source port number";
              }
          }
      }
      container destination {
          uses target {
              leaf address {
                  description "Destination IP address";
              }
              leaf port {
                  description "Destination port number";
              }
          }
      }
  }
```

The "grouping" statement is covered in Section 7.11.

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 which 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".

When an element from one case is created, all elements from all other cases are implicitly deleted. The device handles the enforcement of the constraint, preventing incompatibilities from existing in the configuration.

The choice and case nodes appear only in the schema tree, not in the data tree or XML encoding. The additional levels of hierarchy are not needed beyond the conceptual schema.

YANG Example:

```
     container food {
       choice snack {
           mandatory true;
           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;
                   }
               }
           }
       }
    }
```

NETCONF XML Encoding:

```
     <food>
       <chocolate>first-available</chocolate>
     </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) or an external module. This is useful e.g. 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.

YANG Example:

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

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

If a module augments another model, the XML representation of the data will reflect the prefix of the augmenting model. For example, if the above augmentation were in a module with prefix "other", the XML would look like:

NETCONF XML Encoding:

```
     <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 7.15.

4.2.9. RPC Definitions

YANG allows the definition of NETCONF RPCs. The method names, input parameters and output parameters are modeled using YANG data definition statements.

YANG Example:

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

NETCONF XML Encoding:

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

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

The "rpc" statement is covered in Section 7.13.

4.2.10. Notification Definitions

YANG allows the definition of notifications suitable for NETCONF. 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-index {
        type int32 { range "1 .. max"; }
    }
    leaf if-name {
        type keyref {
            path "/interfaces/interface/name";
        }
    }
}
```

NETCONF XML Encoding:

```
<notification
    xmlns="urn:ietf:params:netconf:capability:notification:1.0">
  <eventTime>2007-09-01T10:00:00Z</eventTime>
  <link-failure xmlns="http://acme.example.com/system">
    <if-name>so-1/2/3.0</if-name>
  </link-failure>
```

```
     </notification>
```

The "notification" statement is covered in Section 7.14.

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 define a complete, cohesive model, or augment an existing data model with additional nodes.

A NETCONF server may implement a number of modules, allowing multiple views of the same data, or multiple views of disjoint subsections of the device's data. Alternatively, the server may implement only one module that defines all available data. Any modules that are implemented MUST be available for all defined datastores.

A module may be divided into submodules, based on the needs of the module owner. The external view remains that of a single module, regardless of the presence or size of its submodules.

A module uses the "include" statement to include its submodules, and the "import" statement to reference external modules. Similarly, a submodule may use the "import" statement to reference other modules, and may use the "include" statement to reference other submodules within its module. A module or submodule may not include submodules from other modules, nor may a submodule import its own module.

The names of all standard modules must be unique, but different revisions of the same module should have the same name. Developers of enterprise modules are encouraged 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.

5.1.1. Module Hierarchies

YANG allows modeling of data in multiple hierarchies, where data may have more than one root node. Models that have multiple roots nodes are sometimes convenient, and are supported by YANG.

5.2. File Layout

YANG modules and submodules are typically stored in files, one module or submodule per file, with the name of the file given by the concatenation of the module or submodule name and the file suffix ".yang". YANG compilers can find imported modules and included submodules via this convention. While the YANG language defines modules, tools may compile submodules independently for performance and manageability reasons. Many errors and warnings that cannot be detected during submodule compilation may be delayed until the

submodules are linked into a cohesive module.

5.3. Object Based View of YANG

While YANG models the configuration as a data tree, it can be used in an object-based manner as well.

The configuration and state data of the device is modeled as a tree of object instances (objects for short). Each object in the tree has a type name (or managed object class name), a namespace, a (possibly empty) set of attributes and a (possibly empty) set of child objects.

A managed object class could be defined as a grouping, containing just one list. Attributes should be defined as leafs inside the list. Child objects should be defined with the corresponding uses statements.

A defined grouping unambiguously defines its properties, it has its own unique name, so when it is referred to in the "uses" statement it is always the same well defined set of properties that we are using.

The data tree can be defined as one or more top level containers containing managed object classes defined as groupings. All further levels of the data tree are defined by managed object classes containing further managed objects.

5.4. XML Namespaces

All YANG definitions are specified within a particular XML Namespace. Each module defines an XML namespace as a globally unique URI [RFC3986]. A NETCONF client or server uses the namespace during XML encoding of data.

The namespace URI is advertised as a capability in the NETCONF <hello> message to indicate support for the YANG module by a NETCONF server. The capability URI advertised SHOULD be on the form:

```
  namespace-uri "?" revision
```

Where "revision" is the revision of the module (see Section 7.1.9) that the server implements.

Namespaces for standard module names will be assigned by IANA. They MUST be unique (but different revisions of the same module should have the same namespace).

Namespaces for private module names will be assigned by the organization owning the module without a central registry. It is

recommended to choose namespaces that will have a low probability of colliding with standard or other enterprise modules, e.g. by using the enterprise or organization name in the namespace.

The "namespace" statement is covered in Section 7.1.3.

5.4.1. YANG Namespace

YANG defines its own namespace for NETCONF <edit-config> operations. This namespace is "urn:ietf:params:xml:ns:yang:1" [XXX IANA].

5.5. Ordering

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

For example, the order in which firewall filters 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 which allow 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 7.7.4.

5.6. Containers with Presence

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

In the first style, the container has no meaning of its own, existing only to contain child nodes. The container data node is implicitly created when the first child data node is created. The data node is implicitly deleted when the last non-key child is deleted, since an empty container has no meaning.

For example, the set of scrambling options for 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 the task of performing this task.

In the second style, the presence of the container itself is configuration data, representing a single bit of configuration data. The container acts as both a configuration knob and a means of organizing related configuration. These containers are explicitly created and deleted.

YANG calls this style a "presence container" and they are 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 device using ssh, but can also contain any ssh-related configuration knobs, such as connection rates or retry limits.

The "presence" statement is covered in Section 7.5.4.

5.7. Scoping

YANG uses static scoping. Grouping definitions are resolved in the context in which they are defined, rather than the context in which they are used. Users of groupings are not required to import modules or include submodules to satisfy all references made by the grouping.

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 original module, not the second module. There is no worry over conflicts if both modules define the type, since there is no ambiguity.

5.8. Nested Typedefs and Groupings

Typedefs and groupings may appear nested under many YANG statements, allowing these to be lexically scoped by the 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 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 group cannot be defined if a higher level in the schema hierarchy has a definition with a matching identifier.

When a YANG implementation resolves a reference to an unprefixed type or grouping, or one which uses the prefix of the local module, it searches up the levels of hierarchy in the schema tree, starting at the current level, for the definition of the type or grouping.

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.

YANG modules are written in the UTF-8 [RFC3629] character set.

6.1. Lexicographical 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 is enclosed within "/*" and "*/".

6.1.2. Tokens

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

6.1.3. Quoting

If a string contains any whitespace characters, a semicolon (";"), curly braces ("{" or "}"), or comment sequences ("//", "/*", or "*/"), then it MUST be enclosed within double or single quotes.

If the double quoted string contains a line break followed by whitespace which is used to indent the text according to the layout in the YANG file, this leading whitespace is stripped from the string, up to at most the same column of the double quote character.

If the double quoted string contains whitespace 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 can not occur in a single quoted string, even when preceded by a backslash.

If a quoted string is followed by a plus character ("+"), followed by another quoted string, the two strings are concatenated into one quoted string, allowing multiple concatenations to build one quoted string. Whitespace trimming of double quoted strings is done before concatenation.

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

\n	new line
\t	a tab character
\"	a double quote
\\	a single backslash

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 upper-case or lower-case 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 63 characters in length. Identifiers are case sensitive. The identifier syntax is formally defined by the rule "identifier" in Section 11. Identifiers can be specified as quoted or unquoted strings.

6.2.1. Identifiers and their namespaces

Each identifier is valid in a namespace which depends on the type of the YANG item being defined:

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 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 the parent node or module.

o All groupings 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 the parent node or module.

o All leafs, leaf-lists, lists, containers, choices, rpcs, and notifications defined 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 parent node of the case node's parent choice node.

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

All identifiers defined in a namespace MUST be unique.

6.3. Statements

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

within curly 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 7.16). 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 module's prefix.

6.4. XPath Evaluations

YANG relies on XPath 1.0 [XPATH] as a notation for specifying many inter-node references and dependencies. NETCONF clients and servers are 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 valid, but any implementation may choose to implement them by hand, rather than using the XPath expression directly.

XPath expressions are evaluated in the context of the current node, with the namespace of the current module defined as the null namespace. References to identifiers in external modules MUST be qualified with appropriate prefixes, and references to the current module and its submodules MAY use a prefix.

7. YANG Statements

The following sections describe all of the YANG core statements.

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

```
  description "some text" {
      acme:documentation-flag 5;
  }
```

7.1. The module Statement

The "module" statement defines the module's name, and groups all statements which belong to the module together. The "module" statement's argument is the name of the module, followed by a block of substatements that hold detailed module information. The module name follows the rules for identifiers in Section 6.2.

Standard module names will be assigned by IANA. The names of all standard modules MUST be unique (but different revisions of the same module should have the same name).

Private module names will be assigned by the organization owning the module without a central registry. It is 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.

A module SHOULD have 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
         <extension statements>
         <typedef statements>
         <grouping statements>
         <container statements>
         <leaf statements>
         <leaf-list statements>
         <list statements>
         <choice statements>
         <uses statements>
         <rpc statements>
         <notification statements>
         <augment statements>
     }
```

7.1.1. The module's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
contact	7.1.8	0..1
container	7.5	0..n
description	7.17.3	0..1
extension	7.16	0..n
grouping	7.11	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	7.14	0..n
organization	7.1.7	0..1
prefix	7.1.4	1
reference	7.17.4	0..1
revision	7.1.9	0..n
rpc	7.13	0..n
typedef	7.3	0..n
uses	7.12	0..n
yang-version	7.1.2	0..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 contains value "1", which is the current yang version and the default value.

This statement is intended for future-proofing the syntax of YANG against possible changes in later versions of YANG. Since the current version is the default value, the statement need not appear in YANG modules until a future version is defined. When a new version is defined, YANG modules can either use version 2 features and add the "yang-version 2" statement, or remain within the version 1 feature set and continue to use the default setting of "yang-version 1".

7.1.3. The namespace Statement

The "namespace" statement defines the XML namespace for all XML elements defined by the module. Its argument is the URI of the

namespace.

See also Section 5.4.

7.1.4. The prefix Statement

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

When used inside the "module" statement, the "prefix" statement defines the prefix to be used when this module is imported. To improve readability of the NETCONF XML, a NETCONF client or server which generates XML or XPath that use prefixes, the prefix defined by a module SHOULD be used, unless there is a conflict.

When used inside the "import" statement, the "prefix" statement defines the prefix to be used when accessing data inside the imported module. When a reference to an identifier from the imported module is used, the prefix string for the module from which objects are being imported is used in combination with a colon (":") and the identifier, 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.

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.

All identifiers contained in an imported module are imported into the current module or submodule, so that they can be referenced by definitions in the current module or submodule. The mandatory "prefix" substatement assigns a prefix for the imported module which is scoped to the importing module or submodule. Multiple "import" statements may be specified to import from different modules.

substatement	section	cardinality
prefix	7.1.4	1

7.1.6. The include Statement

The "include" statement is used to make content from a submodule available to the module. The argument is an identifier which 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. When a submodule includes another submodule, the target submodule's definitions are made available to the current submodule.

7.1.7. The organization Statement

The "organization" statement defines the party responsible for this module. The argument is a string which 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 which is used to specify the name, postal address, telephone number, and electronic mail address of the person to whom technical queries concerning this module should be sent.

7.1.9. The revision Statement

The "revision" statement specifies the editorial revision history of the module, including the initial revision. A series of revisions 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 initial "revision" statement. For every 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 Substatement

substatement	section	cardinality
description	7.17.3	0..1

7.1.10. Usage Example

```
     module acme-system {
         namespace "http://acme.example.com/system";
         prefix "acme";

         import yang-types {
             prefix "yang";
         }

         include acme-types;

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

              ACME, Inc.
              42 Anywhere Drive
              Nowhere, CA 95134
              USA

              Phone: +1 800 555 0815
              EMail: joe@acme.example.com";

         description
             "The module for entities implementing the ACME protocol.";

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

         // definitions follows...
     }
```

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 to split a complex module in several pieces where all the submodules contribute to a single namespace, which is defined by the module

including the submodules.

The "submodule" statement is used to give the submodule a name, and to group all statements which belong to the submodule together.

The "submodule" statement, which must be present at most once, takes as an argument an identifier which is the name of the submodule, followed by a block of substatements that hold detailed submodule information.

Standard submodule names will be assigned by IANA. Name of all standard submodules must be unique and in addition not conflict with module names (but different revisions of the same submodule should have the same name).

Private submodule names will be assigned by the organization owning the submodule without a central registry. It is recommended to choose names for their submodules that will have a low probability of colliding with standard or other enterprise modules and submodules, e.g. by using the enterprise or organization name as a prefix.

A submodule SHOULD have the following layout:

```
  submodule <module-name> {


      <yang-version statement>
```

```
             // module identification
             <belongs-to statement>

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

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

             // revision history
             <revision statements>

             // module definitions
             <extension statements>
             <typedef statements>
             <grouping statements>
             <container statements>
             <leaf statements>
             <leaf-list statements>
             <list statements>
             <choice statements>
             <uses statements>
             <rpc statements>
             <notification statements>
             <augment statements>
         }
```

7.2.1. The submodule's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	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	7.17.3	0..1
extension	7.16	0..n
grouping	7.11	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	7.14	0..n
organization	7.1.7	0..1
reference	7.17.4	0..1
revision	7.1.9	0..n
rpc	7.13	0..n
typedef	7.3	0..n
uses	7.12	0..n
yang-version	7.1.2	0..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 which is the name of the module. Only the module to which a submodule belongs, or another submodule that belongs to the same module, are allowed to include that submodule.

7.2.3. Usage Example

```
     submodule acme-types {

         belongs-to "acme-system";

         import yang-types {
             prefix "yang";
         }

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

              ACME, Inc.
              42 Anywhere Drive
              Nowhere, CA 95134
              USA

              Phone: +1 800 555 0815
              EMail: joe@acme.example.com";

         description
             "This submodule defines common ACME types.";

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

         // definitions follows...
     }
```

7.3. The typedef Statement

The "typedef" statement defines a new type which may be used locally in the module, in modules or submodules which include it, and by other modules which import from it. 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 which 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. For details about scoping for nested typedef, see Section 5.8.

7.3.1. The typedef's Substatements

substatement	section	cardinality
default	7.3.4	0..1
description	7.17.3	0..1
reference	7.17.4	0..1
status	7.17.2	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 which 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 which contains a default value for the new type.

The value of the "default" statement MUST correspond 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. The default value MUST correspond to any restrictions in the derived type.

If the base type's default value does not correspond to the new restrictions, the derived type MUST define a new default value.

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 which is the name of a YANG built-in type (see Section 8) or a derived type (see Section 7.3), followed by an optional block of substatements that are used to put further restrictions on the type.

The restrictions that can be applied depends on the type being restricted. All restriction statements are described in conjunction with the built-in types in Section 8.

7.4.1. The type's Substatements

substatement	section	cardinality
bit	8.6.3	0..n
enum	8.5.3	0..n
length	8.3.3	0..1
path	8.8.2	0..1
pattern	8.3.5	0..n
range	8.1.3	0..1
type	7.4	0..n

7.5. The container Statement

The "container" statement is used to define an interior 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.

By default, a container does not carry any information, but is used to organize and give structure to the data being defined. The "presence" statement (see Section 7.5.4) is used to give semantics to the existence of the container in the data tree.

7.5.1. The container's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
config	7.17.1	0..1
container	7.5	0..n
description	7.17.3	0..1
grouping	7.11	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
must	7.5.2	0..n
presence	7.5.4	0..1
reference	7.17.4	0..1
status	7.17.2	0..1
typedef	7.3	0..n
uses	7.12	0..n

7.5.2. The must Statement

The "must" statement, which is optional, takes as an argument a string which contains an XPath expression. It is used to formally declare a constraint on the configuration data. When a configuration datastore is validated, all "must" constraints are conceptually evaluated once for each corresponding instance in the datastore's data tree, and for all leafs with default values in effect. If an instance does not exist in the data tree, and it does not have a default value, its "must" statements are not evaluated.

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

The "must" statement is ignored if the data does not represent configuration.

The XPath expression is conceptually evaluated in the following context:

o The context node is the node in the data tree for which the "must" statement is defined.

o The accessible tree is made up of all nodes in the data tree, and all leafs with default values.

o The set of namespace declarations is the set of all "import" statements' prefix and namespace pairs, and the "prefix" statement's prefix for the "namespace" statement's URI.

o Elements without a namespace refer to nodes in the current module.

o The function library is the core function library defined in [XPATH], and a function "current()" which returns a node set with the initial context node.

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

If the node with the must statement represents configuration data, any node referenced in the XPath expression MUST also represent configuration.

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

7.5.3. The must's Substatements

substatement	section	cardinality
description	7.17.3	0..1
error-app-tag	7.5.3.2	0..1
error-message	7.5.3.1	0..1
reference	7.17.4	0..1

7.5.3.1. 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>.

7.5.3.2. 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>.

7.5.3.3. 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 " +
              "(ifType = 'ethernet' and ifMTU = 1500)" {
             error-message "An ethernet MTU must be 1500";
         }
         must "ifType != 'atm' or " +
              "(ifType = 'atm' and ifMTU <= 17966 and ifMTU >= 64)" {
             error-message "An atm MTU must be  64 .. 17966";
         }
     }
```

7.5.4. 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 which 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 5.6 for additional information.

7.5.5. The container's Child Node Statements

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

7.5.6. XML Encoding Rules

A container node is encoded as an XML element. The element's name is the container's identifier, and its XML namespace is the module's XML namespace.

The container's child nodes are encoded as subelements to the

container element, in the same order as they are defined within the container statement.

A NETCONF server that replies to a <get> or <get-config> request MAY choose not to send a container element if the container node does not have the "presence" statement and no child nodes exist. Thus, a client that receives an <rpc-reply> for a <get> or <get-config> request, must be prepared to handle the case that a container node without a presence statement is not present in the XML.

7.5.7. NETCONF <edit-config> Operations

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

If the operation is "merge" the node is created if it does not exist.

If the operation is "replace" and the node exists, all child nodes not present in the XML are deleted, and child nodes present in the XML but not present in the datastore are created.

If the operation is "create" the node is created if it does not exist.

If the operation is "delete" the node is deleted if it exists.

If the container has a "presence" statement, it may be implicitly created if it does not exist, even if the operation is "none".

If a container has a "presence" statement and the last child node is deleted, the NETCONF server MAY delete the container.

7.5.8. 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 encoding would look like this:

```
<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="http://example.com/schema/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.

A leaf node exists in zero or one instances in the data tree, depending on the value of the "mandatory" statement.

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

If a leaf has a "default" statement, the leaf's default value is set to 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 set to the type's default value. In all other cases, the leaf does not have a default value.

If the leaf has a default value, the server MUST use this value internally if no value is provided by the NETCONF client when the instance is created.

7.6.1. The leaf's Substatements

substatement	section	cardinality
config	7.17.1	0..1
default	7.6.3	0..1
description	7.17.3	0..1
mandatory	7.6.4	0..1
must	7.5.2	0..n
reference	7.17.4	0..1
status	7.17.2	0..1
type	7.6.2	1
units	7.3.3	0..1

7.6.2. 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.

7.6.3. The leaf's default Statement

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

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

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

7.6.4. The leaf's mandatory Statement

The "mandatory" statement, which is optional, takes as an argument the string "true" or "false". If "mandatory" is "true", the node must exist in a valid configuration if its parent node exists. Since containers without a "presence" statement are implicitly created and deleted when needed, they are ignored when performing mandatory tests for leafs. A mandatory leaf within such a container is mandatory even if the container's data node does not exist.

If not specified, the default is "false".

7.6.5. XML Encoding Rules

A leaf node is encoded as an XML element. The element's name is the leaf's identifier, and its XML namespace is the module's XML namespace.

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

A NETCONF server that replies to a <get> or <get-config> request MAY choose not to send the leaf element if its value is the default value. Thus, a client that receives an <rpc-reply> for a <get> or <get-config> request, must be prepared to handle the case that a leaf node with a default value is not present in the XML. In this case, the value used by the server is known to be the default value.

See Section 7.6.7 for an example.

7.6.6. NETCONF <edit-config> Operations

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

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

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

If the operation is "create" the node is created if it does not exist.

If the operation is "delete" the node is deleted if it exists.

7.6.7. Usage Example

Given the following leaf statement:

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

A corresponding XML encoding:

```
<port>2022</port>
```

To create 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="http://example.com/schema/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.

The values in a leaf-list MUST be unique.

If the type referenced by the leaf-list has a default value, it has no effect in the leaf-list.

7.7.1. The leaf-list's Substatements

substatement	section	cardinality
config	7.17.1	0..1
description	7.17.3	0..1
max-elements	7.7.3	0..1
min-elements	7.7.2	0..1
must	7.5.2	0..n
ordered-by	7.7.4	0..1
reference	7.17.4	0..1
status	7.17.2	0..1
type	7.4	1
units	7.3.3	0..1

7.7.2. The min-elements Statement

The "min-elements" statement, which is optional, takes as an argument a non-negative integer which puts a constraint on a valid configuration. A valid configuration always has at least min-elements values in the leaf-list or list.

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

7.7.3. 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 a valid configuration. A valid configuration always has at most max-elements values in the leaf-list or list.

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

7.7.4. 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, order defaults to "system".

See Section 5.5 for additional information.

7.7.4.1. ordered-by system

The entries in the list are sorted according to an unspecified order. Thus an implementation is free to sort the entries in the most appropriate order. An implementation SHOULD use the same order for the same data, regardless of how the data were created. Using a deterministic order will makes comparisons possible using simple tools like "diff".

This is the default order.

7.7.4.2. ordered-by user

The entries in the list are sorted according to an order defined by the user. This order is controlled by using special XML attributes in the <edit-config> request. See Section 7.7.6 for details.

7.7.5. XML Encoding Rules

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

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

See Section 7.7.7 for an example.

7.7.6. 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 namespace (Section 5.4.1) 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".

In a <copy-config>, or an <edit-config> with a "replace" operation which 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:

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

If the operation is "create" the leaf-list entry is created if it does not exist.

If the operation is "delete" the entry is deleted from the leaf-list if it exists.

7.7.7. Usage Example

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

A corresponding XML encoding:

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

To create a new element in the list:

```
<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="http://example.com/schema/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 ciphers  {
      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="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="http://example.com/schema/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 interior nodes 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.

A list is similar to a table where each list entry is a row in the table.

7.8.1. The list's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
config	7.17.1	0..1
container	7.5	0..n
description	7.17.3	0..1
grouping	7.11	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	7.7.3	0..1
min-elements	7.7.2	0..1
must	7.5.2	0..n
ordered-by	7.7.4	0..1
reference	7.17.4	0..1
status	7.17.2	0..1
typedef	7.3	0..n
unique	7.8.3	0..n
uses	7.12	0..n

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 which specifies a space separated list of 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 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. It also implies that any mandatory statement in the key leafs are ignored.

A leaf that is part of the key can be of any built-in or derived type, except it MUST NOT be the built-in type "empty".

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 11.

7.8.3. The lists's unique Statement

The "unique" statement is used to put constraints on valid configurations. It takes as an argument a string which 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 11). Each such schema node identifier MUST refer to a leaf.

In a valid configuration, the combined values of all the leaf instances specified in the string MUST be unique within all list entry instances.

The unique string syntax is formally defined by the rule "unique-arg" in Section 11.

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>
```

7.8.4. The list's Child Node Statements

Within a list, the "container", "leaf", "list", "leaf-list", "uses", and "choice" 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 name is the list's identifier, and its XML namespace is the module's XML namespace.

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, in the same order as they are defined within the list statement.

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 namespace (Section 5.4.1) 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 8.11) for the list entry.

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

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

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 encoding:

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

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="http://example.com/schema/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="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="http://example.com/schema/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 "name";

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

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

```
     <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="http://example.com/schema/config"
                xmlns:ex="http://example.com/schema/config">
             <user nc:operation="create"
                   yang:insert="after"
                   yang:key="[ex:name='fred']">
               <name>barney</name>
               <type>admin</type>
               <full-name>Barney Rubble</full-name>
             </user>
           </system>
         </config>
       </edit-config>
     </rpc>
```

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

```
     <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="http://example.com/schema/config"
                xmlns:ex="http://example.com/schema/config">
             <user nc:operation="merge"
                   yang:insert="before"
                   yang:key="[ex:name='fred']">
               <name>barney</name>
             </user>
           </system>
         </config>
       </edit-config>
     </rpc>
```

7.9. The choice Statement

The "choice" statement defines a set of alternatives, only one of which may exist at any one time. 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, defined with the case substatement. Each branch contains a number of child nodes. The "choice" statement puts a constraint on a valid configuration. In a valid configuration, the nodes from at most one of the choice's branches exist at the same time.

See Section 4.2.7 for additional information.

7.9.1. The choice's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
case	7.9.2	0..n
config	7.17.1	0..1
container	7.5	0..n
default	7.9.3	0..1
description	7.17.3	0..1
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
mandatory	7.9.4	0..1
reference	7.17.4	0..1
status	7.17.2	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 "anyxml", "container", "leaf", "list", "leaf-list", "uses", and "augment" 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 "anyxml", "container", "leaf", "list", or "leaf-list" statement. In this case, the identifier of the case node is the same as the identifier in the branch statement. 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
anyxml	7.10	0..n
augment	7.15	0..n
container	7.5	0..n
description	7.17.3	0..1
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
reference	7.17.4	0..1
status	7.17.2	0..1
uses	7.12	0..n

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 exists. The argument is the identifier of the "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 values of nodes under the cases. The default values for nodes 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 3.1) 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;
                       default 30;
                       units minutes;
                   }
               }
               case daily {
                   leaf daily {
                       type empty;
                   }
                   leaf time-of-day {
                       type string;
                       units 24-hour-clock;
                       default 1am;
                   }
               }
               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". If "mandatory" is "true", at least one node from exactly one of the choice's case branches MUST exist in a valid configuration. If "mandatory" is "false", a valid configuration MAY have no nodes from the choice's case branches present.

If not specified, the default is "false".

7.9.5. XML Encoding Rules

The choice and case nodes are not visible in XML.

7.9.6. NETCONF <edit-config> operations

Since only one of the choices cases can be valid at any time, the creation of a node from one case implicitly deletes all nodes from all other cases. If an <edit-config> operation creates a node, the

NETCONF server will delete any existing nodes that are defined in other cases inside the choice.

7.9.7. 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 encoding:

```
     <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="http://example.com/schema/config">
             <protocol>
               <udp nc:operation="create"/>
             </protocol>
           </system>
         </config>
       </edit-config>
     </rpc>
```

7.10. 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 in e.g. RPC replies. An example is the <filter> parameter in the <get-config> operation.

An anyxml node cannot be augmented.

It is NOT RECOMMENDED that the anyxml statement is used to represent configuration data.

7.10.1. The anyxml's Substatements

substatement	section	cardinality
config	7.17.1	0..1
description	7.17.3	0..1
mandatory	7.6.4	0..1
reference	7.17.4	0..1
status	7.17.2	0..1

7.10.2. XML Encoding Rules

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

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

7.10.3. 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 within the XML value 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:

If the operation is "merge", 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 operation is "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.

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 operation is "delete" the node is deleted if it exists.

7.10.4. Usage Example

Given the following anyxml statement:

```
  anyxml data;
```

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

```
  <data xmlns:if="http://example.com/ns/interface">
    <if:interface>
      <if:ifIndex>1</if:ifIndex>
    </if:interface>
  </data>

  <data>
    <interface xmlns="http://example.com/ns/interface">
      <ifIndex>1</ifIndex>
    </interface>
  </data>
```

7.11. The grouping Statement

The "grouping" statement is used to define a reusable block of nodes, which may be used locally in the module, in modules which include it, and by other modules which import from it. 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 7.12). 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. For details about scoping for nested groupings, see Section 5.8.

A grouping is more than just a mechanism for textual substitution, but defines a collection of nodes. References from inside the grouping are 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 are applied to the grouping node, not the use node.

7.11.1. The grouping's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
container	7.5	0..n
description	7.17.3	0..1
grouping	7.11	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
reference	7.17.4	0..1
status	7.17.2	0..1
typedef	7.3	0..n
uses	7.12	0..n

7.11.2. Usage Example

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

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

7.12. 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 then updated according to the refinement statements. Thus, the identifiers defined in the grouping are copied into the current module's namespace, even if the grouping is imported from some other module.

7.12.1. The uses's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
choice	7.9	0..n
container	7.5	0..n
description	7.17.3	0..1
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
reference	7.17.4	0..1
status	7.17.2	0..1
uses	7.12	0..n

7.12.2. The uses's Refinement Statements

Some of the properties of each node in the grouping can be refined in substatements to "uses". If a node is not present in a substatement, it is not refined, and thus used exactly as it was defined in the "grouping". A node can be refined only once in "uses".

The following refinements can be done:

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

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

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

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

- A leaf or choice node may get a different "mandatory" statement.

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

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

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

7.12.3. 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.

7.12.4. Usage Example

To use the "address" grouping defined in Section 7.11.2 in a definition of an HTTP server in some other module, we can do:

```
  import acme-system {
      prefix acme;
  }

  container http-server {
      leaf name {
          type string;
      }
      uses acme:address;
  }
```

A corresponding XML encoding:

```
  <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, default can be added:

```
  container http-server {
      leaf name {
          type string;
      }
      uses acme:address {
          leaf port {
              default 80;
          }
      }
  }
```

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

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

The following is an error:

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

7.13. The rpc Statement

The "rpc" statement is used to define a NETCONF RPC method. 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, and is used as the element name directly under the <rpc> element, as designated by the substitution group "rpcOperation" in [RFC4741].

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

7.13.1. The rpc's Substatements

substatement	section	cardinality
description	7.17.3	0..1
grouping	7.11	0..n
input	7.13.2	0..1
output	7.13.3	0..1
reference	7.17.4	0..1
status	7.17.2	0..1
typedef	7.3	0..n

7.13.2. The input Statement

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

If a container in the input tree has a "presence" statement, the container need not be present in a NETCONF RPC invocation.

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

If a leaf in the input tree has a default value, the NETCONF server MUST internally use this default if the leaf is not present in a NETCONF RPC invocation.

If a "config" or "must" statement is present for any node in the input tree, it is ignored.

7.13.2.1. The input's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
container	7.5	0..n
grouping	7.11	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
typedef	7.3	0..n
uses	7.12	0..n

7.13.3. The output Statement

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

If a container in the output tree has a "presence" statement, the container need not be present in a NETCONF RPC reply

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

If a leaf in the output tree has a default value, the NETCONF client MUST internally use this default if the leaf is not present in a NETCONF RPC reply.

If a "config" or "must" statement is present for any node in the output tree, it is ignored.

7.13.3.1. The output's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
container	7.5	0..n
grouping	7.11	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
typedef	7.3	0..n
uses	7.12	0..n

7.13.4. XML Encoding Rules

An rpc node is encoded as a child XML element to the <rpc> element defined in [RFC4741]. The element's name is the rpc's identifier, and its XML namespace is the module's XML namespace.

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 method invocation succeeded, and no output parameters are returned, the <rpc-reply> contains a single <ok/> element defined in [RFC4741]. If output parameters are returned, they are encoded as child elements to the <rpc-reply> element defined in [RFC4741], in the same order as they are defined within the output statement.

7.13.5. Usage Example

The following example defines an RPC method:

```
  module rock {
      namespace "http://example.net/rock";
      prefix rock;

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

```
     }
```

A corresponding XML encoding 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="http://example.net/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>
```

7.14. The notification Statement

The "notification" statement is used to define a NETCONF 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.

If a container in the notification tree has a "presence" statement, the container need not be present in a NETCONF notification.

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

If a leaf in the notification tree has a default value, the NETCONF server MUST internally use this default if the leaf is not present in a NETCONF notification.

If a "config" or "must" statement is present for any node in the notification tree, it is ignored.

7.14.1. The notification's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
choice	7.9	0..n
container	7.5	0..n
description	7.17.3	0..1
grouping	7.11	0..n
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
reference	7.17.4	0..1
status	7.17.2	0..1
typedef	7.3	0..n
uses	7.12	0..n

7.14.2. XML Encoding Rules

A notification node is encoded as a child XML element to the <notification> element defined in [RFC5277]. The element's name is the notification's identifier, and its XML namespace is the module's XML namespace.

The notifications's child nodes are encoded as subelements to the notification node's XML element, in the same order as they are defined within the notification statement.

7.14.3. Usage Example

The following example defines a notification:

```
     module event {

         namespace "http://example.com/event";
         prefix ev;

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

A corresponding XML encoding of the complete notification:

```
     <notification
       xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
       <eventTime>2008-07-08T00:01:00Z</eventTime>
       <event xmlns="http://example.com/event">
         <event-class>fault</event-class>
         <reporting-entity>
           <card>Ethernet0</card>
         </reporting-entity>
         <severity>major</severity>
       </event>
     </notification>
```

7.15. The augment Statement

The "augment" statement allows a module or submodule to add to the schema tree defined in another module or submodule. The argument is a string which 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 augment string is a schema node identifier. The syntax is formally defined by the rule "augment-arg" in Section 11. 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 11) of a schema node identifier MAY be used. Otherwise, the descendant form (defined by the rule "descendant-schema-nodeid" in Section 11) MUST be used.

The syntax for a schema node identifier is a subset of the XPath syntax. It is an absolute or relative XPath location path in abbreviated syntax, where axes and predicates are not permitted.

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 choice node, the "case" statement can be used within the "augment" statement.

If the target node is in another module, then nodes added by the augmentation MUST NOT be mandatory nodes (see Section 3.1).

7.15.1. The augment's Substatements

substatement	section	cardinality
anyxml	7.10	0..n
augment	7.15	0..n
case	7.9.2	0..n
choice	7.9	0..n
container	7.5	0..n
description	7.17.3	0..1
leaf	7.6	0..n
leaf-list	7.7	0..n
list	7.8	0..n
reference	7.17.4	0..1
status	7.17.2	0..1
uses	7.12	0..n
when	7.15.2	0..1

7.15.2. The when Statement

The "when" statement allows the augmentation to be conditional, with the nodes only being valid when a specific criteria is satisfied. The statement's argument is an XPath expression, which is used to formally specify constraints on which instances in the data tree will be augmented by this statement. If the XPath expression conceptually evaluates to "true" for a particular instance, then it is augmented, otherwise it is not.

The XPath expression is conceptually evaluated in the following context:

- 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 which is also a data node.

- The accessible tree is made up of all nodes in the data tree, and all leafs with default values.

- The set of namespace declarations is the set of all "import" statements' prefix and namespace pairs, and the "prefix" statement's prefix for the "namespace" statement's URI.

- Elements without a namespace refer to nodes in the current module.

- The function library is the core function library defined in [XPATH], and a function "current()" which returns a node set with the initial context node.

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

Note that the XPath expression is conceptually evaluated. This means that an implementation does not have to use an XPath evaluator on the device. The augment can very well be implemented with specially written code.

7.15.3. 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 augmented child nodes are encoded after all normal child nodes. If the node is augmented more than once, the blocks of augmented child nodes are sorted (in alphanumeric order) according to their namespace URI and name of the first child node in each block.

7.15.4. Usage Example

In namespace http://example.com/schema/interfaces, 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 http://example.com/schema/ds0, we have:

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

A corresponding XML encoding:

```
     <interfaces xmlns="http://example.com/schema/interfaces"
                 xmlns:ds0="http://example.com/schema/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 7.9.7. 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 encoding:

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

or

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

7.16. 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 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, and an optional block of substatements. The statement's name is created by combining the 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, using some mechanism outside the scope of this specification. Syntactically, the substatements

MUST be core YANG statements, or also defined using "extension" statements. Core YANG statements in extensions MUST follow the syntactical rules in Section 11.

7.16.1. The extension's Substatements

substatement	section	cardinality
argument	7.16.2	0..1
description	7.17.3	0..1
reference	7.17.4	0..1
status	7.17.2	0..1

7.16.2. The argument Statement

The "argument" statement, which is optional, takes as an argument a string which 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 text statement.

7.16.2.1. The argument's Substatements

substatement	section	cardinality
yin-element	7.16.2.2	0..1

7.16.2.2. The yin-element Statement

The "yin-element" statement, which is optional, takes as an argument the string "true" or "false". This statement indicates if the argument should be mapped to an XML element in YIN or to an XML attribute. (see Section 10).

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

7.16.3. Usage Example

To define an extension:

```
     module my-extensions {
       ...

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

To use the extension:

```
     module my-interfaces {
       ...
       import my-extensions {
         prefix "myext";
       }
       ...

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

7.17. Common Statements

This section defines sub-statements common to several other statements.

7.17.1. The config Statement

The "config" statement takes as an argument the string "true" or "false". If "config" is "true", the definition represents configuration, and will be part of the reply to a <get-config> request, and may be sent in a <copy-config> or <edit-config> request. If "config" is "false", it represents state data, and will be part of the reply to a <get>, but not to a <get-config> request.

If "config" is not specified, the default is the same as the parent node's (in the data model) "config" value. If the top node does not specify a "config" statement, the default is "true".

If a node has "config" "false", no node underneath it can have "config" set to "true".

7.17.2. 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 the definition is obsolete and should not be implemented and/or can be removed if previously implemented.

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.

7.17.3. The description Statement

The "description" statement takes as an argument a string which contains a high-level textual description of this definition.

7.17.4. The reference Statement

The "reference" statement takes as an argument a string which is used to specify a textual cross-reference to an external document, either another module which defines related management information, or a document which provides additional information relevant to this definition.

8. 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, 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 (Section 8.3.3, Section 8.3.5) and range restrictions of numeric types (Section 8.1.3).

The lexicographic representation of a value of a certain type is used in the XML encoding over NETCONF, and when specifying default values in a YANG module.

8.1. 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.

8.1.1. Lexicographic Representation

An integer value is lexicographically 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 lexicographic representation can be used, which represents the value in a hexadecimal or octal notation. The hexadecimal notation consists of an optional sign ("+" or "-"), the characters "0x" followed a number of hexadecimal digits, where letters may be upper- or lowercase. The octal notation consists of an optional sign ("+" or "-"), the character "0" followed a number of octal digits.

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
```

8.1.2. Restrictions

All integer types can be restricted with the "range" statement (Section 8.1.3).

8.1.3. 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 floating point built-in types, or types derived from those.

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 value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is 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" means the minimum and maximum value accepted for the type being restricted, respectively.

The range expression syntax is formally defined by the rule "range-arg" in Section 11.

8.1.3.1. The range's Substatements

substatement	section	cardinality
description	7.17.3	0..1
error-app-tag	7.5.3.2	0..1
error-message	7.5.3.1	0..1
reference	7.17.4	0..1

8.1.4. Usage Example

```
  typedef my-base-int32-type {
      type int32 {
          range "1..4 | 10..20";
      }
  }

  type my-base-int32-type {
      // legal range restriction
      range "11..max"; // 11..20
  }

  type int32 {
      // illegal range restriction
      range "11..100";
  }
```

8.2. The Floating Point Built-in Types

The floating point built-in types are float32 and float64. They represent floating point values of single and double precision as defined in [IEEE.754]. Special values are positive and negative infinity, and not-a-number.

8.2.1. Lexicographic Representation

A floating point value is lexicographically represented as consisting of a decimal mantissa followed, optionally, by the character "E" or "e", followed by an integer exponent. The special values positive and negative infinity and not-a-number have lexical representations INF, -INF and NaN, respectively. The minimal value accepted for a float is -INF, and the maximal value accepted for a float is INF.

8.2.2. Restrictions

All floating point types can be restricted with the "range" statement (Section 8.1.3).

8.2.3. Usage Example

```
  type float32 {
      range "1..4.5 | 10 | 20..INF";
  }
```

is equivalent to

```
  type float32 {
      range "1..4.5 | 10 | 20..max";
  }
```

8.3. The string Built-in Type

The string built-in type represents human readable strings in YANG. Legal characters are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646 [ISO.10646]:

```
  // any Unicode character, excluding the surrogate blocks,
  // FFFE, and FFFF.
  string = *char
  char = %x9 / %xA / %xD / %x20-DFFF / %xE000-FFFD /
         %x10000-10FFFF
```

8.3.1. Lexicographic Representation

A string value is lexicographically represented as character data in the XML encoding.

8.3.2. Restrictions

A string can be restricted with the "length" (Section 8.3.3) and "pattern" (Section 8.3.5) statements.

8.3.3. 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 type "string", or types derived from "string".

A "length" statement restricts the number of 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 an already length restricted type, the new restriction MUST be equal or more limiting, that is, 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" means the minimum and maximum length 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 11.

8.3.3.1. The length's Substatements

substatement	section	cardinality
description	7.17.3	0..1
error-app-tag	7.5.3.2	0..1
error-message	7.5.3.1	0..1
reference	7.17.4	0..1

8.3.4. Usage Example

```
     typedef my-base-str-type {
         type string {
             length "1..255";
         }
     }

     type my-base-str-type {
         // legal length refinement
         length "11 | 42..max"; // 11 | 42..255
     }

     type my-base-str-type {
         // illegal length refinement
         length "1..999";
     }
```

8.3.5. 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 completely matches the pattern.

If the type has multiple "pattern" statements, the expressions are AND:ed together, i.e. all such expressions have to match.

8.3.5.1. The pattern's Substatements

substatement	section	cardinality
description	7.17.3	0..1
error-app-tag	7.5.3.2	0..1
error-message	7.5.3.1	0..1
reference	7.17.4	0..1

8.3.6. Usage Example

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
  xx00        // illegal
```

8.4. The boolean Built-in Type

The boolean built-in type represents a boolean value.

8.4.1. Lexicographic Representation

The lexicographical representation of a boolean value is the strings "true" and "false".

8.4.2. Restrictions

A boolean cannot be restricted.

8.5. The enumeration Built-in Type

The enumeration built-in type represents values from a set of assigned names.

8.5.1. Lexicographic Representation

The lexicographical representation of an enumeration value is the assigned name string.

8.5.2. Restrictions

An enumeration cannot be restricted.

8.5.3. 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 which is the assigned name. It is optionally followed by a block of substatements which holds detailed enum information.

All assigned names in an enumeration MUST be unique.

8.5.3.1. The enum's Substatements

substatement	section	cardinality
description	7.17.3	0..1
reference	7.17.4	0..1
status	7.17.2	0..1
value	8.5.3.2	0..1

8.5.3.2. 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 sub-statement is the first one defined, the assigned value is zero (0), otherwise the assigned value is one greater than the current highest enum value.

If the current highest value is equal to 2147483647, then an enum value MUST be specified for enum sub-statements following the one with the current highest value.

8.5.4. Usage Example

```
type enumeration {
    enum enabled {
        value 1;
    }
    enum disabled {
        value 2;
    }
}


type enumeration {
    enum zero;
    enum one;
    enum seven {
        value 7;
    }
}
```

8.6. 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.

8.6.1. Restrictions

A bits type cannot be restricted.

8.6.2. Lexicographic Representation

The lexicographical representation of the bits type is a space separated list of the individual bit values that are set. An empty string thus represents a value where no bits are set.

8.6.3. 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 which is the assigned name of the bit. It is followed by a block of substatements which holds detailed bit information. A bit name follows the same syntax rules as an identifier (see Section 6.2).

All bit names in a bits type MUST be unique.

8.6.3.1. The bit's Substatements

substatement	section	cardinality
description	7.17.3	0..1
reference	7.17.4	0..1
status	7.17.2	0..1
position	8.6.3.2	0..1

8.6.3.2. The position Statement

The "position" statement, which is optional, takes as an argument a non-negative integer value which 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. The value is unused by YANG and the XML encoding, but is carried as a convenience to implementors.

If a bit position is not specified, then one will be automatically assigned. If the bit sub-statement is the first one defined, the assigned value is zero (0), otherwise the assigned value is one greater than the current highest bit position.

If the current highest bit position value is equal to 4294967295, then a position value MUST be specified for bit sub-statements following the one with the current highest position value.

8.6.4. Usage Example

Given the following type:

```
  leaf mybits {
      type bits {
          bit disable-nagle {
              position 0;
          }
          bit auto-sense-speed {
              position 1;
          }
          bit 10-Mb-only {
              position 2;
          }
      }
      default "auto-sense-speed";
  }
```

The lexicographic representation of this leaf with bit values disable-nagle and 10-Mb-only set would be:

```
  <mybits>disable-nagle 10-Mb-only</mybits>
```

8.7. The binary Built-in Type

The binary built-in type represents any binary data, i.e. a sequence of octets.

8.7.1. Restrictions

A binary can be restricted with the "length" (Section 8.3.3) statement. The length of a binary value is the number of octets it contains.

8.7.2. Lexicographic Representation

Binary values are encoded with the base64 encoding scheme [RFC4648].

8.8. The keyref Built-in Type

The keyref type is used to reference a particular list entry in the data tree. Its value is constrained to be the same as the key of an existing list entry.

If the leaf with the keyref type represents configuration, the list entry it refers to MUST also represent configuration. Such a leaf puts a constraint on a valid configuration. In a valid configuration, all keyref nodes MUST reference existing list entries.

8.8.1. Restrictions

A keyref cannot be restricted.

8.8.2. The path Statement

The "path" statement, which is a substatement to the "type" statement, MUST be present if the type is "keyref". It takes as an argument a string which MUST refer to one key node of a list entry.

The syntax for a path argument is a subset of the XPath syntax. It is an absolute or relative XPath location path in abbreviated syntax, where axes are not permitted, and predicates are used only for constraining the values for the key nodes for list entries. Each predicate consists of at most one equality test per key.

The predicates are only used when more than one key reference is needed to uniquely identify a list entry. This occurs if the list has multiple keys, or a reference to a list within a list is needed. In these cases, multiple keyref leafs are typically specified, and predicates are used to tie them together.

The syntax is formally defined by the rule "path-arg" in Section 11.

8.8.3. Lexicographic Representation

A keyref value is encoded the same way as the key it references.

8.8.4. Usage Example

With the following list:

```
     list interface {
         key "name";
         leaf name {
             type string;
         }
         list address {
             key "ip";
             leaf ip {
                 type yang:ip-address;
             }
         }
     }
```

The following keyref refers to an existing interface:

```
     leaf mgmt-interface {
         type keyref {
             path "../interface/name";
         }
     }
```

A corresponding XML snippet is e.g.:

```
     <interface>
       <name>eth0</name>
     </interface>
     <interface>
       <name>lo</name>
     </interface>

     <mgmt-interface>eth0</mgmt-interface>
```

The following keyrefs refer to an existing address of an interface:

```
     container default-address {
         leaf ifname {
             type keyref {
                 path "../../interface/name";
             }
         }
         leaf address {
             type keyref {
                 path "../../interface[name = current()/../ifname]"
                    + "/address/ip";
             }
         }
     }
```

A corresponding XML snippet is e.g.:

```
     <interface>
       <name>eth0</name>
       <address>
         <ip>192.0.2.1</ip>
       </address>
       <address>
         <ip>192.0.2.2</ip>
       </address>
     </interface>
     <interface>
       <name>lo</name>
       <address>
         <ip>127.0.0.1</ip>
       </address>
     </interface>

     <default-address>
       <ifname>eth0</ifname>
       <address>192.0.2.2</address>
     </default-address>
```

8.9. 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.

8.9.1. Restrictions

An empty type cannot be restricted.

8.9.2. Lexicographic Representation

Not applicable.

8.9.3. Usage Example

The following leaf

```
     leaf enable-qos {
         type empty;
     }
```

will be encoded as

```
     <enable-qos/>
```

if it exists.

8.10. 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 used to repeatedly specify each member type of the union. It takes as an argument a string which is the name of a member type.

A member type can be of any built-in or derived type, except it MUST NOT be one of the built-in types "empty" or "keyref".

Example:

```
     type union {
         type int32;
         type enumeration {
             enum "unbounded";
         }
     }
```

8.10.1. Restrictions

A union can not be restricted. However, each member type can be restricted, based on the rules defined in Section 8 chapter.

8.10.2. Lexicographic Representation

The lexicographical representation of an union is a value that corresponds to the representation of any one of the member types.

8.11. 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 syntax, which is used to uniquely identify a node in the data tree. It is an absolute XPath location path in abbreviated syntax, where axes are not permitted, and predicates are used only for specifying the values for the key nodes for list entries, or a value of a leaf-list. Each predicate consists of one equality test per key. Each key MUST have a corresponding predicate.

The syntax is formally defined by the rule "absolute-instid" in Section 11.

8.11.1. Restrictions

An instance-identifier cannot be restricted.

8.11.2. Lexicographic Representation

An instance-identifier value is lexicographically represented as a string in the XML encoding. The namespace prefixes used in the encoding MUST be declared in the XML namespace scope in the instance-idenfitier'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.

8.11.3. Usage Example

The following are examples of instance identifiers:

```
  /ex:system/ex:services/ex:ssh/ex:port

  /ex:system/ex:user[ex:name='fred']

  /ex:system/ex:user[ex:name='fred']/ex:type

  /ex:system/ex:server[ex:ip='192.0.2.1'][ex:port='80']

  /ex:system/ex:services/ex:ssh/ex:cipher[.='blowfish-cbc']
```

9. Updating a Module

[Editor's Note: add versioning rules, i.e. what can be done w/o changing the module name and the namespace]

10. YIN

A YANG module can be specified in an alternative XML-based syntax called YIN. This appendix describes symmetric mapping rules between the two formats.

The YANG and YIN formats contain equivalent information using different notations. The purpose of the YIN notation is to allow the user to translate YANG into YIN, use the rich set of XML based tools on the YIN format to transform, or filter the model information. Tools like XSLT or XML validators can be utilized. After this the model can be transformed back to the YANG format if needed, which provides a more concise and readable format.

The YANG-2-YIN and the YIN-2-YANG transformations will not modify the information content of the model.

10.1. Formal YIN Definition

YIN is described by an algorithm that transforms YANG to YIN.

10.2. Transformation Algorithm YANG-2-YIN

Every keyword results in a new XML element. The name of the element is the keyword. All core YANG elements are defined in the namespace "urn:ietf:params:xml:ns:yang:yin:1". [XXX IANA]

The top-level element is always <module> or <submodule>.

Elements which represent keywords that are imported extensions from other modules MUST be properly namespace qualified, where the namespace is the namespace of the imported module. Translators SHOULD use the same prefix as used in the YANG module.

If the keyword has an argument, its encoding depends on the value of the argument's "yin-element". If "yin-element" is false, the argument is encoded as an XML attribute to the keyword's element. If "yin-element" is true, the argument is encoded as a subelement to the keyword's element. The name of the attribute or element is the name of the argument.

The core YANG keywords have arguments according to the table below. Extension keywords have arguments according to Section 7.16.2.

YANG to YIN keyword map

keyword	argument name	yin-element
anyxml	name	false
argument	name	false
augment	target-node	false
belongs-to	module	false
bit	name	false
case	name	false
choice	name	false
config	value	false
contact	info	true
container	name	false
default	value	false
description	text	true
enum	name	false
error-app-tag	value	false
error-message	value	true
extension	name	false
grouping	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
module	name	false
must	condition	false
namespace	uri	false
notification	name	false
ordered-by	value	false
organization	info	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	info	false
revision	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 30

If a statement is followed by substatements, those substatements are subelements in the YIN mapping.

Comments in YANG MAY be transformed into XML comments.

10.2.1. Usage Example

The following YANG snippet:

```
leaf mtu {
    type uint32;
    description "The MTU of the interface.";
}
```

is translated into the following YIN snippet:

```
<leaf name="mtu">
  <type name="uint32"/>
  <description>
    <text>The MTU of the interface."</text>
  </description>
</leaf>
```

10.3. Transformation Algorithm YIN-2-YANG

The transformation is based on a recursive algorithm that is started on the <module> or <submodule> element.

The element is transformed into a YANG keyword. If the keyword in Table 30 is marked as yin-element true, the subelement with the keyword's argument name in Table 30 contains the YANG keyword's argument as text content. If the keyword in Table 30 is marked as yin-element false, the element's attribute with keyword's argument name in Table 30 contains the YANG keyword's argument.

If there are no other subelements to the element, the YANG statement is closed with a ";". Otherwise, each such subelement is transformed, according to the same algorithm, as substatements to the current YANG statement, enclosed within "{" and "}".

XML comments in YIN MAY be transformed into YANG comments.

10.3.1. Tabulation, Formatting

To get a readable YANG module the YANG output will have to be indented with appropriate whitespace characters.

11. YANG ABNF Grammar

In YANG, almost all statements are unordered. The ABNF grammar [RFC5234] 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.

```
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 stmtsep]
                           namespace-stmt stmtsep
                           prefix-stmt stmtsep

submodule-header-stmts = ;; these stmts can appear in any order
                          [yang-version-stmt stmtsep]
                           belongs-to-stmt stmtsep

meta-stmts              = ;; these stmts can appear in any order
                          [organization-stmt stmtsep]
                          [contact-stmt stmtsep]
                          [description-stmt stmtsep]
                          [reference-stmt stmtsep]

linkage-stmts           = ;; these stmts can appear in any order
```

```
                                *(import-stmt stmtsep)
                                *(include-stmt stmtsep)

   revision-stmts            = *(revision-stmt stmtsep)

   body-stmts                = *((extension-stmt /
                                   typedef-stmt /
                                   grouping-stmt /
                                   data-def-stmt /
                                   rpc-stmt /
                                   notification-stmt) stmtsep)

   data-def-stmt             = container-stmt /
                                leaf-stmt /
                                leaf-list-stmt /
                                list-stmt /
                                choice-stmt /
                                anyxml-stmt /
                                uses-stmt /
                                augment-stmt

   case-data-def-stmt        = container-stmt /
                                leaf-stmt /
                                leaf-list-stmt /
                                list-stmt /
                                anyxml-stmt /
                                uses-stmt /
                                augment-stmt

   yang-version-stmt         = yang-version-keyword sep yang-version-arg-str
                                optsep stmtend

   yang-version-arg-str      = < a string which matches the rule
                                  yang-version-arg >

   yang-version-arg          = "1"

   import-stmt               = import-keyword sep identifier-arg-str optsep
                                "{" stmtsep
                                    prefix-stmt stmtsep
                                "}"

   include-stmt              = include-keyword sep identifier-arg-str optsep
                                stmtend

   namespace-stmt            = namespace-keyword sep uri-str optsep stmtend

   uri-str                   = < a string which matches the rule
```

```
                             URI in RFC 3986 >

   prefix-stmt            = prefix-keyword sep prefix-arg-str
                              optsep stmtend

   belongs-to-stmt        = belongs-to-keyword sep identifier-arg-str
                              optsep stmtend

   organization-stmt      = organization-keyword sep string
                              optsep stmtend

   contact-stmt           = contact-keyword sep string optsep stmtend

   description-stmt       = description-keyword sep string optsep
                              stmtend

   reference-stmt         = reference-keyword sep string optsep stmtend

   units-stmt             = units-keyword sep string optsep stmtend

   revision-stmt          = revision-keyword sep date-arg-str optsep
                            (";" /
                             "{" stmtsep
                                 [description-stmt stmtsep]
                             "}")

   extension-stmt         = extension-keyword sep identifier-arg-str optsep
                            (";" /
                             "{" stmtsep
                                ;; these stmts can appear in any order
                                [argument-stmt stmtsep]
                                [status-stmt stmtsep]
                                [description-stmt stmtsep]
                                [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 which matches the rule
                              yin-element-arg >
```

```
yin-element-arg        = true-keyword / false-keyword

typedef-stmt           = typedef-keyword sep identifier-arg-str optsep
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             type-stmt stmtsep
                             [units-stmt stmtsep]
                             [default-stmt stmtsep]
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                          "}"

type-stmt              = type-keyword sep identifier-ref-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ( numerical-restrictions /
                                string-restrictions /
                                enum-specification /
                                keyref-specification /
                                bits-specification /
                                union-specification )
                              stmtsep
                          "}")

numerical-restrictions = range-stmt stmtsep

range-stmt             = range-keyword sep range-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [error-message-stmt stmtsep]
                              [error-app-tag-stmt stmtsep]
                              [description-stmt stmtsep]
                              [reference-stmt stmtsep]
                           "}")

string-restrictions    = ;; these stmts can appear in any order
                         [length-stmt stmtsep]
                         *(pattern-stmt stmtsep)

length-stmt            = length-keyword sep length-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [error-message-stmt stmtsep]
                              [error-app-tag-stmt stmtsep]
                              [description-stmt stmtsep]
```

```
                              [reference-stmt stmtsep]
                           "}")

   pattern-stmt           = pattern-keyword sep string optsep
                            (";" /
                             "{" stmtsep
                                 ;; these stmts can appear in any order
                                 [error-message-stmt stmtsep]
                                 [error-app-tag-stmt stmtsep]
                                 [description-stmt stmtsep]
                                 [reference-stmt stmtsep]
                              "}")

   default-stmt           = default-keyword sep string stmtend

   enum-specification     = 1*(enum-stmt stmtsep)

   enum-stmt              = enum-keyword sep identifier-arg-str optsep
                            (";" /
                             "{" stmtsep
                                 ;; these stmts can appear in any order
                                 [value-stmt stmtsep]
                                 [status-stmt stmtsep]
                                 [description-stmt stmtsep]
                                 [reference-stmt stmtsep]
                              "}")

   keyref-specification   = path-stmt stmtsep

   path-stmt              = path-keyword sep path-arg-str stmtend

   union-specification    = 1*(type-stmt stmtsep)

   bits-specification     = 1*(bit-stmt stmtsep)

   bit-stmt               = bit-keyword sep identifier-arg-str optsep
                            (";" /
                             "{" stmtsep
                                 ;; these stmts can appear in any order
                                 [position-stmt stmtsep]
                                 [status-stmt stmtsep]
                                 [description-stmt stmtsep]
                                 [reference-stmt stmtsep]
                               "}"
                             "}")

   position-stmt          = position-keyword sep
                            position-value-arg-str stmtend
```

```
position-value-arg-str = < a string which matches the rule
                           position-value-arg >

position-value-arg     = non-negative-decimal-value

status-stmt            = status-keyword sep status-arg-str stmtend

status-arg-str         = < a string which 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 which 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 which 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 which 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 stmtsep]
                              [error-app-tag-stmt stmtsep]
                              [description-stmt stmtsep]
                              [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 which matches the rule
                           min-value-arg >

min-value-arg          = non-negative-decimal-value

max-elements-stmt      = max-elements-keyword sep
                         max-value-arg-str stmtend;

max-value-arg-str      = < a string which matches the rule
                           max-value-arg >

max-value-arg          = unbounded-keyword /
                         positive-decimal-value

value-stmt             = value-keyword sep decimal-value stmtend

grouping-stmt          = grouping-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *((typedef-stmt /
                                grouping-stmt) stmtsep)
                             *(data-def-stmt stmtsep)
                          "}")

container-stmt         = container-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             *(must-stmt stmtsep)
                             [presence-stmt stmtsep]
                             [config-stmt stmtsep]
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *((typedef-stmt /
```

```
                                  grouping-stmt) stmtsep)
                               *(data-def-stmt stmtsep)
                            "}")

   leaf-stmt              = leaf-keyword sep identifier-arg-str optsep
                              "{" stmtsep
                                  ;; these stmts can appear in any order
                                  type-stmt stmtsep
                                  [units-stmt stmtsep]
                                  *(must-stmt stmtsep)
                                  [default-stmt stmtsep]
                                  [config-stmt stmtsep]
                                  [mandatory-stmt stmtsep]
                                  [status-stmt stmtsep]
                                  [description-stmt stmtsep]
                                  [reference-stmt stmtsep]
                               "}"

   leaf-list-stmt         = leaf-list-keyword sep identifier-arg-str optsep
                              "{" stmtsep
                                  ;; these stmts can appear in any order
                                  type-stmt stmtsep
                                  [units-stmt stmtsep]
                                  *(must-stmt stmtsep)
                                  [config-stmt stmtsep]
                                  [min-elements-stmt stmtsep]
                                  [max-elements-stmt stmtsep]
                                  [ordered-by-stmt stmtsep]
                                  [status-stmt stmtsep]
                                  [description-stmt stmtsep]
                                  [reference-stmt stmtsep]
                               "}"

   list-stmt              = list-keyword sep identifier-arg-str optsep
                              "{" stmtsep
                                  ;; these stmts can appear in any order
                                  *(must-stmt stmtsep)
                                  [key-stmt stmtsep]
                                  *(unique-stmt stmtsep)
                                  [config-stmt stmtsep]
                                  [min-elements-stmt stmtsep]
                                  [max-elements-stmt stmtsep]
                                  [ordered-by-stmt stmtsep]
                                  [status-stmt stmtsep]
                                  [description-stmt stmtsep]
                                  [reference-stmt stmtsep]
                                  *((typedef-stmt /
                                     grouping-stmt) stmtsep)
```

```
                             1*(data-def-stmt stmtsep)
                          "}"

key-stmt               = key-keyword sep key-arg-str stmtend

key-arg-str            = < a string which matches the rule
                           key-arg >

key-arg                = identifier *(sep identifier)

unique-stmt            = unique-keyword sep unique-arg-str stmtend

unique-arg-str         = < a string which 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
                             [default-stmt stmtsep]
                             [config-stmt stmtsep]
                             [mandatory-stmt stmtsep]
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *((short-case-stmt / case-stmt) stmtsep)
                          "}")

short-case-stmt        = container-stmt /
                         leaf-stmt /
                         leaf-list-stmt /
                         list-stmt /
                         anyxml-stmt

case-stmt              = case-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *(case-data-def-stmt stmtsep)
                          "}")

anyxml-stmt            = anyxml-keyword sep identifier-arg-str optsep
```

```
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [config-stmt stmtsep]
                              [mandatory-stmt stmtsep]
                              [status-stmt stmtsep]
                              [description-stmt stmtsep]
                              [reference-stmt stmtsep]
                           "}")

uses-stmt              = uses-keyword sep identifier-ref-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *(refinement-stmt stmtsep)
                          "}")

refinement-stmt        = refine-container-stmt /
                         refine-leaf-stmt /
                         refine-leaf-list-stmt /
                         refine-list-stmt /
                         refine-choice-stmt /
                         refine-anyxml-stmt

refine-leaf-stmt       = leaf-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             *(must-stmt stmtsep)
                             [default-stmt stmtsep]
                             [config-stmt stmtsep]
                             [mandatory-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                          "}")

refine-leaf-list-stmt  = leaf-list-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             *(must-stmt stmtsep)
                             [config-stmt stmtsep]
                             [min-elements-stmt stmtsep]
                             [max-elements-stmt stmtsep]
                             [description-stmt stmtsep]
```

```
                             [reference-stmt stmtsep]
                          "}")

refine-list-stmt       = list-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             *(must-stmt stmtsep)
                             [config-stmt stmtsep]
                             [min-elements-stmt stmtsep]
                             [max-elements-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *(refinement-stmt stmtsep)
                          "}")

refine-choice-stmt     = choice-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                             ;; these stmts can appear in any order
                             [default-stmt stmtsep]
                             [mandatory-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *(refine-case-stmt stmtsep)
                          "}")

refine-case-stmt       = case-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                              *(refinement-stmt stmtsep)
                          "}")


refine-container-stmt  = container-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             *(must-stmt stmtsep)
                             [presence-stmt stmtsep]
                             [config-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             *(refinement-stmt stmtsep)
                          "}")
```

```
refine-anyxml-stmt     = anyxml-keyword sep identifier-arg-str optsep
                        (";" /
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             [config-stmt stmtsep]
                             [mandatory-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                          "}")

unknown-statement      = prefix ":" identifier [sep string] optsep
                         (";" / "{" *unknown-statement "}")

augment-stmt           = augment-keyword sep augment-arg-str optsep
                         "{" stmtsep
                             ;; these stmts can appear in any order
                             [when-stmt stmtsep]
                             [status-stmt stmtsep]
                             [description-stmt stmtsep]
                             [reference-stmt stmtsep]
                             1*((data-def-stmt stmtsep) /
                                (case-stmt stmtsep))
                          "}"

augment-arg-str        = < a string which matches the rule
                           augment-arg >

augment-arg            = absolute-schema-nodeid /
                         descendant-schema-nodeid

when-stmt              = when-keyword sep string stmtend

rpc-stmt               = rpc-keyword sep identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [status-stmt stmtsep]
                              [description-stmt stmtsep]
                              [reference-stmt stmtsep]
                              *((typedef-stmt /
                                 grouping-stmt) stmtsep)
                              [input-stmt stmtsep]
                              [output-stmt stmtsep]
                          "}")

input-stmt             = input-keyword optsep
                         "{" stmtsep
                              ;; these stmts can appear in any order
```

```
                             *((typedef-stmt /
                                grouping-stmt) stmtsep)
                             1*(data-def-stmt stmtsep)
                         "}"

   output-stmt           = output-keyword optsep
                         "{" stmtsep
                              ;; these stmts can appear in any order
                             *((typedef-stmt /
                                grouping-stmt) stmtsep)
                             1*(data-def-stmt stmtsep)
                         "}"

   notification-stmt     = notification-keyword sep
                         identifier-arg-str optsep
                         (";" /
                          "{" stmtsep
                              ;; these stmts can appear in any order
                              [status-stmt stmtsep]
                              [description-stmt stmtsep]
                              [reference-stmt stmtsep]
                              *((typedef-stmt /
                                 grouping-stmt) stmtsep)
                              *(data-def-stmt stmtsep)
                          "}")

   ;; Ranges

   range-arg-str         = < a string which matches the rule
                           range-arg >

   range-arg             = range-part *(optsep "|" optsep range-part)

   range-part            = range-boundary
                         [optsep ".." optsep range-boundary]

   range-boundary        = neginf-keyword / posinf-keyword /
                         min-keyword / max-keyword /
                         decimal-value / float-value

   ;; Lengths

   length-arg-str        = < a string which 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-decimal-value

;; Date

date-arg-str           = < a string which matches the rule
                           date-arg >

date-arg               = 4DIGIT "-" 2DIGIT "-" 2DIGIT

;; Schema Node Identifiers

schema-nodeid          = absolute-schema-nodeid /
                         relative-schema-nodeid

absolute-schema-nodeid
                       = 1*("/" node-identifier)

relative-schema-nodeid
                       = descendant-schema-nodeid /
                         (("." / "..") "/"
                         *relative-schema-nodeid)

descendant-schema-nodeid
                       = node-identifier
                         absolute-schema-nodeid

node-identifier        = [prefix ":"] identifier


;; Instance Identifiers

instance-identifier-str
                       = < a string which matches the rule
                           instance-identifier >

instance-identifier    = absolute-instid / relative-instid

absolute-instid        = 1*("/" (node-identifier *predicate))

relative-instid        = descendant-instid /
                         (("." / "..") "/"
                          *relative-instid)

descendant-instid      = node-identifier *predicate
                         absolute-instid
```

```
predicate              = "[" *WSP predicate-expr *WSP "]"

predicate-expr         = (node-identifier / ".") *WSP "=" *WSP
                         ((DQUOTE string DQUOTE) /
                          (SQUOTE string SQUOTE))

;; keyref path

path-arg-str           = < a string which matches the rule
                           path-arg >

path-arg               = absolute-path / relative-path

absolute-path          = 1*("/" (node-identifier *path-predicate))

relative-path          = descendant-path /
                         (".." "/"
                         *relative-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          = this-variable-keyword "/" rel-path-keyexpr

rel-path-keyexpr       = 1*(".." "/") *(node-identifier "/")
                         node-identifier

;;; Keywords, using abnfgen's syntax for case-sensitive strings

;; statment keywords
anyxml-keyword         = 'anyxml'
argument-keyword       = 'argument'
augment-keyword        = 'augment'
belongs-to-keyword     = 'belongs-to'
bit-keyword            = 'bit'
case-keyword           = 'case'
choice-keyword         = 'choice'
config-keyword         = 'config'
contact-keyword        = 'contact'
container-keyword      = 'container'
default-keyword        = 'default'
description-keyword    = 'description'
enum-keyword           = 'enum'
error-app-tag-keyword  = 'error-app-tag'
```

```
error-message-keyword  = 'error-message'
extension-keyword      = 'extension'
grouping-keyword       = 'grouping'
import-keyword         = 'import'
include-keyword        = 'include'
input-keyword          = 'input'
key-keyword            = 'key'
leaf-keyword           = 'leaf'
leaf-list-keyword      = 'leaf-list'
length-keyword         = 'length'
list-keyword           = 'list'
mandatory-keyword      = 'mandatory'
max-elements-keyword   = 'max-elements'
min-elements-keyword   = 'min-elements'
module-keyword         = 'module'
must-keyword           = 'must'
namespace-keyword      = 'namespace'
notification-keyword   = 'notification'
ordered-by-keyword     = 'ordered-by'
organization-keyword   = 'organization'
output-keyword         = 'output'
path-keyword           = 'path'
pattern-keyword        = 'pattern'
position-keyword       = 'position'
prefix-keyword         = 'prefix'
presence-keyword       = 'presence'
range-keyword          = 'range'
reference-keyword      = 'reference'
revision-keyword       = 'revision'
rpc-keyword            = 'rpc'
status-keyword         = 'status'
submodule-keyword      = 'submodule'
type-keyword           = 'type'
typedef-keyword        = 'typedef'
unique-keyword         = 'unique'
units-keyword          = 'units'
uses-keyword           = 'uses'
value-keyword          = 'value'
when-keyword           = 'when'
yang-version-keyword   = 'yang-version'
yin-element-keyword    = 'yin-element'

;; other keywords

current-keyword        = 'current'
deprecated-keyword     = 'deprecated'
false-keyword          = 'false'
max-keyword            = 'max'
```

```
min-keyword            = 'min'
nan-keyword            = 'NaN'
neginf-keyword         = '-INF'
obsolete-keyword       = 'obsolete'
posinf-keyword         = 'INF'
system-keyword         = 'system'
this-variable-keyword  = '$this'
true-keyword           = 'true'
unbounded-keyword      = 'unbounded'
user-keyword           = 'user'

;; Basic Rules

keyword                = [prefix ":"] identifier

prefix-arg-str         = < a string which matches the rule
                           prefix-arg >

prefix-arg             = prefix

prefix                 = identifier

identifier-arg-str     = < a string which matches the rule
                           identifier-arg >

identifier-arg         = identifier

identifier             = (ALPHA / "_")
                         *(ALPHA / DIGIT / "_" / "-" / ".")

identifier-ref-arg-str = < a string which matches the rule
                           identifier-ref-arg >

identifier-ref-arg     = [prefix ":"] identifier

string                 = < an unquoted string as returned by
                           the scanner >

decimal-value          = ("-" non-negative-decimal-value)  /
                          non-negative-decimal-value

non-negative-decimal-value = "0" / positive-decimal-value

positive-decimal-value = (non-zero-digit *DIGIT)

zero-decimal-value     = 1*DIGIT

stmtend                = ";" / "{" *unknown-statement "}"
```

```
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

float-value            = neginf-keyword /
                          posinf-keyword /
                          nan-keyword /
                          decimal-value "." zero-decimal-value
                             *1("E" ("+"/"-") zero-decimal-value)

SQUOTE                 = %x27
                          ; ' (Single Quote)

;;
;; RFC 4234 core rules.
;;

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)

HEXDIG                 = DIGIT /
                          %x61 / %x62 / %x63 / %x64 / %x65 / %x66
                          ; only lower-case a..f

HTAB                   = %x09
                          ; horizontal tab

LF                     = %x0A
                          ; linefeed
```

```
SP                      = %x20
                          ; space

VCHAR                   = %x21-7E
                          ; visible (printing) characters

WSP                     = SP / HTAB
                          ; white space
```

12. 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.

12.1. Error Message for Data that Violates a YANG unique Statement:

If a NETCONF operation would result in configuration data where a unique constraint is invalidated, the following error is returned:

```
  Tag:             operation-failed
  Error-app-tag:   data-not-unique
  Error-info:      <non-unique>: Contains an instance identifier which
                   points to a leaf which invalidates the unique
                   constraint. This element is present once for each
                   leaf invalidating the unique constraint.

                   The <non-unique> element is in the YANG
                   namespace ("urn:ietf:params:xml:ns:yang:1"
                   [XXX IANA]).
```

12.2. Error Message for Data that Violates a YANG 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 is returned:

```
  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 are more than one extra child present.

12.3. Error Message for Data that Violates a YANG 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 is returned:

```
  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 are more than one child missing.

12.4. Error Message for Data that Violates a YANG must statement:

If a NETCONF operation would result in configuration data where the restrictions imposed by a "must" statement is violated the following error is returned, unless a specific "error-app-tag" substatement is present for the "must" statement.

Tag:	operation-failed
Error-app-tag:	must-violation

12.5. 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 a non-existing instance, the following error is returned:

Tag:	bad-attribute
Error-app-tag:	missing-instance

13. IANA Considerations

This document registers two URIs for the YANG XML namespace in the IETF XML registry [RFC3688].

```
URI: urn:ietf:params:xml:ns:yang:yin:1

URI: urn:ietf:params:xml:ns:yang:1
```

14. 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.

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.

YANG is dependent upon:

- the security of the transmission infrastructure used to send sensitive information

- the security of applications which store or release such sensitive information.

- adequate authentication and access control mechanisms to restrict the usage of sensitive data.

15. Contributors

The following people all contributed significantly to the initial YANG draft:

- Andy Bierman (andybierman.com)
- Balazs Lengyel (Ericsson)
- David Partain (Ericsson)
- Juergen Schoenwaelder (Jacobs University Bremen)
- Phil Shafer (Juniper Networks)

16. References

16.1. Normative References

[IEEE.754]
Institute of Electrical and Electronics Engineers, "Standard for Binary Floating-Point Arithmetic", IEEE Standard 754, August 1985.

[ISO.10646]
International Organization for Standardization, "Information Technology - Universal Multiple-octet coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", ISO Standard 10646-1, May 1993.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC4741] Enns, R., "NETCONF Configuration Protocol", RFC 4741, December 2006.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5277] Chisholm, S. and H. Trevino, "NETCONF Event Notifications", RFC 5277, July 2008.

[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 V. and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition", W3C REC REC-xmlschema-2-20041028,

October 2004.

16.2. Non-Normative References

[RFC2578] McCloghrie, K., Ed., Perkins, D., Ed., and J. Schoenwaelder, Ed., "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, April 1999.

[RFC2579] McCloghrie, K., Ed., Perkins, D., Ed., and J. Schoenwaelder, Ed., "Textual Conventions for SMIv2", STD 58, RFC 2579, April 1999.

[RFC3780] Strauss, F. and J. Schoenwaelder, "SMIng - Next Generation Structure of Management Information", RFC 3780, May 2004.

Appendix A. ChangeLog

A.1. Version -01

- Removed "Appendix A. Derived YANG Types".
- Removed "Appendix C. XML Schema Considerations".
- Removed "Appendix F. Why We Need a New Modeling Language".
- Moved "Appendix B. YIN" to its own section.
- Moved "Appendix D. YANG ABNF Grammar" to its own section.
- Moved "Appendix E. Error Responses for YANG Related Errors" into its own section.
- The "input" and "output" nodes are now implicitly created by the "rpc" statement, in order for augmentation of these nodes to work correctly.
- Allow "config" in "choice".
- Added reference to XPath 1.0.
- Using an XPath function "current()" instead of the variable "$this".
- Clarified that a "must" expression in a configuration node must not reference non-configuration nodes.
- Added XML encoding rules and usage examples for rpc and notification.
- Removed requirement that refinements are specified in the same order as in the original grouping's definition.
- Fixed whitespace issues in the ABNF grammar.
- Added the term "mandatory node", and refer to it in the description of augment (see Section 7.15), and choice (see Section 7.9.3).
- Added support for multiple "pattern" statements in "type".
- Several clarifications and fixed typos.

A.2. Version -00

Changes from draft-bjorklund-netconf-yang-02.txt

- Fixed bug in grammar for bit-stmt
- Fixed bugs in example XPath expressions
- Added keyword 'presence' to the YIN mapping table

Author's Address

Martin Bjorklund (editor)
Tail-f Systems

Email: mbj@tail-f.com

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).