

Network Working Group  
Internet-Draft  
Expires: January 18, 2002

F. Strauss  
J. Schoenwaelder  
TU Braunschweig  
July 20, 2001

**SMIng - Next Generation Structure of Management Information  
draft-ietf-sming-02**

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of [Section 10 of RFC2026](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/1id-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on January 18, 2002.

Copyright Notice

Copyright (C) The Internet Society (2001). All Rights Reserved.

Abstract

This memo presents an object-oriented data definition language for the specification of various kinds of management information. It is independent of management protocols and applications. Protocol mappings are defined as extensions to this language in separate memos. The language builds on experiences gained with the SMIV2 and its derivative SPPI. It is expected that the language presented in this memo along with its protocol mappings will replace the SMIV2 and the SPPI in the long term.



Table of Contents

- [1.](#) Introduction . . . . . [4](#)
- [2.](#) SMIng Data Modelling . . . . . [5](#)
- [2.1](#) Identifiers . . . . . [6](#)
- [3.](#) Base Types and Derived Types . . . . . [7](#)
- [3.1](#) OctetString . . . . . [7](#)
- [3.2](#) Pointer . . . . . [8](#)
- [3.3](#) Object Identifier . . . . . [9](#)
- [3.4](#) Integer32 . . . . . [10](#)
- [3.5](#) Integer64 . . . . . [10](#)
- [3.6](#) Unsigned32 . . . . . [11](#)
- [3.7](#) Unsigned64 . . . . . [12](#)
- [3.8](#) Float32 . . . . . [13](#)
- [3.9](#) Float64 . . . . . [14](#)
- [3.10](#) Float128 . . . . . [15](#)
- [3.11](#) Enumeration . . . . . [16](#)
- [3.12](#) Bits . . . . . [17](#)
- [3.13](#) Display Formats . . . . . [18](#)
- [4.](#) The SMIng File Structure . . . . . [20](#)
- [4.1](#) Comments . . . . . [20](#)
- [4.2](#) Statements and Arguments . . . . . [20](#)
- [5.](#) The module Statement . . . . . [20](#)
- [5.1](#) The module's import Statement . . . . . [21](#)
- [5.2](#) The module's organization Statement . . . . . [22](#)
- [5.3](#) The module's contact Statement . . . . . [22](#)
- [5.4](#) The module's description Statement . . . . . [22](#)
- [5.5](#) The module's reference Statement . . . . . [22](#)
- [5.6](#) The module's revision Statement . . . . . [22](#)
- [5.6.1](#) The revision's date Statement . . . . . [22](#)
- [5.6.2](#) The revision's description Statement . . . . . [23](#)
- [5.7](#) Usage Example . . . . . [23](#)
- [6.](#) The extension Statement . . . . . [24](#)
- [6.1](#) The extension's status Statement . . . . . [24](#)
- [6.2](#) The extension's description Statement . . . . . [24](#)
- [6.3](#) The extension's reference Statement . . . . . [24](#)
- [6.4](#) The extension's abnf Statement . . . . . [25](#)
- [6.5](#) Usage Example . . . . . [25](#)
- [7.](#) The typedef Statement . . . . . [25](#)
- [7.1](#) The typedef's type Statement . . . . . [25](#)
- [7.2](#) The typedef's default Statement . . . . . [26](#)
- [7.3](#) The typedef's format Statement . . . . . [26](#)
- [7.4](#) The typedef's units Statement . . . . . [26](#)
- [7.5](#) The typedef's status Statement . . . . . [27](#)
- [7.6](#) The typedef's description Statement . . . . . [27](#)
- [7.7](#) The typedef's reference Statement . . . . . [27](#)
- [7.8](#) Usage Examples . . . . . [27](#)
- [8.](#) The identity Statement . . . . . [28](#)



- [8.1](#) The identity's status Statement . . . . . [29](#)
- [8.2](#) The identity' description Statement . . . . . [29](#)
- [8.3](#) The identity's reference Statement . . . . . [29](#)
- [8.4](#) Usage Examples . . . . . [30](#)
- [9.](#) The class Statement . . . . . [30](#)
- [9.1](#) The class' attribute Statement . . . . . [30](#)
- [9.1.1](#) The attribute's access Statement . . . . . [31](#)
- [9.1.2](#) The attribute's default Statement . . . . . [31](#)
- [9.1.3](#) The attribute's format Statement . . . . . [31](#)
- [9.1.4](#) The attribute's units Statement . . . . . [32](#)
- [9.1.5](#) The attribute's status Statement . . . . . [32](#)
- [9.1.6](#) The attribute's description Statement . . . . . [32](#)
- [9.1.7](#) The attribute's reference Statement . . . . . [33](#)
- [9.2](#) The class' unique Statement . . . . . [33](#)
- [9.3](#) The class' event Statement . . . . . [33](#)
- [9.3.1](#) The event's status Statement . . . . . [33](#)
- [9.3.2](#) The event's description Statement . . . . . [34](#)
- [9.3.3](#) The event's reference Statement . . . . . [34](#)
- [9.4](#) The class' status Statement . . . . . [34](#)
- [9.5](#) The class' description Statement . . . . . [34](#)
- [9.6](#) The class's reference Statement . . . . . [35](#)
- [9.7](#) Usage Example . . . . . [35](#)
- [10.](#) Extending a Module . . . . . [36](#)
- [11.](#) SMIng Language Extensibility . . . . . [37](#)
- [12.](#) Security Considerations . . . . . [39](#)
- [13.](#) Acknowledgements . . . . . [39](#)
- References . . . . . [40](#)
- Authors' Addresses . . . . . [41](#)
- [A.](#) SMIng ABNF Grammar . . . . . [41](#)
- [B.](#) OPEN ISSUES . . . . . [51](#)
- Full Copyright Statement . . . . . [54](#)



## **1. Introduction**

In traditional management systems management information is viewed as a collection of managed objects, residing in a virtual information store, termed the Management Information Base (MIB). Collections of related objects are defined in MIB modules. These modules are written conforming to a specification language, the Structure of Management Information (SMI). There are different versions of the SMI. The SMI version 1 (SMIv1) is defined in [9], [10], [11] and the SMI version 2 (SMIv2) in [5], [6], [7]. Both are based on adapted subsets of OSI's Abstract Syntax Notation One, ASN.1 [13].

In a similar fashion policy provisioning information is viewed as a collection of Provisioning Classes (PRCs) and Provisioning Instances (PRIs) residing in a virtual information store, termed the Policy Information Base (PIB). Collections of related Provisioning Classes are defined in PIB modules. PIB modules are written using the Structure of Policy Provisioning Information (SPPI) [8] which is an adapted subset of SMIv2.

The SMIv1 and the SMIv2 are bound to the Simple Network Management Protocol (SNMP) while the the SPPI is bound to the Common Open Policy Service Provisioning (COPS-PR) protocol. Even though the languages have common rules, it is hard to use common data definitions with both protocols. It is the purpose of this document to define a common object-oriented data definition language, named SMIng, that allows to formally specify data models independent of specific protocols and applications. Companion documents contain

- o core modules that supply common SMIng definitions [1][2],
- o a SMIng language extension to define SNMP specific mappings of SMIng definitions in way compatible to SMIv2 MIBs [3], and
- o a SMIng language extension to define COPS-PR specific mappings of SMIng definition in a way compatible to SPPI PIBs.

[Section 2](#) gives an overview of the basic concepts of data modelling using SMIng while the subsequent sections present the concepts of the SMIng language in detail: the base types, the SMIng file structure, and all SMIng core statements.

The remainder of the document describes extensibility features of the language and rules to follow when changes are applied to a module. [Appendix A](#) contains the grammar of SMIng in ABNF [12] notation.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this





document are to be interpreted as described in [\[4\]](#).

## 2. SMIng Data Modelling

SMIng is a language designed to specify management information in a structured way readable to computer programs, e.g. MIB compilers, as well as to human readers.

Management information is modeled in classes in an object-oriented manner. Classes can be defined from scratch or by inheritance from a parent class. Multiple inheritance is not possible. The concept of classes is described in [Section 9](#).

Each class has a number of attributes. Each attribute represents an atomic piece of information of a base type, a sub-type of a base type, or another class. The concept of attributes is described in [Section 9.1](#).

The base types of SMIng include signed and unsigned integers, octet strings, enumeration types, bitset types, and pointers. Pointers are references to class instances, attributes of class instances, or arbitrary identities. The SMIng type system is described in [Section 3](#).

Related class and type definitions are defined in modules. A module may refer to definitions from other modules by importing identifiers from those modules. Each module may serve one or multiple purposes:

- o the definition of management classes,
- o the definition of events,
- o the definition of derived types,
- o the definition of arbitrary untyped identities serving as values of pointers,
- o the definition of SMIng extensions to allow the local module or other modules to specify information beyond the scope of the base SMIng in a machine readable notation. Some extensions for the application of SMIng in the SNMP framework are defined in [\[3\]](#),
- o the definition of information beyond the scope of the base SMIng statements, based on locally defined or imported SMIng extensions.

Each module is identified by an upper-case identifier. The names of all standard modules must be unique (but different versions 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.

## **2.1 Identifiers**

Identifiers are used to identify different kinds of SMIng items by name. Each identifier is valid in a namespace which depends on the type of the SMIng item being defined:

- o The global namespace contains all module identifiers.
- o Each module defines a new namespace. A module's namespace may contain definitions of extension identifiers, derived type identifiers, identity identifiers, and class identifiers. Furthermore, a module may import identifiers of these kinds from other modules. All these identifiers are also visible within all inner namespaces of the module.
- o Each class within a module defines a new namespace. A class' namespace may contain definitions of attribute identifiers and event identifiers.
- o Each enumeration type and bitset type defines a new namespace of its named numbers. These named numbers are visible in each expression of a corresponding value, e.g., default values and sub-typing restrictions.
- o Extensions may define additional namespaces and have additional rules of other namespaces' visibility.

Within every namespace each identifier MUST be unique.

Each identifier starts with an upper-case or lower-case character, dependent on the kind of SMIng item, followed by zero or more letters, digits and hyphens.

All identifiers defined in a namespace MUST be unique and SHOULD NOT only differ in case. Identifiers MUST NOT exceed 64 characters in length. Furthermore, the set of all identifiers defined in all modules of a single standardization body or organization SHOULD be unique and mnemonic. This promotes a common language for humans to use when discussing a module.

To reference an item that is defined in the local module, its definition MUST sequentially precede the reference. Thus, there MUST NOT be any forward references.



To reference an item, that is defined in an external module it MUST be imported into the local module's namespace ([Section 5.1](#)). Identifiers that are neither defined nor imported MUST NOT be visible in the local module. On the other hand, all items defined in a module are implicitly exported.

When identifiers from external modules are referenced, there is the possibility of name collisions. As such, if different items with the same identifier are imported or if imported identifiers collide with identifiers of locally defined items, then this ambiguity is resolved by prefixing those identifiers with the names of their modules and the namespace operator `::', i.e. `Module::item'. Of course, this notation can be used to refer to identifiers even when there is no name collision.

Note that SMIng core language keywords MUST NOT be imported. See the `...Keyword' rules of the SMIng ABNF grammar in [Appendix A](#) for a list of those keywords.

### **3. Base Types and Derived Types**

SMIng has a minimal but complete set of base 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 base types or from other derived types. Derived types may use subtyping to formally restrict the set of possible values. An initial set of commonly used derived types is defined in the SMIng standard module IETF-SMING [[1](#)].

The different base types and their derived types allow different kinds of subtyping, namely size restrictions and range restrictions. See the following sections on base types ([Section 3.1](#) through [Section 3.12](#)) for details.

#### **[3.1](#) OctetString**

The OctetString base type represents arbitrary binary or textual data. Although SMIng has a theoretical size limitation of  $2^{16}-1$  (65535) octets for this base type, module designers should realize that there may be implementation and interoperability limitations for sizes in excess of 255 octets.

Values of octet strings may be denoted as textual data enclosed in double quotes or as arbitrary binary data denoted as a `0x'-prefixed hexadecimal value of an even number of at least two hexadecimal digits, where each pair of hexadecimal digits represents a single



octet. Letters in hexadecimal values MAY be upper-case but lower-case characters are RECOMMENDED. Textual data may contain any number (possibly zero) of any 7-bit displayable ASCII characters except double quote `\"`, including tab characters, spaces and line terminator characters (nl or cr & nl). Textual data may span multiple lines, where each subsequent line prefix containing only white space up to the column where the first line's data starts SHOULD be skipped by parsers for a better text formatting.

When defining a type derived (directly or indirectly) from the OctetString base type, the size in octets may be restricted by appending a list of size ranges or explicit size values, separated by pipe `|` characters and the whole list enclosed in parenthesis. A size range consists of a lower bound, two consecutive dots `..` and an upper bound. Each value can be given in decimal or `0x`-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. Size 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 size restriction is applied to an already size restricted octet string the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit size values or ranges, or splitting ranges into multiple ranges with intermediate gaps.

Value Examples:

```
"This is a multiline
textual data example." // legal
"This is "illegally" quoted." // illegal quotes
"But this is 'ok'." // legal apostrophe quoting
"" // legal zero length
0x123 // illegal odd hex length
0x534d496e670a // legal octet string
```

Restriction Examples:

```
OctetString (0 | 4..255) // legal size spec
OctetString (4) // legal exact size
OctetString (-1 | 1) // illegal negative size
OctetString (5 | 0) // illegal ordering
OctetString (1 | 1..10) // illegal overlapping
```

### [3.2 Pointer](#)

The Pointer base type represents values that reference class instances, attributes of class instances, or arbitrary identities.





The only values of the Pointer type that can be present in a module can refer to identities. They are denoted as identifiers of the concerned identities.

When defining a type derived (directly or indirectly) from the Pointer base type, the values may be restricted to a specific class, attribute or identity and all (directly or indirectly) derived items thereof by appending the identifier of the appropriate construct enclosed in parenthesis.

Value Examples:

```
    null                // legal identity name
    snmpUDPDomain      // legal identity name
```

Restriction Examples:

```
    Pointer (snmpTransportDomain) // legal restriction
```

### **3.3 Object Identifier**

The ObjectIdentifier base type represents administratively assigned names for use with SNMP and COPS-PR. This type SHOULD NOT be used in protocol independent SMIng modules. It is meant to be used in SNMP and COPS-PR mappings of attributes of type Pointer ([Section 3.2](#)).

Values of this type may be denoted as a sequence of numerical non-negative sub-identifier values which each MUST NOT exceed  $2^{32}-1$  (4294967295). Sub-identifiers may be denoted decimal or `\0x'`-prefixed hexadecimal. They are separated by single dots and without any intermediate white space. Alternatively (and preferred in most cases), the first element may be a previously defined or imported lower-case identifier, representing a static object identifier prefix. The total number of sub-identifiers MUST NOT exceed 128 including the expanded identifier.

Object identifier derived types cannot be restricted in any way.

Value Examples:

```
    1.3.6.1            // legal numerical oid
    mib-2.1           // legal oid with identifier prefix
    internet.4.1.0x0627.0x01 // legal oid with hex subids
    iso.-1            // illegal negative subid
    iso.org.6         // illegal non-heading identifier
    IF-MIB::ifNumber.0 // legal fully qualified instance oid
```



### 3.4 Integer32

The Integer32 base type represents integer values between  $-2^{31}$  (-2147483648) and  $2^{31}-1$  (2147483647).

Values of type Integer32 may be denoted as decimal or hexadecimal numbers, where only decimal numbers can be negative. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by ``0x'` and MUST have an even number of at least two hexadecimal digits, where letters MAY be upper-case but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Integer32 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe ``|'` characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots ``..'`` and an upper bound. Each value can be given in decimal or ``0x'`-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. 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.

Value Examples:

```
015           // illegal leading zero
-123          // legal negative value
- 1           // illegal intermediate space
0xabc         // illegal hexadecimal value length
-0xff         // illegal sign on hex value
0x80000000    // illegal value, too large
0xf00f        // legal hexadecimal value
```

Restriction Examples:

```
Integer32 (0 | 5..10) // legal range spec
Integer32 (5..10 | 2..3) // illegal ordering
Integer32 (4..8 | 5..10) // illegal overlapping
```

### 3.5 Integer64

The Integer64 base type represents integer values between  $-2^{63}$  (-9223372036854775808) and  $2^{63}-1$  (9223372036854775807).



Values of type Integer64 may be denoted as decimal or hexadecimal numbers, where only decimal numbers can be negative. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of hexadecimal digits, where letters MAY be upper-case but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Integer64 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..' and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. 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.

Value Examples:

```
015           // illegal leading zero
-123          // legal negative value
- 1           // illegal intermediate space
0xabc         // illegal hexadecimal value length
-0xff         // illegal sign on hex value
0x80000000    // legal value
```

Restriction Examples:

```
Integer64 (0 | 5..10) // legal range spec
Integer64 (5..10 | 2..3) // illegal ordering
Integer64 (4..8 | 5..10) // illegal overlapping
```

### **3.6 Unsigned32**

The Unsigned32 base type represents positive integer values between 0 and  $2^{32}-1$  (4294967295).

Values of type Unsigned32 may be denoted as decimal or hexadecimal numbers. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of hexadecimal digits, where letters MAY be upper-case but lower-case characters are RECOMMENDED.



When defining a type derived (directly or indirectly) from the Unsigned32 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..' and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. 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.

Value Examples:

```
015           // illegal leading zero
-123          // illegal negative value
0xabc         // illegal hexadecimal value length
0x80000000    // legal hexadecimal value
0x8080000000  // illegal value, too large
```

Restriction Examples:

```
Unsigned32 (0 | 5..10)    // legal range spec
Unsigned32 (5..10 | 2..3) // illegal ordering
Unsigned32 (4..8 | 5..10) // illegal overlapping
```

### **3.7 Unsigned64**

The Unsigned64 base type represents positive integer values between 0 and  $2^{64}-1$  (18446744073709551615).

Values of type Unsigned64 may be denoted as decimal or hexadecimal numbers. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of hexadecimal digits, where letters MAY be upper-case but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Unsigned64 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..' and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number





of at least two digits. 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.

Value Examples:

```
015           // illegal leading zero
-123          // illegal negative value
0xabc         // illegal hexadecimal value length
0x8080000000 // legal hexadecimal value
```

Restriction Examples:

```
Unsigned64 (1..10000000000) // legal range spec
Unsigned64 (5..10 | 2..3)   // illegal ordering
```

### **3.8 Float32**

The Float32 base type represents floating point values of single precision as described by [15].

Values of type Float32 may be denoted as a decimal fraction with an optional exponent as known from many programming languages. See the grammar rule ``floatValue'` of [Appendix A](#) for the detailed syntax. Special values are ``snan'` (signaling Not-a-Number), ``qnan'` (quiet Not-a-Number), ``neginf'` (negative infinity), and ``posinf'` (positive infinity). Note that `-0.0` and `+0.0` are different floating point values. `0.0` is equal to `+0.0`.

When defining a type derived (directly or indirectly) from the Float32 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe ``|'` characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots ``..'` and an upper bound. 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. The special values ``snan'`, ``qnan'`, ``neginf'`, and ``posinf'` must be explicitly listed in restrictions if they shall be included, where ``snan'` and ``qnan'` cannot be used in ranges.



Note that encoding is not subject to this specification. It has to be described by protocols that transport objects of type Float32. Note also that most floating point encodings disallow the representation of many values that can be written as decimal fractions as used in SMIng for human readability. Therefore, explicit values in floating point type restrictions should be handled with care.

Value Examples:

```
00.1           // illegal leading zero
3.1415        // legal value
-2.5E+3       // legal negative exponential value
```

Restriction Examples:

```
Float32 (-1.0..1.0) // legal range spec
Float32 (1 | 3.3 | 5) // legal, probably unrepresentable 3.3
Float32 (neginf..-0.0) // legal range spec
Float32 (-10.0..10.0 | 0) // illegal overlapping
```

### [3.9](#) Float64

The Float64 base type represents floating point values of double precision as described by [\[15\]](#).

Values of type Float64 may be denoted as a decimal fraction with an optional exponent as known from many programming languages. See the grammar rule 'floatValue' of [Appendix A](#) for the detailed syntax. Special values are 'snan' (signaling Not-a-Number), 'qnan' (quiet Not-a-Number), 'neginf' (negative infinity), and 'posinf' (positive infinity). Note that -0.0 and +0.0 are different floating point values. 0.0 is equal to +0.0.

When defining a type derived (directly or indirectly) from the Float64 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe '|' characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots '..' and an upper bound. 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. The special values 'snan', 'qnan', 'neginf', and 'posinf' must be explicitly listed in restrictions if they shall be included, where 'snan' and



`qnan' cannot be used in ranges.

Note that encoding is not subject to this specification. It has to be described by protocols that transport objects of type Float64. Note also that most floating point encodings disallow the representation of many values that can be written as decimal fractions as used in SMIng for human readability. Therefore, explicit values in floating point type restrictions should be handled with care.

Value Examples:

```
00.1           // illegal leading zero
3.1415         // legal value
-2.5E+3        // legal negative exponential value
```

Restriction Examples:

```
Float64 (-1.0..1.0) // legal range spec
Float64 (1 | 3.3 | 5) // legal, probably unrepresentable 3.3
Float64 (neginf..-0.0) // legal range spec
Float64 (-10.0..10.0 | 0) // illegal overlapping
```

### **[3.10](#) Float128**

The Float128 base type represents floating point values of quadruple precision as described by [\[15\]](#).

Values of type Float128 may be denoted as a decimal fraction with an optional exponent as known from many programming languages. See the grammar rule `floatValue' of [Appendix A](#) for the detailed syntax. Special values are `snan' (signaling Not-a-Number), `qnan' (quiet Not-a-Number), `neginf' (negative infinity), and `posinf' (positive infinity). Note that  $-0.0$  and  $+0.0$  are different floating point values.  $0.0$  is equal to  $+0.0$ .

When defining a type derived (directly or indirectly) from the Float128 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..' and an upper bound. 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. The special



values ``snan'`, ``qnan'`, ``neginf'`, and ``posinf'` must be explicitly listed in restrictions if they shall be included, where ``snan'` and ``qnan'` cannot be used in ranges.

Note that encoding is not subject to this specification. It has to be described by protocols that transport objects of type `Float128`. Note also that most floating point encodings disallow the representation of many values that can be written as decimal fractions as used in SMIng for human readability. Therefore, explicit values in floating point type restrictions should be handled with care.

Value Examples:

```
00.1           // illegal leading zero
3.1415         // legal value
-2.5E+3        // legal negative exponential value
```

Restriction Examples:

```
Float128 (-1.0..1.0) // legal range spec
Float128 (1 | 3.3 | 5) // legal, probably unrepresentable 3.3
Float128 (neginf..-0.0) // legal range spec
Float128 (-10.0..10.0 | 0) // illegal overlapping
```

### **3.11 Enumeration**

The Enumeration base type represents values from a set of integers in the range between  $-2^{31}$  (-2147483648) and  $2^{31}-1$  (2147483647), where each value has an assigned name. The list of those named numbers has to be comma-separated, enclosed in parenthesis and appended to the ``Enumeration'` keyword. Each named number is denoted by its lower-case identifier followed by the assigned integer value, denoted as a decimal or ``0x'`-prefixed hexadecimal number, enclosed in parenthesis. Hexadecimal numbers must have an even number of at least two digits. Every name and every number in an enumeration type MUST be unique. It is RECOMMENDED that values are positive and start at 1 and be numbered contiguously. All named numbers MUST be given in ascending order.

Values of enumeration types may be denoted as decimal or ``0x'`-prefixed hexadecimal numbers or preferably as their assigned names. Hexadecimal numbers must have an even number of at least two digits.

When defining a type derived (directly or indirectly) from an enumeration type, the set of named numbers may be equal or restricted by removing one or more named numbers. But no named numbers may be





added or changed regarding its name, value, or both.

Type and Value Examples:

```
Enumeration (up(1), down(2), testing(3))
Enumeration (down(2), up(1)) // illegal order

0 // legal (though not recommended) value
up // legal value given by name
2 // legal value given by number
```

### [3.12 Bits](#)

The Bits base type represents bit sets. That is, a Bits value is a set of flags identified by small integer numbers starting at 0. Each bit number has an assigned name. The list of those named numbers has to be comma-separated, enclosed in parenthesis and appended to the 'Bits' keyword. Each named number is denoted by its lower-case identifier followed by the assigned integer value, denoted as a decimal or '0x'-prefixed hexadecimal number, enclosed in parenthesis. Hexadecimal numbers must have an even number of at least two digits. Every name and every number in a bits type MUST be unique. It is RECOMMENDED that numbers start at 0 and be numbered contiguously. Negative numbers are forbidden. All named numbers MUST be given in ascending order.

Values of bits types may be denoted as a comma-separated list of decimal or '0x'-prefixed hexadecimal numbers or preferably their assigned names enclosed in parenthesis. Hexadecimal numbers must have an even number of at least two digits. There MUST NOT be any element (by name or number) listed more than once. Elements MUST be listed in ascending order.

When defining a type derived (directly or indirectly) from a bits type, the set of named numbers may be restricted by removing one or more named numbers. But no named numbers may be added or changed regarding its name, value, or both.

Type and Value Examples:

```
Bits (readable(0), writeable(1), executable(2))
Bits (writeable(1), readable(0)) // illegal order

() // legal empty value
(readable, writeable, 2) // legal value
(0, readable, executable) // illegal, readable(0) appears twice
(writeable, 4) // illegal, element 4 out of range
```



### **3.13 Display Formats**

Attribute definitions and type definitions allow the specification of a format to be used, when a value of that attribute or an attribute of that type is displayed. Format specifications are represented as textual data.

When the attribute or type has an underlying base type of Integer32, Integer64, Unsigned32, or Unsigned64, the format consists of an integer-format specification, containing two parts. The first part is a single character suggesting a display format, either: `'x'` for hexadecimal, or `'d'` for decimal, or `'o'` for octal, or `'b'` for binary. For all types, when rendering the value, leading zeros are omitted, and for negative values, a minus sign is rendered immediately before the digits. The second part is always omitted for `'x'`, `'o'` and `'b'`, and need not be present for `'d'`. If present, the second part starts with a hyphen and is followed by a decimal number, which defines the implied decimal point when rendering the value. For example `'d-2'` suggests that a value of 1234 be rendered as `'12.34'`.

When the attribute or type has an underlying base type of OctetString, the format consists of one or more octet-format specifications. Each specification consists of five parts, with each part using and removing zero or more of the next octets from the value and producing the next zero or more characters to be displayed. The octets within the value are processed in order of significance, most significant first.

The five parts of a octet-format specification are:

1. the (optional) repeat indicator; if present, this part is a `'*'`, and indicates that the current octet of the value is to be used as the repeat count. The repeat count is an unsigned integer (which may be zero) which specifies how many times the remainder of this octet-format specification should be successively applied. If the repeat indicator is not present, the repeat count is one.
2. the octet length: one or more decimal digits specifying the number of octets of the value to be used and formatted by this octet-specification. Note that the octet length can be zero. If less than this number of octets remain in the value, then the lesser number of octets are used.
3. the display format, either: `'x'` for hexadecimal, `'d'` for decimal, `'o'` for octal, `'a'` for ASCII, or `'t'` for UTF-8 [[16](#)]. If the octet length part is greater than one, and the display format part refers to a numeric format, then network byte-ordering (big-



endian encoding) is used interpreting the octets in the value. The octets processed by the 't' display format do not necessarily form an integral number of UTF-8 characters. Trailing octets which do not form a valid UTF-8 encoded character are discarded.

- 4. the (optional) display separator character; if present, this part is a single character which is produced for display after each application of this octet-specification; however, this character is not produced for display if it would be immediately followed by the display of the repeat terminator character for this octet specification. This character can be any character other than a decimal digit and a '\*'.
- 5. the (optional) repeat terminator character, which can be present only if the display separator character is present and this octet specification begins with a repeat indicator; if present, this part is a single character which is produced after all the zero or more repeated applications (as given by the repeat count) of this octet specification. This character can be any character other than a decimal digit and a '\*'.

Output of a display separator character or a repeat terminator character is suppressed if it would occur as the last character of the display.

If the octets of the value are exhausted before all the octet format specification have been used, then the excess specifications are ignored. If additional octets remain in the value after interpreting all the octet format specifications, then the last octet format specification is re-interpreted to process the additional octets, until no octets remain in the value.

Note that for some types no format specifications are defined and SHOULD be omitted. Implementations MUST ignore format specifications they cannot interpret. Also note that the SMIng grammar (Appendix A) does not specify the syntax of format specifications.

Display Format Examples:

Base Type	Format	Example Value	Rendered Value
OctetString	255a	"Hello World."	Hello World.
OctetString	1x:	"Hello!"	48:65:6c:6c:6f:21
OctetString	1d:1d:1d.1d,1a1d:1d	0x0d1e0f002d0400	13:30:15.0, -4:0
OctetString	1d.1d.1d.1d/2d	0x0a0000010400	10.0.0.1/1024
OctetString	*1x:/1x:	0x02aabbccdee	aa:bb/cc:dd:ee
Integer32	d-2	1234	12.34



## **4. The SMIng File Structure**

The topmost container of SMIng information is a file. An SMIng file may contain zero, one or more modules. It is RECOMMENDED to separate modules into files named by their modules, where possible. Though, for dedicated purposes it may be reasonable to collect several modules in a single file.

The top level SMIng construct is the ``module'` statement ([Section 5](#)) that defines a single SMIng module. A module contains a sequence of sections in an obligatory order with different kinds of definitions. Whether these sections contain statements or remain empty mainly depends on the purpose of the module.

### **4.1 Comments**

Comments can be included at any position in an SMIng file, except in between the characters of a single token like those of a quoted string. However, it is RECOMMENDED that all substantive descriptions be placed within an appropriate description clause, so that the information is available to SMIng parsers.

Comments commence with a pair of adjacent slashes ``//'` and end at the end of the line.

### **4.2 Statements and Arguments**

SMIng has a very small set of basic grammar rules based on the concept of statements. Each statement starts with a lower-case keyword identifying the statement followed by a number (possibly zero) of arguments. An argument may be quoted text, an identifier, a value of any base type, a list of identifiers enclosed in parenthesis ``( )'` or a statement block enclosed in curly braces ``{ }'`. Since statement blocks are valid arguments, it is possible to nest statement sequences. Each statement is terminated by a semicolon ``;'.`

The core set of statements may be extended using the SMIng ``extension'` statement. See [Section 6](#) and [Section 11](#) for details.

At places where a statement is expected, but an unknown lower-case word is read, those statements MUST be skipped up to the proper semicolon, including nested statement blocks.

## **5. The module Statement**

The ``module'` statement is used as a container of all definitions of a single SMIng module. It gets two arguments: an upper-case module





name and a statement block that contains mandatory and optional statements and sections of statements in an obligatory order:

```
module <MODULE-NAME> {  
  
    <optional import statements>  
    <organization statement>  
    <contact statement>  
    <description statement>  
    <optional reference statement>  
    <at least one revision statement>  
  
    <optional extension statements>  
  
    <optional typedef statements>  
  
    <optional class statements>  
  
};
```

The optional ``import'` statements are followed by the mandatory ``organization'`, ``contact'`, and ``description'` statements and the optional ``reference'` statement, which in turn are followed by the mandatory ``revision'` statements. This part defines the module's meta information while the following sections contain its main definitions.

See the ``moduleStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``module'` statement.

### **5.1 The module's import Statement**

The optional module's ``import'` statement is used to import identifiers from external modules into the local module's namespace. It gets two arguments: the name of the external module and a comma-separated list of one or more identifiers to be imported enclosed in parenthesis.

Multiple ``import'` statements for the same module but with disjoint lists of identifiers are allowed, though NOT RECOMMENDED. Anyhow, the same identifier from the same module MUST NOT be imported multiple times. To import identifiers with the same name from different modules might be necessary and is allowed. To distinguish them in the local module, they have to be referred by qualified names. It is NOT RECOMMENDED to import identifiers not used in the local module.

See the ``importStatement'` rule of the SMIng grammar (Appendix A) for



the formal syntax of the ``import'` statement.

### **5.2 The module's organization Statement**

The module's ``organization'` statement, which must be present, gets one argument which is used to specify a textual description of the organization(s) under whose auspices this module was developed.

### **5.3 The module's contact Statement**

The module's ``contact'` statement, which must be present, gets one argument 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.

### **5.4 The module's description Statement**

The module's ``description'` statement, which must be present, gets one argument which is used to specify a high-level textual description of the contents of this module.

### **5.5 The module's reference Statement**

The module's ``reference'` statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related management information, or some other document which provides additional information relevant to this module.

### **5.6 The module's revision Statement**

The module's ``revision'` statement is repeatedly used to specify the editorial revisions of the module, including the initial revision. It gets one argument which is a statement block that holds detailed information in an obligatory order. A module **MUST** have at least one initial ``revision'` statement. For every editorial change, a new one **MUST** be added in front of the revisions sequence, so that all revisions are in reverse chronological order.

See the ``revisionStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``revision'` statement.

#### **5.6.1 The revision's date Statement**

The revision's ``date'` statement, which must be present, gets one argument which is used to specify the date and time of the revision in the format ``YYYY-MM-DD HH:MM'` or ``YYYY-MM-DD'` which implies the time ``00:00'`. The time is always given in UTC.



See the `date` rule of the SMIng grammar (Appendix A) for the formal syntax of the revision's `date` statement.

### **5.6.2 The revision's description Statement**

The revision's `description` statement, which must be present, gets one argument which is used to specify a high-level textual description of the revision.

### **5.7 Usage Example**

Consider how a skeletal module might be constructed:

```
module FIZBIN {  
  
    import IETF-SMING (DisplayString);  
  
    organization  
        "IETF Next Generation Structure of  
        Management Information Working Group (SMING)";  
  
    contact  
        "Frank Strauss  
  
        TU Braunschweig  
        Bueltenweg 74/75  
        38106 Braunschweig  
        Germany  
  
        Phone: +49 531 391-3266  
        EMail: strauss@ibr.cs.tu-bs.de";  
  
    description  
        "The module for entities implementing  
        the xxxx protocol.";  
    reference  
        "RFC 2578, Section 5.7.";  
  
    revision {  
        date          "2001-03-02";  
        description  
            "Initial revision, published as RFC XXXX.";  
    };  
  
    // ... further definitions ...  
  
}; // end of module FIZBIN.
```



## **6. The extension Statement**

The ``extension'` statement is used to define new statements to be used in the local module following this extension statement definition or in external modules that may import this extension statement definition. The ``extension'` statement gets two arguments: a lower-case extension statement identifier and a statement block that holds detailed extension information in an obligatory order.

Extension statement identifiers SHOULD NOT contain any upper-case characters.

Note that the SMIng extension feature does not allow to formally specify the context, argument syntax and semantics of an extension. Its only purpose is to declare the existence of an extension and to allow a unique reference to an extension. See [Section 11](#) for detailed information on extensions and [3] for mappings of SMIng definitions to SNMP which is formally defined as an extension.

See the ``extensionStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``extension'` statement.

### **6.1 The extension's status Statement**

The extension's ``status'` statement, which must be present, gets one argument which is used to specify whether this extension definition is current or historic. The value ``current'` means that the definition is current and valid. The value ``obsolete'` means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value ``deprecated'` also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

### **6.2 The extension's description Statement**

The extension's ``description'` statement, which must be present, gets one argument which is used to specify a high-level textual description of the extension statement.

It is RECOMMENDED to include information on the extension's context, its semantics, and implementation conditions. See also [Section 11](#).

### **6.3 The extension's reference Statement**

The extension's ``reference'` statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related





extension definitions, or some other document which provides additional information relevant to this extension.

#### **6.4 The extension's abnf Statement**

The extension's ``abnf'` statement, which need not be present, gets one argument which is used to specify a formal ABNF [12] grammar definition of the extension. This grammar can reference rule names from the core SMIng grammar [Appendix A](#).

Note that the ``abnf'` statement should contain only pure ABNF and no additional text, though comments prefixed by semicolon are allowed but should probably be moved to the description statement. Note that double quotes are not allowed inside textual descriptions which are itself enclosed in double quotes. So they have to be replaced by single quotes.

#### **6.5 Usage Example**

```
extension severity {
  status current;
  description
    "The optional severity extension statement can only
    be applied to the statement block of an SMIng class'
    event definition. If it is present it denotes the
    severity level of the event in a range from 0
    (emergency) to 7 (debug).";
  abnf
    "severityStatement = severityKeyword sep number optsep ';'
    severityKeyword    = 'severity'";
};
```

### **7. The typedef Statement**

The ``typedef'` statement is used to define new data types to be used in the local module or in external modules. It gets two arguments: an upper-case type identifier and a statement block that holds detailed type information in an obligatory order.

Type identifiers SHOULD NOT consist of all upper-case characters and SHOULD NOT contain hyphens.

See the ``typedefStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``typedef'` statement.

#### **7.1 The typedef's type Statement**

The typedef's ``type'` statement, which must be present, gets one



argument which is used to specify the type from which this type is derived. Optionally, type restrictions may be applied to the new type by appending subtyping information according to the rules of the base type. See [Section 3](#) for SMIng base types and their type restrictions.

## **[7.2](#) The typedef's default Statement**

The typedef's ``default'` statement, which need not be present, gets one argument which is used to specify an acceptable default value for attributes of this type. A default value may be used when an attribute instance is created. That is, the value is a "hint" to implementors.

The value of the ``default'` statement must, of course, correspond to the (probably restricted) type specified in the typedef's ``type'` statement.

The default value of a type may be overwritten by a default value of an attribute of this type.

Note that for some types, default values make no sense.

## **[7.3](#) The typedef's format Statement**

The typedef's ``format'` statement, which need not be present, gets one argument which is used to give a hint as to how the value of an instance of an attribute of this type might be displayed. See [Section 3.13](#) for a description of format specifications.

If no format is specified, it is inherited from the type given in the ``type'` statement. On the other hand, the format specification of a type may be semantically refined by a format specification of an attribute of this type.

## **[7.4](#) The typedef's units Statement**

The typedef's ``units'` statement, which need not be present, gets one argument which is used to specify a textual definition of the units associated with attributes of this type.

If no units are specified, they are inherited from the type given in the ``type'` statement. On the other hand, the units specification of a type may be semantically refined by a units specification of an attribute of this type.

The units specification has to be appropriate for values displayed according to the typedef's format specification, if present. E.g.,



if the type defines frequency values of type `Unsigned64` measured in thousands of Hertz, the format specification should be ``d-3'` and the units specification should be ``Hertz'` or ``Hz'`. If the format specification would be omitted, the units specification should be ``Milli-Hertz'` or ``mHz'`. Authors of SMIng modules should pay attention to keep format and units specifications synced. Application implementors **MUST NOT** implement units specifications without implementing format specifications.

### **7.5 The typedef's status Statement**

The typedef's ``status'` statement, which must be present, gets one argument which is used to specify whether this type definition is current or historic. The value ``current'` means that the definition is current and valid. The value ``obsolete'` means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value ``deprecated'` also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Derived types **SHOULD NOT** be defined as ``current'` if their underlying type is ``deprecated'` or ``obsolete'`. Similarly, they **SHOULD NOT** be defined as ``deprecated'` if their underlying type is ``obsolete'`. Nevertheless, subsequent revisions of the underlying type cannot be avoided, but **SHOULD** be taken into account in subsequent revisions of the local module.

### **7.6 The typedef's description Statement**

The typedef's ``description'` statement, which must be present, gets one argument which is used to specify a high-level textual description of the newly defined type.

It is **RECOMMENDED** to include all semantic definitions necessary for implementation, and to embody any information which would otherwise be communicated in any commentary annotations associated with this type definition.

### **7.7 The typedef's reference Statement**

The typedef's ``reference'` statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related type definitions, or some other document which provides additional information relevant to this type definition.

### **7.8 Usage Examples**



```
typedef RptrOperStatus {
    type          Enumeration (other(1), ok(2), rptrFailure(3),
                             groupFailure(4), portFailure(5),
                             generalFailure(6));
    default       other;      // undefined by default.
    status        deprecated;
    description   "A type to indicate the operational state
                 of a repeater.";
    reference     "[IEEE 802.3 Mgt], 30.4.1.1.5, aRepeaterHealthState.";
};

typedef SnmpTransportDomain {
    type          Pointer (snmpTransportDomain);
    status        current;
    description   "A pointer to an SNMP transport domain identity.";
};

typedef DateAndTime {
    type          OctetString (8 | 11);
    format        "2d-1d-1d,1d:1d:1d.1d,1a1d:1d";
    status        current;
    description   "A date-time specification.
                 ...
                 Note that if only local time is known, then timezone
                 information (fields 8-10) is not present.";
    reference     "RFC 2579, SNMPv2-TC.DateAndTime.";
};

typedef Frequency {
    type          Unsigned64;
    format        "d-3"
    units         "Hertz";
    status        current;
    description   "A wide-range frequency specification measured
                 in thousands of Hertz.";
};
```

## 8. The identity Statement

The 'identity' statement is used to define a new abstract and untyped identity. Its only purpose is to denote its name, semantics and existence. An identity can be defined either from scratch or derived





from a parent identity. The ``identity'` statement gets the following two or four arguments: The first argument is a lower-case identity identifier and the last argument is a statement block that holds detailed identity information in an obligatory order. In case of derived identities there are two tokens inbetween: a single colon ``:'` and the identifier of the parent identity.

See the ``identityStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``identity'` statement.

### **8.1 The identity's status Statement**

The identity's ``status'` statement, which must be present, gets one argument which is used to specify whether this identity definition is current or historic. The value ``current'` means that the definition is current and valid. The value ``obsolete'` means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value ``deprecated'` also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Derived identities SHOULD NOT be defined as ``current'` if their parent identity is ``deprecated'` or ``obsolete'`. Similarly, they SHOULD NOT be defined as ``deprecated'` if their parent identity is ``obsolete'`. Nevertheless, subsequent revisions of the parent identity cannot be avoided, but SHOULD be taken into account in subsequent revisions of the local module.

### **8.2 The identity' description Statement**

The identity's ``description'` statement, which must be present, gets one argument which is used to specify a high-level textual description of the newly defined identity.

It is RECOMMENDED to include all semantic definitions necessary for implementation, and to embody any information which would otherwise be communicated in any commentary annotations associated with this identity definition.

### **8.3 The identity's reference Statement**

The identity's ``reference'` statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related identity definitions, or some other document which provides additional information relevant to this identity definition.



## **8.4 Usage Examples**

```
identity null {
  status current;
  description
    "An identity used to represent null pointer values.";
};

identity snmpTransportDomain {
  status current;
  description
    "A generic SNMP transport domain identity.";
};

identity snmpUDPDomain : snmpTransportDomain {
  status current;
  description
    "The SNMP over UDP transport domain.";
};
```

## **9. The class Statement**

The ``class'` statement is used to define a new class, that represents a container of related attributes and events ([Section 9.1](#), [Section 9.3](#)) in an object-oriented manner. Thus, a class can be defined either from scratch or derived from a parent class. A derived class inherits all attributes and events of the parent class and can be extended by additional attributes and events. Furthermore, parent attributes can be refined by new attributes of the same name that are more specific in their formal type restrictions or their semantics specified in the attribute description clause. Similarly, parent events can be refined by new events of the same name that are more specific in their semantics specified in the event description clause.

The ``class'` statement gets the following two or four arguments: The first argument is an upper-case class identifier and the last argument is a statement block that holds detailed class information in an obligatory order. In case of derived classes there are two tokens inbetween: a single colon ``:'` and the identifier of the parent class.

See the ``classStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``class'` statement.

### **9.1 The class' attribute Statement**

The class' ``attribute'` statement, which can be present zero, one or



multiple times, gets three arguments: a type or class name, the attribute name, and a statement block that holds detailed attribute information in an obligatory order.

#### **9.1.1 The attribute's access Statement**

The attribute's ``access'` statement must be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify whether it makes sense to read and/or write an instance of the attribute, or to include its value in an event. This is the maximal level of access for the attribute. This maximal level of access is independent of any administrative authorization policy.

The value ``readwrite'` indicates that read and write access makes sense. The value ``readonly'` indicates that read access makes sense, but write access is never possible. The value ``eventonly'` indicates an object which is accessible only via an event.

These values are ordered, from least to greatest access level: ``eventonly'`, ``readonly'`, ``readwrite'`.

#### **9.1.2 The attribute's default Statement**

The attribute's ``default'` statement need not be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify an acceptable default value for this attribute. A default value may be used when an attribute instance is created. That is, the value is a "hint" to implementors.

The value of the ``default'` statement must, of course, correspond to the (probably restricted) type specified in the attribute's ``type'` statement.

The attribute's default value overrides the default value of the underlying type definition if both are present.

#### **9.1.3 The attribute's format Statement**

The attribute's ``format'` statement need not be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to give a hint as to how the value of an instance of this attribute might be displayed. See [Section 3.13](#) for a description of format specifications.

The attribute's format specification overrides the format



specification of the underlying type definition if both are present.

#### **9.1.4 The attribute's units Statement**

The attribute's ``units'` statement need not be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify a textual definition of the units associated with this attribute.

The attribute's units specification overrides the units specification of the underlying type definition if both are present.

The units specification has to be appropriate for values displayed according to the attribute's format specification if present. E.g., if the attribute represents a frequency value of type `Unsigned64` measured in thousands of Hertz, the format specification should be ``d-3'` and the units specification should be ``Hertz'` or ``Hz'`. If the format specification would be omitted the units specification should be ``Milli-Hertz'` or ``mHz'`. Authors of SMIng modules should pay attention to keep format and units specifications of type and attribute definitions synced. Application implementors **MUST NOT** implement units specifications without implementing format specifications.

#### **9.1.5 The attribute's status Statement**

The attribute's ``status'` statement must be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify whether this attribute definition is current or historic. The value ``current'` means that the definition is current and valid. The value ``obsolete'` means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value ``deprecated'` also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Attributes **SHOULD NOT** be defined as ``current'` if their type or their containing class is ``deprecated'` or ``obsolete'`. Similarly, they **SHOULD NOT** be defined as ``deprecated'` if their type or their containing class is ``obsolete'`. Nevertheless, subsequent revisions of used type definition cannot be avoided, but **SHOULD** be taken into account in subsequent revisions of the local module.

#### **9.1.6 The attribute's description Statement**

The attribute's ``description'` statement, which must be present, gets





one argument which is used to specify a high-level textual description of this attribute.

It is RECOMMENDED to include all semantic definitions necessary for the implementation of this attribute.

#### **9.1.7 The attribute's reference Statement**

The attribute's ``reference'` statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related attribute definitions, or some other document which provides additional information relevant to this attribute definition.

#### **9.2 The class' unique Statement**

The class' ``unique'` statement, which need not be present, gets one argument that specifies a comma-separated list of attributes of this class, enclosed in parenthesis. If present, this list of attributes makes up a unique identification of all possible instances of this class. It can be used as a unique key in underlying protocols.

If the list is empty the class should be regarded as a scalar class with only a single static instance.

If the ``unique'` statement is not present the class is not meant to be instantiated directly, but just to be contained in other classes or to be the parent class of other refining classes.

If present, the attribute list MUST NOT contain any attribute more than once and the attributes should be ordered where appropriate so that the attributes that are most significant in most situations appear first.

#### **9.3 The class' event Statement**

The class' ``event'` statement is used to define an event related to an instance of this class that can occur asynchronously. It gets two arguments: a lower-case event identifier and a statement block that holds detailed information in an obligatory order.

See the ``eventStatement'` rule of the SMIng grammar (Appendix A) for the formal syntax of the ``event'` statement.

##### **9.3.1 The event's status Statement**

The event's ``status'` statement, which must be present, gets one argument which is used to specify whether this event definition is



current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

### **9.3.2 The event's description Statement**

The event's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of this event.

It is RECOMMENDED to include all semantic definitions necessary for the implementation of this event. In particular, it SHOULD be documented which instance of the class is associated with an event of this type.

### **9.3.3 The event's reference Statement**

The event's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related event definitions, or some other document which provides additional information relevant to this event definition.

## **9.4 The class' status Statement**

The class' `status' statement, which must be present, gets one argument which is used to specify whether this class definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Derived classes SHOULD NOT be defined as `current' if their parent class is `deprecated' or `obsolete'. Similarly, they SHOULD NOT be defined as `deprecated' if their parent class is `obsolete'. Nevertheless, subsequent revisions of the parent class cannot be avoided, but SHOULD be taken into account in subsequent revisions of the local module.

## **9.5 The class' description Statement**

The class' `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of



the newly defined class.

It is RECOMMENDED to include all semantic definitions necessary for implementation, and to embody any information which would otherwise be communicated in any commentary annotations associated with this class definition.

### **[9.6](#) The class's reference Statement**

The class's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related class definitions, or some other document which provides additional information relevant to this class definition.

### **[9.7](#) Usage Example**

Consider how an event might be described that signals a status change of an interface:

```
class Interface {
  // ...
  attribute Gauge32 speed {
    access      readonly;
    units       "bps";
    status      current;
    description
      "An estimate of the interface's current bandwidth
       in bits per second.";
  };
  // ...
  attribute AdminStatus adminStatus {
    access      readwrite;
    status      current;
    description
      "The desired state of the interface.";
  };
  attribute OperStatus operStatus {
    access      readonly;
    status      current;
    description
      "The current operational state of the interface.";
  };
};

event linkDown {
  status      current;
  description
    "A linkDown event signifies that the ifOperStatus
```



```
        attribute for this interface instance is about to
        enter the down state from some other state (but not
        from the notPresent state). This other state is
        indicated by the included value of ifOperStatus.";
};

status      current;
description
    "A physical or logical network interface.";
};
```

## **10. Extending a Module**

As experience is gained with a module, it may be desirable to revise that module. However, changes are not allowed if they have any potential to cause interoperability problems between an implementation using an original specification and an implementation using an updated specification(s).

For any change, some statements near the top of the module MUST be updated to include information about the revision: specifically, a new `revision' statement ([Section 5.6](#)) must be included in front of the `revision' statements. Furthermore, any necessary changes MUST be applied to other statements, including the `organization' and `contact' statements ([Section 5.2](#), [Section 5.3](#)).

Note that any definition contained in a module is available to be imported by any other module, and is referenced in an `import' statement via the module name. Thus, a module name MUST NOT be changed. Specifically, the module name (e.g., `FIZBIN' in the example of [Section 5.7](#)) MUST NOT be changed when revising a module (except to correct typographical errors), and definitions MUST NOT be moved from one module to another.

Also note, that obsolete definitions MUST NOT be removed from modules since their identifiers may still be referenced by other modules.

A definition may be revised in any of the following ways:

- o In `typedef' statement blocks, a `type' statement containing an `Enumeration' or `Bits' type may have new named numbers added.
- o In `typedef' statement blocks, the value of a `type' statement may be replaced by another type if the new type is derived (directly or indirectly) from the same base type, has the same set of values, and has identical semantics.





- o In ``attribute'` statements where the first argument specifies a class, the class may be replaced by another class if the new class is inherited (directly or indirectly) from the base class and both classes have identical semantics.
- o In ``attribute'` statements where the first argument specifies a type, the type may be replaced by another type if the new type is derived (directly or indirectly) from the same base type, has the same set of values, and has identical semantics.
- o In any statement block, a ``status'` statement value of ``current'` may be revised as ``deprecated'` or ``obsolete'`. Similarly, a ``status'` statement value of ``deprecated'` may be revised as ``obsolete'`. When making such a change, the ``description'` statement SHOULD be updated to explain the rationale.
- o In ``typedef'` and ``attribute'` statement blocks, a ``default'` statement may be added or updated.
- o In ``typedef'` and ``attribute'` statement blocks, a ``units'` statement may be added.
- o A class may be augmented by adding new attributes.
- o In any statement block, clarifications and additional information may be included in the ``description'` statement.
- o In any statement block, a ``reference'` statement may be added or updated.
- o Entirely new extensions, types, identities, and classes may be defined, using previously unassigned identifiers.

Otherwise, if the semantics of any previous definition are changed (i.e., if a non-editorial change is made to any definition other than those specifically allowed above), then this MUST be achieved by a new definition with a new identifier. In case of a class where the semantics of any attributes are changed, the new class can be defined by inheritance from the old class and refining the changed attributes.

Note that changing the identifier associated with an existing definition is considered a semantic change, as these strings may be used in an ``import'` statement.

## **11. SMIng Language Extensibility**

While the core SMIng language has a well defined set of statements



([Section 5](#) through [Section 9.3](#)) that are used to specify those aspects of management information commonly regarded as necessary without management protocol specific information, there may be further information, people wish to express. To describe additional information informally in description statements has the disadvantage that this information cannot be parsed by any program.

SMIng allows modules to include statements that are unknown to a parser but fulfill some core grammar rules ([Section 4.2](#)). Furthermore, additional statements may be defined by the 'extension' statement ([Section 6](#)). Extensions can be used in the local module or in other modules, that import the extension. This has some advantages:

- o A parser can differentiate between statements known as extensions and unknown statements. This enables the parser to complain about unknown statements, e.g. due to typos.
- o If an extension's definition contains a formal ABNF grammar definition and a parser is able to interpret this ABNF definition, this enables the parser also to complain about wrong usage of an extension.
- o Since, there might be some common need for extensions, there is a relatively high probability of extension name collisions originated by different organizations, as long as there is no standardized extension for that purpose. The requirement to explicitly import extension statements allows to distinguish those extensions.
- o The supported extensions of an SMIng implementation, e.g. a SMIng module compiler, can be clearly expressed.

The only formal effect of an extension statement definition is to declare its existence and its status, and optionally its ABNF grammar. All additional aspects SHOULD be described in the 'description' statement:

- o The detailed semantics of the new statement SHOULD be described.
- o The contexts in which the new statement can be used, SHOULD be described, e.g., a new statement may be designed to be used only in the statement block of a module, but not in other nested statement blocks. Others may be applicable in multiple contexts. In addition, the point in the sequence of an obligatory order of other statements, where the new statement may be inserted, might be prescribed.



- o The circumstances that make the new statement mandatory or optional SHOULD be described.
- o The syntax of the new statement SHOULD at least be described informally, if not supplied formally in an `abnf' statement.
- o It might be reasonable to give some suggestions under which conditions the implementation of the new statement is adequate and how it could be integrated into existent implementations.

Some possible extension applications are:

- o The formal mappings of SMIng definitions into the SNMP ([\[3\]](#)) and COPS-PR frameworks are defined as SMIng extensions.
- o Inlined annotations to definitions. E.g., a vendor may wish to describe additional information to class and attribute definitions in private modules. An example are severity levels of events in the statement block of an `event' statement.
- o Arbitrary annotations to external definitions. E.g., a vendor may wish to describe additional information to definitions in a "standard" module. This allows a vendor to implement "standard" modules as well as additional private features, without redundant module definitions, but on top of "standard" module definitions.

## **[12.](#) 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.

## **[13.](#) Acknowledgements**

Since SMIng started as a close successor of SMIV2, some paragraphs and phrases are directly taken from the SMIV2 specifications [\[5\]](#), [\[6\]](#), [\[7\]](#) written by Jeff Case, Keith McCloghrie, David Perkins, Marshall T. Rose, Juergen Schoenwaelder, and Steven L. Waldbusser.

The authors would like to thank all participants of the 7th NMRG meeting held in Schloss Kleinheubach from 6-8 September 2000, which was a major step towards the current status of this memo, namely Heiko Dassow, David Durham, and Bert Wijnen.

Marshall T. Rose's work on an XML framework for RFC authors [\[17\]](#) made the writing of an Internet standards document much more comfortable.



## References

- [1] Strauss, F. and J. Schoenwaelder, "SMIng Core Modules", [draft-ietf-sming-modules-02.txt](#), July 2001.
- [2] Strauss, F. and J. Schoenwaelder, "SMIng Internet Protocol Core Modules", [draft-ietf-sming-inet-modules-02.txt](#), July 2001.
- [3] Strauss, F. and J. Schoenwaelder, "SMIng Extension for SNMP Mappings", [draft-ietf-sming-snmp-02.txt](#), July 2001.
- [4] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [RFC 2119](#), [BCP 14](#), March 1997.
- [5] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Structure of Management Information Version 2 (SMIv2)", [RFC 2578](#), STD 58, April 1999.
- [6] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Textual Conventions for SMIv2", [RFC 2579](#), STD 59, April 1999.
- [7] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Conformance Statements for SMIv2", [RFC 2580](#), STD 60, April 1999.
- [8] McCloghrie, K., Fine, M., Seligson, J., Chan, K., Hahn, S., Sahita, R., Smith, A. and F. Reichmeyer, "Structure of Policy Provisioning Information (SPPI)", [draft-ietf-rap-sppi-07.txt](#), May 2001.
- [9] Rose, M. and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based Internets", [RFC 1155](#), STD 16, May 1990.
- [10] Rose, M. and K. McCloghrie, "Concise MIB Definitions", [RFC 1212](#), STD 16, March 1991.
- [11] Rose, M., "A Convention for Defining Traps for use with the SNMP", [RFC 1215](#), March 1991.
- [12] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", [RFC 2234](#), November 1997.
- [13] International Organization for Standardization, "Specification of Abstract Syntax Notation One (ASN.1)", International Standard 8824, December 1987.





- [14] Harrington, D., Presuhn, R. and B. Wijnen, "An Architecture for Describing SNMP Management Frameworks", [RFC 2271](#), January 1999.
- [15] Institute of Electrical and Electronics Engineers, "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, August 1985.
- [16] Yergeau, F., "UTF-8, a transformation format of ISO 10646", [RFC 2279](#), January 1998.
- [17] Rose, M., "Writing I-Ds and RFCs using XML", [RFC 2629](#), June 1999.

#### Authors' Addresses

Frank Strauss  
TU Braunschweig  
Bueltenweg 74/75  
38106 Braunschweig  
Germany

Phone: +49 531 391-3266  
EMail: [strauss@ibr.cs.tu-bs.de](mailto:strauss@ibr.cs.tu-bs.de)  
URI: <http://www.ibr.cs.tu-bs.de/>

Juergen Schoenwaelder  
TU Braunschweig  
Bueltenweg 74/75  
38106 Braunschweig  
Germany

Phone: +49 531 391-3289  
EMail: [schoenw@ibr.cs.tu-bs.de](mailto:schoenw@ibr.cs.tu-bs.de)  
URI: <http://www.ibr.cs.tu-bs.de/>

#### [Appendix A](#). SMIng ABNF Grammar

The SMIng grammar conforms to the Augmented Backus-Naur Form (ABNF) [\[12\]](#).

```
;;  
;; sming.abnf -- SMIng grammar in ABNF notation (RFC 2234).  
;;  
;; @(#) $Id: sming.abnf,v 1.24 2001/07/20 14:13:10 strauss Exp $  
;;  
;; Copyright (C) The Internet Society (2001). All Rights Reserved.
```



```
;;

;;
;; This file is WORK IN PROGRESS.
;;

smingFile          = optsep *(moduleStatement optsep)

;;
;; Statement rules.
;;

moduleStatement    = moduleKeyword sep ucIdentifier optsep
                    "{" stmtsep
                    *(importStatement stmtsep)
                    organizationStatement stmtsep
                    contactStatement stmtsep
                    descriptionStatement stmtsep
                    *1(referenceStatement stmtsep)
                    1*(revisionStatement stmtsep)
                    *(extensionStatement stmtsep)
                    *(typedefStatement stmtsep)
                    *(identityStatement stmtsep)
                    *(classStatement stmtsep)
                    "}" optsep ";"

extensionStatement = extensionKeyword sep lcIdentifier optsep
                    "{" stmtsep
                    statusStatement stmtsep
                    descriptionStatement stmtsep
                    *1(referenceStatement stmtsep)
                    *1(abnfStatement stmtsep)
                    "}" optsep ";"

typedefStatement   = typedefKeyword sep ucIdentifier optsep
                    "{" stmtsep
                    typedefTypeStatement stmtsep
                    *1(defaultStatement stmtsep)
                    *1(formatStatement stmtsep)
                    *1(unitsStatement stmtsep)
                    statusStatement stmtsep
                    descriptionStatement stmtsep
                    *1(referenceStatement stmtsep)
                    "}" optsep ";"

identityStatement  = identityKeyword sep lcIdentifier optsep
                    *1(":" optsep qlcIdentifier optsep)
                    "{" stmtsep
```



```

        statusStatement stmtsep
        descriptionStatement stmtsep
        *1(referenceStatement stmtsep)
    "}" optsep ";"

classStatement      = classKeyword sep ucIdentifier optsep
                    *1(":" optsep qcIdentifier optsep)
                    "{" stmtsep
                    *(attributeStatement stmtsep)
                    *1(uniqueStatement stmtsep)
                    *(eventStatement stmtsep)
                    statusStatement stmtsep
                    descriptionStatement stmtsep
                    *1(referenceStatement stmtsep)
                    "}" optsep ";"

attributeStatement  = attributeKeyword sep
                    qcIdentifier sep
                    lcIdentifier optsep
                    "{" stmtsep
                    *1(accessStatement stmtsep)
                    *1(defaultStatement stmtsep)
                    *1(formatStatement stmtsep)
                    *1(unitsStatement stmtsep)
                    statusStatement stmtsep
                    descriptionStatement stmtsep
                    *1(referenceStatement stmtsep)
                    "}" optsep ";"

uniqueStatement     = uniqueKeyword optsep
                    "(" optsep qlcIdentifierList
                    optsep ")" optsep ";"

eventStatement      = eventKeyword sep lcIdentifier
                    optsep "{" stmtsep
                    statusStatement stmtsep
                    descriptionStatement stmtsep
                    *1(referenceStatement stmtsep)
                    "}" optsep ";"

importStatement     = importKeyword sep ucIdentifier optsep
                    "(" optsep
                    identifierList optsep
                    ")" optsep ";"

revisionStatement   = revisionKeyword optsep "{" stmtsep
                    dateStatement stmtsep
                    descriptionStatement stmtsep
```



```
        "}" optsep ";"

typedefTypeStatement = typeKeyword sep refinedBaseType optsep ";"
dateStatement       = dateKeyword sep date optsep ";"
organizationStatement = organizationKeyword sep text optsep ";"
contactStatement    = contactKeyword sep text optsep ";"
formatStatement     = formatKeyword sep format optsep ";"
unitsStatement      = unitsKeyword sep units optsep ";"
statusStatement     = statusKeyword sep status optsep ";"
accessStatement     = accessKeyword sep access optsep ";"
defaultStatement    = defaultKeyword sep anyValue optsep ";"
descriptionStatement = descriptionKeyword sep text optsep ";"
referenceStatement   = referenceKeyword sep text optsep ";"
abnfStatement       = abnfKeyword sep text optsep ";"

;;
;;
;;

refinedBaseType = IdentityKeyword /
                 ObjectIdentifierKeyword /
                 OctetStringKeyword *1(optsep numberSpec) /
                 PointerKeyword *1(optsep pointerSpec) /
                 Integer32Keyword *1(optsep numberSpec) /
                 Unsigned32Keyword *1(optsep numberSpec) /
                 Integer64Keyword *1(optsep numberSpec) /
                 Unsigned64Keyword *1(optsep numberSpec) /
                 Float32Keyword *1(optsep floatSpec) /
                 Float64Keyword *1(optsep floatSpec) /
                 Float128Keyword *1(optsep floatSpec) /
                 EnumerationKeyword
                               optsep namedSignedNumberSpec /
                 BitsKeyword optsep namedNumberSpec

refinedType = qucIdentifier *1(optsep anySpec)

anySpec = pointerSpec / numberSpec / floatSpec
```





```
pointerSpec      = "(" optsep qlcIdentifier optsep ")"
numberSpec       = "(" optsep numberElement
                  *furtherNumberElement
                  optsep ")"
furtherNumberElement = optsep "|" optsep numberElement
numberElement    = signedNumber *1numberUpperLimit
numberUpperLimit = optsep ".." optsep signedNumber
floatSpec        = "(" optsep floatElement
                  *furtherFloatElement
                  optsep ")"
furtherFloatElement = optsep "|" optsep floatElement
floatElement       = floatValue *1floatUpperLimit
floatUpperLimit    = optsep ".." optsep floatValue
namedNumberSpec   = "(" optsep namedNumberList optsep ")"
namedNumberList   = namedNumberItem
                  *(optsep "," optsep namedNumberItem)
                  *1(optsep ",")
namedNumberItem   = lcIdentifier optsep "(" optsep number
                  optsep ")"
namedSignedNumberSpec = "(" optsep namedSignedNumberList optsep ")"
namedSignedNumberList = namedSignedNumberItem
                      *(optsep "," optsep
                        namedSignedNumberItem)
                      *1(optsep ",")
namedSignedNumberItem = lcIdentifier optsep "(" optsep signedNumber
                      optsep ")"
identifierList      = identifier
                      *(optsep "," optsep identifier)
                      *1(optsep ",")
qIdentifierList     = qIdentifier
                      *(optsep "," optsep qIdentifier)
                      *1(optsep ",")
```



```
qlcIdentifierList      = qlcIdentifier
                        *(optsep "," optsep qlcIdentifier)
                        *1(optsep ",")

bitsValue              = "(" optsep bitsList optsep ")"

bitsList               = *1(lcIdentifier
                            *(optsep "," optsep lcIdentifier))
                            *1(optsep ",")

;;
;; Other basic rules.
;;

identifier             = ucIdentifier / lcIdentifier

qIdentifier            = qucIdentifier / qlcIdentifier

ucIdentifier           = ucAlpha *63(ALPHA / DIGIT / "-")

qucIdentifier          = *1(ucIdentifier "::") ucIdentifier

lcIdentifier           = lcAlpha *63(ALPHA / DIGIT / "-")

qlcIdentifier          = *1(ucIdentifier "::") lcIdentifier

attrIdentifier         = lcIdentifier *("." lcIdentifier)

qattrIdentifier        = *1(ucIdentifier ".") attrIdentifier

text                   = textSegment *(optsep textSegment)

textSegment            = DQUOTE *textAtom DQUOTE

textAtom               = textVChar / HTAB / SP / lineBreak

date                   = DQUOTE 4DIGIT "-" 2DIGIT "-" 2DIGIT
                        *1(" " 2DIGIT ":" 2DIGIT)
                        DQUOTE
                        ; always in UTC

format                 = textSegment

units                  = textSegment

anyValue               = bitsValue /
                        signedNumber /
                        hexadecimalNumber /
```



```

floatValue /
text /
objectIdentifier
; Note: `objectIdentifier' includes the
; syntax of enumeration labels and
; identities.
; They are not named literally to
; avoid reduce/reduce conflicts when
; building LR parsers based on this
; grammar.

status          = currentKeyword /
                 deprecatedKeyword /
                 obsoleteKeyword

access          = eventonlyKeyword /
                 readonlyKeyword /
                 readwriteKeyword

objectIdentifier = (qlcIdentifier / subid "." subid)
                  *127("." subid)

subid           = decimalNumber

number          = hexadecimalNumber / decimalNumber

negativeNumber  = "-" decimalNumber

signedNumber    = number / negativeNumber

decimalNumber   = "0" / (nonZeroDigit *DIGIT)

zeroDecimalNumber = 1*DIGIT

hexadecimalNumber = %x30 %x78 ; "0x" with x only lower-case
                    1*(HEXDIG HEXDIG)

floatValue      = neginfKeyword /
                 posinfKeyword /
                 snanKeyword /
                 qnanKeyword /
                 signedNumber "." zeroDecimalNumber
                 *1("E" ("+" / "-") zeroDecimalNumber)

;;
;; Rules to skip unknown statements
;; with arbitrary arguments and blocks.
;;

```



```

unknownStatement      = unknownKeyword optsep *unknownArgument
                        optsep ";"

unknownArgument       = ("(" optsep unknownList optsep ")") /
                        ("{" optsep *unknownStatement optsep "}") /
                        qucIdentifier /
                        anyValue /
                        anySpec

unknownList           = namedNumberList /
                        qIdentifierList

unknownKeyword        = lcIdentifier

```

```

;;
;; Keyword rules.
;;
;;
;; Typically, keywords are represented by tokens returned from the
;; lexical analyzer. Note, that the lexer has to be stateful to
;; distinguish keywords from identifiers depending on the context
;; position in the input stream.
;;
;;
;; Also note, that these keyword definitions are represented in
;; cleartext for readability, while SMIng keywords are meant to be
;; case-sensitive, although ABNF makes quoted strings like these to
;; be case-insensitive.
;;

```

```

;; Statement keywords. They must be lower-case.

```

```

moduleKeyword        = %x6D %x6F %x64 %x75 %x6C %x65
importKeyword        = %x69 %x6D %x70 %x6F %x72 %x74
revisionKeyword      = %x72 %x65 %x76 %x69 %x73 %x69 %x6F %x6E
dateKeyword          = %x64 %x61 %x74 %x65
organizationKeyword  = %x6F %x72 %x67 %x61 %x6E %x69 %x7A %x61 %x74
                    %x69 %x6F %x6E
contactKeyword       = %x63 %x6F %x6E %x74 %x61 %x63 %x74
descriptionKeyword   = %x64 %x65 %x73 %x63 %x72 %x69 %x70 %x74 %x69
                    %x6F %x6E
referenceKeyword     = %x72 %x65 %x66 %x65 %x72 %x65 %x6E %x63 %x65
extensionKeyword     = %x65 %x78 %x74 %x65 %x6E %x73 %x69 %x6F %x6E
typedefKeyword       = %x74 %x79 %x70 %x65 %x64 %x65 %x66
typeKeyword          = %x74 %x79 %x70 %x65
identityKeyword      = %x69 %x64 %x65 %x6E %x74 %x69 %x74 %x79
classKeyword         = %x63 %x6C %x61 %x73 %x73
attributeKeyword     = %x61 %x74 %x74 %x72 %x69 %x62 %x75 %x74 %x65
uniqueKeyword        = %x75 %x6E %x69 %x71 %x75 %x65
eventKeyword         = %x65 %x76 %x65 %x6E %x74

```





```

formatKeyword      = %x66 %x6F %x72 %x6D %x61 %x74
unitsKeyword       = %x75 %x6E %x69 %x74 %x73
statusKeyword      = %x73 %x74 %x61 %x74 %x75 %x73
accessKeyword      = %x61 %x63 %x63 %x65 %x73 %x73
defaultKeyword     = %x64 %x65 %x66 %x61 %x75 %x6C %x74
abnfKeyword        = %x61 %x62 %x6E %x66

```

;; Base type keywords.

```

OctetStringKeyword = %x4F %x63 %x74 %x65 %x74 %x53 %x74 %x72 %x69
                    %x6E %x67
PointerKeyword      = %x50 %x6F %x69 %x6E %x74 %x65 %x72
IdentityKeyword     = %x49 %x64 %x65 %x6E %x74 %x69 %x74 %x79
ObjectIdentifierKeyword = %x4F %x62 %x6A %x65 %x63 %x74 %x49 %x64
                    %x65 %x6E %x74 %x69 %x66 %x69 %x65 %x72
Integer32Keyword    = %x49 %x6E %x74 %x65 %x67 %x65 %x72 %x33 %x32
Unsigned32Keyword   = %x55 %x6E %x73 %x69 %x67 %x6E %x65 %x64 %x33
                    %x32
Integer64Keyword    = %x49 %x6E %x74 %x65 %x67 %x65 %x72 %x36 %x34
Unsigned64Keyword   = %x55 %x6E %x73 %x69 %x67 %x6E %x65 %x64 %x36
                    %x34
Float32Keyword      = %x46 %x6C %x6F %x61 %x74 %x33 %x32
Float64Keyword      = %x46 %x6C %x6F %x61 %x74 %x36 %x34
Float128Keyword     = %x46 %x6C %x6F %x61 %x74 %x31 %x32 %x38
BitsKeyword         = %x42 %x69 %x74 %x73
EnumerationKeyword  = %x45 %x6E %x75 %x6D %x65 %x72 %x61 %x74 %x69
                    %x6F %x6E

```

;; Status keywords.

```

currentKeyword      = %x63 %x75 %x72 %x72 %x65 %x6E %x74
deprecatedKeyword   = %x64 %x65 %x70 %x72 %x65 %x63 %x61 %x74 %x65
                    %x64
obsoleteKeyword     = %x6F %x62 %x73 %x6F %x6C %x65 %x74 %x65

```

;; Access keywords.

```

eventonlyKeyword    = %x65 %x76 %x65 %x6E %x74 %x6F %x6E %x6C %x79
readonlyKeyword     = %x72 %x65 %x61 %x64 %x6F %x6E %x6C %x79
readwriteKeyword    = %x72 %x65 %x61 %x64 %x77 %x72 %x69 %x74 %x65

```

;; Special floating point values' keywords.

```

neginfKeyword       = %x6E %x65 %x67 %x69 %x6E %x66
posinfKeyword       = %x70 %x6F %x73 %x69 %x6E %x66
snanKeyword         = %x73 %x6E %x61 %x6E
qnanKeyword         = %x71 %x6E %x61 %x6E

```



```
;;
;; Some low level rules.
;; These tokens are typically skipped by the lexical analyzer.
;;

sep                = 1*(comment / lineBreak / WSP)
                   ; unconditional separator

optsep            = *(comment / lineBreak / WSP)

stmtsep           = *(comment /
                   lineBreak /
                   WSP /
                   unknownStatement)

comment           = "//" *(WSP / VCHAR) lineBreak

lineBreak         = CRLF / LF

;;
;; Encoding specific rules.
;;

textVChar         = %x21 / %x23-7E
                   ; any VCHAR except DQUOTE

ucAlpha           = %x41-5A

lcAlpha           = %x61-7A

nonZeroDigit     = %x31-39

;;
;; RFC 2234 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 / %x63 / %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

;;
;; EOF
;;
```

## [Appendix B. OPEN ISSUES](#)

Set of Documents - What is the expected set of documents specifying the SMIng? Currently, it looks like we are going to define these:

- (a) a core SMIng language specification,
- (b) specification of a core SMIng module,
- (c) language extensions for SNMP mappings,
- (d) language extensions for COPS-PR mappings,
- (e) maybe, an SMIng guidelines document,
- (f) specification of a basic inet modules, that not only contain basic definitions but also document the usage SMIng.

How Generic Shall the Core Language be? - If we focus strictly on SNMP and COPS-PR, we can build on some common characteristics in these two related worlds, e.g., the concept and common definitions of OIDs, and conformance statements. On the other hand, if we feel closer to OO modeling concepts that should remain applicable also to other environments (AAA/DIAMETER, XML-style definitions), more information has to be specified in the mappings to those



environments, while the core language cannot be very expressive.

Floating Point Types - Shall we include Float32/64/128 in the base type system? Maybe only Float32/64? If we do, shall we disallow restrictions? See also the requirements document.

Events / NOTIFICATIONS - SMIV2 NOTIFICATIONS contain objects. How about SMIng? Assume, the clause is named `event'. Shall events carry a set of attributes? How about those attributes identifying an instance of a class? Currently, events are associated with a class. What attributes are carried with an event is subject to the protocol mapping.

Display Formats - Should display hints be usable in a reversed way? Check all variants carefully. Is the optional repeat indicator `\*' necessary? Would `u' for unsigned integers be useful?

Discriminated Unions - How to specify unions and their discriminators? `typemap' statement? What are the specific requirements? See also the requirements document.

How To Read - Add a section on how to read this set of documents.

Annotations - Make annotations a core feature of SMIng? They are used to add information to an existent definition in an external module, e.g., a vendor or user can add specific severity level information to standard event definitions.

Glossary - Add/Update the glossary of terms.

Module Naming Scheme - Propose well known module name suffixes: `-MIB' for SNMP mapping modules? `-PIB' for COPS-PR mapping modules? `-EXT' for modules that define extensions (e.g. snmp)? no extension for modules that define general classes and types? This should go to the Guidelines document.

`Extending A Module' - Carefully adjust the rules, e.g., `new named numbers may be added to enumeration types' is in contradiction with `attributes may get a new type only if the set of values remains equal'.

ABNF Statement - Is the `abnf' statement really meaningful? Someone stated that it could be abused.

7-bit ASCII texts - See requirements.

Module Namespaces - Should we introduce domain-based namespaces for module names? E.g., DISMAN-SCRIPT-MIB.ietf.org? Mapping to





SMIV2/SPPI module names? Which parts are case-sensitive? Separator char between module name and domain name (@/.)? Or should we enforce organization prefixes (also for the IETF), like IETF-DISMAN-SCRIPT-MIB?

Learn from ODL, XML, ODBMS - Look at the ODL proposal from TINAC. Look at the XML schema work from W3C. Look at the ODBMS work.

Inheritance - Inheritance is a powerful technique in software development. But is it really what we want to have in management data modeling? If it is not easy to find good examples for inheritance, can we expect that people will know how to use it? Or would it be more likely that it will be misused? Maybe, containment/discriminated unions are what we really need.

Examples for Primary goals: MIBs/PIBs - Keep in mind that the primary goal is to derive modules for use with SNMP and COPS-PR from common definitions. If we cannot easily give good examples, we have failed.

Classes or Interfaces - Are classes really classes or are they more interfaces?

Reusable event definitions - Currently events are defined within a class. Do we need to be able to reuse event definitions in multiple classes? This potentially requires to give events their own names, independent of any class definitions. Or is it sufficient to use inheritance/containment to handle 99 % of the cases?

Extensions - Optionally require the understanding of imported extensions (similar to the marvelous diameter M bit ;-)

Extension Context - Do we need a mechanism to allow an extension to specify the context in which it can be used (the containing statement block in the position within this block)?

`Static' Definitions - Is it useful to make specific definitions non-exported (like `static' in C)? Or would it be useful to make only those definitions be exported that are explicitly marked (`public')?

More Formal Restrictions? - Do we need further formal restrictions on type definitions, e.g. subtyping not allowed on TimeTicks, max-access read-only on Counters, no default values on Counters?



## Full Copyright Statement

Copyright (C) The Internet Society (2001). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

