INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.680

(07/2002)

SERIES X: DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

OSI networking and system aspects – Abstract Syntax Notation One (ASN.1)

Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation

ITU-T Recommendation X.680

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

PUBLIC DATA NETWORKS	
Services and facilities	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalling and switching	X.50–X.89
Network aspects	X.90–X.149
Maintenance	X.150–X.179
Administrative arrangements	X.180–X.199
OPEN SYSTEMS INTERCONNECTION	
Model and notation	X.200–X.209
Service definitions	X.210–X.219
Connection-mode protocol specifications	X.220–X.229
Connectionless-mode protocol specifications	X.230–X.239
PICS proformas	X.240–X.259
Protocol Identification	X.260–X.269
Security Protocols	X.270–X.279
Layer Managed Objects	X.280–X.289
Conformance testing	X.290–X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300–X.349
Satellite data transmission systems	X.350–X.369
IP-based networks	X.370–X.399
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600–X.629
Efficiency	X.630–X.639
Quality of service	X.640–X.649
Naming, Addressing and Registration	X.650–X.679
Abstract Syntax Notation One (ASN.1)	**X.680–X.699**
OSI MANAGEMENT	
Systems Management framework and architecture	X.700–X.709
Management Communication Service and Protocol	X.710–X.719
Structure of Management Information	X.720–X.729
Management functions and ODMA functions	X.730–X.799
SECURITY	X.800–X.849
OSI APPLICATIONS	
Commitment, Concurrency and Recovery	X.850–X.859
Transaction processing	X.860–X.879
Remote operations	X.880–X.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999

For further details, please refer to the list of ITU-T Recommendations.

INTERNATIONAL STANDARD ISO/IEC 8824-1

ITU-T RECOMMENDATION X.680

Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation

Summary

This Recommendation | International Standard provides a notation called Abstract Syntax Notation One (ASN.1) for defining the syntax of information data. It defines a number of simple data types and specifies a notation for referencing these types and for specifying values of these types.

The ASN.1 notations can be applied whenever it is necessary to define the abstract syntax of information without constraining in any way how the information is encoded for transmission.

Source

ITU-T Recommendation X.680 was prepared by ITU-T Study Group 17 (2001-2004) and approved on 14 July 2002. An identical text is also published as ISO/IEC 8824-1.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU [had/had not] received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

Introduction

This Recommendation | International Standard presents a standard notation for the definition of data types and values. A *data type* (or *type* for short) is a category of information (for example, numeric, textual, still image or video information). A *data value* (or *value* for short) is an instance of such a type. This Recommendation | International Standard defines several basic types and their corresponding values, and rules for combining them into more complex types and values.

In some protocol architectures, each message is specified as the binary value of a sequence of octets. However, standards-writers need to define quite complex data types to carry their messages, without concern for their binary representation. In order to specify these data types, they require a notation that does not necessarily determine the representation of each value. ASN.1 is such a notation. This notation is supplemented by the specification of one or more algorithms called *encoding rules* that determine the value of the octets that carry the application semantics (called the *transfer syntax*). ITU-T Rec. X.690 | ISO/IEC 8825-1, ITU-T Rec. X.691 | ISO/IEC 8825-2 and ITU-T Rec. X.693 | ISO/IEC 8825-4 specify three families of standardized encoding rules, called *Basic Encoding Rules (BER)*, *Packed Encoding Rules (PER)*, and *XML Encoding Rules (XER)*.

Some users wish to redefine their legacy protocols using ASN.1, but cannot use standardized encoding rules because they need to retain their existing binary representations. Other users wish to have more complete control over the exact layout of the bits on the wire (the transfer syntax). These requirements are addressed by ITU-T Rec. X.692 | ISO/IEC 8825-3 which specifies an *Encoding Control Notation (ECN)* for ASN.1. ECN enables designers to formally specify the abstract syntax of a protocol using ASN.1, but to then (if they so wish) take complete or partial control of the bits on the wire by writing an accompanying ECN specification (which may reference standardized Encoding Rules for some parts of the encoding).

A very general technique for defining a complicated type at the abstract level is to define a small number of *simple types* by defining all possible values of the simple types, then combining these simple types in various ways. Some of the ways of defining new types are as follows:

a) given an (ordered) list of existing types, a value can be formed as an (ordered) sequence of values, one from each of the existing types; the collection of all possible values obtained in this way is a new type (if the existing types in the list are all distinct, this mechanism can be extended to allow omission of some values from the list);

b) given an unordered set of (distinct) existing types, a value can be formed as an (unordered) set of values, one from each of the existing types; the collection of all possible unordered sets of values obtained in this way is a new type (the mechanism can again be extended to allow omission of some values);

c) given a single existing type, a value can be formed as an (ordered) list or (unordered) set of zero, one or more values of the existing type; the collection of all possible lists or sets of values obtained in this way is a new type;

d) given a list of (distinct) types, a value can be chosen from any one of them; the set of all possible values obtained in this way is a new type;

e) given a type, a new type can be formed as a subset of it by using some structure or order relationship among the values.

An important aspect of combining types in this way is that encoding rules should recognize the combining constructs, providing unambiguous encodings of the collection of values of the basic types. Thus, every basic type defined using the notation specified in this Recommendation | International Standard is assigned a *tag* to aid in the unambiguous encoding of values.

Tags are mainly intended for machine use, and are not essential for the human notation defined in this Recommendation | International Standard. Where, however, it is necessary to require that certain types be distinct, this is expressed by requiring that they have distinct tags. The allocation of tags is therefore an important part of the use of this notation, but (since 1994) it is possible to specify the automatic allocation of tags.

NOTE 1 – Within this Recommendation | International Standard, tag values are assigned to all simple types and construction mechanisms. The restrictions placed on the use of the notation ensure that tags can be used in transfer for unambiguous identification of values.

An ASN.1 specification will initially be produced with a set of fully defined ASN.1 types. At a later stage, however, it may be necessary to change those types (usually by the addition of extra components in a sequence or set type). If this is to be possible in such a way that implementations using the old type definitions can interwork with implementations using the new type definitions in a defined way, encoding rules need to provide appropriate support. The ASN.1 notation supports the inclusion of an *extension marker* on a number of types. This signals to encoding rules the intention of the designer that this type is one of a series of related types (i.e., versions of the same initial type) called an *extension series*, and that the encoding rules are required to enable information transfer between implementations using different types that are related by being part of the same extension series.

Clauses 10 to 31 (inclusive) define the simple types supported by ASN.1, and specify the notation to be used for referencing simple types and for defining new types using them. Clauses 10 to 31 also specify notations to be used for specifying values of types defined using ASN.1. Two value notations are provided. The first is called the basic ASN.1 value notation, and has been part of the ASN.1 notation since its first introduction. The second is called the XML ASN.1 Value Notation, and provides a value notation using Extensible Markup Language (XML).

NOTE 2 – The XML Value Notation provides a means of representing ASN.1 values using XML. Thus, an ASN.1 type definition also specifies the structure and content of an XML element. This makes ASN.1 a simple schema language for XML.

Clauses 33 to 34 (inclusive) define the types supported by ASN.1 for carrying within them the complete encoding of ASN.1 types.

Clauses 35 to 40 (inclusive) define the character string types.

Clauses 41 to 44 (inclusive) define certain types which are considered to be of general utility, but which require no additional encoding rules.

Clauses 45 to 47 (inclusive) define a notation which enables subtypes to be defined from the values of a parent type.

Clause 48 defines a notation which allows ASN.1 types specified in a "version 1" specification to be identified as likely to be extended in "version 2", and for additions made in subsequent versions to be separately listed and identified with their version number.

Clause 49 defines a notation which allows ASN.1 type definitions to contain an indication of the intended error handling if encodings are received for values which lie outside those specified in the current standardized definition.

Annex A forms an integral part of this Recommendation | International Standard, and specifies ASN.1 regular expressions.

Annex B forms an integral part of this Recommendation | International Standard, and specifies rules for type and value compatibility.

Annex C forms an integral part of this Recommendation | International Standard, and records object identifier and object descriptor values assigned in the ASN.1 series of Recommendations | International Standards.

Annex D does not form an integral part of this Recommendation | International Standard, and describes the top-level arcs of the registration tree for object identifiers.

Annex E does not form an integral part of this Recommendation | International Standard, and provides examples and hints on the use of the ASN.1 notation.

Annex F does not form an integral part of this Recommendation | International Standard, and provides a tutorial on ASN.1 character strings.

Annex G does not form an integral part of this Recommendation | International Standard, and provides a tutorial on the ASN.1 model of type extension.

Annex H does not form an integral part of this Recommendation | International Standard, and provides a summary of ASN.1 using the notation of clause 5.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation

1 Scope

This Recommendation | International Standard provides a standard notation called Abstract Syntax Notation One (ASN.1) that is used for the definition of data types, values, and constraints on data types.

This Recommendation | International Standard:

- defines a number of simple types, with their tags, and specifies a notation for referencing these types and for specifying values of these types;
- defines mechanisms for constructing new types from more basic types, and specifies a notation for defining such types and assigning them tags, and for specifying values of these types;
- defines character sets (by reference to other Recommendations and/or International Standards) for use within ASN.1.

The ASN.1 notation can be applied whenever it is necessary to define the abstract syntax of information.

The ASN.1 notation is referenced by other standards which define encoding rules for the ASN.1 types.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

- CCITT Recommendation X.660 (1992) | ISO/IEC 9834-1:1993*, Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: General procedures: (plus amendments)*.
- ITU-T Recommendation X.681 (2002) | ISO/IEC 8824-2:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.
- ITU-T Recommendation X.682 (2002) | ISO/IEC 8824-3:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification*.
- ITU-T Recommendation X.683 (2002) | ISO/IEC 8824-4:2002, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.
- ITU-T Recommendation X.690 (2002) | ISO/IEC 8825-1:2002, *Information technology – ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*.
- ITU-T Recommendation X.691 (2002) | ISO/IEC 8825-2:2002, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*.
- ITU-T Recommendation X.692 (2002) | ISO/IEC 8825-3:2002, *Information technology – ASN.1 encoding rules: Specification of Encoding Control Notation (ECN)*.

– ITU-T Recommendation X.693 (2001) | ISO/IEC 8825-4:2002, *Information technology – ASN.1 encoding rules: XML Encoding Rules (XER)*.

2.2 Additional references

– ITU-R Recommendation TF.460-5 (1997), *Standard-frequency and time-signal emissions.*

– CCITT Recommendation T.100 (1988), *International information exchange for interactive videotex.*

– ITU-T Recommendation T.101 (1994), *International interworking for videotex services.*

– ISO *International Register of Coded Character Sets to be used with Escape Sequences*.

– ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*.

– ISO/IEC 2022:1994, *Information technology – Character code structure and extension techniques*.

– ISO/IEC 6523:1998, *Data interchange – Structures for the identification of organizations*.

– ISO/IEC 7350:1991, *Information technology – Registration of repertoires of graphic characters from ISO/IEC 10367*.

– ISO 8601:2000, *Data elements and interchange formats – Information interchange – Representation of dates and times*.

– ISO/IEC 10646-1:2000, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

– The Unicode Standard, Version 3.2.0:2002. The Unicode Consortium. (Reading, MA, Addison-Wesley)

NOTE 1 – The above reference is included because it provides names for control characters.

– W3C XML 1.0:2000, *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, Copyright © [6 October 2000] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), http://www.w3.org/TR/2000/REC-xml-20001006.*

NOTE 2 – The reference to a document within this Recommendation | International Standard does not give it, as a stand-alone document, the status of a Recommendation or International Standard.

3 Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply.

3.1 Information object specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.681 | ISO/IEC 8824-2:

a) information object;

b) information object class;

c) information object set;

d) instance-of type;

e) object class field type.

3.2 Constraint specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.682 | ISO/IEC 8824-3:

a) component relation constraint;

b) table constraint.

3.3 Parameterization of ASN.1 specification

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

a) parameterized type;

b) parameterized value.

3.4 Structure for identification of organizations

This Recommendation | International Standard uses the following terms defined in ISO/IEC 6523:

a) issuing organization;
b) organization code;
c) International Code Designator.

3.5 Universal Multiple-Octet Coded Character Set (UCS)

This Recommendation | International Standard uses the following terms defined in ISO/IEC 10646-1:

a) Basic Multilingual Plane (BMP);
b) cell;
c) combining character;
d) graphic symbol;
e) group;
f) limited subset;
g) plane;
h) row;
i) selected subset.

3.6 Additional definitions

3.6.1 abstract character: An abstract value which is used for the organization, control or representation of textual data.

NOTE – Annex F provides a more complete description of the term abstract character.

3.6.2 abstract value: A value whose definition is based only on the type used to carry some semantics, independently of how it is represented in any encoding.

NOTE – Examples of abstract values are the values of the integer type, the boolean type, a character string type, or of a type which is a sequence (or a choice) of an integer and a boolean.

3.6.3 ASN.1 character set: The set of characters, specified in clause 10, used in the ASN.1 notation.

3.6.4 ASN.1 specification: A collection of one or more ASN.1 modules.

3.6.5 associated type: A type which is used only for defining the value and subtype notation for a type.

NOTE – Associated types are defined in this Recommendation | International Standard when it is necessary to make it clear that there may be a significant difference between how the type is defined in ASN.1 and how it is encoded. Associated types do not appear in user specifications.

3.6.6 bitstring type: A simple type whose distinguished values are an ordered sequence of zero, one or more bits.

NOTE – Where there is a need to carry embedded encodings of an abstract value, the use of a bitstring (or an octetstring) type without a contents constraint (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11) is deprecated. Otherwise, the use of the embedded-pdv type (see clause 33) provides a more flexible mechanism, allowing the announcement of the abstract syntax and of the encoding of the abstract value that is embedded.

3.6.7 boolean type: A simple type with two distinguished values.

3.6.8 character property: The set of information associated with a cell in a table defining a character repertoire.

NOTE – The information will normally include some or all of the following items:

a) a graphic symbol;
b) a character name;
c) the definition of functions associated with the character when used in particular environments;
d) whether it represents a digit;
e) an associated character differing only in (upper/lower) case.

3.6.9 character abstract syntax: Any abstract syntax whose values are specified as the set of character strings of zero, one or more characters from some specified collection of characters.

3.6.10 character repertoire: The characters in a character set without any implication on how such characters are encoded.

3.6.11 character string types: Simple types whose values are strings of characters from some defined character set.

3.6.12 character transfer syntax: Any transfer syntax for a character abstract syntax.

NOTE – ASN.1 does not support character transfer syntaxes which do not encode all character strings as an integral multiple of 8 bits.

3.6.13 choice types: Types defined by referencing a list of distinct types; each value of the choice type is derived from the value of one of the component types.

3.6.14 component type: One of the types referenced when defining a **CHOICE**, **SET**, **SEQUENCE**, **SET OF**, or **SEQUENCE OF**.

3.6.15 constraint: A notation which can be used in association with a type, to define a subtype of that type.

3.6.16 contents constraint: A constraint on a bit string or octet string type that specifies either that the contents are to be an encoding of a specified ASN.1 type, or that specified procedures are to be used to produce and process the contents.

3.6.17 control characters: Characters appearing in some character repertoires that have been given a name (and perhaps a defined function in relation to certain environments) but which have not been assigned a graphic symbol, and which are not spacing characters.

NOTE – HORIZONTAL TABULATION (9) and LINE FEED (10) are examples of control characters that have been assigned a formatting function in a printing environment. DATA LINK ESCAPE (16) is an example of a control character that has been assigned a function in a communication environment.

3.6.18 Coordinated Universal Time (UTC): The time scale maintained by the Bureau International de l'Heure (International Time Bureau) that forms the basis of a coordinated dissemination of standard frequencies and time signals.

NOTE 1 – The source of this definition is ITU-R Rec. TF.460-5. ITU-R has also defined the acronym for Coordinated Universal Time as UTC.

NOTE 2 – UTC and Greenwich Mean Time (GMT) are two alternative time standards which for most practical purposes determine the same time.

3.6.19 element: A value of a governing type or an information object of a governing information object class, distinguable from all other values of the same type or information objects of the same class, respectively.

3.6.20 element set: A set of elements, all of which are values of a governing type, or information objects of a governing class.

NOTE – Governing class is defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 3.4.7.

3.6.21 embedded-pdv type: A type whose set of values is formally the union of the sets of values in all possible abstract syntaxes. This type can be used in an ASN.1 specification that wishes to carry in its protocol an abstract value whose type may be defined externally to that ASN.1 specification. It carries an identification of the abstract syntax (the type) of the abstract value being carried, as well as an identification of the encoding rules used to encode that abstract value.

3.6.22 encoding: The bit-pattern resulting from the application of a set of encoding rules to an abstract value.

3.6.23 (ASN.1) encoding rules: Rules which specify the representation during transfer of the values of ASN.1 types. Encoding rules also enable the values to be recovered from the representation, given knowledge of the type.

NOTE – For the purpose of specifying encoding rules, the various referenced type (and value) notations, which can provide alternative notations for built-in types (and values), are not relevant.

3.6.24 enumerated types: Simple types whose values are given distinct identifiers as part of the type notation.

3.6.25 extension addition: One of the added notations in an extension series. For set, sequence and choice types, each extension addition is the addition of either a single extension addition group or a single component type. For enumerated types it is the addition of a single further enumeration. For a constraint it is the addition of (only) one subtype element.

NOTE – Extension additions are both textually ordered (following the extension marker) and logically ordered (having increasing enumeration values, and, in the case of **CHOICE** alternatives, increasing tags).

3.6.26 extension addition group: One or more components of a set, sequence or choice type grouped within version brackets. An extension addition group is used to clearly identify the components of a set, sequence or choice type that were added in a particular version of an ASN.1 module, and can identify that version with a simple integer.

3.6.27 extension addition type: A type contained within an extension addition group or a single component type that is itself an extension addition (in such a case it is not contained within an extension addition group).

3.6.28 extensible constraint: A subtype constraint with an extension marker at the outer level, or that is extensible through the use of set arithmetic with extensible sets of values.

3.6.29 extension insertion point (or insertion point): The location within a type definition where extension additions are inserted. This location is the end of the type notation of the immediately preceding type in the extension series if there is

a single ellipsis in the type definition, or immediately before the second ellipsis if there is an extension marker pair in the definition of the type.

NOTE – There can be at most one insertion point within the components of any choice, sequence, or set type.

3.6.30 extension marker: A syntactic flag (an ellipsis) that is included in all types that form part of an extension series.

3.6.31 extension marker pair: A pair of extension markers between which extension additions are inserted.

3.6.32 extension-related: Two types that have the same extension root, where one was created by adding zero or more extension additions to the other.

3.6.33 extension root: An extensible type that is the first type in an extension series. It carries either the extension marker with no additional notation other than comments and white-space between the extension marker and the matching "}" or ")", or an extension marker pair with no additional notation other than a single comma, comments and white-space between the extension markers.

NOTE – Only an extension root can be the first type in an extension series.

3.6.34 extension series: A series of ASN.1 types which can be ordered in such a way that each successive type in the series is formed by the addition of text at the extension insertion point.

3.6.35 extensible type: A type with an extension marker, or to which an extensible constraint has been applied.

3.6.36 external reference: A type reference, value reference, information object class reference, information object reference, or information object set reference (which may be parameterized), that is defined in some other module than the one in which it is being referenced, and which is being referred to by prefixing the module name to the referenced item.

EXAMPLE – **`ModuleName.TypeReference`**

3.6.37 external type: A type which is a part of an ASN.1 specification that carries a value whose type may be defined externally to that ASN.1 specification. It also carries an identification of the type of the value being carried.

3.6.38 false: One of the distinguished values of the boolean type (see also "true").

3.6.39 governing (type); governor: A type definition or reference which affects the interpretation of a part of the ASN.1 syntax, requiring that part of the ASN.1 syntax to reference values in the governing type.

3.6.40 identical type definitions: Two instances of the ASN.1 "Type" production (see clause 16) are defined as identical type definitions if, after performing the transformations specified in Annex B, they are identical ordered lists of identical lexical items (see clause 11).

3.6.41 integer type: A simple type with distinguished values which are the positive and negative whole numbers, including zero (as a single value).

NOTE – When particular encoding rules limit the range of an integer, such limitations are chosen so as not to affect any user of ASN.1.

3.6.42 lexical item: A named sequence of characters from the ASN.1 character set, specified in clause 11, which is used in forming the ASN.1 notation.

3.6.43 module: One or more instances of the use of the ASN.1 notation for type, value, value set, information object class, information object, and information object set (as well as the parameterized variant of those), encapsulated using the ASN.1 module notation (see clause 12).

NOTE – The terms information object class (etc.) are specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, and parameterization is specified in ITU-T Rec. X.683 | ISO/IEC 8824-4.

3.6.44 null type: A simple type consisting of a single value, also called null.

3.6.45 object: A well-defined piece of information, definition, or specification which requires a name in order to identify its use in an instance of communication.

NOTE – Such an object may be an information object as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2.

3.6.46 object descriptor type: A type whose distinguished values are human-readable text providing a brief description of an object (see 3.6.45).

NOTE – An object descriptor value is usually associated with a single object. Only an object identifier value unambiguously identifies an object.

3.6.47 object identifier: A globally unique value associated with an object to unambiguously identify it.

3.6.48 object identifier type: A simple type whose values are the set of all object identifiers allocated in accordance with the rules of ITU-T Rec. X.660 | ISO/IEC 9834 series.

NOTE – The rules of ITU-T Rec. X.660 | ISO/IEC 9834-1 permit a wide range of authorities to independently associate object identifiers with objects.

3.6.49 octetstring type: A simple type whose distinguished values are an ordered sequence of zero, one or more octets, each octet being an ordered sequence of eight bits.

3.6.50 open systems interconnection: An architecture for computer communication which provides a number of terms which are used in this Recommendation | International Standard preceded by the abbreviation "OSI".

NOTE – The meaning of such terms can be obtained from the ITU-T Rec. X.200 series and equivalent ISO/IEC Standards if needed. The terms are only applicable if ASN.1 is used in an OSI environment.

3.6.51 open type notation: An ASN.1 notation used to denote a set of values from more than one ASN.1 type.

NOTE 1 – The term "open type" is used synonymously with "open type notation" in the body of this Recommendation | International Standard.

NOTE 2 – All ASN.1 encoding rules provide unambiguous encodings for the values of a single ASN.1 type. They do not necessarily provide unambiguous encodings for "open type notation", which carries values from ASN.1 types that are not normally determined at specification time. Knowledge of the type of the value being encoded in the "open type notation" is needed before the abstract value for that field can be unambiguously determined.

NOTE 3 – The only notation in this Recommendation | International Standard which is an open type notation is the "ObjectClassFieldType" specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 14, where the "FieldName" denotes either a type field or a variable-type value field.

3.6.52 parent type (of a subtype): The type that is being constrained when defining a subtype, and which governs the subtype notation.

NOTE – The parent type may itself be a subtype of some other type.

3.6.53 production: A part of the formal notation (also called grammar or Backus-Naur Form, BNF) used to specify ASN.1.

3.6.54 real type: A simple type whose distinguished values (specified in clause 20) are members of the set of real numbers.

3.6.55 recursive definition (of a type): A set of ASN.1 definitions which cannot be reordered so that all types used in a construction are defined before the definition of the construction.

NOTE – Recursive definitions are allowed in ASN.1: the user of the notation has the responsibility for ensuring that those values (of the resulting types) which are used have a finite representation and that the value set associated with the type contains at least one value.

3.6.56 relative object identifier: A value which identifies an object by its position relative to some known object identifier (see 3.6.47).

3.6.57 relative object identifier type: A simple type whose values are the set of all possible relative object identifiers.

3.6.58 restricted character string type: A character string type whose characters are taken from a fixed character repertoire identified in the type specification.

3.6.59 selection types: Types defined by reference to a component type of a choice type, and whose values are precisely the values of that component type.

3.6.60 sequence types: Types defined by referencing a fixed, ordered list of types (some of which may be declared to be optional); each value of the sequence type is an ordered list of values, one from each component type.

NOTE – Where a component type is declared to be optional, a value of the sequence type need not contain a value of that component type.

3.6.61 sequence-of types: Types defined by referencing a single component type; each value in the sequence-of type is an ordered list of zero, one or more values of the component type.

3.6.62 serial application (of constraints): The application of a constraint to a parent type which is already constrained.

3.6.63 set arithmetic: The formation of new sets of values or information objects using the operations of union, intersection and set difference (use of **EXCEPT**) as specified in 46.2.

NOTE – The result of serial application of constraints is not covered by the term "set arithmetic".

3.6.64 set types: Types defined by referencing a fixed, unordered, list of types (some of which may be declared to be optional); each value in the set type is an unordered list of values, one from each component type.

NOTE – Where a component type is declared to be optional, a value of the set type need not contain a value of that component type.

3.6.65 set-of types: Types defined by referencing a single component type; each value in the set-of type is an unordered list of zero, one or more values of the component type.

3.6.66 simple types: Types defined by directly specifying the set of their values.

3.6.67 spacing character: A character in a character repertoire which is intended for inclusion with graphic characters in the printing of a character string but which is represented in the physical rendition by empty space; it is not normally considered to be a control character (see 3.6.17).

NOTE – There may be a single spacing character in the character repertoire, or there may be multiple spacing characters with varying widths.

3.6.68 subtype (of a parent type): A type whose values are a subset (or the complete set) of the values of some other type (the parent type).

3.6.69 tag: A type denotation which is associated with every ASN.1 type.

3.6.70 tagged types: A type defined by referencing a single existing type and a tag; the new type is isomorphic to the existing type, but is distinct from it.

3.6.71 tagging: Replacing the existing (possibly the default) tag of a type by a specified tag.

3.6.72 transfer syntax: The set of bit strings used to exchange the abstract values in an abstract syntax, usually obtained by application of encoding rules to an abstract syntax.

NOTE – The term "transfer syntax" is synonymous with "encoding".

3.6.73 true: One of the distinguished values of the boolean type (see also "false").

3.6.74 type: A named set of values.

3.6.75 type reference name: A name associated uniquely with a type within some context.

NOTE – Reference names are assigned to the types defined in this Recommendation | International Standard; these are universally available within ASN.1. Other reference names are defined in other Recommendations | International Standards, and are applicable only in the context of that Recommendation | International Standard.

3.6.76 unrestricted character string type: A type whose abstract values are values from a character abstract syntax, together with an identification of the character abstract syntax and of the character transfer syntax to be used in its encoding.

3.6.77 user (of ASN.1): The individual or organization that defines the abstract syntax of a particular piece of information using ASN.1.

3.6.78 value mapping: A 1-1 relationship between values in two types that enables a reference to one of those values to be used as a reference to the other value. This can, for example, be used in specifying subtypes and default values (see Annex B).

3.6.79 value reference name: A name associated uniquely with a value within some context.

3.6.80 value set: A collection of values of a type. Semantically equivalent to a subtype.

3.6.81 version brackets: A pair of adjacent left and right brackets ("**[[**" or "**]]**") used to delineate the start and end of an extension addition group. The pair of left brackets can optionally be followed by a number giving a version number for the extension addition group.

3.8.82 version number: A number which can be associated with a version bracket (see G.1.8).

NOTE – A version number cannot be added to an extension addition which is not part of an extension addition group, nor to extension additions to any type other than choice, sequence, or set.

3.6.83 white-space: Any formatting action that yields a space on a printed page, such as spaces or tabs.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules of ASN.1
BMP	Basic Multilingual Plane
DCC	Data Country Code
DNIC	Data Network Identification Code

ECN	Encoding Control Notation of ASN.1
ICD	International Code Designator
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
OID	Object Identifier
OSI	Open Systems Interconnection
PER	Packed Encoding Rules of ASN.1
ROA	Recognized Operating Agency
UCS	Universal Multiple-Octet Coded Character Set
UTC	Coordinated Universal Time
XML	Extensible Markup Language

5 Notation

5.1 General

5.1.1 The ASN.1 notation consists of a sequence of characters from the ASN.1 character set specified in clause 10.

5.1.2 Each use of the ASN.1 notation contains characters from the ASN.1 character set grouped into lexical items. Clause 11 specifies all the sequences of characters forming lexical items, and names each item.

5.1.3 The ASN.1 notation is specified in clause 12 (and following clauses) by specifying and naming those sequences of lexical items which form valid instances of the ASN.1 notation, and by specifying the ASN.1 semantics of each sequence.

5.1.4 In order to specify the permitted sequences of lexical items, this Recommendation | International Standard uses a formal notation defined in the following subclauses.

5.2 Productions

5.2.1 All lexical items are named (see clause 11), and permitted sequences of lexical items are named.

5.2.2 A new (more complex) permitted sequence of lexical items is defined by means of a production. This uses the names of lexical items and of permitted sequences of lexical items and forms a new named permitted sequence of lexical items.

5.2.3 Each production consists of the following parts, on one or several lines, in order:

a) a name for the new permitted sequence of lexical items;

b) the characters

::=

c) one or more alternative sequences of lexical items, as defined in 5.3, separated by the character

|

5.2.4 A sequence of lexical items is present in the new permitted sequence of lexical items if it is present in one or more of the alternatives. The new permitted sequence of lexical items is referenced in this Recommendation | International Standard by the name in 5.2.3 a) above.

NOTE – If the same sequence of lexical items appears in more than one alternative, any semantic ambiguity in the resulting notation is resolved by associated text.

5.3 The alternative collections

5.3.1 Each alternative in a production (see 5.2.3.c) is specified by a list of names. Each name is either the name of a lexical item, or is the name of a permitted sequence of lexical items defined and named by some other production.

5.3.2 The permitted sequence of lexical items defined by each alternative consists of all sequences obtained by taking any one of the sequences (or the lexical item) associated with the first name, in combination with (and followed

by) any one of the sequences (or lexical item) associated with the second name, in combination with (and followed by) any one of the sequences (or lexical item) associated with the third name, and so on up to and including the last name (or lexical item) in the alternative.

5.4 Non-spacing indicator

If the non-spacing indicator "**&**" (AMPERSAND) is inserted between these items in production sequences, then the lexical item that precedes it and the lexical item that follows it shall not be separated by white-space.

NOTE – This indicator is only used in productions that describe the XML value notation. For example, it is used to specify that the lexical item "<" is to be immediately followed by an XML tag name.

5.5 Example of a production

5.5.1 The production:

```
ExampleProduction    ::=
        bstring
    |   hstring
    |   "{" IdentifierList "}"
```

associates the name "ExampleProduction" with the following sequences of lexical items:

a) any "bstring" (a lexical item); or

b) any "hstring" (a lexical item); or

c) any sequence of lexical items associated with "IdentifierList", preceded by a "**{**" and followed by a "**}**".

NOTE – "**{**" and "**}**" are the names of lexical items containing the single characters **{** and **}** (see 11.26).

5.5.2 In this example, "IdentifierList" would be defined by a further production, either before or after the production defining "ExampleProduction".

5.6 Layout

Each production used in this Recommendation | International Standard is preceded and followed by an empty line. Empty lines do not appear within productions. The production may be on a single line, or may be spread over several lines. Layout is not significant.

5.7 Recursion

The productions in this Recommendation | International Standard are frequently recursive. In this case the productions are to be continuously reapplied until no new sequences are generated.

NOTE – In many cases, such reapplication results in an infinite set of permitted sequences of lexical items. Some or all of the sequences in the set may themselves contain an unbounded number of lexical items. This is not an error.

5.8 References to permitted sequences of lexical items

This Recommendation | International Standard references a permitted sequence of lexical items (part of the ASN.1 notation) by referencing the name that appears before the "::=" in a production; the name is surrounded by the QUOTATION MARK (34) character (") to distinguish it from natural language text, unless it appears as part of a production.

5.9 References to a lexical item

This Recommendation | International Standard references a lexical item by using the name of the lexical item; when the name appears in natural language text, and could be confused with such text, then it is surrounded by the QUOTATION MARK (34) character (").

5.10 Short-hand notations

In order to make productions more concise and more readable, the following short-hand notations are used in the definition of permitted sequences of lexical items in this Recommendation | International Standard and also in ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3 and ITU-T Rec. X.683 | ISO/IEC 8824-4:

a) An asterisk (*) following two names, "A" and "B", denotes the "empty" lexical item (see 11.7), or one of the permitted sequences of lexical items associated with "A", or an alternating series of one of the

sequences of lexical items associated with "A" and one of the sequences of lexical items associated with "B", both starting and finishing with one associated with "A". Thus:

C ::= A B *

is equivalent to:

C ::= D | empty
D ::= A | A B D

"D" being an auxiliary name not appearing elsewhere in the productions.

EXAMPLE – "C ::= A B *" is the shorthand notation for the following alternatives of C:

empty
A
A B A
A B A B A
A B A B A B A
...

b) A plus sign (+) is similar to the asterisk in a), except that the "empty" lexical item is excluded. Thus:

E ::= A B +

is equivalent to:

E ::= A | A B E

EXAMPLE – "E ::= A B +" is the shorthand notation for the following alternatives of E:

A
A B A
A B A B A
A B A B A B A
...

c) A question mark (?) following a name denotes either the "empty" lexical item (see 11.7) or a permitted sequence of lexical items associated with "A". Thus:

F ::= A ?

is equivalent to:

F ::= empty | A

NOTE – These short-hand notations take precedence over the juxtaposition of lexical items in production sequences (see 5.2.2).

5.11 Value references and the typing of values

5.11.1 The ASN.1 value assignment notation enables a name to be given to a value of a specified type. This name can be used wherever a reference to that value is needed. Annex B describes and specifies the value mapping mechanism that allows a value reference name for a value of one type to identify a value of a second (similar) type. Thus, a reference to the first value can be used wherever a reference to a value in the second type is required.

5.11.2 In the body of the ASN.1 standards normal English text is used to specify legality (or otherwise) of constructs where more than one type is involved. These legality specifications generally require that two or more types be "compatible". For example, the type used in defining a value reference is required to be "compatible with" the governing type when the value reference is used. The normative Annex B uses the value mapping concept to give a precise statement about whether any given ASN.1 construct is legal or not.

6 The ASN.1 model of type extension

When decoding an extensible type, a decoder may detect:

a) the absence of expected extension additions in a sequence or set type; or

b) the presence of arbitrary unexpected extension additions above those defined (if any) in a sequence or set type, or of an unknown alternative in a choice type, or an unknown enumeration in an enumerated type, or of an unexpected length or value of a type whose constraint is extensible.

In formal terms, an abstract syntax defined by the extensible type **X** contains not only the values of type **X**, but also the values of all types that are extension-related to **X**. Thus, the decoding process never signals an error when either of the above situations (a or b) is detected. The action that is taken in each situation is determined by the ASN.1 specifier.

NOTE – Frequently the action will be to ignore the presence of unexpected additional extensions, and to use a default value or a "missing" indicator for expected extension additions that are absent.

Unexpected extension additions detected by a decoder in an extensible type can later be included in a subsequent encoding of that type (for transmission back to the sender, or to some third party), provided that the same transfer syntax is used on the subsequent transmission.

7 Extensibility requirements on encoding rules

NOTE – These requirements apply to standardized encoding rules. They do not apply to encoding rules defined using ECN (see ITU-T Rec. X.692 | ISO/IEC 8825-3).

7.1 All ASN.1 encoding rules shall allow the encoding of values of an extensible type **X** in such a way that they can be decoded using an extensible type **Y** that is extension-related to **X**. Further, the encoding rules shall allow the values that were decoded using **Y** to be re-encoded (using **Y**) and decoded using a third extensible type **Z** that is extension related to **Y** (and hence **X** also).

NOTE – Types **X**, **Y** and **Z** may appear in any order in the extension series.

If a value of an extensible type **X** is encoded and then relayed (directly or through a relaying application using extension-related type **Z**) to another application that decodes the value using extensible type **Y** that is extension-related to **X**, then the decoder using type **Y** obtains an abstract value composed of:

a) an abstract value of the extension root type;

b) an abstract value of each extension addition that is present in both **X** and **Y**;

c) delimited encoding for each extension addition (if any) that is in **X** but not in **Y**.

The encodings in c) shall be capable of being included in a later encoding of a value of **Y**, if so required by the application. That encoding shall be a valid encoding of a value of **X**.

Tutorial example: If system A is using an extensible root type (type **X**) that is a sequence type or a set type with an extension addition of an optional integer type, while system B is using an extension-related type (type **Y**) that has two extension additions where each is an optional integer type, then transmission by B of a value of **Y** which omits the integer value of the first extension addition and includes the second must not be confused by A with the presence of the first (only) extension addition of **X** that it knows about. Moreover, A must be able to re-encode the value of **X** with a value present for the first integer type, followed by the second integer value received from B, if so required by the application protocol.

7.2 All ASN.1 encoding rules shall specify the encoding and decoding of the value of an enumerated type and a choice type in such a way that if a transmitted value is in the set of extension additions held in common by the encoder and the decoder, then it is successfully decoded; otherwise, it shall be possible for the decoder to delimit the encoding of it and to identify it as a value of an (unknown) extension addition.

7.3 All ASN.1 encoding rules shall specify the encoding and decoding of types with extensible constraints in such a way that if a transmitted value is in the set of extension additions held in common by the encoder and the decoder, then it is successfully decoded, otherwise it shall be possible for the decoder to delimit the encoding of and to identify it as a value of an (unknown) extension addition.

In all cases, the presence of extension additions shall not affect the ability to recognize later material when a type with an extension marker is nested inside some other type.

NOTE 1 – All variants of the Basic Encoding Rules of ASN.1 and the Packed Encoding Rules of ASN.1 satisfy all these requirements. Encoding rules defined using ECN do not necessarily satisfy all these requirements, but may do so.

NOTE 2 – PER and BER do not identify the version number in the encoding of an extension addition. Encodings specified using ECN may or may not provide such identification.

8 Tags

8.1 A tag is specified by giving a class and a number within the class. The class is one of:

- universal;
- application;
- private;
- context-specific.

8.2 The number is a non-negative integer, specified in decimal notation.

8.3 Restrictions on tags assigned by the user of ASN.1 are specified in clause 30.

NOTE – Clause 30 includes the restriction that users of this notation are not allowed to explicitly specify universal class tags in their ASN.1 specifications. There is no formal difference between use of tags from the other three classes. Where application class tags are employed, a private or context-specific class tag could generally be applied instead, as a matter of user choice and style. The presence of the three classes is largely for historical reasons, but guidance is given in E.2.12 on the way in which the classes are usually employed.

8.4 Table 1 summarizes the assignment of tags in the universal class which are specified in this Recommendation | International Standard.

Table 1 – Universal class tag assignments

UNIVERSAL 0	Reserved for use by the encoding rules
UNIVERSAL 1	Boolean type
UNIVERSAL 2	Integer type
UNIVERSAL 3	Bitstring type
UNIVERSAL 4	Octetstring type
UNIVERSAL 5	Null type
UNIVERSAL 6	Object identifier type
UNIVERSAL 7	Object descriptor type
UNIVERSAL 8	External type and Instance-of type
UNIVERSAL 9	Real type
UNIVERSAL 10	Enumerated type
UNIVERSAL 11	Embedded-pdv type
UNIVERSAL 12	UTF8String type
UNIVERSAL 13	Relative object identifier type
UNIVERSAL 14-15	Reserved for future editions of this Recommendation | International Standard
UNIVERSAL 16	Sequence and Sequence-of types
UNIVERSAL 17	Set and Set-of types
UNIVERSAL 18-22, 25-30	Character string types
UNIVERSAL 23-24	Time types
UNIVERSAL 31-...	Reserved for addenda to this Recommendation | International Standard

8.5 Some encoding rules require a canonical order for tags. To provide uniformity, a canonical order for tags is defined in 8.6.

8.6 The canonical order for tags is based on the outermost tag of each type and is defined as follows:

a) those elements or alternatives with universal class tags shall appear first, followed by those with application class tags, followed by those with context-specific tags, followed by those with private class tags;

b) within each class of tags, the elements or alternatives shall appear in ascending order of their tag numbers.

9 Use of the ASN.1 notation

9.1 The ASN.1 notation for a type definition shall be "Type" (see 16.1).

9.2 The ASN.1 notation for a value of a type shall be "Value" (see 16.7).

NOTE – It is not in general possible to interpret the value notation without knowledge of the type.

9.3 The ASN.1 notation for assigning a type to a type reference name shall be either "TypeAssignment" (see 15.1), "ValueSetTypeAssignment" (see 15.6), "ParameterizedTypeAssignment" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2), or "ParameterizedValueSetTypeAssignment" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2).

9.4 The ASN.1 notation for assigning a value to a value reference name shall be either "ValueAssignment" (see 15.2) or "ParameterizedValueAssignment" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2).

9.5 The production alternatives of the notation "Assignment" shall only be used within the notation "ModuleDefinition" (except as specified in NOTE 2 of 12.1).

10 The ASN.1 character set

10.1 A lexical item shall consist of a sequence of the characters listed in Table 2 except as specified in 10.2 and 10.3. In Table 2, characters are identified by the names they are given in ISO/IEC 10646-1.

Table 2 – ASN.1 characters

A to Z	(LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z)
a to z	(LATIN SMALL LETTER A to LATIN SMALL LETTER Z)
0 to 9	(DIGIT ZERO to DIGIT 9)
!	(EXCLAMATION MARK)
"	(QUOTATION MARK)
&	(AMPERSAND)
'	(APOSTROPHE)
(	(LEFT PARENTHESIS)
)	(RIGHT PARENTHESIS)
*****	(ASTERISK)
,	(COMMA)
-	(HYPHEN-MINUS)
.	(FULL STOP)
/	(SOLIDUS)
:	(COLON)
;	(SEMICOLON)
<	(LESS-THAN SIGN)
=	(EQUALS SIGN)
>	(GREATER-THAN SIGN)
@	(COMMERCIAL AT)
[	(LEFT SQUARE BRACKET)
]	(RIGHT SQUARE BRACKET)
^	(CIRCUMFLEX ACCENT)
_	(LOW LINE)
{	(LEFT CURLY BRACKET)
|	(VERTICAL LINE)
}	(RIGHT CURLY BRACKET)

NOTE – Where equivalent derivative standards are developed by national standards bodies, additional characters may appear in the following lexical items:

- typereference (see 11.2);
- identifier (see 11.3);
- valuereference (see 11.4);
- modulereference (see 11.5).

When additional characters are introduced to accommodate a language in which the distinction between upper-case and lower-case letters is without meaning, the syntactic distinction achieved by dictating the case of the first character of certain of the above lexical items has to be achieved in some other way. This is to allow valid ASN.1 specifications to be written in various languages.

10.2 Where the notation is used to specify the value of a character string type, all characters for the defined character set can appear in the ASN.1 notation, surrounded by the QUOTATION MARK (34) characters (") (see 11.14).

10.3 Additional (arbitrary) graphic symbols may appear in the "comment" lexical item (see 11.6).

10.4 There shall be no significance placed on the typographical style, size, colour, intensity, or other display characteristics.

10.5 The upper and lower-case letters shall be regarded as distinct.

10.6 ASN.1 definitions can also contain white-space characters (see 11.1.6) between lexical items.

11 ASN.1 lexical items

11.1 General rules

11.1.1 The following subclauses specify the characters in lexical items. In each case the name of the lexical item is given, together with the definition of the character sequences which form the lexical item.

11.1.2 The lexical items specified in the subclauses of this clause 11 (except multiple-line "comment", "bstring", "hstring" and "cstring") shall not contain white-space (see 11.6, 11.10, 11.12 and 11.14).

11.1.3 The length of a line is not restricted.

11.1.4 Lexical items may be separated by one or more occurrences of white-space (see 11.1.6) or comments (see 11.6) except when the non-spacing indicator "**&**" (see 5.4) is used. Within an "XMLTypedValue" production (see 15.2), white-space may appear between lexical items, but the "comment" lexical item shall not be present.

NOTE – This is to avoid ambiguity resulting from the presence of adjacent hyphens or asterisk and solidus within an "xmlcstring" lexical item. Such characters never indicate the start of a "comment" lexical item when they appear within an "XMLTypedValue" production.

11.1.5 A lexical item shall be separated from a following lexical item by one or more instances of white-space or comment if the initial character (or characters) of the following lexical item is a permitted character (or characters) for inclusion at the end of the characters in the earlier lexical item.

11.1.6 This Recommendation | International Standard uses the terms "newline", and "white-space". In representing white-space and newline (end of line) in machine-readable specifications, any one or more of the following characters may be used in any combination (for each character, the character name and character code specified in The Unicode Standard are given):

For white-space:

HORIZONTAL TABULATION (9)

LINE FEED (10)

VERTICAL TABULATION (11)

FORM FEED (12)

CARRIAGE RETURN (13)

SPACE (32)

For newline:

LINE FEED (10)

VERTICAL TABULATION (11)

FORM FEED (12)

CARRIAGE RETURN (13)

NOTE – Any character or character sequence that is a valid newline is also a valid white-space.

11.2 Type references

Name of lexical item – typereference

11.2.1 A "typereference" shall consist of an arbitrary number (one or more) of letters, digits, and hyphens. The initial character shall be an upper-case letter. A hyphen shall not be the last character. A hyphen shall not be immediately followed by another hyphen.

NOTE – The rules concerning hyphen are designed to avoid ambiguity with (possibly following) comment.

11.2.2 A "typereference" shall not be one of the reserved character sequences listed in 11.27.

11.3 Identifiers

Name of lexical item – identifier

An "identifier" shall consist of an arbitrary number (one or more) of letters, digits, and hyphens. The initial character shall be a lower-case letter. A hyphen shall not be the last character. A hyphen shall not be immediately followed by another hyphen.

NOTE – The rules concerning hyphen are designed to avoid ambiguity with (possibly following) comment.

11.4 Value references

Name of lexical item – valuereference

A "valuereference" shall consist of the sequence of characters specified for an "identifier" in 11.3. In analysing an instance of use of this notation, a "valuereference" is distinguished from an "identifier" by the context in which it appears.

11.5 Module references

Name of lexical item – modulereference

A "modulereference" shall consist of the sequence of characters specified for a "typereference" in 11.2. In analysing an instance of use of this notation, a "modulereference" is distinguished from a "typereference" by the context in which it appears.

11.6 Comments

Name of lexical item – comment

11.6.1 A "comment" is not referenced in the definition of the ASN.1 notation. It may, however, appear at any time between other lexical items, and has no syntactic significance.

NOTE – Nonetheless, in the context of a Recommendation | International Standard that uses ASN.1, an ASN.1 comment may contain normative text related to the application semantics, or constraints on the syntax.

11.6.2 The lexical item "comment" can have two forms:

a) One-line comments which begin with "`--`" as defined in 11.6.3;

b) Multiple-line comments which begin with "`/*`" as defined in 11.6.4.

11.6.3 Whenever a "comment" begins with a pair of adjacent hyphens, it shall end with the next pair of adjacent hyphens or at the end of the line, whichever occurs first. A comment shall not contain a pair of adjacent hyphens other than the pair which starts it and the pair, if any, which ends it. If a comment beginning with "`--`" includes the adjacent characters "`/*`" or "`*/`", these have no special meaning and are considered part of the comment. The comment may include graphic symbols which are not in the character set specified in 10.1 (see 10.3).

11.6.4 Whenever a "comment" begins with "`/*`", it shall end with a corresponding "`*/`", whether this "`*/`" is on the same line or not. If another "`/*`" is found before a "`*/`", then the comment terminates when a matching "`*/`" has been found for each "`/*`". If a comment beginning with "`/*`" includes two adjacent hyphens "`--`", these hyphens have no special meaning and are considered part of the comment. The comment may include graphic symbols which are not in the character set specified in 10.1 (see 10.3).

NOTE – This allows the user to comment parts of an ASN.1 module that already contain comments (whether they begin with "`--`" or "`/*`") by simply inserting "`/*`" at the beginning of the part to be commented and "`*/`" at its end, provided there are no character string values within the part to be commented out that contain "`/*`" or "`*/`".

11.7 Empty lexical item

Name of lexical item – empty

The "empty" item contains no characters. It is used in the notation of clause 5 when alternative sets of production sequences are specified, to indicate that absence of all alternatives is possible.

11.8 Numbers

Name of lexical item – number

A "number" shall consist of one or more digits. The first digit shall not be zero unless the "number" is a single digit.

NOTE – The "number" lexical item is always mapped to an integer value by interpreting it as decimal notation.

11.9 Real numbers

Name of lexical item – realnumber

A "realnumber" shall consist of an integer part that is a series of one or more digits, and optionally a decimal point (.). The decimal point can optionally be followed by a fractional part which is one or more digits. The integer part, decimal point or fractional part (whichever is last present) can optionally be followed by an e or E and an optionally-signed exponent which is one or more digits. The leading digit of the exponent shall not be zero unless the exponent is a single digit.

11.10 Binary strings

Name of lexical item – bstring

A "bstring" shall consist of an arbitrary number (possibly zero) of the characters:

0 1

possibly intermixed with white-space, preceded by an APOSTROPHE (39) character (**'**) and followed by the pair of characters:

'B

EXAMPLE – **'01101100'B**

Occurrences of white-space within a binary string lexical item have no significance.

11.11 XML binary string item

Name of item – xmlbstring

An "xmlbstring" shall consist of an arbitrary number (possibly zero) of zeros, ones or white-space. Any white-space characters that appear within a binary string item have no significance.

EXAMPLE – 01101100

This sequence of characters is also a valid instance of "xmlhstring" and "xmlcstring". In analysing an instance of use of this notation, an "xmlbstring" is distinguished from an "xmlhstring" or "xmlcstring" by the context in which it appears.

11.12 Hexadecimal strings

Name of lexical item – hstring

11.12.1 An "hstring" shall consist of an arbitrary number (possibly zero) of the characters:

A B C D E F 0 1 2 3 4 5 6 7 8 9

possibly intermixed with white-space, preceded by an APOSTROPHE (39) character (**'**) and followed by the pair of characters:

'H

EXAMPLE – **'AB0196'H**

Occurrences of white-space within a hexadecimal string lexical item have no significance.

11.12.2 Each character is used to denote the value of a semi-octet using a hexadecimal representation.

11.13 XML hexadecimal string item

Name of item – xmlhstring

11.13.1 An "xmlhstring" shall consist of an arbitrary number (possibly zero) of the characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

or white-space. Any white-space characters that appear within a hexadecimal string item have no significance.

EXAMPLE – Ab0196

11.13.2 Each character is used to denote the value of a semi-octet using a hexadecimal representation.

11.13.3 Some instances of "xmlhstring" are also valid instances of "xmlbstring" and "xmlcstring". In analysing an instance of use of this notation, an "xmlhstring" is distinguished from an "xmlbstring" or "xmlcstring" by the context in which it appears.

11.14 Character strings

Name of lexical item – cstring

11.14.1 A "cstring" shall consist of an arbitrary number (possibly zero) of graphic symbols and spacing characters from the character set referenced by the character string type, preceded and followed by a QUOTATION MARK (34) character ("). If the character set includes a QUOTATION MARK (34) character, this character (if present in the character string being represented by the "cstring") shall be represented in the "cstring" by a pair of QUOTATION MARK (34) characters on the same line with no intervening spacing character. The "cstring" may span more than one line of text, in which case the character string being represented shall not include spacing characters in the position prior to or following the end of line in the "cstring". Any spacing characters that appear immediately prior to or following the end of line in the "cstring" have no significance.

NOTE 1 – The "cstring" can only be used to unambiguously represent (on a printed page) character strings for which every character in the string being represented has either been assigned a graphic symbol, or is a spacing character. Where a character string containing control characters needs to be denoted in a printed representation, alternative ASN.1 syntax is available (see clause 35).

NOTE 2 – The character string represented by a "cstring" consists of the characters associated with the graphic symbols and spacing characters. Spacing characters immediately preceding or following any end of line in the "cstring" are not part of the character string being represented (they are ignored). Where spacing characters are included in the "cstring", or where the graphic symbols in the character repertoire are not unambiguous in a printed representation, the character string denoted by "cstring" may be ambiguous in that printed representation.

EXAMPLE 1 – **"屎 屍 市 弒"**

EXAMPLE 2 – The "cstring":

```
"ABCDE    FGH
IJK""XYZ"
```

can be used to represent a character string value of type **IA5String**. The value represented consists of the characters:

```
ABCDE    FGHIJK"XYZ
```

where the precise number of spaces intended between **E** and **F** can be ambiguous in a printed representation if a proportional spacing font (such as is used above) is used in the printed specification, or if the character repertoire contains multiple spacing characters of different widths.

11.14.2 When a character is a combining character (see Annex F) it shall be denoted in a printed representation of the "cstring" as an individual character. It shall not be overprinted with the characters with which it combines. (This ensures that the order of combining characters in the string value is unambiguously defined in the printed version.)

EXAMPLE – Lower case "**e**" and the accent combining character are two characters in ISO/IEC 10646-1, and thus a corresponding "cstring" should be printed as two characters and not as the single character **é**.

11.15 XML character string item

Name of item – xmlcstring

11.15.1 An "xmlcstring" shall consist of an arbitrary number (possibly zero) of the following ISO/IEC 10646-1 characters:

a) HORIZONTAL TABULATION (9);

b) LINE FEED (10);

c) CARRIAGE RETURN (13);

d) any character whose ISO/IEC 10646-1 character code is in the range 32 (20 hex) to 55295 (D7FF hex), inclusive;

e) any character whose ISO/IEC 10646-1 character code is in the range 57344 (E000 hex) to 65533 (FFFD hex), inclusive;

f) any character whose ISO/IEC 10646-1 character code is in the range 65536 (10000 hex) to 1114111 (10FFFF hex), inclusive.

NOTE – Additional restrictions are imposed by the requirement that the "xmlcstring", in an instance of use, shall contain only characters permitted by the governing character string type.

11.15.2 The characters "&" (AMPERSAND), "<" (LESS-THAN SIGN) or ">" (GREATER-THAN SIGN) shall appear only as part of one of the character sequences specified in 11.15.4 or 11.15.5.

11.15.3 An "xmlcstring" is used to represent the value of a restricted character string (see 37.9), and can be used to represent all combinations of ISO/IEC 10646-1 characters, either directly, or by using the escape sequences specified below.

NOTE 1 – An "xmlcstring" cannot be used to represent characters that are not present in ISO/IEC 10646-1, such as some of the control characters which can appear in **`GeneralString`**, nor can it represent characters which might be defined with ISO/IEC 10646-1 character codes above 10FFFF hex.

NOTE 2 – The characters LINE FEED (10) and CARRIAGE RETURN (13) and the pair CARRIAGE RETURN + LINE FEED are not distinguished when processed by conforming XML processors.

11.15.4 If the characters "&" (AMPERSAND), "<" (LESS-THAN SIGN) or ">" (GREATER-THAN SIGN) are present in an abstract character string value being represented by "xmlcstring" (see 37.9), they shall be represented in the "xmlcstring" by either

a) the escape sequences specified in 11.15.8; or

b) the escape sequences "&", "<" or ">" respectively. These escape sequences shall not contain white-space (see 11.1.6).

11.15.5 If a character with an ISO/IEC 10646-1 character code in column 1 of Table 3 is present in the abstract character string value being represented by the "xmlcstring" (see 37.9), it shall be represented by the character sequence in column 2 of Table 3. These character sequences shall not contain white-space (see 11.1.6).

NOTE – This does not include characters with decimal character codes 9, 10, and 13, and all the letters in these character sequences are lowercase.

Table 3 – Escape sequences for control characters in an "xmlcstring"

ISO/IEC 10646-1 character code	"xmlcstring" representation	ISO/IEC 10646-1 character code	"xmlcstring" representation
0 (0 hex)	<nul/>	17 (11 hex)	<dc1/>
1 (1 hex)	<soh/>	18 (12 hex)	<dc2/>
2 (2 hex)	<stx/>	19 (13 hex)	<dc3/>
3 (3 hex)	<etx/>	20 (14 hex)	<dc4/>
4 (4 hex)	<eot/>	21 (15 hex)	<nak/>
5 (5 hex)	<enq/>	22 (16 hex)	<syn/>
6 (6 hex)	<ack/>	23 (17 hex)	<etb/>
7 (7 hex)	<bel/>	24 (18 hex)	<can/>
8 (8 hex)	<bs/>	25 (19 hex)	
11 (B hex)	<vt/>	26 (1A hex)	<sub/>
12 (C hex)	<ff/>	27 (1B hex)	<esc/>
14 (E hex)	<so/>	28 (1C hex)	<is4/>
15 (F hex)	<si/>	29 (1D hex)	<is3/>
16 (10 hex)	<dle/>	30 (1E hex)	<is2/>
		31 (1F hex)	<is1/>

11.15.6 When "xmlcstring" is used within an "XMLTypedValue" (see 15.2) forming part of an XER encoding (see ITU-T Rec. X.693 | ISO/IEC 8825-4), it may contain adjacent HYPHEN-MINUS (45) characters. When used within an instance of XML value notation in an ASN.1 module, it shall not contain two adjacent HYPHEN-MINUS characters. If this character sequence is present in an abstract character string value being represented by the "xmlcstring" in an ASN.1 module, then at least one of the adjacent HYPHEN-MINUS characters shall be represented by the escape sequences specified in 11.15.8.

11.15.7 When "xmlcstring" is used within an "XMLTypedValue" forming part of an XER encoding (see ITU-T Rec. X.693 | ISO/IEC 8825-4), it may contain adjacent ASTERISK (42) and SOLIDUS (47) characters in any order. When used within an instance of XML value notation in an ASN.1 module, it shall not contain adjacent ASTERISK and SOLIDUS characters (in any order). If this character sequence is present in an abstract character string value being represented by the "xmlcstring", then at least one of the adjacent ASTERISK and SOLIDUS characters shall be represented by the escape sequences specified in 11.15.8.

11.15.8 Any character that can appear directly in an "xmlcstring" can also be represented in the "xmlcstring" by an escape sequence of the form "&#n;" (where n is the ISO/IEC 10646-1 character code in decimal notation) or of the form "&#xn;" (where n is the ISO/IEC 10646-1 character code in hexadecimal notation). These escape sequences shall not contain white-space (see 11.1.6).

NOTE 1 – Leading zeros are permitted in the decimal and hexadecimal values of "n" and both lowercase and uppercase letters "A"-"F" can be used in the hexadecimal value.

NOTE 2 – If the escape sequences "&#n" and "&#xn" are used for ISO/IEC 10646-1 characters which are not in the Basic Multilingual Plane (BMP), the value of "n" will be greater than 65535 (FFFF hex).

EXAMPLE – The "xmlcstring":

ABCDé FGHîJK&XYZ

can be used to represent a character string value of type UTF8String. The value represented consists of the characters:

ABCDé FGHîJK&XYZ

where the precise space characters between é and F can be ambiguous in print media if a proportional spacing font (such as above) is used in the specification.

11.16 Assignment lexical item

Name of lexical item – "**::=**"

This lexical item shall consist of the sequence of characters:

::=

NOTE – This sequence does not contain white-space (see 11.1.2).

11.17 Range separator

Name of lexical item – "**..**"

This lexical item shall consist of the sequence of characters:

..

NOTE – This sequence does not contain white-space (see 11.1.2).

11.18 Ellipsis

Name of lexical item – "**...**"

This lexical item shall consist of the sequence of characters:

...

NOTE – This sequence does not contain white-space (see 11.1.2).

11.19 Left version brackets

Name of lexical item – "**[[**"

This lexical item shall consist of the sequence of characters:

[[

NOTE – This sequence does not contain white-space (see 11.1.2).

11.20 Right version brackets

Name of lexical item – "**]]**"

This lexical item shall consist of the sequence of characters:

]]

NOTE – This sequence does not contain white-space (see 11.1.2).

11.21 XML end tag start item

Name of item – "</"

This item shall consist of the sequence of characters:

</

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.22 XML single tag end item

Name of item – "/>"

This item shall consist of the sequence of characters:

/>

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.23 XML boolean true item

Name of item – "true"

11.23.1 This item shall consist of the sequence of characters:

true

11.23.2 In analysing an instance of use of this notation, a "true" is distinguished from a "valuereference" or an "identifier" by the context in which it appears.

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.24 XML boolean false item

Name of item – "false"

11.24.1 This item shall consist of the sequence of characters:

false

11.24.2 In analyzing an instance of use of this notation, a "false" is distinguished from a "valuereference" or an "identifier" by the context in which it appears.

NOTE – This sequence does not contain any white-space characters (see 11.1.2).

11.25 XML tag names for ASN.1 types

Name of item – xmlasn1typename

11.25.1 This Recommendation | International Standard uses the item "xmlasn1typename" when ASN.1 built-in types are to be used as XML tag names.

11.25.2 Table 4 lists the character sequences that are to form the "xmlasn1typename" for each of the ASN.1 built-in types listed in 16.2. The ASN.1 built-in type is identified in column 1 of Table 4 by its production name. The character sequence which shall be used for "xmlasn1typename" is identified in column 2 of Table 4, with no white-space before or after these character sequences.

11.25.3 The "xmlasn1typename" for the "UsefulType"s (see 41.1) shall be the "typereference" used in their definition.

11.25.4 The character sequence in the "xmlasn1typename" item for the "ObjectClassFieldType" and for the "InstanceOfType" are specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.1 and Annex C.

11.25.5 If the ASN.1 built-in type is a "TaggedType" then the type which determines the "xmlasn1typename" shall be "Type" in the "TaggedType" (see 30.1). If this is itself a "TaggedType", then this subclause 11.25.5 shall be recursively applied.

Table 4 – Characters in xmlasn1typename

ASN.1 type production name	Characters in xmlasn1typename
BitStringType	BIT_STRING
BooleanType	BOOLEAN
ChoiceType	CHOICE
EmbeddedPDVType	SEQUENCE
EnumeratedType	ENUMERATED
ExternalType	SEQUENCE
InstanceOfType	SEQUENCE
IntegerType	INTEGER
NullType	NULL
ObjectClassFieldType	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.10 and 14.11*
ObjectIdentifierType	OBJECT_IDENTIFIER
OctetStringType	OCTET_STRING
RealType	REAL
RelativeOIDType	RELATIVE_OID
RestrictedCharacterStringType	*The type name (e.g.* `IA5String`*)*
SequenceType	SEQUENCE
SequenceOfType	SEQUENCE_OF
SetType	SET
SetOfType	SET_OF
TaggedType	*See 11.25.5*
UnrestrictedCharacterStringType	SEQUENCE

11.26 Single character lexical items

Names of lexical items –

"{"
"}"
"<"
">"
","
"."
"("
")"
"["
"]"
"-" (HYPEN-MINUS)
":"
"="
""" (QUOTATION MARK)
"'" (APOSTROPHE)
" " (SPACE)
"."
"@"
"|"
"!"
"^"

A lexical item with any of the names listed above shall consist of the single character without the quotation marks.

11.27 Reserved words

Names of reserved words –

ABSENT	**ENCODED**	**INTEGER**	**RELATIVE-OID**
ABSTRACT-SYNTAX	**END**	**INTERSECTION**	**SEQUENCE**
ALL	**ENUMERATED**	**ISO646String**	**SET**
APPLICATION	**EXCEPT**	**MAX**	**SIZE**
AUTOMATIC	**EXPLICIT**	**MIN**	**STRING**
BEGIN	**EXPORTS**	**MINUS-INFINITY**	**SYNTAX**
BIT	**EXTENSIBILITY**	**NULL**	**T61String**
BMPString	**EXTERNAL**	**NumericString**	**TAGS**
BOOLEAN	**FALSE**	**OBJECT**	**TeletexString**
BY	**FROM**	**ObjectDescriptor**	**TRUE**
CHARACTER	**GeneralizedTime**	**OCTET**	**TYPE-IDENTIFIER**
CHOICE	**GeneralString**	**OF**	**UNION**
CLASS	**GraphicString**	**OPTIONAL**	**UNIQUE**
COMPONENT	**IA5String**	**PATTERN**	**UNIVERSAL**
COMPONENTS	**IDENTIFIER**	**PDV**	**UniversalString**
CONSTRAINED	**IMPLICIT**	**PLUS-INFINITY**	**UTCTime**
CONTAINING	**IMPLIED**	**PRESENT**	**UTF8String**
DEFAULT	**IMPORTS**	**PrintableString**	**VideotexString**
DEFINITIONS	**INCLUDES**	**PRIVATE**	**VisibleString**
EMBEDDED	**INSTANCE**	**REAL**	**WITH**

Lexical items with the above names shall consist of the sequence of characters in the name, and are reserved character sequences.

NOTE 1 – White-space does not occur in these sequences.

NOTE 2 – The keywords **CLASS**, **CONSTRAINED**, **CONTAINING**, **ENCODED**, **INSTANCE**, **SYNTAX** and **UNIQUE** are not used in this Recommendation | International Standard; they are used in ITU-T Rec. X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3 and ITU-T Rec. X.683 | ISO/IEC 8824-4.

12 Module definition

12.1 A "ModuleDefinition" is specified by the following productions:

```
ModuleDefinition ::=
        ModuleIdentifier
        DEFINITIONS
        TagDefault
        ExtensionDefault
        "::="
        BEGIN
        ModuleBody
        END

ModuleIdentifier ::=
        modulereference
        DefinitiveIdentifier

DefinitiveIdentifier ::=
        "{" DefinitiveObjIdComponentList "}"
    |   empty

DefinitiveObjIdComponentList  ::=
        DefinitiveObjIdComponent
    |   DefinitiveObjIdComponent DefinitiveObjIdComponentList
```

```
DefinitiveObjIdComponent ::=
        NameForm
    |   DefinitiveNumberForm
    |   DefinitiveNameAndNumberForm

DefinitiveNumberForm    ::= number

DefinitiveNameAndNumberForm ::= identifier "(" DefinitiveNumberForm ")"

TagDefault ::=
        EXPLICIT TAGS
    |   IMPLICIT TAGS
    |   AUTOMATIC TAGS
    |   empty

ExtensionDefault ::=
        EXTENSIBILITY IMPLIED
    |   empty

ModuleBody ::=
        Exports Imports AssignmentList
    |   empty

Exports ::=
        EXPORTS SymbolsExported ";"
    |   EXPORTS ALL ";"
    |   empty

SymbolsExported ::=
        SymbolList
    |   empty

Imports ::=
        IMPORTS SymbolsImported ";"
    |   empty

SymbolsImported ::=
        SymbolsFromModuleList
    |   empty

SymbolsFromModuleList ::=
        SymbolsFromModule
    |   SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=
        SymbolList FROM GlobalModuleReference

GlobalModuleReference ::=
        modulereference AssignedIdentifier

AssignedIdentifier ::=
        ObjectIdentifierValue
    |   DefinedValue
    |   empty

SymbolList ::=
        Symbol
    |   SymbolList "," Symbol

Symbol ::=
        Reference
    |   ParameterizedReference

Reference ::=
        typereference
    |   valuereference
    |   objectclassreference
    |   objectreference
    |   objectsetreference
```

AssignmentList ::=
    Assignment
  | **AssignmentList Assignment**

Assignment ::=
    TypeAssignment
  | **ValueAssignment**
  | **XMLValueAssignment**
  | **ValueSetTypeAssignment**
  | **ObjectClassAssignment**
  | **ObjectAssignment**
  | **ObjectSetAssignment**
  | **ParameterizedAssignment**

NOTE 1 – The use of a "ParameterizedReference" in the "Exports" and "Imports" lists is specified in ITU-T Rec. X.683 | ISO/IEC 8824-4.

NOTE 2 – For examples (and for the definition in this Recommendation | International Standard of types with universal class tags), the "ModuleBody" can be used outside of a "ModuleDefinition".

NOTE 3 – "TypeAssignment", "ValueAssignment", "XMLValueAssignment" and "ValueSetTypeAssignment" productions are specified in clause 15.

NOTE 4 – The value of "TagDefault" for the module definition affects only those types defined explicitly in the module. It does not affect the interpretation of imported types.

NOTE 5 – The character semicolon does not appear in the assignment list specification or any of its subordinate productions, and is reserved for use by ASN.1 tool developers.

12.2 The "TagDefault" is taken as **`EXPLICIT TAGS`** if it is "empty".

NOTE – Clause 30 gives the meaning of **`EXPLICIT TAGS`**, **`IMPLICIT TAGS`**, and **`AUTOMATIC TAGS`**.

12.3 When the **`AUTOMATIC TAGS`** alternative of "TagDefault" is selected, automatic tagging is said to be selected for the module, otherwise it is said to be not selected. Automatic tagging is a syntactical transformation which is applied (with additional conditions) to the "ComponentTypeLists" and "AlternativeTypeLists" productions occurring within the definition of the module. This transformation is formally specified by 24.7 to 24.9, 26.3 and 28.2 to 28.5 regarding the notations for sequence types, set types and choice types, respectively.

12.4 The **`EXTENSIBILITY IMPLIED`** option is equivalent to the textual insertion of an extension marker ("**`...`**") in the definition of each type in the module for which it is permitted. The location of the implied extension marker is the last position in the type where an explicitly specified extension marker is allowed. The absence of **`EXTENSIBILITY IMPLIED`** means that extensibility is only provided for those types within the module where an extension marker is explicitly present.

NOTE – **`EXTENSIBILITY IMPLIED`** affects only types. It has no effect on object sets and subtype constraints.

12.5 The "modulereference" appearing in the "ModuleIdentifier" production is called the module name.

NOTE – The possibility of defining a single ASN.1 module by the use of several occurrences of "ModuleBody" assigned the same "modulereference" was (arguably) permitted in earlier specifications. It is not permitted by this Recommendation | International Standard.

12.6 Module names shall be used only once (except as specified in 12.9) within the sphere of interest of the definition of the module.

12.7 If the "DefinitiveIdentifier" is not empty, the denoted object identifier value unambiguously and uniquely identifies the module being defined. No defined value may be used in defining the object identifier value.

NOTE – The question of what changes to a module require a new "DefinitiveIdentifier" is not addressed in this Recommendation | International Standard.

12.8 If the "AssignedIdentifier" is not empty, the "ObjectIdentifierValue" and the "DefinedValue" alternatives unambiguously and uniquely identify the module from which reference names are being imported. When the "DefinedValue" alternative of "AssignedIdentifier" is used, it shall be a value of type object identifier. Each "valuereference" which textually appears within an "AssignedIdentifier" shall satisfy one of the following rules:

a) It is defined in the "AssignmentList" of the module being defined, and all "valuereference"s which textually appear on the right side of the assignment statement also satisfy this rule (rule "a") or the next rule (rule "b").

b) It appears as a "Symbol" in a "SymbolsFromModule" whose "AssignedIdentifier" does not textually contain any "valuereference"s.

NOTE – It is recommended that an object identifier be assigned so that others can unambiguously refer to the module.

12.9 The "GlobalModuleReference" in a "SymbolsFromModule" shall appear in the "ModuleDefinition" of another module, except that if it includes a non-empty "DefinitiveIdentifier", the "modulereference" may differ in the two cases.

NOTE – A different "modulereference" from that used in the other module should only be used when symbols are to be imported from two modules with the same name (the modules being named in disregard of 12.6). The use of alternative distinct names makes these names available for use in the body of the module (see 12.15).

12.10 When both a "modulereference" and a non-empty "AssignedIdentifier" are used in referencing a module, the latter shall be considered definitive.

12.11 When the referenced module has a non-empty "DefinitiveIdentifier", the "GlobalModuleReference" referencing that module shall not have an empty "AssignedIdentifier".

12.12 When the "SymbolsExported" alternative of "Exports" is selected:

a) each "Symbol" in "SymbolsExported" shall satisfy one and only one of the following conditions:

   i) is only defined in the module being constructed; or

   ii) appears exactly once in the "SymbolsImported" alternative of "Imports";

b) every "Symbol" to which reference from outside the module is appropriate shall be included in the "SymbolsExported" and only these "Symbol"s may be referenced from outside the module (subject to the relaxation specified in 12.13); and

c) if there are no such "Symbol"s, then the empty alternative of "SymbolsExported" (not of "Exports") shall be selected.

12.13 When either the "empty" alternative or the **`EXPORTS ALL`** alternative of "Exports" is selected, every "Symbol" defined in the module or imported by the module may be referenced from other modules subject to the restriction specified in 12.12 a).

NOTE – The "empty" alternative of "Exports" is included for backwards compatibility.

12.14 Identifiers that appear in a "NamedNumberList", "Enumeration" or "NamedBitList" are implicitly exported if the typereference that defines them is exported or appears as a component (or subcomponent) within an exported type.

12.15 When the "SymbolsImported" alternative of "Imports" is selected:

a) Each "Symbol" in "SymbolsFromModule" shall either be defined in the module body, or be present in the "Imports" clause, of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule". Importing a "Symbol" present in the "Imports" clause of the referenced module is only allowed if there is only one occurrence of the "Symbol" in that clause, and the "Symbol" is not defined in the referenced module.

   NOTE 1 – This does not prohibit the same symbol name defined in two different modules from being imported into another module. However, if the same "Symbol" name appears more than once in the "Imports" clause of module A, that "Symbol" name cannot be exported from A for import to another module B.

b) If the "SymbolsExported" alternative of "Exports" is selected in the definition of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule" the "Symbol" shall appear in its "SymbolsExported".

c) Only those "Symbol"s that appear amongst the "SymbolList" of a "SymbolsFromModule" may appear as the symbol in any "External<X>Reference" which has the "modulereference" denoted by the "GlobalModuleReference" of that "SymbolsFromModule" (where <X> is "Value", "Type", "Object", "Objectclass", or "Objectset").

d) If there are no such "Symbol"s, then the "empty" alternative of "SymbolsImported" shall be selected.

   NOTE 2 – An effect of c) and d) is that the statement **`IMPORTS;`** implies that the module cannot contain an "External<X>Reference".

e) All the "SymbolsFromModule" in the "SymbolsFromModuleList" shall include occurrences of "GlobalModuleReference" such that:

   i) the "modulereference" in them are all different from each other and from the "modulereference" associated with the referencing module; and

   ii) the "AssignedIdentifier", when non-empty, denotes object identifier values which are all different from each other and from the object identifier value (if any) associated with the referencing module.

12.16 When the "empty" alternative of "Imports" is selected, the module may still reference "Symbols" defined in other modules by means of an "External<X>Reference".

NOTE – The "empty" alternative of "Imports" is included for backwards compatibility.

12.17 Identifiers that appear in a "NamedNumberList", "Enumeration" or "NamedBitList" are implicitly imported if the typereference that defines them is imported or appears as a component (or subcomponent) within an imported type.

12.18 A "Symbol" in a "SymbolsFromModule" may appear in "ModuleBody" as a "Reference". The meaning associated with the "Symbol" is that which it has in the module denoted by the corresponding "GlobalModuleReference".

12.19 Where the "Symbol" also appears in an "AssignmentList" (deprecated), or appears in one or more other instances of "SymbolsFromModule", it shall only be used in an "External<X>Reference". Where it does not so appear, it shall be used directly as a "Reference".

12.20 The various alternatives for "Assignment" are defined in the following clauses in this Recommendation | International Standard, except as noted otherwise:

Assignment alternative	*Defining subclause*
"TypeAssignment"	15.1
"ValueAssignment"	15.2
"XMLValueAssignment"	15.2
"ValueSetTypeAssignment"	15.6
"ObjectClassAssignment"	ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.1
"ObjectAssignment"	ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.1
"ObjectSetAssignment"	ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.1
"ParameterizedAssignment"	ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.1

The first symbol of every "Assignment" is one of the alternatives of "Reference", denoting the reference name being defined. In no two assignments within an "AssignmentList" shall the reference names be the same.

13 Referencing type and value definitions

13.1 The defined type and value productions**:**

```
DefinedType ::=
        ExternalTypeReference
    |   Typereference
    |   ParameterizedType
    |   ParameterizedValueSetType

DefinedValue ::=
        ExternalValueReference
    |   Valuereference
    |   ParameterizedValue
```

specify the sequences which shall be used to reference type and value definitions. The type identified by a "ParameterizedType" and "ParameterizedValueSetType", and the value identified by a "ParameterizedValue" are specified in ITU-T Rec. X.683 | ISO/IEC 8824-4.

13.2 The "NonParameterizedTypeName" production:

```
NonParameterizedTypeName ::=
        ExternalTypeReference
    |   typereference
    |   xmlasn1typename
```

is used when an XML tag name is needed to represent an ASN.1 type.

13.3 The third alternative shall not be used as the "NonParameterizedTypeName" in the "XMLTypedValue" of "XMLValueAssignment" (see 15.2) or of "XMLOpenTypeFieldVal" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.6) when the XML value notation is used in an ASN.1 module if the "xmlasn1typename" is "CHOICE", "ENUMERATED", "SEQUENCE", "SEQUENCE_OF", "SET" or "SET_OF".

NOTE – This restriction is imposed in XML value notation used in an ASN.1 module because these "xmlasn1typename"s do not define an ASN.1 type. The restriction is not present for use of this notation in encoding rules (such as XER, see ITU-T Rec. X.693 | ISO/IEC 8825-4) because XML tags formed from "xmlasn1typename"s are not used to determine the types that are being encoded.

13.4 Except as specified in 12.18, the "typereference", "valuereference", "ParameterizedType", "ParameterizedValueSetType" or "ParameterizedValue" alternatives shall not be used unless the reference is within the "ModuleBody" in which a type or value is assigned (see 15.1 and 15.2) to the "typereference" or "valuereference".

13.5 The "ExternalTypeReference" and "ExternalValueReference" shall not be used unless the corresponding "typereference" or "valuereference":

a) has been assigned a type or value respectively (see 15.1 and 15.2); or

b) are present in the "Imports" clause,

within the "ModuleBody" used to define the corresponding "modulereference". Referencing a name in the "Imports" clause of another module shall only be allowed if there is no more than one occurrence of the "Symbol" in that clause.

NOTE – This does not prohibit the same "Symbol" defined in two different modules from being imported into another module. However, if the same "Symbol" appears more than once in the **IMPORTS** clause of a module **A**, then that "Symbol" cannot be referenced using module **A** in an external reference.

13.6 An external reference shall be used in a module only to refer to a reference name which is defined in a different module, and is specified by the following productions:

```
ExternalTypeReference ::=
        modulereference
        "."
        typereference

ExternalValueReference ::=
        modulereference
        "."
        valuereference
```

NOTE – Additional external reference productions ("ExternalClassReference", "ExternalObjectReference" and "ExternalObjectSetReference") are specified in ITU-T Rec. X.681 | ISO/IEC 8824-2.

13.7 When the referencing module is defined using the "SymbolsImported" alternative of "Imports", the "modulereference" in the external reference shall appear in the "GlobalModuleReference" of exactly one of the "SymbolsFromModule" in the "SymbolsImported". When the referencing module is defined using the "empty" alternative of "Imports", the "modulereference" in the external reference shall appear in the "ModuleDefinition" of the module (different from the referencing module) where the "Reference" is defined.

13.8 Where a "DefinedType" is used as part of notation governed by a "Type" (for example, in a "SubtypeConstraint") then the "DefinedType" shall be compatible with the governing "Type" as specified in clause B.6.2.

13.9 Every occurrence within an ASN.1 specification of a "DefinedValue" is governed by a "Type", and that "DefinedValue" shall reference a value of a type that is compatible with the governing "Type" as specified in clause B.6.2.

14 Notation to support references to ASN.1 components

14.1 There is a requirement for formal reference to components of ASN.1 types, values, etc. for many purposes. One such instance is the need to write text to identify a specific type within some ASN.1 module. This clause defines a notation which can be used to provide such references.

14.2 The notation enables any component of a set or sequence type (which is either mandatorily or optionally present in the type) to be identified.

14.3 Any part of any ASN.1 type definition can be referenced by use of the "AbsoluteReference" syntactic construct:

```
AbsoluteReference ::= "@" ModuleIdentifier
        "."
        ItemSpec

ItemSpec ::=
        typereference
    |   ItemId "." ComponentId

ItemId ::= ItemSpec
```

```
ComponentId ::=
        identifier
    |   number
    |   "*"
```

NOTE – The AbsoluteReference production is not used elsewhere in this Recommendation | International Standard. It is provided for the purposes stated in 14.1.

14.4 The "ModuleIdentifier" identifies an ASN.1 module (see 12.1).

14.5 When the first altenative of "DefinitiveIdentifier" is used as part of the "ModuleIdentifier", the "DefinitiveIdentifier" unambiguously and uniquely identifies the module from which a name is being referenced.

14.6 The "typereference" references any ASN.1 type defined in the module identified by "ModuleIdentifier".

14.7 The "ComponentId" in each "ItemSpec" identifies a component of the type which has been identified by the "ItemId". It shall be the last "ComponentId" if the component it identifies is not a set, sequence, set-of, sequence-of, or choice type.

14.8 The "identifier" form of "ComponentId" can be used if the parent "ItemId" is a set or sequence type, and is required to be one of the "identifier"s of the "NamedType" in the "ComponentTypeLists" of that set or sequence. It can also be used if the "ItemId" identifies a choice type, and is then required to be one of the "identifier"s of a "NamedType" in the "AlternativeTypeLists" of that choice type. It cannot be used in any other circumstance.

14.9 The number form of "ComponentId" can be used only if the "ItemId" is a sequence-of or set-of type. The value of the number identifies the instance of the type in the sequence-of or set-of, with the value "1" identifying the first instance of the type. The value zero identifies a conceptual integer type component (not explicitly present in transfer) that contains a count of the number of instances of the type in the sequence-of or set-of that are present in the value of the enclosing type.

14.10 The "*****" form of "ComponentId" can be used only if the "ItemId" is a sequence-of or set-of. Any semantics associated with the use of the "*****" form of "ComponentId" apply to all components of the sequence-of and set-of.

NOTE – In the following example:

```
M DEFINITIONS ::= BEGIN
      T ::= SEQUENCE {
            a     BOOLEAN,
            b     SET OF INTEGER
      }
END
```

the components of "**T**" could be referenced by text outside an ASN.1 module (or in a comment), such as:

```
-- if (@M.T.b.0 is odd) then:
--        (@M.T.b.* shall be an odd integer)
```

which is used to state that if the number of components in **b** is odd, all components of **b** must be odd.

15 Assigning types and values

15.1 A "typereference" shall be assigned a type by the notation specified by the "TypeAssignment" production:

```
TypeAssignment ::=
        typereference
        "::="
        Type
```

The "typereference" shall not be an ASN.1 reserved word (see 11.27).

15.2 A "valuereference" shall be assigned a value by the notation specified by either the "ValueAssignment" or "XMLValueAssignment" productions:

```
ValueAssignment ::=
        valuereference
        Type
        "::="
        Value
```

```
XMLValueAssignment ::=
        valuereference
        ":::="
        XMLTypedValue

XMLTypedValue ::=
        "<" & NonParameterizedTypeName ">"
        XMLValue
        "</" & NonParameterizedTypeName ">"
    |   "<" & NonParameterizedTypeName "/>"
```

The value being assigned to the "valuereference" in the "ValueAssignment" is "Value", and is governed by "Type" and shall be a notation for a value of the type defined by "Type" (as specified in 15.3). The value being assigned to the "valuereference" in the "XMLValueAssignment" is "XMLValue" (see 16.7), and shall be a notation for a value of the type defined by "NonParameterizedTypeName" (as specified in 15.4). If this is the "xmlasn1typename" item, then it identifies the ASN.1 built-in type in the corresponding row of Table 4 (see also 13.3).

15.3 "Value" is a notation for a value of a type as specified in 16.7.

15.4 "XMLValue" is a notation for a value of a type if "XMLValue" is an "XMLBuiltinValue" notation for the type (see 16.10).

15.5 The second alternative of "XMLTypedValue" (use of an XML empty-element tag) can be used only if an instance of the "XMLValue" production is empty.

NOTE – If the "XMLValue" production was an "xmlcstring" containing only white-space, this would not be empty, and the second alternative could not be used.

15.6 A "typereference" can be assigned a value set by the notation specified by the "ValueSetTypeAssignment" production:

```
ValueSetTypeAssignment ::=
        typereference
        Type
        ":::="
        ValueSet
```

This notation assigns to "typereference" the type defined as a subtype of the type denoted by "Type" and which contains exactly the values which are specified in or allowed by "ValueSet". The "typereference" shall not be an ASN.1 reserved word (see 11.27), and may be referenced as a type. "ValueSet" is defined in 15.7.

15.7 A value set governed by some type shall be specified by the notation "ValueSet":

```
ValueSet ::= "{" ElementSetSpecs "}"
```

The value set comprises all of the values, of which there shall be at least one, specified by "ElementSetSpecs" (see clause 46).

15.8 The "ValueSetTypeAssignment" production expands into:

```
typereference
    Type
    ":::="
    "{" ElementSetSpecs "}"
```

For all purposes, including the application of encoding rules, this is defined to be exactly equivalent to the use of the production:

```
typereference
    ":::="
    Type
    "(" ElementSetSpecs ")"
```

with the same "Type" and "ElementSetSpecs" specifications.

16 Definition of types and values

16.1 A type shall be specified by the notation "Type":

Type ::= BuiltinType | ReferencedType | ConstrainedType

16.2 The built-in types of ASN.1 are specified by the notation "BuiltinType", defined as follows:

BuiltinType ::=
BitStringType
| **BooleanType**
| **CharacterStringType**
| **ChoiceType**
| **EmbeddedPDVType**
| **EnumeratedType**
| **ExternalType**
| **InstanceOfType**
| **IntegerType**
| **NullType**
| **ObjectClassFieldType**
| **ObjectIdentifierType**
| **OctetStringType**
| **RealType**
| **RelativeOIDType**
| **SequenceType**
| **SequenceOfType**
| **SetType**
| **SetOfType**
| **TaggedType**

The various "BuiltinType" notations are defined in the following clauses (in this Recommendation | International Standard unless otherwise stated):

BitStringType	21
BooleanType	17
CharacterStringType	36
ChoiceType	28
EmbeddedPDVType	33
EnumeratedType	19
ExternalType	34
InstanceOfType	ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex C
IntegerType	18
NullType	23
ObjectClassFieldType	ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.1
ObjectIdentifierType	31
OctetStringType	22
RealType	20
RelativeOIDType	32
SequenceType	24
SequenceOfType	25
SetType	26
SetOfType	27
TaggedType	30

16.3 The referenced types of ASN.1 are specified by the notation "ReferencedType":

ReferencedType ::=
DefinedType
| **UsefulType**
| **SelectionType**
| **TypeFromObject**
| **ValueSetFromObjects**

The "ReferencedType" notation provides an alternative means of referring to some other type (and ultimately to a built-in type). The various "ReferencedType" notations, and the way in which the type to which they refer is determined, are specified in the following places in this Recommendation | International Standard unless otherwise stated:

DefinedType	13.1
UsefulType	41.1
SelectionType	29
TypeFromObject	ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 15
ValueSetFromObjects	ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 15

16.4 The "ConstrainedType" is defined in clause 45.

16.5 This Recommendation | International Standard requires the use of the notation "NamedType" in specifying the components of the set types, sequence types and choice types. The notation for "NamedType" is:

NamedType ::= identifier Type

16.6 The "identifier" is used to unambiguously refer to components of a set type, sequence type or choice type in the value notation, in inner subtype constraints and in component relation constraints (see ITU-T Rec. X.682 | ISO/IEC 8824-3). It is not part of the type, and has no effect on the type.

16.7 A value of some type shall be specified by the notation "Value" or by the notation "XMLValue":

Value ::=
BuiltinValue
| **ReferencedValue**
| **ObjectClassFieldValue**

XMLValue ::=
XMLBuiltinValue
| **XMLObjectClassFieldValue**

NOTE 1 – "ObjectClassFieldValue" and "XMLObjectClassFieldValue" are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.6.

NOTE 2 – "XMLValue" is only used in "XMLTypedValue".

16.8 If any part of the "XMLValue" production results in an XML start-tag immediately followed by an XML end-tag, possibly separated by white-space inserted as permitted by 11.1.4 (for example, <field1></field1>), these two XML tags, and any intervening white-space, can be replaced by a single XML empty-element tag (<field1/>).

NOTE – If any white-space character, except white-space inserted as permitted by 11.1.4, is present between the final ">" character of the start tag and the initial "<" character of the end-tag, the condition above is not satisfied.

16.9 Values of the built-in types of ASN.1 can be specified by the notation "XMLBuiltinValue" (see 16.10) or "BuiltinValue", defined as follows:

BuiltinValue ::=
BitStringValue
| **BooleanValue**
| **CharacterStringValue**
| **ChoiceValue**
| **EmbeddedPDVValue**
| **EnumeratedValue**
| **ExternalValue**
| **InstanceOfValue**
| **IntegerValue**
| **NullValue**
| **ObjectIdentifierValue**
| **OctetStringValue**
| **RealValue**
| **RelativeOIDValue**
| **SequenceValue**
| **SequenceOfValue**
| **SetValue**
| **SetOfValue**
| **TaggedValue**

Each of the various "BuiltinValue" notations is defined in the same subclause as the corresponding "BuiltinType" notation, as listed in 16.2.

16.10 "XMLBuiltinValue" is defined as follows:

XMLBuiltinValue ::=
XMLBitStringValue
| **XMLBooleanValue**
| **XMLCharacterStringValue**
| **XMLChoiceValue**
| **XMLEmbeddedPDVValue**
| **XMLEnumeratedValue**
| **XMLExternalValue**
| **XMLInstanceOfValue**
| **XMLIntegerValue**
| **XMLNullValue**
| **XMLObjectIdentifierValue**
| **XMLOctetStringValue**
| **XMLRealValue**
| **XMLRelativeOIDValue**
| **XMLSequenceValue**
| **XMLSequenceOfValue**
| **XMLSetValue**
| **XMLSetOfValue**
| **XMLTaggedValue**

Each of the various "XMLBuiltinValue" notations is defined in the same clause as the corresponding "BuiltinType" notation, as listed in 16.2 above.

16.11 The referenced values of ASN.1 are specified by the notation "ReferencedValue":

ReferencedValue ::=
DefinedValue
| **ValueFromObject**

The "ReferencedValue" notation provides an alternative means of referring to some other value (and ultimately to a built-in value). The various "ReferencedValue" notations, and the way in which the value to which they refer is determined, are specified in the following places (in this Recommendation | International Standard unless otherwise stated):

DefinedValue	13.1
ValueFromObject	ITU-T Rec. X.681 | ISO/IEC 8824-2, clause 15

16.12 Regardless of whether or not a type is a "BuiltinType", "ReferencedType" or "ConstrainedType", its values can be specified by either a "BuiltinValue" or "ReferencedValue" of that type.

16.13 The value of a type referenced using the "NamedType" notation shall be defined by the notation "NamedValue", or when used as part of an "XMLValue", by the notation "XMLNamedValue". These productions are:

NamedValue ::= identifier Value

XMLNamedValue ::= "<" & identifier ">" XMLValue "</" & identifier ">"

where the "identifier" is the same as that used in the "NamedType" notation.

NOTE – The "identifier" is part of the notation, it does not form part of the value itself. It is used to unambiguously refer to the components of a set type, sequence type or choice type.

16.14 The implied (see 12.4) or explicit presence of an extension marker (see clause 6) in the definition of a type has no effect on the value notation. That is, the value notation for a type with an extension marker is exactly the same as if the extension marker was absent.

NOTE – Subclause 46.8 prohibits value notation used in a subtype constraint from referencing a value that is not in the extension root of the parent type.

17 Notation for the boolean type

17.1 The boolean type (see 3.6.7) shall be referenced by the notation "BooleanType":

BooleanType ::= `BOOLEAN`

17.2 The tag for types defined by this notation is universal class, number 1.

17.3 The value of a boolean type (see 3.6.73 and 3.6.38) shall be defined by the notation "BooleanValue", or when used as an "XMLValue", by the notation "XMLBooleanValue". These productions are:

```
BooleanValue ::= TRUE | FALSE

XMLBooleanValue ::=
        "<" & "true" "/>"
    |   "<" & "false" "/>"
```

18 Notation for the integer type

18.1 The integer type (see 3.6.41) shall be referenced by the notation "IntegerType":

```
IntegerType ::=
        INTEGER
    |   INTEGER "{" NamedNumberList "}"

NamedNumberList ::=
        NamedNumber
    |   NamedNumberList "," NamedNumber

NamedNumber ::=
        identifier "(" SignedNumber ")"
    |   identifier "(" DefinedValue ")"

SignedNumber ::=
        number
    |   "-" number
```

18.2 The second alternative of "SignedNumber" shall not be used if the "number" is zero.

18.3 The "NamedNumberList" is not significant in the definition of a type. It is used solely in the value notation specified in 18.9.

18.4 The "valuereference" in "DefinedValue" shall be of type integer.

NOTE – Since an "identifier" cannot be used to specify the value associated with "NamedNumber", the "DefinedValue" can never be misinterpreted as an "IntegerValue". Therefore in the following case

```
a INTEGER ::= 1
T1 ::= INTEGER { a(2) }
T2 ::= INTEGER { a(3), b(a) }
c T2 ::= b
d T2 ::= a
```

c denotes the value 1, since it cannot be a reference to the second nor the third occurrence of **a**, and **d** denotes the value 3.

18.5 The value of each "SignedNumber" or "DefinedValue" appearing in the "NamedNumberList" shall be different, and represents a distinguished value of the integer type.

18.6 Each "identifier" appearing in the "NamedNumberList" shall be different.

18.7 The order of the "NamedNumber"s in the "NamedNumberList" is not significant.

18.8 The tag for types defined by this notation is universal class, number 2.

18.9 The value of an integer type shall be defined by the notation "IntegerValue", or when used as an "XMLValue", by the notation "XMLIntegerValue". These productions are:

```
IntegerValue ::=
        SignedNumber
    |   identifier

XMLIntegerValue ::=
        SignedNumber
    |   "<" & identifier "/>"
```

18.10 The "identifier" in "IntegerValue" and in "XMLIntegerValue" shall be one of the "identifier"s in the "IntegerType" with which the value is associated, and shall represent the corresponding number.

NOTE – When referencing an integer value for which an "identifier" has been defined, use of the "identifier" form of "IntegerValue" and "XMLIntegerValue" should be preferred.

18.11 Within an instance of value notation for an integer type with a "NamedNumberList", any occurrence of a name that is both an "identifier" from the "NamedNumberList" and a reference name shall be interpreted as the "identifier".

19 Notation for the enumerated type

19.1 The enumerated type (see 3.6.24) shall be referenced by the notation "EnumeratedType":

EnumeratedType ::=
ENUMERATED "{" Enumerations "}"

Enumerations ::=
RootEnumeration
| **RootEnumeration "," "..." ExceptionSpec**
| **RootEnumeration "," "..." ExceptionSpec "," AdditionalEnumeration**

RootEnumeration ::= Enumeration

AdditionalEnumeration ::= Enumeration

Enumeration ::= EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem ::= identifier | NamedNumber

NOTE 1 – Each value of an "EnumeratedType" has an identifier which is associated with a distinct integer. However, the values themselves are not expected to have any integer semantics. Specifying the "NamedNumber" alternative of "EnumerationItem" provides control of the representation of the value in order to facilitate compatible extensions.

NOTE 2 – The numeric values inside the "NamedNumber"s in the "RootEnumeration" are not necessarily ordered or contiguous, and the numeric values inside the "NamedNumber"s in the "AdditionalEnumeration" are ordered but not necessarily contiguous.

19.2 For each "NamedNumber", the "identifier" and the "SignedNumber" shall be distinct from all other "identifier"s and "SignedNumber"s in the "Enumeration". Subclauses 18.2 and 18.4 also apply to each "NamedNumber".

19.3 Each "EnumerationItem" (in an "EnumeratedType") which is an "identifier" is successively assigned a distinct non-negative integer. For the "RootEnumeration", the successive integers starting with 0, but excluding any which are employed in "EnumerationItem"s which are "NamedNumber"s, are assigned.

NOTE – An integer value is associated with an "EnumerationItem" to assist in the definition of encoding rules. It is not otherwise used in the ASN.1 specification.

19.4 The value of each new "EnumerationItem" shall be greater than all previously defined "AdditionalEnumeration"s in the type.

19.5 When a "NamedNumber" is used in defining an "EnumerationItem" in the "AdditionalEnumeration", the value associated with it shall be different from the value of all previously defined "EnumerationItem"s (in this type) regardless of whether the previously defined "EnumerationItem"s occur in the enumeration root or not. For example:

```
A ::= ENUMERATED {a, b, ..., c(0)}      -- invalid, since both 'a' and 'c' equal 0
B ::= ENUMERATED {a, b, ..., c, d(2)}   -- invalid, since both 'c' and 'd' equal 2
C ::= ENUMERATED {a, b(3), ..., c(1)}   -- valid, 'c' = 1
D ::= ENUMERATED {a, b, ..., c(2)}      -- valid, 'c' = 2
```

19.6 The value associated with the first "EnumerationItem" in the "AdditionalEnumeration" alternative that is an "identifier" (not a "NamedNumber") shall be the smallest value for which an "EnumerationItem" is not defined in the "RootEnumeration" and all preceding "EnumerationItem"s in the "AdditionalEnumeration" (if any) are smaller. For example, the following are all valid:

```
A ::= ENUMERATED {a, b, ..., c}         -- c = 2
B ::= ENUMERATED {a, b, c(0), ..., d}   -- d = 3
C ::= ENUMERATED {a, b, ..., c(3), d}   -- d = 4
D ::= ENUMERATED {a, z(25), ..., d}     -- d = 1
```

19.7 The enumerated type has a tag which is universal class, number 10.

19.8 The value of an enumerated type shall be defined by the notation "EnumeratedValue", or when used as an "XMLValue", by the notation "XMLEnumeratedValue". These productions are:

EnumeratedValue ::= identifier

XMLEnumeratedValue ::= "<" & identifier "/>"

19.9 The "identifier" in "EnumeratedValue" and "XMLEnumeratedValue" shall be equal to that of an "identifier" in the "EnumeratedType" sequence with which the value is associated.

19.10 Within an instance of value notation for an enumerated type, any occurrence of a name that is both an "identifier" from the "Enumeration" and a reference name shall be interpreted as the "identifier".

20 Notation for the real type

20.1 The real type (see 3.6.54) shall be referenced by the notation "RealType":

RealType ::= REAL

20.2 The real type has a tag which is universal class, number 9.

20.3 The values of the real type are the values **PLUS-INFINITY** and **MINUS-INFINITY** together with the real numbers capable of being specified by the following formula involving three integers, M, B and E:

$$M \times B^{E}$$

where M is called the mantissa, B the base, and E the exponent.

20.4 The real type has an associated type which is used to give precision to the definition of the abstract values of the real type and is also used to support the value and subtype notations of the real type.

NOTE – Encoding rules may define a different type which is used to specify encodings, or may specify encodings without reference to the associated type. In particular, the encoding in BER and PER provides a Binary-Coded Decimal (BCD) encoding if "base" is 10, and an encoding which permits efficient transformation to and from hardware floating point representations if "base" is 2.

20.5 The associated type for value definition and subtyping purposes is (with normative comments):

```
SEQUENCE {
     mantissa   INTEGER,
     base  INTEGER (2|10),
     exponent   INTEGER
-- The associated mathematical real number is "mantissa"
-- multiplied by "base" raised to the power "exponent"
}
```

NOTE 1 – Non-zero values represented by "base" 2 and by "base" 10 are considered to be distinct abstract values even if they evaluate to the same real number value, and may carry different application semantics.

NOTE 2 – The notation **REAL (WITH COMPONENTS { ... , base (10)})** can be used to restrict the set of values to base 10 abstract values (and similarly for base 2 abstract values).

NOTE 3 – This type is capable of carrying an exact finite representation of any number which can be stored in typical floating point hardware, and of any number with a finite character-decimal representation.

20.6 The value of a real type shall be defined by the notation "RealValue", or when used in an "XMLValue", by the notation "XMLRealValue":

```
RealValue ::=
        NumericRealValue
    |   SpecialRealValue

NumericRealValue ::=
        realnumber
    |   "-" realnumber
    |   SequenceValue          -- Value of the associated sequence type

SpecialRealValue ::=
        PLUS-INFINITY
    |   MINUS-INFINITY
```

The second and third alternatives of "NumericRealValue" shall not be used for zero values.

```
XMLRealValue ::=
        XMLNumericRealValue | XMLSpecialRealValue

XMLNumericRealValue ::=
        realnumber
    |   "-" realnumber
```

The second alternative of "XMLNumericRealValue" shall not be used for zero values.

```
XMLSpecialRealValue ::=
        "<" & PLUS-INFINITY "/>" | "<" & MINUS-INFINITY "/>"
```

20.7 When the "realnumber" notation is used it identifies the corresponding "base" 10 abstract value. If the "RealType" is constrained to "base" 2, the "realnumber" identifies the "base" 2 abstract value corresponding either to the decimal value specified by the "realnumber" or to a locally-defined precision if an exact representation is not possible.

21 Notation for the bitstring type

21.1 The bitstring type (see 3.6.6) shall be referenced by the notation "BitStringType":

```
BitStringType ::=
        BIT STRING
    |   BIT STRING "{" NamedBitList "}"

NamedBitList ::=
        NamedBit
    |   NamedBitList "," NamedBit

NamedBit ::=
        identifier "(" number ")"
    |   identifier "(" DefinedValue ")"
```

21.2 The first bit in a bit string is called the leading bit. The final bit in a bit string is called the trailing bit.

NOTE – This terminology is used in specifying the value notation and in defining encoding rules.

21.3 The "DefinedValue" shall be a reference to a non-negative value of type integer.

21.4 The value of each "number" or "DefinedValue" appearing in the "NamedBitList" shall be different, and is the number of a distinguished bit in a bitstring value. The leading bit of the bit string is identified by the "number" zero, with succeeding bits having successive values.

21.5 Each "identifier" appearing in the "NamedBitList" shall be different.

NOTE 1 – The order of the "NamedBit" production sequences in the "NamedBitList" is not significant.

NOTE 2 – Since an "identifier" that appears within the "NamedBitList" cannot be used to specify the value associated with a "NamedBit", the "DefinedValue" can never be misinterpreted as an "IntegerValue". Therefore in the following case:

```
a  INTEGER ::= 1
T1 ::= INTEGER { a(2) }
T2 ::= BIT STRING { a(3), b(a) }
```

the last occurrence of **a** denotes the value 1, as it cannot be a reference to the second nor the third occurrence of **a**.

21.6 The presence of a "NamedBitList" has no effect on the set of abstract values of this type. Values containing 1 bits other than the named bits are permitted.

21.7 When a "NamedBitList" is used in defining a bitstring type ASN.1 encoding rules are free to add (or remove) arbitrarily any trailing 0 bits to (or from) values that are being encoded or decoded. Application designers should therefore ensure that different semantics are not associated with such values which differ only in the number of trailing 0 bits.

21.8 This type has a tag which is universal class, number 3.

21.9 The value of a bitstring type shall be defined by the notation "BitStringValue", or when used as an "XMLValue", by the notation "XMLBitStringValue". These productions are:

```
BitStringValue ::=
        bstring
    |   hstring
    |   "{" IdentifierList "}"
    |   "{" "}"
    |   CONTAINING Value

IdentifierList ::=
        identifier
    |   IdentifierList "," identifier
```

XMLBitStringValue ::=
    XMLTypedValue
  | **Xmlbstring**
  | **XMLIdentifierList**
  | **empty**

XMLIdentifierList ::=
    "<" & identifier "/>"
  | **XMLIdentifierList "<" & identifier "/>"**

21.10 The "XMLTypedValue" alternative shall not be used unless the bitstring has a contents constraint which includes an ASN.1 type and does not include an **ENCODED BY**. If this alternative is used, the "XMLTypedValue" shall be a value of the ASN.1 type in the contents constraint.

21.11 The "XMLIdentifierList" alternative shall not be used unless the bitstring has a "NamedBitList".

21.12 Each "identifier" in "BitStringValue" or "XMLBitStringValue" shall be the same as an "identifier" in the "BitStringType" production sequence with which the value is associated.

21.13 The "empty" alternative denotes a bitstring with no bits.

21.14 If the bitstring has named bits, the "BitStringValue" or "XMLBitStringValue" notation denotes a bitstring value with ones in the bit positions specified by the numbers corresponding to the "identifier"s, and with all other bits zero.

NOTE – For a "BitStringType" that has a "NamedBitList", the "**{**" "**}**" production sequence in "BitStringValue" and the "empty" in "XMLBitStringValue" are used to denote the bitstring which contains no one bits.

21.15 When using the "bstring" or "xmlbstring" notation, the leading bit of the bitstring value is on the left, and the trailing bit of the bitstring value is on the right.

21.16 When using the "hstring" notation, the most significant bit of each hexadecimal digit corresponds to the leftmost bit in the bitstring.

NOTE – This notation does not, in any way, constrain the way encoding rules place a bitstring into octets for transfer.

21.17 The "hstring" notation shall not be used unless the bitstring value consists of a multiple of four bits.

EXAMPLE

```
'A98A'H
```

and

```
'1010100110001010'B
```

are alternative notations for the same bitstring value. If the type was defined using a "NamedBitList", the (single) trailing zero does not form part of the value, which is thus 15 bits in length. If the type was defined without a "NamedBitList", the trailing zero does form part of the value, which is thus 16 bits in length.

21.18 The **CONTAINING** alternative can only be used if there is a contents constraint on the bitstring type which includes **CONTAINING**. The "Value" shall then be value notation for a value of the "Type" in the "ContentsConstraint" (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11).

NOTE – This value notation can never appear in a subtype constraint because ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11.3 forbids further constraints after a "ContentsConstraint", and the above text forbids its use unless the governor has a "ContentsConstraint".

21.19 The **CONTAINING** alternative shall be used if there is a contents constraint on the bitstring type which does not contain **ENCODED BY**.

22 Notation for the octetstring type

22.1 The octetstring type (see 3.6.49) shall be referenced by the notation "OctetStringType":

OctetStringType ::= OCTET STRING

22.2 This type has a tag which is universal class, number 4.

22.3 The value of an octetstring type shall be defined by the notation "OctetStringValue", or when used as an "XMLValue", by the notation "XMLOctetStringValue". These productions are:

OctetStringValue ::=
    bstring
  | **hstring**
  | **CONTAINING Value**

XMLOctetStringValue ::=
    XMLTypedValue
  | **xmlhstring**

22.4 The "XMLTypedValue" alternative shall not be used unless the octetstring has a contents constraint which includes an ASN.1 type and does not include an **ENCODED BY**. If this alternative is used, the "XMLTypedValue" shall be a value of the ASN.1 type in the contents constraint.

22.5 In specifying the encoding rules for an octetstring, the octets are referenced by the terms first octet and trailing octet, and the bits within an octet are referenced by the terms most significant bit and least significant bit.

22.6 When using the "bstring" notation, the left-most bit of the "bstring" notation shall be the most significant bit of the first octet of the octetstring value. If the "bstring" is not a multiple of eight bits, it shall be interpreted as if it contained additional zero trailing bits to make it the next multiple of eight.

22.7 When using the "hstring" or "xmlhstring" notation, the left-most hexadecimal digit shall be the most significant semi-octet of the first octet.

22.8 If the "hstring" is an odd number of hexadecimal digits, it shall be interpreted as if it contained a single additional trailing zero hexadecimal digit. The "xmlhstring" shall not be an odd number of hexadecimal digits.

22.9 The **CONTAINING** alternative can only be used if there is a contents constraint on the octetstring type which includes **CONTAINING**. The "Value" shall then be value notation for a value of the "Type" in the "ContentsConstraint" (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11).

NOTE – This value notation can never appear in a subtype constraint because ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 11.3 forbids further constraints after a "ContentsConstraint", and the above text forbids its use unless the governor has a "ContentsConstraint".

22.10 The **CONTAINING** alternative shall be used if there is a contents constraint on the octetstring type which does not contain **ENCODED BY**.

23 Notation for the null type

23.1 The null type (see 3.6.44) shall be referenced by the notation "NullType":

NullType ::= NULL

23.2 This type has a tag which is universal class, number 5.

23.3 The value of a null type shall be referenced by the notation "NullValue", or when used as an "XMLValue", by the notation "XMLNullValue". These productions are:

NullValue ::= NULL

XMLNullValue ::= empty

24 Notation for sequence types

24.1 The notation for defining a sequence type (see 3.6.60) shall be the "SequenceType":

SequenceType ::=
    SEQUENCE "{" "}"
  | **SEQUENCE "{" ExtensionAndException OptionalExtensionMarker "}"**
  | **SEQUENCE "{" ComponentTypeLists "}"**

ExtensionAndException ::= "..." | "..." ExceptionSpec

OptionalExtensionMarker ::= "," "..." | empty

ComponentTypeLists ::=
    RootComponentTypeList
  | **RootComponentTypeList "," ExtensionAndException ExtensionAdditions**

```
            OptionalExtensionMarker
    |   RootComponentTypeList "," ExtensionAndException ExtensionAdditions
            ExtensionEndMarker "," RootComponentTypeList
    |   ExtensionAndException ExtensionAdditions ExtensionEndMarker ","
            RootComponentTypeList
    |   ExtensionAndException ExtensionAdditions OptionalExtensionMarker

RootComponentTypeList ::= ComponentTypeList

ExtensionEndMarker ::= "," "..."

ExtensionAdditions ::=
        "," ExtensionAdditionList
    |   empty

ExtensionAdditionList ::=
        ExtensionAddition
    |   ExtensionAdditionList "," ExtensionAddition

ExtensionAddition ::=
        ComponentType
    |   ExtensionAdditionGroup

ExtensionAdditionGroup ::= "[[" VersionNumber ComponentTypeList "]]"

VersionNumber ::= empty | number ":"

ComponentTypeList ::=
        ComponentType
    |   ComponentTypeList "," ComponentType

ComponentType ::=
        NamedType
    |   NamedType OPTIONAL
    |   NamedType DEFAULT Value
    |   COMPONENTS OF Type
```

24.2 When the "ComponentTypeLists" production occurs within the definition of a module for which automatic tagging is selected (see 12.3), and none of the occurrences of "NamedType" in any of the first three alternatives for "ComponentType" contains a "TaggedType", then automatic tagging transformation is selected for the entire "ComponentTypeLists", otherwise it is not.

NOTE 1 – The use of the "TaggedType" notation within the definition of the list of components for a sequence type gives control of tags to the specifier, as opposed to automatic assignment by the automatic tagging mechanism. Therefore, in the following case:

```
T ::= SEQUENCE { a INTEGER, b [1] BOOLEAN, c OCTET STRING }
```

no automatic tagging is applied to the list of components **a**, **b**, **c**, even if this definition of sequence type **T** occurs within a module for which automatic tagging is selected.

NOTE 2 – Only those occurrences of the "ComponentTypeLists" production appearing within a module where automatic tagging is selected are candidates for transformation by automatic tagging.

24.3 The decision to apply the automatic tagging transformation is taken individually for each occurrence of "ComponentTypeLists" and *prior* to the **COMPONENTS OF** transformation specified by 24.4. However, as specified in 24.7 to 24.9, the automatic tagging transformation (if applied) is applied *after* the **COMPONENTS OF** transformation.

NOTE – The effect of this is that the application of automatic tags is suppressed by tags explicitly present in the "ComponentTypeLists", but not by tags present in the "Type" following **COMPONENTS OF**.

24.4 "Type" in the "**COMPONENTS OF** Type" notation shall be a sequence type. The "**COMPONENTS OF** Type" notation shall be used to define the inclusion, at this point in the list of components, of all the component types of the referenced type, except for any extension marker and extension additions that may be present in the "Type". (Only the "RootComponentTypeList" of the "Type" in the "**COMPONENTS OF** Type" is included; extension markers and extension additions, if any, are ignored by the "**COMPONENTS OF** Type" notation.) Any subtype constraint applied to the referenced type is ignored by this transformation.

NOTE – This transformation is logically completed prior to the satisfaction of the requirements in the following subclauses.

24.5 The following subclauses each identify a series of occurrences of "ComponentType" in either the root or the extension additions or both. The rule of 24.5.1 shall apply to all such series.

24.5.1 Where there are one or more consecutive occurrences of "ComponentType" that are all marked **OPTIONAL** or **DEFAULT**, the tags of those "ComponentType"s and of any immediately following component type in the series shall be distinct (see clause 30). If automatic tagging was selected, the requirement that tags be distinct applies only after automatic tagging has been performed, and will always be satisfied.

24.5.2 Subclause 24.5.1 shall apply to the series of "ComponentType"s in the root.

24.5.3 Subclause 24.5.1 shall apply to the complete series of "ComponentType"s in the root or in the extension additions, in the textual order of their occurrence in the type definition (ignoring all version brackets and ellipsis notation). (See also 48.7.)

24.6 When the third or fourth alternative of "ComponentTypeLists" is used, all "ComponentType"s in extension additions shall have tags which are distinct from the tags of the textually following "ComponentType"s up to and including the first such "ComponentType" that is not marked **OPTIONAL** or **DEFAULT** in the trailing "RootComponentTypeList", if any. (See also 48.7.)

24.7 The automatic tagging transformation of an occurrence of "ComponentTypeLists" is logically performed *after* the transformation specified by 24.4, but only if 24.2 determines that it shall apply to that occurrence of "ComponentTypeLists". Automatic tagging transformation impacts each "ComponentType" of the "ComponentTypeLists" by replacing the "Type" originally in the "NamedType" production with a replacement "TaggedType" occurrence specified in 24.9.

24.8 If automatic tagging is in effect and the "ComponentType"s in the extension root have no tags, then no "ComponentType" within the "ExtensionAdditionList" shall be a "TaggedType".

24.9 If automatic tagging is in effect, the replacement "TaggedType" is specified as follows:

a) the replacement "TaggedType" notation uses the "Tag Type" alternative;

b) the "Class" of the replacement "TaggedType" is empty (i.e., tagging is context-specific);

c) the "ClassNumber" in the replacement "TaggedType" is tag value zero for the first "ComponentType" in the "RootComponentTypeList", one for the second, and so on, proceeding with increasing tag numbers;

d) the "ClassNumber" in the replacement "TaggedType" of the first "ComponentType" in the "ExtensionAdditionList" is zero if the "RootComponentTypeList" is missing, else it is one greater than the largest "ClassNumber" in the "RootComponentTypeList", with the next "ComponentType" in the "ExtensionAdditionList" having a "ClassNumber" one greater than the first, and so on, proceeding with increasing tag numbers;

e) the "Type" in the replacement "TaggedType" is the original "Type" being replaced.

NOTE 1 – The rules governing specification of implicit tagging or explicit tagging for replacement "TaggedType"s are provided by 30.6. Automatic tagging is always implicit tagging unless the "Type" is a choice type or an open type notation, or a "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3), in which case it is explicit tagging.

NOTE 2 – Once 24.7 is satisfied, the tags of the components are completely determined, and are not modified even when the sequence type is referenced in the definition of a component within another "ComponentTypeLists" for which automatic tagging transformation applies. Thus, in the following case:

```
T ::= SEQUENCE { a   Ta,  b   Tb,  c   Tc }
E ::= SEQUENCE { f1  E1,  f2  T,   f3  E3 }
```

automatic tagging applied to the components of **E** never affects the tags attached to components **a**, **b** and **c** of **T**, whatever the tagging environment of **T**. If **T** is defined in an automatic tagging environment and **E** is not in an automatic tagging environment, automatic tagging is still applied to components **a**, **b** and **c** of **T**.

NOTE 3 – When a sequence type appears as the "Type" in "**COMPONENTS OF** Type", each occurrence of "ComponentType" in it is duplicated by the application of 24.4 prior to the possible application of automatic tagging to the referencing sequence type. Thus, in the following case:

```
T ::= SEQUENCE { a  Ta,  b  SEQUENCE { b1  T1,  b2  T2,  b3  T3},  c Tc }
W ::= SEQUENCE { x  Wx,  COMPONENTS OF  T, y  Wy }
```

the tags of **a**, **b**, and **c** within **T** need not be the same as the tags of **a**, **b**, and **c** within **W** if **W** has been defined in an automatic tagging environment, but the tags of **b1**, **b2** and **b3** are the same in both **T** and **W**. In other words, the automatic tagging transformation is only applied once to a given "ComponentTypeLists".

NOTE 4 – Subtyping has no impact on automatic tagging.

NOTE 5 – When automatic tagging is in place, insertion of new components at any location other than the extension insertion point (see 3.6.29) may result in changes to other components due to the side effect of modifying the tags thus causing interworking problems with an older version of the specification.

24.10 If **OPTIONAL** or **DEFAULT** are present, the corresponding value may be omitted from a value of the new type.

24.11 If **DEFAULT** occurs, the omission of a value for that type shall be exactly equivalent to the insertion of the value defined by "Value", which shall be a value notation for a value of the type defined by "Type" in the "NamedType" production sequence.

24.12 The value corresponding to an "ExtensionAdditionGroup" (all components together) is optional. However, if such a value is present, then the value corresponding to the components within the bracketed "ComponentTypeList" that are not marked **OPTIONAL** or **DEFAULT** shall be present.

24.13 The "identifier"s in all "NamedType" production sequences of the "ComponentTypeLists" (together with those obtained by expansion of **COMPONENTS OF**) shall all be distinct.

24.14 A value for a given extension addition type shall not be specified unless there are values specified for all extension addition types not marked **OPTIONAL** or **DEFAULT** that lie logically between the extension addition type and the extension root.

NOTE 1 – Where the type has grown from the extension root (version 1) through version 2 to version 3 by the addition of extension additions, the presence in an encoding of any addition from version 3 requires the presence of an encoding of all additions in version 2 that are not marked **OPTIONAL** or **DEFAULT**.

NOTE 2 – "ComponentType"s that are extension additions but not contained within an "ExtensionAdditionGroup" should always be encoded if they are not marked **OPTIONAL** or **DEFAULT**, except when the abstract value is being relayed from a sender that is using an earlier version of the abstract syntax in which the "ComponentType" is not defined.

NOTE 3 – Use of the "ExtensionAdditionGroup" production is recommended because:

a) it can result in more compact encodings depending on the encoding rules (e.g., PER);

b) the syntax is more precise in that it clearly indicates that a value of a type defined in the "ExtensionAdditionList" and not marked **OPTIONAL** or **DEFAULT** should always be present in an encoding if the extension addition group in which it is defined is encoded (compare with Note 1);

c) the syntax makes it clear which types in an "ExtensionAdditionList" must as a group be supported by an application.

24.15 A "VersionNumber" shall be used only if all "ExtensionAdditions"s and "ExtensionAdditionAlternatives", within the module are "ExtensionAdditionGroup"s or "ExtensionAdditionAlternativesGroup"s with "VersionNumber"s. The "number" in each "VersionNumber" of an "ExtensionAdditionGroup" shall be greater than or equal to two, and shall be greater than the "number" in any preceding "ExtensionAdditionGroup" within an insertion point.

NOTE 1 – The convention used here is that the specification with no extension addition groups is version 1, thus the first added extension addition group will have a number greater than or equal to 2. Where a single "ExtensionAddition" is needed for an "ExtensionAdditions", an "ExtensionAdditionGroup" can be used with a single "ExtensionAddition".

NOTE 2 – The restrictions on use of "VersionNumber" apply only within a single module and impose no constraints on imported types.

24.16 All sequence types have a tag which is universal class, number 16.

NOTE – Sequence-of types have the same tag as sequence types (see 25.2).

24.17 The notation for defining a value of a sequence type shall be "SequenceValue", or when used as an "XMLValue", "XMLSequenceValue". These productions are:

```
SequenceValue ::=
        "{" ComponentValueList "}"
    |   "{" "}"

ComponentValueList ::=
        NamedValue
    |   ComponentValueList "," NamedValue

XMLSequenceValue ::=
        XMLComponentValueList
    |   empty

XMLComponentValueList ::=
        XMLNamedValue
    |   XMLComponentValueList XMLNamedValue
```

24.18 The "**{**" "**}**" or "empty" notation shall only be used if:

a) all "ComponentType" sequences in the "SequenceType" are marked **DEFAULT** or **OPTIONAL**, and all values are omitted; or

b) the type notation was **SEQUENCE{}**.

24.19 There shall be one "NamedValue" or "XMLNamedValue" for each "NamedType" in the "SequenceType" which is not marked **OPTIONAL** or **DEFAULT**, and the values shall be in the same order as the corresponding "NamedType" sequences.

25 Notation for sequence-of types

25.1 The notation for defining a sequence-of type (see 3.6.61) from another type shall be the "SequenceOfType".

SequenceOfType ::= SEQUENCE OF Type | SEQUENCE OF NamedType

NOTE – If an initial letter which is upper-case is needed for an XML tag name used in XML Value Notation for the "SequenceOfType", then the first alternative should be used. (The XML tag name is then formed from the name of the "Type".)

25.2 All sequence-of types have a tag which is universal class, number 16.

NOTE – Sequence types have the same tag as sequence-of types (see 24.16).

25.3 The notation for defining a value of a sequence-of type shall be the "SequenceOfValue", or when used as an "XMLValue", "XMLSequenceOfValue". These productions are:

```
SequenceOfValue ::=
        "{" ValueList "}"
    |   "{" NamedValueList "}"
    |   "{"  "}"

ValueList ::=
        Value
    |   ValueList "," Value

NamedValueList ::=
        NamedValue
    |   NamedValueList "," NamedValue

XMLSequenceOfValue ::=
        XMLValueList
    |   XMLDelimitedItemList
    |   XMLSpaceSeparatedList
    |   empty

XMLValueList ::=
        XMLValueOrEmpty
    |   XMLValueOrEmpty XMLValueList

XMLValueOrEmpty ::=
        XMLValue
    |   "<" & NonParameterizedTypeName "/>"

XMLSpaceSeparatedList ::=
        XMLValueOrEmpty
    |   XMLValueOrEmpty " " XMLSpaceSeparatedList

XMLDelimitedItemList ::=
        XMLDelimitedItem
    |   XMLDelimitedItem XMLDelimitedItemList

XMLDelimitedItem ::=
        "<" & NonParameterizedTypeName ">" XMLValue
            "</" & NonParameterizedTypeName ">"
    |   "<" & identifier ">" XMLValue "</" & identifier ">"
```

The "**{**" "**}**" or "empty" notation is used when the "SequenceOfValue" or "XMLSequenceOfValue" is an empty list.

NOTE 1 – Semantic significance may be placed on the order of these values.

NOTE 2 – The "XMLSpaceSeparatedList" production is not used in this Recommendation | International Standard, and is not used in XML Value Notation. It is provided in order to allow specification of the use of "XMLSpaceSeparatedList" in encodings of the "IntegerType", "RealType", "ObjectIdentifierType", "RelativeOIDType", and the **GeneralizedTime** and **UTCTime** useful types. It is also possible to specify use of "XMLValueList" instead of "XMLDelimitedItemList" for some instances of "**SEQUENCE OF SEQUENCE**" and "**SEQUENCE OF SET**".

25.4 If the"XMLValue" for the component is "empty", then the second alternative of "XMLValueOrEmpty" shall be chosen to represent that value of the component.

25.5 The "XMLValueList" or "XMLDelimitedItemList" productions shall be used in accordance with column 2 of Table 5, where the "Type" of the component is listed in column 1.

Table 5 – "XMLSequenceOfValue" and "XMLSetOfValue" notation for ASN.1 types

ASN.1 type	XML value notation
BitStringType	XMLDelimitedItemList
BooleanType	XMLValueList
CharacterStringType	XMLDelimitedItemList
ChoiceType	XMLValueList
ConstrainedType	*See 25.7*
DefinedType	*See 25.9*
EmbeddedPDVType	XMLDelimitedItemList
EnumeratedType	XMLValueList
ExternalType	XMLDelimitedItemList
InstanceOfType	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, C.9*
IntegerType	XMLDelimitedItemList
NullType	XMLValueList
ObjectClassFieldType	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.10 and 14.11*
ObjectIdentifierType	XMLDelimitedItemList
OctetStringType	XMLDelimitedItemList
RealType	XMLDelimitedItemList
RelativeOIDType	XMLDelimitedItemList
SelectionType	*See 25.8*
SequenceType	XMLDelimitedItemList
SequenceOfType	XMLDelimitedItemList
SetType	XMLDelimitedItemList
SetOfType	XMLDelimitedItemList
TaggedType	*See 25.6*
UsefulType (GeneralizedTime)	XMLDelimitedItemList
UsefulType (UTCTime)	XMLDelimitedItemList
UsefulType (ObjectDescriptor)	XMLDelimitedItemList
TypeFromObject	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 15.6*
ValueSetFromObjects	*See ITU-T Rec. X.681 | ISO/IEC 8824-2, 15.6*

25.6 If the "Type" of the component is a "TaggedType" then the type which determines the "XMLSequenceOfValue" notation shall be the "Type" in the "TaggedType" (see 30.1). If this is itself a "TaggedType", then this subclause 25.6 shall be recursively applied.

25.7 If the "Type" of the component is a "ConstrainedType" then the type which determines the "XMLSequenceOfValue" notation shall be the "Type" in the "ConstrainedType" (see 45.1). If this is itself a "ConstrainedType", then this subclause 25.7 shall be recursively applied.

25.8 If the "Type" of the component is a "SelectionType" then the type which determines the "XMLSequenceOfValue" notation shall be the type referenced by the "SelectionType" (see clause 29).

25.9 If the "Type"of the component is a "DefinedType" then the type which determines the "XMLSequenceOfValue" notation shall be the type referenced by the "DefinedType" (see 13.1).

25.10 The second alternative of "XMLDelimitedItem" shall be used if and only if the "SequenceOfType" contains an "identifier", and the "identifier" in the "XMLDelimitedItem" shall be that "identifier".

25.11 If the first alternative of "XMLDelimitedItem" is used, then if the component of the sequence-of type (after ignoring any tags) is a "typereference" or an "ExternalTypeReference", then the "NonParameterizedTypeName" shall be that "typereference" or "ExternalTypeReference", otherwise it shall be the "xmlasn1typename" specified in Table 4 corresponding to the built-in type of the component.

25.12 If the first alternative of "SequenceOfType" is used, then the first alternative of "SequenceOfValue" shall be used. Each "Value" in the "ValueList" of "SequenceOfValue", and each "XMLValue" in the alternatives of "XMLSequenceOfValue" shall be of the type specified in the "SequenceOfType".

25.13 If the second alternative of "SequenceOfType" is used, then the second alternative of "SequenceOfValue" shall be used, and each "NamedValue" in the "NamedValueList" shall contain a "Value" of the type specified in the "NamedType" of the "SequenceOfType". The "identifier" in the "NamedValue"s shall be the "identifier" in the "NamedType" of the "SequenceOfType".

26 Notation for set types

26.1 The notation for defining a set type (see 3.6.64) from other types shall be the "SetType":

```
SetType ::=
        SET "{" "}"
    |   SET "{" ExtensionAndException OptionalExtensionMarker "}"
    |   SET "{" ComponentTypeLists "}"
```

"ComponentTypeLists", "ExtensionAndException" and "OptionalExtensionMarker" are specified in 24.1.

26.2 "Type" in the "**COMPONENTS OF** Type" notation shall be a set type. The "**COMPONENTS OF** Type" notation shall be used to define the inclusion, at this point in the list of components, of all the component types of the referenced type, except for any extension marker and extension additions that may be present in the "Type". (Only the "RootComponentTypeList" of the "Type" in the "**COMPONENTS OF** Type" is included; extension markers and extension additions, if any, are ignored by the "**COMPONENTS OF** Type" notation.) Any subtype constraint applied to the referenced type is ignored by this transformation.

NOTE – This transformation is logically completed prior to the satisfaction of the requirements in the following subclauses.

26.3 The "ComponentType" types in a set type shall all have different tags (see clause 30). The tag of each new "ComponentType" added to the "ExtensionAdditions" shall be canonically greater (see 8.6) than those of the other components in the "ExtensionAdditions".

NOTE – Where the "TagDefault" for the module in which this notation appears is **AUTOMATIC TAGS**, this is achieved regardless of the actual "ComponentType"s, as a result of the application of 24.7. (See also 48.7.)

26.4 Subclauses 24.2 and 24.7 to 24.13 also apply to set types.

26.5 All set types have a tag which is universal class, number 17.

NOTE – Set-of types have the same tag as set types (see 27.2).

26.6 There shall be no semantics associated with the order of values in a set type.

26.7 The notation for defining the value of a set type shall be "SetValue", or when used as an "XMLValue", "XMLSetValue". These productions are:

```
SetValue ::=
        "{" ComponentValueList "}"
    |   "{" "}"

XMLSetValue ::=
        XMLComponentValueList
    |   empty
```

"ComponentValueList" and "XMLComponentValueList" are specified in 24.17.

26.8 The "SetValue" and "XMLSetValue" shall only be "**{**" "**}**" and "empty" respectively if:

a) all "ComponentType" sequences in the "SetType" are marked **DEFAULT** or **OPTIONAL**, and all values are omitted; or

b) the type notation was **SET{}**.

26.9 There shall be one "NamedValue" or "XMLNamedValue" for each "NamedType" in the "SetType" which is not marked **OPTIONAL** or **DEFAULT**.

NOTE – These "NamedValue"s or "XMLNamedValue"s may appear in any order.

27 Notation for set-of types

27.1 The notation for defining a set-of type (see 3.6.65) from another type shall be the "SetOfType":

SetOfType ::=
SET OF Type
| **SET OF NamedType**

NOTE – If an initial letter which is upper-case is needed for an XML tag name used in XML Value Notation for the "SetOfType", then the first alternative should be used. (The XML tag name is then formed from the name of the "Type".)

27.2 All set-of types have a tag which is universal class, number 17.

NOTE – Set types have the same tag as set-of types (see 26.5).

27.3 The notation for defining a value of a set-of type shall be the "SetOfValue", or when used as an "XMLValue", "XMLSetOfValue". These productions are:

SetOfValue ::=
"{" ValueList "}"
| **"{" NamedValueList "}"**
| **"{" "}"**

XMLSetOfValue ::=
XMLValueList
| **XMLDelimitedItemList**
| **XMLSpaceSeparatedList**
| **empty**

"ValueList", "NamedValueList" and the alternatives of "XMLSetOfValue" are specified in 25.3. The "**{**" "**}**" or "empty" notation is used when the "SetOfValue" or "XMLSetOfValue" is an empty list.

NOTE 1 – Semantic significance should not be placed on the order of these values.

NOTE 2 – Encoding rules are not required to preserve the order of these values.

NOTE 3 – The set-of type is not a mathematical set of values, thus, as an example, for **SET OF INTEGER** the values **{ 1 }** and **{ 1 1 }** are distinct.

27.4 If the first alternative of "SetOfType" is used, then the first alternative of "SetOfValue" shall be used. Each "Value" in the "ValueList" of "SetOfValue", and each "XMLValue" in the alternatives of "XMLSetOfValue" shall be of the type specified in the "SetOfType".

27.5 If the second alternative of "SetOfType" is used, then the second alternative of "SetOfValue" shall be used, and each "NamedValue" sequence in the "NamedValueList" shall contain a "Value" of the type specified in the "NamedType" of the "SetOfType". The "identifier" in the "NamedValue"s shall be the "identifier" in the "NamedType" of the "SetOfType".

28 Notation for choice types

28.1 The notation for defining a choice type (see 3.6.13) from other types shall be the "ChoiceType":

ChoiceType ::= CHOICE "{" AlternativeTypeLists "}"

AlternativeTypeLists ::=
RootAlternativeTypeList
| **RootAlternativeTypeList "," ExtensionAndException ExtensionAdditionAlternatives OptionalExtensionMarker**

RootAlternativeTypeList ::= AlternativeTypeList

ExtensionAdditionAlternatives ::=
"," ExtensionAdditionAlternativesList
| **empty**

ExtensionAdditionAlternativesList ::=
ExtensionAdditionAlternative
| **ExtensionAdditionAlternativesList "," ExtensionAdditionAlternative**

```
ExtensionAdditionAlternative ::=
        ExtensionAdditionAlternativesGroup
    |   NamedType

ExtensionAdditionAlternativesGroup ::=
        "[[" VersionNumber AlternativeTypeList "]]"

AlternativeTypeList ::=
        NamedType
    |   AlternativeTypeList "," NamedType
```

NOTE – "**T ::= CHOICE { a A }**" and **A** are not the same type, and may be encoded differently by encoding rules.

28.2 When the "AlternativeTypeLists" production occurs within the definition of a module for which automatic tagging is selected (see 12.3), and none of the occurrences of "NamedType" in any "AlternativeTypeList" contains a "TaggedType", the automatic tagging transformation is selected for the entire "AlternativeTypeLists", otherwise it is not.

28.3 The types defined in the "AlternativeTypeList" productions in an "AlternativeTypeLists" shall have distinct tags (see clause 30, and 48.7). If automatic tagging was selected, the requirement that tags be distinct applies only after automatic tagging has been performed, and will always be satisfied.

28.4 If automatic tagging is in effect and the "NamedType"s in the extension root have no tags, then no "NamedType" within the "ExtensionAdditionAlternativesList" shall be a tagged type.

28.5 The automatic tagging transformation impacts each "NamedType" of the "AlternativeTypeLists" by replacing the "Type" originally in the "NamedType" production with a replacement "TaggedType". The replacement "TaggedType" is specified as follows:

a) the replacement "TaggedType" notation uses the "Tag Type" alternative;

b) the "Class" of the replacement "TaggedType" is empty (i.e., tagging is context-specific);

c) the "ClassNumber" in the replacement "TaggedType" is tag value zero for the first "NamedType" in the "RootAlternativeTypeList", one for the second, and so on, proceeding with increasing tag numbers;

d) the "ClassNumber" in the replacement "TaggedType" of the first "NamedType" in the "ExtensionAdditionAlternativesList" is one greater than the largest "ClassNumber" in the "RootAlternativeTypeList", with the next "NamedType" in the "ExtensionAdditionAlternativesList" having a "ClassNumber" one greater than the first, and so on, proceeding with increasing tag numbers;

e) the "Type" in the replacement "TaggedType" is the original "Type" being replaced.

NOTE 1 – The rules governing specification of implicit tagging or explicit tagging for replacement "TaggedType"s are provided by 30.6. Automatic tagging is always implicit tagging unless the "Type" is an untagged choice type or an untagged open type notation, or an untagged "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3), in which case it is explicit tagging.

NOTE 2 – Once automatic tagging has been applied, the tags of the components are completely determined, and are not modified even when the choice type is referenced in the definition of an alternative within another "AlternativeTypeLists" for which automatic tagging transformation applies. Thus, in the following case:

```
T ::= CHOICE { a Ta, b Tb, c Tc }

E ::= CHOICE {f1 E1, f2 T, f3 E3}
```

automatic tagging applied to the components of **E** never affects the tags attached to components **a**, **b** and **c** of **T**, whatever the tagging environment of **T**. If **T** is defined in an automatic tagging environment and **E** is not in an automatic tagging environment, automatic tagging is still applied to components **a**, **b** and **c** of **T**.

NOTE 3 – Subtyping does not affect automatic tagging.

NOTE 4 – When automatic tagging is in place, insertion of new alternatives at any location other than the extension insertion point (see 3.6.29) may result in changes to other alternatives due to the side effect of modifying the tags thus causing interworking problems with an older version of the specification.

28.5 "VersionNumber" is defined in 24.1, and the restrictions on consistent use of "VersionNumber" throughout a module that are specified in 24.15 shall apply to the use of "number"s within this production.

28.6 The tag of each new "NamedType" added to the "ExtensionAdditionAlternativesList" shall be canonically greater (see 8.6) than those of the other alternatives in the "ExtensionAdditionAlternativesList", and shall be the last "NamedType" in the "ExtensionAdditionAlternativesList".

28.7 The choice type contains values which do not all have the same tag. (The tag depends on the alternative which contributed the value to the choice type.)

28.8 When this type does not have an extension marker and is used in a place where this Recommendation | International Standard requires the use of types with distinct tags (see 28.3), all possible tags of values of the choice type shall be considered in such requirement. The following examples which assume that the "TagDefault" is not **AUTOMATIC TAGS** illustrate this requirement.

EXAMPLES

1
```
A ::= CHOICE {
b     B,
c     NULL}

B ::= CHOICE {
d     [0] NULL,
e     [1] NULL}
```

2
```
A ::= CHOICE {
b     B,
c     C}

B ::= CHOICE {
d     [0] NULL,
e     [1] NULL}

C ::= CHOICE {
f     [2] NULL,
g     [3] NULL}
```

3 (Incorrect)
```
      A ::= CHOICE {
b     B,
c     C}

      B ::= CHOICE {
d     [0] NULL,
e     [1] NULL}

      C ::= CHOICE {
f     [0] NULL,
g     [1] NULL}
```

Examples 1 and 2 are correct uses of the notation. Example 3 is incorrect without automatic tagging, as the tags for types **d** and **f** are identical, as well as for **e** and **g**.

28.9 The "identifier"s of all "NamedType"s in the "AlternativeTypeLists" shall differ from those of the other "NamedType"s in that list.

28.10 The notation for defining the value of a choice type shall be the "ChoiceValue", or when used as an "XMLValue", "XMLChoiceValue". These productions are:

ChoiceValue ::= identifier ":" Value

XMLChoiceValue ::= "<" & identifier ">" XMLValue "</" & identifier ">"

28.11 "Value" or "XMLValue" shall be a notation for a value of the type in the "AlternativeTypeLists" that is named by the "identifier".

29 Notation for selection types

29.1 The notation for defining a selection type (see 3.6.59) shall be "SelectionType":

SelectionType ::= identifier "<" Type

where "Type" denotes a choice type, and "identifier" is that of some "NamedType" appearing in the "AlternativeTypeLists" of the definition of that choice type.

29.2 When "Type" denotes a constrained type, the selection is performed on the parent type, ignoring any subtype constraint on the parent type.

29.3 Where the "SelectionType" is used as a "NamedType", the "identifier" of the "NamedType" is present, as well as the "identifier" of the "SelectionType".

29.4 Where the "SelectionType" is used as a "Type", the "identifier" is retained and the type denoted is that of the selected alternative.

29.5 The notation for a value of a selection type shall be the notation for a value of the type referenced by the "SelectionType".

30 Notation for tagged types

A tagged type (see 3.6.70) is a new type which is isomorphic with an old type, but which has a different tag. The tagged type is mainly of use where this Recommendation | International Standard requires the use of types with distinct tags (see 24.5 to 24.6, 26.3 and 28.3). The use of a "TagDefault" of **AUTOMATIC TAGS** in a module allows this to be accomplished without the explicit appearance of tagged type notation in that module.

NOTE – Where a protocol determines that values from several data types may be transmitted at any moment in time, distinct tags may be needed to enable the recipient to correctly decode the value.

30.1 The notation for a tagged type shall be "TaggedType":

```
TaggedType ::=
        Tag Type
    |   Tag IMPLICIT Type
    |   Tag EXPLICIT Type

Tag ::= "[" Class ClassNumber "]"

ClassNumber ::=
        number
    |   DefinedValue

Class ::=
        UNIVERSAL
    |   APPLICATION
    |   PRIVATE
    |   empty
```

30.2 The "valuereference" in "DefinedValue" shall be of type integer, and assigned a non-negative value.

30.3 The new type is isomorphic with the old type, but has a tag with class "Class" and number "ClassNumber", except when "Class" is "empty", in which case the tag is context-specific class and number is "ClassNumber".

30.4 The "Class" shall not be **UNIVERSAL** except for types defined in this Recommendation | International Standard.

NOTE 1 – Use of universal class tags are agreed from time-to-time by ITU-T and ISO.

NOTE 2 – Subclause E.2.12 contains guidance and hints on stylistic use of tag classes.

30.5 All application of tags is either implicit tagging or explicit tagging. Implicit tagging indicates, for those encoding rules which provide the option, that explicit identification of the original tag of the "Type" in the "TaggedType" is not needed during transfer.

NOTE – It can be useful to retain the old tag where this was universal class, and hence unambiguously identifies the old type without knowledge of the ASN.1 definition of the new type. Minimum transfer octets is, however, normally achieved by the use of **IMPLICIT**. An example of an encoding using **IMPLICIT** is given in ITU-T Rec. X.690 | ISO/IEC 8825-1.

30.6 The tagging construction specifies explicit tagging if any of the following holds:

a) the "Tag **EXPLICIT** Type" alternative is used;

b) the "Tag Type" alternative is used and the value of "TagDefault" for the module is either **EXPLICIT TAGS** or is empty;

c) the "Tag Type" alternative is used and the value of "TagDefault" for the module is **IMPLICIT TAGS** or **AUTOMATIC TAGS**, but the type defined by "Type" is an untagged choice type, an untagged open type, or an untagged "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3).

The tagging construction specifies implicit tagging otherwise.

30.7 If the "Class" is "empty", there are no restrictions on the use of "Tag", other than those implied by the requirement for distinct tags in 24.5 to 24.6, 26.3 and 28.3.

30.8 The **IMPLICIT** alternative shall not be used if the type defined by "Type" is an untagged choice type or an untagged open type or an untagged "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3).

30.9 The notation for a value of a "TaggedType" shall be "TaggedValue", or when used as an "XMLValue", "XMLTaggedValue". These productions are:

```
TaggedValue ::= Value

XMLTaggedValue ::= XMLValue
```

where "Value" or "XMLValue" is a notation for a value of the "Type" in the "TaggedType".

NOTE – The "Tag" does not appear in this notation.

31 Notation for the object identifier type

31.1 The object identifier type (see 3.6.48) shall be referenced by the notation "ObjectIdentifierType":

```
ObjectIdentifierType ::=
        OBJECT IDENTIFIER
```

31.2 This type has a tag which is universal class, number 6.

31.3 The value notation for an object identifier shall be "ObjectIdentifierValue", or when used as an "XMLValue", "XMLObjectIdentifierValue". These productions are:

```
ObjectIdentifierValue ::=
        "{" ObjIdComponentsList "}"
    |   "{" DefinedValue ObjIdComponentsList "}"

ObjIdComponentsList ::=
        ObjIdComponents
    |   ObjIdComponents ObjIdComponentsList

ObjIdComponents ::=
        NameForm
    |   NumberForm
    |   NameAndNumberForm
    |   DefinedValue

NameForm ::= identifier

NumberForm ::= number | DefinedValue

NameAndNumberForm ::=
        identifier "(" NumberForm ")"

XMLObjectIdentifierValue ::=
        XMLObjIdComponentList

XMLObjIdComponentList ::=
        XMLObjIdComponent
    |   XMLObjIdComponent & "." & XMLObjIdComponentList

XMLObjIdComponent ::=
        NameForm
    |   XMLNumberForm
    |   XMLNameAndNumberForm

XMLNumberForm ::= number

XMLNameAndNumberForm ::=
        identifier & "(" & XMLNumberForm & ")"
```

31.4 The "valuereference" in "DefinedValue" of "NumberForm" shall be of type integer, and assigned a non-negative value.

31.5 The "valuereference" in "DefinedValue" of "ObjectIdentifierValue" shall be of type object identifier.

31.6 The "DefinedValue" of "ObjIdComponents" shall be of type relative object identifier, and shall identify an ordered set of arcs from some starting node in the object identifier tree to some later node in the object identifier tree. The starting node is identified by the earlier "ObjIdComponents"s, and later "ObjIdComponents"s (if any) identify arcs from the later node. The starting node is required to be neither the root, nor a node immediately beneath the root.

NOTE – A relative object identifier value has to be associated with a specific object identifier value so as to unambiguously identify an object. Object identifier values are required (see 31.10) to have at least two components. This is why there is a restriction on the starting node.

31.7 The "NameForm" shall be used only for those object identifier components whose numeric value and identifier are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, Annexes A to C (see also Annex D of this Recommendation | International Standard), and shall be one of the identifiers specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, Annexes A to C. Where ITU-T Rec. X.660 | ISO/IEC 9834-1 specifies synonymous identifiers, any synonym may be used with the

same semantics. Where the same name is both an identifier specified in ITU-T Rec. X.660 | ISO/IEC 9834-1 and an ASN.1 value reference within the module containing the "NameForm", the name within the object identifier value shall be treated as an ITU-T Rec. X.660 | ISO/IEC 9834-1 identifier.

31.8 The "number" in the "NumberForm" and "XMLNumberForm" shall be the numeric value assigned to the object identifier component.

31.9 The "identifier" in the "NameAndNumberForm" and "XMLNameAndNumberForm" shall be specified when a numeric value is assigned to the object identifier component.

NOTE – The authorities allocating numeric values to object identifier components are identified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

31.10 The semantics associated with an object identifier value are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

NOTE – ITU-T Rec. X.660 | ISO/IEC 9834-1 requires that an object identifier value shall contain at least two arcs.

31.11 The significant part of the object identifier component is the "NameForm" or "NumberForm" or "XMLNumberForm" which it reduces to, and which provides the numeric value for the object identifier component. Except for the arcs specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, Annexes A to C (see also Annex D of this Recommendation | International Standard), the numeric value of the object identifier component is always present in an instance of object identifier value notation.

31.12 Where the "ObjectIdentifierValue" includes a "DefinedValue" for an object identifier value, the list of object identifier components to which it refers is prefixed to the components explicitly present in the value.

NOTE – ITU-T Rec. X.660 | ISO/IEC 9834-1 recommends that whenever an object identifier value is assigned to identify an object, an object descriptor value is also assigned.

EXAMPLES

With identifiers assigned as specified in ITU-T Rec. X.660 | ISO/IEC 9834-1, the values:

```
{ iso standard 8571 pci (1) }
```

and

```
{ 1 0 8571 1 }
```

would each identify an object, **pci**, defined in ISO 8571, as would

```
iso.standard.8571.pci(1)
```

and

```
1.0.8571.1
```

in an "XMLObjectIdentifierValue".

With the following additional definition:

```
ftam OBJECT IDENTIFIER ::= { iso standard 8571 }
```

the following value is equivalent to those above:

```
{ ftam pci(1) }
```

32 Notation for the relative object identifier type

32.1 The relative object identifier type (see 3.6.57) shall be referenced by the notation "RelativeOIDType":

```
RelativeOIDType ::= RELATIVE-OID
```

32.2 This type has a tag which is universal class, number 13.

32.3 The value notation for a relative object identifier shall be "RelativeOIDValue", or when used as "XMLValue", "XMLRelativeOIDValue". These productions are:

```
RelativeOIDValue ::=
        "{" RelativeOIDComponentsList "}"

RelativeOIDComponentsList ::=
        RelativeOIDComponents
    |   RelativeOIDComponents RelativeOIDComponentsList
```

RelativeOIDComponents ::=
NumberForm
| **NameAndNumberForm**
| **DefinedValue**

XMLRelativeOIDValue ::=
XMLRelativeOIDComponentList

XMLRelativeOIDComponentList ::=
XMLRelativeOIDComponent
| **XMLRelativeOIDComponent & "." & XMLRelativeOIDComponentList**

XMLRelativeOIDComponent ::=
XMLNumberForm
| **XMLNameAndNumberForm**

32.4 The productions "NumberForm", "NameAndNumberForm", "XMLNumberForm", "XMLNameAndNumberForm", and their semantics, are defined in subclauses 31.3 to 31.11.

32.5 The "DefinedValue" of "RelativeOIDComponents" shall be of type relative object identifier, and shall identify an ordered set of arcs from some starting node in the object identifier tree to some later node in the object identifier tree. The starting node is identified by the earlier "RelativeOIDComponents"s (if any), and later "RelativeOIDComponents"s (if any) identify arcs from the later nodes.

32.6 The first "RelativeOIDComponents" or "XMLRelativeOIDComponent" identifies one or more arcs from some starting node in the object identifier tree to some later node in the object identifier tree. The starting point can be defined by comments associated with the type definition. If there is no definition of the starting node within comments associated with the type definition, then it needs to be transmitted as an object identifier value in an instance of communication (see E.2.19). The starting node is required to be neither the root, nor a node immediately beneath the root.

NOTE – A relative object identifier value has to be associated with a specific object identifier value so as to unambiguously identify an object. Object identifier values are required (see 31.10) to have at least two components. This is why there is a restriction on the starting node.

EXAMPLE

With the following definitions:

```
thisUniversity  OBJECT IDENTIFIER       ::=
            {iso member-body country(29) universities(56) thisuni(32)}

firstgroup   RELATIVE-OID   ::=  {science-fac(4) maths-dept(3)}
```

or in XML value notation:

```
thisUniversity ::= <OBJECT_IDENTIFIER>1.2.29.56.32</OBJECT_IDENTIFIER>

firstgroup ::= <RELATIVE_OID>4.3</RELATIVE_OID>
```

the relative object identifier:

```
relOID   RELATIVE-OID  ::=  {firstgroup room(4) socket(6)}
```

or in XML value notation:

```
relOID ::= <RELATIVE_OID>4.3.4.6</RELATIVE_OID>
```

can be used instead of the **OBJECT IDENTIFIER** value **{1 2 29 56 32 4 3 4 6}** if the current root (known by the application or transmitted by the application) is **thisUniversity**.

33 Notation for the embedded-pdv type

33.1 The embedded-pdv type (see 3.6.21) shall be referenced by the notation "EmbeddedPDVType":

EmbeddedPDVType ::= EMBEDDED PDV

NOTE – The term "Embedded PDV" means an abstract value from a possibly different abstract syntax (essentially, the value and encoding of a message defined in a separate – and identified – protocol) that is embedded in a message. Historically, it meant "Embedded Presentation Data Value" from its use in the OSI Presentation Layer, but this expansion is not used today, and it should be interpreted as "embedded value".

33.2 This type has a tag which is universal class, number 11.

33.3 The type consists of values representing:

a) an encoding of a single data value that may, but need not, be the value of an ASN.1 type; and

b) identification (separately or together) of:

1) an abstract syntax; and

2) the transfer syntax.

NOTE 1 – The data value may be the value of an ASN.1 type, or may, for example, be the encoding of a still image or a moving picture. The identification consists of either one or two object identifiers, or (in an OSI environment) references an OSI presentation context identifier which specifies the abstract and transfer syntaxes.

NOTE 2 – The identification of the abstract syntax and/or the encoding may also be determined by the application designer as a fixed value, in which case it is not encoded in an instance of communication.

33.4 The embedded-pdv type has an associated type. This associated type is used to support the value and subtype notations of the embedded-pdv type.

33.5 The associated type for value definition and subtyping, assuming an automatic tagging environment, is (with normative comments):

```
SEQUENCE {
      identification                          CHOICE {
            syntaxes                             SEQUENCE {
                  abstract                          OBJECT IDENTIFIER,
                  transfer                          OBJECT IDENTIFIER }
            -- Abstract and transfer syntax object identifiers --,

            syntax                               OBJECT IDENTIFIER
            -- A single object identifier for identification of the abstract
            -- and transfer syntaxes --,

            presentation-context-id              INTEGER
            -- (Applicable only to OSI environments)
            -- The negotiated OSI presentation context identifies the
            -- abstract and transfer syntaxes --,

            context-negotiation                  SEQUENCE {
                  presentation-context-id           INTEGER,
                  transfer-syntax                   OBJECT IDENTIFIER }
            -- (Applicable only to OSI environments)
            -- Context-negotiation in progress, presentation-context-id
            -- identifies only the abstract syntax
            -- so the transfer syntax shall be specified --,

            transfer-syntax                      OBJECT IDENTIFIER
            -- The type of the value (for example, specification that it is
            -- the value of an ASN.1 type)
            -- is fixed by the application designer (and hence known to both
            -- sender and receiver). This
            -- case is provided primarily to support
            -- selective-field-encryption (or other encoding
            -- transformations) of an ASN.1 type --,

            fixed                                NULL
            -- The data value is the value of a fixed ASN.1 type (and hence
            -- known to both sender
            -- and receiver) -- },

      data-value-descriptor                   ObjectDescriptor  OPTIONAL
      -- This provides human-readable identification of the class of the
      -- value --,
      data-value                              OCTET STRING }

( WITH COMPONENTS {
            ... ,
            data-value-descriptor  ABSENT } )
```

NOTE – The embedded-pdv type does not allow the inclusion of a **data-value-descriptor** value. However, the definition of the associated type provided here underlies the commonalities which exist between the embedded-pdv type, the external type and the unrestricted character string type.

33.6 The **presentation-context-id** alternative is only applicable in an OSI environment, when the integer value shall be an OSI presentation context identifier in the OSI defined context set. This alternative shall not be used during OSI context negotiation.

33.7 The **context-negotiation** alternative is only applicable in an OSI environment, and shall only be used during OSI context negotiation. The integer value shall be an OSI presentation context identifier proposed for addition to the OSI defined context set. The object identifier **transfer-syntax** shall identify a proposed transfer syntax for that OSI presentation context which is to be used to encode the value.

33.8 The notation for a value of the embedded-pdv type shall be the value notation for the associated type defined in 33.5, where the value of the **data-value** component of type **OCTET STRING** represents an encoding using the transfer syntax specified in **identification**.

EmbeddedPdvValue ::= SequenceValue *-- value of associated type defined in 33.5*

XMLEmbeddedPDVValue ::= XMLSequenceValue *-- value of associated type defined in 33.5*

EXAMPLE – If a single option is to be enforced, such as use of **syntaxes**, then this can be done by writing:

```
EMBEDDED PDV (WITH COMPONENTS {
                ... ,
                identification  (WITH COMPONENTS {
                      syntaxes  PRESENT } ) } )
```

34 Notation for the external type

34.1 The external type (see 3.6.37) shall be referenced by the notation "ExternalType":

ExternalType ::= EXTERNAL

34.2 This type has a tag which is universal class, number 8.

34.3 The type consists of values representing:

a) an encoding of a single data value that may, but need not, be the value of an ASN.1 type; and

b) identification of:

1) an abstract syntax; and

2) the transfer syntax; and

c) (optionally) an object descriptor which provides a human-readable description of the category of the data value. The optional object descriptor shall not be present unless explicitly permitted by comment associated with use of the "ExternalType" notation.

NOTE – Note 1 in 33.3 also applies to the external type.

34.4 The external type has an associated type. This type is used to give precision to the definition of the abstract values of the external type and is also used to support the value and subtype notations of the external type.

NOTE – Encoding rules may define a different type which is used to derive encodings, or may specify encodings without reference to any associated type. For example, the encoding in BER uses a different sequence type for historical reasons.

34.5 The associated type for value definition and subtyping, assuming an automatic tagging environment, is (with normative comments):

```
SEQUENCE {
      identification                          CHOICE {
            syntaxes                             SEQUENCE {
                  abstract                          OBJECT IDENTIFIER,
                  transfer                          OBJECT IDENTIFIER }
            -- Abstract and transfer syntax object identifiers --,

            syntax                               OBJECT IDENTIFIER
            -- A single object identifier for identification of the abstract
            -- and transfer syntaxes --,

            presentation-context-id              INTEGER
            -- (Applicable only to OSI environments)
            -- The negotiated OSI presentation context identifies the
            -- abstract and transfer syntaxes --,

            context-negotiation                  SEQUENCE {
                  presentation-context-id           INTEGER,
                  transfer-syntax                   OBJECT IDENTIFIER }
```

```
            -- (Applicable only to OSI environments)
            -- Context-negotiation in progress, presentation-context-id
            -- identifies only the abstract syntax
            -- so the transfer syntax shall be specified --,

            transfer-syntax                         OBJECT IDENTIFIER
            -- The type of the value (for example, specification that it is
            -- the value of an ASN.1 type)
            -- is fixed by the application designer (and hence known to both
            -- sender and receiver). This
            -- case is provided primarily to support
            -- selective-field-encryption (or other encoding
            -- transformations) of an ASN.1 type --,

            fixed                                   NULL
            -- The data value is the value of a fixed ASN.1 type (and hence
            -- known to both sender
            -- and receiver) -- },

      data-value-descriptor                 ObjectDescriptor  OPTIONAL
            -- This provides human-readable identification of the class of
            -- the value --,

      data-value                            OCTET STRING }
            ( WITH COMPONENTS {
                  ... ,
                  identification  (WITH COMPONENTS {
                        ... ,
                        syntaxes                            ABSENT,
                        transfer-syntax                     ABSENT,
                        fixed                         ABSENT } ) } )
```

NOTE – For historical reasons, the external type does not allow the **syntaxes**, **transfer-syntax** or **fixed** alternatives of **identification**. Application designers requiring these options should use the embedded-pdv type. The definition of the associated type provided here underlies the commonalities which exist between the external type, the unrestricted character string type and the embedded-pdv type.

34.6 The text of 33.6 and 33.7 also applies to the external type.

34.7 The notation for a value of the external type shall be the value notation for the associated type defined in 34.5, where the value of the **data-value** component of type **OCTET STRING** represents an encoding using the transfer syntax specified in **identification**.

ExternalValue ::= SequenceValue -- *value of associated type defined in 34.5*

XMLExternalValue ::= XMLSequenceValue -- *value of associated type defined in 34.5*

NOTE – For historical reasons, encoding rules are able to transfer embedded values in **EXTERNAL** whose encodings are not an exact multiple of eight bits. Such values cannot be represented in value notation using the above associated type.

35 The character string types

These types consist of strings of characters from some specified character repertoire. It is normal to define a character repertoire and its encoding by use of cells in one or more tables, each cell corresponding to a character in the repertoire. A graphic symbol and a character name are also usually assigned to each cell, although in some repertoires, cells are left empty, or have names but no shapes (examples of cells with names but no shape include control characters such as EOF in ISO/IEC 646 and spacing characters such as THIN-SPACE and EN-SPACE in ISO/IEC 10646-1).

In general, the information associated with a cell denotes a distinct abstract character in the repertoire even if that information is null (no graphic symbol or name is assigned to that cell).

The ASN.1 basic value notation for character string types has three variants (which can be combined), specified formally below:

a) A representation of the characters in the string using assigned graphic symbols, possibly including spacing characters; this is the "cstring" notation.

   NOTE 1 – Such a representation can be ambiguous in a printed representation when the same graphic symbol is used for more than one character in the repertoire.

   NOTE 2 – Such a representation can be ambiguous in a printed representation when spacing characters of different widths are present in the repertoire or the specification is printed with a proportional-spacing font.

b) A listing of the characters in the character string value by giving a series of ASN.1 value references that have been assigned the character; a set of such value references is defined in the module

ASN1-CHARACTER-MODULE in clause 38 for the ISO/IEC 10646-1 character repertoire and for the **IA5String** character repertoire; this form is not available for other character repertoires unless the user assigns to such value references using the value notation described in a) above or c) below.

c) A listing of the characters in the character string value by identifying each abstract character by the position of its cell in the character repertoire table(s); this form is available only for **IA5String**, **UniversalString**, **UTF8String** and **BMPString**.

The ASN.1 XML value notation for character string types uses the "xmlcstring" notation, which includes the ability to use escape sequences for certain special characters, and for specification of characters using decimal or hexadecimal (see 11.15).

36 Notation for character string types

36.1 The notation for referencing a character string type (see 3.6.11) shall be:

```
CharacterStringType ::=
        RestrictedCharacterStringType
    |   UnrestrictedCharacterStringType
```

"RestrictedCharacterStringType" is the notation for a restricted character string type and is defined in clause 37. "UnrestrictedCharacterStringType" is the notation for the unrestricted character string type and is defined in 40.1.

36.2 The tag of each restricted character string type is specified in 37.1. The tag of the unrestricted character string type is specified in 40.2.

36.3 The notation for a character string value shall be:

```
CharacterStringValue ::=
        RestrictedCharacterStringValue
    |   UnrestrictedCharacterStringValue

XMLCharacterStringValue ::=
        XMLRestrictedCharacterStringValue
    |   XMLUnrestrictedCharacterStringValue
```

"RestrictedCharacterStringValue" and "XMLRestrictedCharacterStringValue" are defined in 37.8 and 37.9 respectively. "UnrestrictedCharacterStringValue" and "XMLUnrestrictedCharacterStringValue" are notations for an unrestricted character string value and they are defined in 40.7.

37 Definition of restricted character string types

This clause defines types whose values are restricted to sequences of zero, one or more characters from some specified collection of characters. The notation for referencing a restricted character string type shall be "RestrictedCharacterStringType":

```
RestrictedCharacterStringType ::=
        BMPString
    |   GeneralString
    |   GraphicString
    |   IA5String
    |   ISO646String
    |   NumericString
    |   PrintableString
    |   TeletexString
    |   T61String
    |   UniversalString
    |   UTF8String
    |   VideotexString
    |   VisibleString
```

Each "RestrictedCharacterStringType" alternative is defined by specifying:

a) the tag assigned to the type; and

b) a name (e.g., **NumericString**) by which the type is referenced; and

c) the characters in the collection of characters used in defining the type, by reference to a table listing the character graphics or by reference to a registration number in the ISO International Register of Coded Character Sets (see *ISO International Register of Coded Character Sets to be used with Escape Sequences*), or by reference to ISO/IEC 10646-1.

37.1 Table 6 lists the name by which each restricted character string type is referenced, the number of the universal class tag assigned to the type, the defining registration number or table, or the defining text clause, and, where necessary, identification of a Note relating to the entry in the table. Where a synonymous name is defined in the notation, this is listed in parentheses.

Table 6 – List of restricted character string types

Name for referencing the type	Universal class number	Defining registration number[a], table number, or ITU-T Rec. X.680 | ISO/IEC 8824-1 clause	Notes
UTF8String	12	Subclause 37.16	
NumericString	18	Table 7	(Note 1)
PrintableString	19	Table 8	(Note 1)
TeletexString (**T61String**)	20	6, 87, 102, 103, 106, 107, 126, 144, 150, 153, 156, 164, 165, 168 + SPACE + DELETE	(Note 2)
VideotexString	21	1, 13, 72, 73, 87, 89, 102, 108, 126, 128, 129, 144, 150, 153, 164, 165, 168 + SPACE + DELETE	(Note 3)
IA5String	22	1, 6 + SPACE + DELETE	
GraphicString	25	All G sets + SPACE	
VisibleString (**ISO646String**)	26	6 + SPACE	
GeneralString	27	All G and all C sets + SPACE + DELETE	
UniversalString	28	See 37.6	
BMPString	30	See 37.15	

a) The defining registration numbers are listed in ISO International Register of Coded Character Sets to be used with Escape Sequences.

NOTE 1 – The type-style, size, colour, intensity, or other display characteristics are not significant.

NOTE 2 – Register entries 6 and 156 can be used instead of 102 and 103.

NOTE 3 – The entries corresponding to these registration numbers provide the functionality of CCITT Rec. T.100 and ITU-T Rec. T.101.

37.2 Table 7 lists the characters which can appear in the **NumericString** type and **NumericString** character abstract syntax.

Table 7 – NumericString

Name	Graphic
Digits	**0**, **1**, ... **9**
Space	(space)

37.3 The following object identifier and object descriptor values are assigned to identify and describe the **NumericString** character abstract syntax:

```
{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0) }
```

and

```
"NumericString character abstract syntax"
```

NOTE 1 – This object identifier value can be used in **CHARACTER STRING** values and in other cases where there is a need to carry the identification of the character string type separate from the value.

NOTE 2 – A value of a **NumericString** character abstract syntax may be encoded by:

a) One of the rules given in ISO/IEC 10646-1 for encoding the abstract characters. In this case the character transfer syntax is identified by the object identifier associated with those rules in ISO/IEC 10646-1, Annex N.

b) The ASN.1 encoding rules for the built-in type **NumericString**. In this case the character transfer syntax is identified by the object identifier value **{joint-iso-itu-t asn1(1) basic-encoding(1)}**.

37.4 Table 8 lists the characters which can appear in the **PrintableString** type and **PrintableString** character abstract syntax.

Table 8 – PrintableString

Name	Graphic
Latin capital letters	**A**, **B**, ... **Z**
Latin small letters	**a**, **b**, ... **z**
Digits	**0**, **1**, ... **9**
SPACE	(space)
APOSTROPHE	**'**
LEFT PARENTHESIS	**(**
RIGHT PARENTHESIS	**)**
PLUS SIGN	**+**
COMMA	**,**
HYPHEN-MINUS	**-**
FULL STOP	**.**
SOLIDUS	**/**
COLON	**:**
EQUALS SIGN	**=**
QUESTION MARK	**?**

37.5 The following object identifier and object descriptor values are assigned to identify and describe the **PrintableString** character abstract syntax:

```
{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1) }
```

and

```
"PrintableString character abstract syntax"
```

NOTE 1 – This object identifier value can be used in **CHARACTER STRING** values and in other cases where there is a need to carry the identification of the character string type separate from the value.

NOTE 2 – A value of a **PrintableString** character abstract syntax may be encoded by:

a) One of the rules given in ISO/IEC 10646-1 for encoding the abstract characters. In this case the character transfer syntax is identified by the object identifier associated with those rules in ISO/IEC 10646-1, Annex N.

b) The ASN.1 encoding rules for the built-in type **PrintableString**. In this case the character transfer syntax is identified by the object identifier **{ joint-iso-itu-t asn1(1) basic-encoding(1) }**.

37.6 The characters which can appear in the **UniversalString** type are any of the characters allowed by ISO/IEC 10646-1.

37.7 Use of this type invokes the conformance requirements specified in ISO/IEC 10646-1.

NOTE – Clause 38 defines an ASN.1 module containing a number of subtypes of this type for the "Collections of graphics characters for subsets" defined in ISO/IEC 10646-1, Annex A.

37.8 The "RestrictedCharacterStringValue" notation for the restricted character string types shall be "cstring" (see 11.14), "CharacterStringList", "Quadruple", or "Tuple". "Quadruple" is only capable of defining a character string of length one, and can only be used in value notation for **UniversalString**, **UTF8String** or **BMPString** types. "Tuple" is only capable of defining a character string of length one, and can only be used in value notation for **IA5String** types.

```
RestrictedCharacterStringValue ::=
        cstring
    |   CharacterStringList
    |   Quadruple
    |   Tuple

CharacterStringList ::= "{" CharSyms "}"

CharSyms ::=
        CharsDefn
    |   CharSyms "," CharsDefn
```

CharsDefn ::=
cstring
| Quadruple
| Tuple
| DefinedValue

Quadruple ::= "{" Group "," Plane "," Row "," Cell "}"

Group ::= number

Plane ::= number

Row ::= number

Cell ::= number

Tuple ::= "{" TableColumn "," TableRow "}"

TableColumn ::= number

TableRow ::= number

NOTE 1 – The "cstring" notation can only be used unambiguously on a medium capable of displaying the graphic symbols for the characters which are present in the value. Conversely, if the medium has no such capability, the only means of unambiguously specifying a character string value that uses such graphic symbols is by means of the "CharacterStringList" notation, and only if the type is **UniversalString**, **UTF8String**, **BMPString** or **IA5String**, and the "DefinedValue" alternative of "CharsDefn" is used (see 38.1.2).

NOTE 2 – Clause 38 defines a number of "valuereference"s which denote single characters (strings of size 1) of type **BMPString** (and hence **UniversalString** and **UTF8String**) and **IA5String**.

EXAMPLE – Suppose that one wishes to specify a value of "abcΣdef" for a **UniversalString** where the character "Σ" is not representable on the available medium, this value can also be expressed as:

```
IMPORTS BasicLatin, greekCapitalLetterSigma FROM ASN1-CHARACTER-MODULE
     { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) };

MyAlphabet ::= UniversalString (FROM (BasicLatin | greekCapitalLetterSigma))

mystring MyAlphabet ::= { "abc" , greekCapitalLetterSigma , "def" }
```

NOTE 3 – When specifying the value of a **UniversalString**, **UTF8String** or **BMPString** type, the "cstring" notation should not be used unless ambiguities arising from different graphic characters with similar shapes have been resolved.

EXAMPLE – The following "cstring" notation should not be used because the graphic symbols 'H', 'O', 'P' and 'E' occur in the BASIC LATIN, CYRILLIC and BASIC GREEK alphabets and thus are ambiguous.

```
IMPORTS BasicLatin, Cyrillic, BasicGreek FROM ASN1-CHARACTER-MODULE
     { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) };

MyAlphabet ::= UniversalString (FROM (BasicLatin | Cyrillic | BasicGreek))

mystring MyAlphabet ::= "HOPE"
```

An alternative unambiguous definition of **mystring** would be:

```
mystring  MyAlphabet(BasicLatin) ::= "HOPE"
```

Formally, **mystring** is a value reference to a value of a subset of **MyAlphabet**, but it can, by the value mapping rules of Annex B, be used wherever a value reference is needed to this value within **MyAlphabet**.

37.9 The "XMLRestrictedCharacterStringValue" notation is:

XMLRestrictedCharacterStringValue ::= xmlcstring

37.10 There are characters which cannot be directly represented in "xmlcstring". These shall be represented using the escape sequences specified in 11.15.

NOTE – If the restricted character string value contains characters which are not ISO/IEC 10646-1 characters specified in 11.15.1, these cannot be represented in "xmlcstring", and such values cannot be transferred using XML Encoding Rules (see ITU-T Rec. X.693 | ISO/IEC 8825-4).

37.11 The "DefinedValue" in "CharsDefn" shall be a reference to a value of that type.

37.12 The "number" in the "Plane", "Row" and "Cell" productions shall be less than 256, and in the "Group" production it shall be less than 128.

37.13 The "Group" specifies a group in the coding space of the UCS, the "Plane" specifies a plane within the group, the "Row" specifies a row within the plane, and the "Cell" specifies a cell within the row. The abstract character identified by this notation is the abstract character for the cell specified by the "Group", "Plane", "Row", and "Cell" values. In all cases, the set of permitted characters may be restricted by subtyping.

NOTE – Application designers should consider carefully the conformance implications when using open-ended character string types such as **GeneralString**, **GraphicString**, and **UniversalString** without the application of constraints. Careful text on conformance is also needed for bounded but large character string types such as **TeletexString**.

37.14 The "number" in the "TableColumn" production shall be in the range zero to seven, and the "number" in the "TableRow" production shall be in the range zero to fifteen. The "TableColumn" specifies a column and the "TableRow" specifies a row of a character code table in accordance with Figure 1 of ISO/IEC 2022. This notation is used only for **IA5String** when the code table contains Register Entry 1 in columns 0 and 1 and Register Entry 6 in columns 2 to 7 (see the *ISO International Register of Coded Character Sets to be used with Escape Sequences*).

37.15 **BMPString** is a subtype of **UniversalString** that has its own unique tag and contains only the characters in the Basic Multilingual Plane (those corresponding to the first 64K-2 cells, less cells whose encoding is used to address characters outside the Basic Multilingual Plane) of ISO/IEC 10646-1. It has an associated type defined as:

```
UniversalString (Bmp)
```

where **Bmp** is defined in the ASN.1 module **ASN1-CHARACTER-MODULE** (see clause 38) as the subtype of **UniversalString** corresponding to the "BMP" collection name defined in ISO/IEC 10646-1, Annex A.

NOTE 1 – Since **BMPString** is a built-in type, it is not defined in **ASN1-CHARACTER-MODULE**.

NOTE 2 – The purpose of defining **BMPString** as a built-in type is to enable encoding rules (such as BER) that do not take account of constraints to use 16-bit rather than 32-bit encodings.

NOTE 3 – In the value notation all **BMPString** values are valid **UniversalString** and **UTF8String** values.

37.16 **UTF8String** is synonymous with **UniversalString** at the abstract level and can be used wherever **UniversalString** is used (subject to rules requiring distinct tags) but has a different tag and is a distinct type.

NOTE – The encoding of **UTF8String** used by BER and PER is different from that of **UniversalString**, and for most text will be less verbose.

38 Naming characters and collections defined in ISO/IEC 10646-1

This clause specifies an ASN.1 built-in module which contains the definition of a value reference name for each character from ISO/IEC 10646-1, where each name references a **UniversalString** value of size 1. This module also contains the definition of a type reference name for each collection of characters from ISO/IEC 10646-1, where each name references a subset of the **UniversalString** type.

NOTE – These values are available for use in the value notation of the **UniversalString** type and types derived from it. All of the value and type references defined in the module specified in 38.1 are exported and must be imported by any module that uses them.

38.1 Specification of the ASN.1 Module "ASN1-CHARACTER-MODULE"

The module is not printed here in full. Instead, the means by which it is defined is specified.

38.1.1 The module begins as follows:

```
ASN1-CHARACTER-MODULE { joint-iso-itu-t asn1(1) specification(0) modules(0)
iso10646(0) }
      DEFINITIONS ::= BEGIN
      -- All of the value references and type references defined within this
      -- module are implicitly exported, and are available for import by any module.
      -- ISO/IEC 646 control characters:

      nul   IA5String ::= {0, 0}
      soh   IA5String ::= {0, 1}
      stx   IA5String ::= {0, 2}
      etx   IA5String ::= {0, 3}
      eot   IA5String ::= {0, 4}
      enq   IA5String ::= {0, 5}
      ack   IA5String ::= {0, 6}
      bel   IA5String ::= {0, 7}
      bs    IA5String ::= {0, 8}
      ht    IA5String ::= {0, 9}
      lf    IA5String ::= {0,10}
      vt    IA5String ::= {0,11}
      ff    IA5String ::= {0,12}
      cr    IA5String ::= {0,13}
      so    IA5String ::= {0,14}
      si    IA5String ::= {0,15}
      dle   IA5String ::= {1, 0}
      dc1   IA5String ::= {1, 1}
```

```
dc2   IA5String ::= {1, 2}
dc3   IA5String ::= {1, 3}
dc4   IA5String ::= {1, 4}
nak   IA5String ::= {1, 5}
syn   IA5String ::= {1, 6}
etb   IA5String ::= {1, 7}
can   IA5String ::= {1, 8}
em    IA5String ::= {1, 9}
sub   IA5String ::= {1,10}
esc   IA5String ::= {1,11}
is4   IA5String ::= {1,12}
is3   IA5String ::= {1,13}
is2   IA5String ::= {1,14}
is1   IA5String ::= {1,15}
del   IA5String ::= {7,15}
```

38.1.2 For each entry in each list of character names for the graphic characters (glyphs) shown in clauses 24 and 25 of ISO/IEC 10646-1, the module includes a statement of the form:

```
<namedcharacter> BMPString ::= <tablecell>
-- represents the character <iso10646name>, see ISO/IEC 10646-1
```

where:

a) <*iso10646name*> is the character name derived from one listed in ISO/IEC 10646-1;

b) <*namedcharacter*> is a string obtained by applying to <*iso10646name*> the procedures specified in 38.2;

c) <*tablecell*> is the glyph in the table cell in ISO/IEC 10646-1 corresponding to the list entry.

EXAMPLE

```
latinCapitalLetterA  BMPString  ::=  {0, 0, 0, 65}
-- represents the character LATIN CAPITAL LETTER A, see ISO/IEC 10646-1

greekCapitalLetterSigma BMPString  ::=  {0, 0, 3, 163}
-- represents the character GREEK CAPITAL LETTER SIGMA, see ISO/IEC 10646-1
```

38.1.3 For each name for a collection of graphic characters specified in ISO/IEC 10646-1, Annex A, a statement is included in the module of the form:

```
<namedcollectionstring> ::= BMPString
      (FROM (<alternativelist>))
      -- represents the collection of characters <collectionstring>,
      -- see ISO/IEC 10646-1.
```

where:

a) <*collectionstring*> is the name for the collection of characters assigned in ISO/IEC 10646-1;

b) <*namedcollectionstring*> is formed by applying to <*collectionstring*> the procedures of 38.3;

c) <*alternativelist*> is formed by using the <*namedcharacter*>s as generated in 38.2 for each of the characters specified by ISO/IEC 10646-1.

The resulting type reference, <*namedcollectionstring*>, forms a limited subset. (See the tutorial in Annex F.)

NOTE – A limited subset is a list of characters in a specified subset. Contrast this to a selected subset, which is a collection of characters listed in ISO/IEC 10646-1, Annex A, plus the BASIC LATIN collection.

EXAMPLE (partial)

```
space BMPString              ::= {0, 0, 0, 32}
exclamationMark BMPString    ::= {0, 0, 0, 33}
quotationMark BMPString      ::= {0, 0, 0, 34}
...                          -- and so on
tilde BMPString              ::= {0, 0, 0, 126}

BasicLatin ::= BMPString
      (FROM (space
      | exclamationMark
      | quotationMark
      | ...       -- and so on
      | tilde)
      )
```

```
-- represents the collection of characters BASIC LATIN, see ISO/IEC 10646-1.
-- The ellipsis in this example is used for brevity and means "and so on";
-- you cannot use this in an actual ASN.1 module.
```

38.1.4 ISO/IEC 10646-1 defines three levels of implementation. By default all types defined in **ASN1-CHARACTER-MODULE**, except for **Level1** and **Level2** conform to implementation level 3, since such types have no restriction on use of combining characters. **Level1** indicates that implementation level 1 is required, **Level2** indicates that implementation level 2 is required, and **Level3** indicates that implementation level 3 is required. Thus, the following are defined in **ASN1-CHARACTER-MODULE**:

```
Level1 ::= BMPString (FROM (ALL EXCEPT CombiningCharacters))

Level2 ::= BMPString (FROM (ALL EXCEPT CombiningCharactersType-2))

Level3 ::= BMPString
```

NOTE 1 – **CombiningCharacters** and **CombiningCharactersType-2** are the <namedcollectionstring>s corresponding to "COMBINING CHARACTERS" and "COMBINING CHARACTERS B-2", respectively, defined in ISO/IEC 10646-1, Annex A.

NOTE 2 – **Level1** and **Level2** will be used either following an "IntersectionMark" (see clause 46) or as the only constraint in a "ConstraintSpec". (See E.2.7.1 for an example.)

NOTE 3 – See F.2.5 for more information on this topic.

38.1.5 The module is terminated by the statement:

```
END
```

38.1.6 A user-defined equivalent of the example in 38.1.3 is:

```
BasicLatin ::= BMPString (FROM (space..tilde))
-- represents the collection of characters BASIC LATIN,
-- see ISO/IEC 10646-1.
```

38.2 A <*namedcharacter*> is the string obtained by taking an <*iso10646name*> (see 38.1.2) and applying the following algorithm:

a) each upper-case letter of the <*iso10646name*> is transformed into the corresponding lower-case letter, unless the upper-case letter is preceded by a SPACE, in which case the upper-case letter is kept unchanged;

b) each digit and each HYPHEN-MINUS is kept unchanged;

c) each SPACE is deleted.

NOTE – The above algorithm, taken in conjunction with the character naming guidelines in Annex K of ISO/IEC 10646-1 will always result in unambiguous value notation for every character name listed in ISO/IEC 10646-1.

EXAMPLE – The character from ISO/IEC 10646-1, row 0, cell 60, which is named "LESS-THAN SIGN" and has the graphic representation "<" can be referenced using the "DefinedValue" of:

```
less-thanSign
```

38.3 A <*namedcollectionstring*> is the string obtained by taking <*collectionstring*> and applying the following algorithm:

a) each upper-case letter of the ISO/IEC 10646-1 collection name is transformed into the corresponding lower-case letter, unless the upper-case letter is preceded by a SPACE or it is the first letter of the name, in which case the upper-case letter is kept unchanged;

b) each digit and each HYPHEN-MINUS is kept unchanged;

c) each SPACE is deleted.

EXAMPLES

1) The collection identified in Annex A of ISO/IEC 10646-1 as:

BASIC LATIN

has the ASN.1 type reference:

```
BasicLatin
```

2) A character string type consisting of the characters in the BASIC LATIN collection, together with the BASIC ARABIC collection, could be defined as follows:

```
My-Character-String ::= BMPString (FROM (BasicLatin | BasicArabic) )
```

NOTE – The above construction is necessary because the apparently simpler construction of:

```
My-Character-String ::= BMPString (BasicLatin | BasicArabic)
```

would allow only strings which were entirely BASIC LATIN or BASIC ARABIC but not a mixture of both.

39 Canonical order of characters

39.1 For the purpose of "ValueRange" subtyping and for possible use by encoding rules, a canonical ordering of characters is specified for **UniversalString**, **UTF8String**, **BMPString**, **NumericString**, **PrintableString**, **VisibleString**, and **IA5String**.

39.2 For the purpose of this clause only, a character is in one-to-one correspondence with a cell in a code table, whether that cell has been assigned a character name or shape, and whether it is a control character or printing character, combining or non-combining character.

39.3 The canonical order of an abstract character is defined by the canonical order of its value in the 32-bit representation of ISO/IEC 10646-1, with low numbers appearing first and high numbers appearing last in the canonical order.

39.4 Endpoints of "ValueRanges" within "PermittedAlphabet" notations (or individual characters) can be specified using either the ASN.1 value reference defined in the module **ASN1-CHARACTER-MODULE** or (where the graphic symbol is unambiguous in the context of the specification and the medium used to represent it) by giving the graphic symbol in a "cstring" (**ASN1-CHARACTER-MODULE** is defined in 38.1) , or by use of the "Quadruple" or "Tuple" notation of 37.8.

39.6 For **NumericString**, the canonical ordering, increasing from left to right, is defined (see Table 7 of 37.2) as:

(space) **0 1 2 3 4 5 6 7 8 9**

The entire character set contains precisely 11 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring".

NOTE – This order is the same as the order of the corresponding characters in the BASIC LATIN collection of ISO/IEC 10646-1.

39.7 For **PrintableString**, the canonical ordering, increasing from left to right and top to bottom, is defined (see Table 8 of 37.4) as:

(SPACE) (APOSTROPHE) (LEFT PARENTHESIS) (RIGHT PARENTHESIS) (PLUS SIGN) (COMMA) (HYPHEN-MINUS) (FULL STOP) (SOLIDUS) **0123456789** (COLON) (EQUAL SIGN) (QUESTION MARK) **ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz**

The entire character set contains precisely 74 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring".

NOTE – This order is the same as the order of the corresponding characters in the BASIC LATIN collection of ISO/IEC 10646-1.

39.8 For **VisibleString**, the canonical order of the cells is defined from the ISO/IEC 646 encoding (called ISO 646 ENCODING) as follows:

(ISO 646 ENCODING) - 32

NOTE – That is, the canonical order is the same as the characters in cells 2/0-7/14 of the ISO/IEC 646 code table.

The entire character set contains precisely 95 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring".

39.9 For **IA5String**, the canonical order of the cells is defined from the ISO/IEC 646 encoding as follows:

(ISO 646 ENCODING)

The entire character set contains precisely 128 characters. The endpoint of a "ValueRange" (or individual characters) can be specified using the graphic symbol in a "cstring" or an ISO 646 control character value reference defined in 38.1.1.

40 Definition of unrestricted character string types

This clause defines a type whose values are the values of any character abstract syntax. In an OSI environment, this abstract syntax may be part of the OSI defined context set. Otherwise, it is referenced directly for each instance of use of the unrestricted character string type.

NOTE 1 – A character abstract syntax (and one or more corresponding character transfer syntaxes) can be defined by any organization able to allocate ASN.1 **OBJECT IDENTIFIER**s.

NOTE 2 – Profiles produced by a community of interest will normally determine the character abstract syntaxes and character transfer syntaxes that are to be supported for specific instances or groups of instances of **CHARACTER STRING**. It will be usual in OSI applications to include reference to supported syntaxes in an OSI Protocol Implementation Conformance Statement.

40.1 The unrestricted character string type (see 3.6.76) shall be referenced by the notation "UnrestrictedCharacterStringType":

UnrestrictedCharacterStringType ::= CHARACTER STRING

40.2 This type has a tag which is universal class, number 29.

40.3 The type consists of values representing:

a) a character string value that may, but need not, be the value of an ASN.1 character string type; and

b) identification (separately or together) of:

1) a character abstract syntax; and

2) the character transfer syntax.

40.4 The unrestricted character string type has an associated type. This associated type is used to support its value and subtype notations.

40.5 The associated type for value definition and subtyping, assuming an automatic tagging environment, is (with normative comments):

```
SEQUENCE {
     identification                          CHOICE {
          syntaxes                             SEQUENCE {
               abstract                           OBJECT IDENTIFIER,
               transfer                           OBJECT IDENTIFIER }
               --  Abstract and transfer syntax object identifiers --,

          syntax                                 OBJECT IDENTIFIER
               --  A single object identifier for identification of the
               -- abstract and transfer syntaxes --,
          presentation-context-id                INTEGER
               --  (Applicable only to OSI environments)
               --  The negotiated OSI presentation context identifies the
               -- abstract and transfer syntaxes  --,

          context-negotiation                    SEQUENCE {
               presentation-context-id              INTEGER,
               transfer-syntax                      OBJECT IDENTIFIER }
          -- (Applicable only to OSI environments)
          -- Context-negotiation in progress, presentation-context-id
          -- identifies only the
          -- abstract-syntax, so the transfer syntax shall be specified --,

          transfer-syntax                          OBJECT IDENTIFIER
          --  The type of the value (for example, specification that it is
          -- the value of an ASN.1 type) is fixed by the application
          -- designer (and hence known to both sender and receiver). This
          -- case is provided primarily to support
          -- selective-field-encryption (or other encoding
          --  transformations) of an ASN.1 type  --,

          fixed                                    NULL
          -- The data value is the value of a fixed ASN.1 type (and hence
          -- known to both sender and receiver)  -- },

     data-value-descriptor                  ObjectDescriptor  OPTIONAL
          -- This provides human-readable identification of the class of
          -- the value  --,
     string-value                            OCTET STRING }
     ( WITH COMPONENTS {
          ... ,
          data-value-descriptor  ABSENT } )
```

NOTE – The unrestricted character string type does not allow the inclusion of a **data-value-descriptor** value together with the **identification**. However, the definition of the associated type provided here underlies the commonalities which exist between the embedded-pdv type, the external type and the unrestricted character string type.

40.6 The text of 33.6 and 33.7 also applies to the unrestricted character string type.

40.7 The value notation shall be the value notation for the associated type, where the value of the **string-value** component of type **OCTET STRING** represents an encoding using the transfer syntax specified in **identification**.

UnrestrictedCharacterStringValue ::= SequenceValue -- *value of associated type defined in 40.5*

XMLUnrestrictedCharacterStringValue ::=
XMLSequenceValue -- *value of associated type defined in 40.5*

40.8 An example of the unrestricted character string type is given in E.2.8.

41 Notation for types defined in clauses 42 to 44

41.1 The notation for referencing a type defined in clauses 42 to 44 shall be:

UsefulType ::= typereference

where "typereference" is one of those defined in clauses 42 to 44 using the ASN.1 notation.

41.2 The tag of each "UsefulType" is specified in clauses 42 to 44.

42 Generalized time

42.1 This type shall be referenced by the name:

GeneralizedTime

42.2 The type consists of values representing:

a) a calendar date, as defined in ISO 8601; and

b) a time of day, to any of the precisions defined in ISO 8601, except for the hours value 24 which shall not be used; and

c) the local time differential factor as defined in ISO 8601.

42.3 The type is defined, using ASN.1, as follows:

```
GeneralizedTime ::=    [UNIVERSAL 24] IMPLICIT VisibleString
```

with the values of the **VisibleString** restricted to strings of characters which are either

a) a string representing the calendar date, as specified in ISO 8601, with a four-digit representation of the year, a two-digit representation of the month and a two-digit representation of the day, without use of separators, followed by a string representing the time of day, as specified in ISO 8601, without separators other than decimal comma or decimal period (as provided for in ISO 8601), and with no terminating Z (as provided for in ISO 8601); or

b) the characters in a) above followed by an upper-case letter **Z**; or

c) the characters in a) above followed by a string representing a local time differential, as specified in ISO 8601, without separators.

In case a), the time shall represent the local time. In case b), the time shall represent coordinated universal time. In case c), the part of the string formed as in case a) represents the local time (t_1), and the time differential (t_2) enables coordinated universal time to be determined as follows:

coordinated universal time is $t_1 - t_2$

EXAMPLES

Case a)

"19851106210627.3"
local time 6 minutes, 27.3 seconds after 9 pm on 6 November 1985.

Case b)

"19851106210627.3Z"
coordinated universal time as above.

Case c)

"19851106210627.3-0500"
local time as in example a), with local time 5 hours retarded in relation to coordinated universal time.

42.4 The tag shall be as defined in 42.3.

42.5 The value notation shall be the value notation for the **VisibleString** defined in 42.3.

43 Universal time

43.1 This type shall be referenced by the name:

UTCTime

43.2 The type consists of values representing:

a) calendar date; and

b) time to a precision of one minute or one second; and

c) (optionally) a local time differential from coordinated universal time.

43.3 The type is defined, using ASN.1, as follows:

```
UTCTime ::= [UNIVERSAL 23] IMPLICIT VisibleString
```

with the values of the **VisibleString** restricted to strings of characters which are the juxtaposition of:

a) the six digits YYMMDD where YY is the two low-order digits of the Christian year, MM is the month (counting January as 01), and DD is the day of the month (01 to 31); and

b) either:

1) the four digits hhmm where hh is hour (00 to 23) and mm is minutes (00 to 59); or

2) the six digits hhmmss where hh and mm are as in 1) above, and ss is seconds (00 to 59); and

c) either:

1) the character **Z**; or

2) one of the characters **+** or **-**, followed by hhmm, where hh is hour and mm is minutes.

The alternatives in b) above allow varying precisions in the specification of the time.

In alternative c) 1), the time is coordinated universal time. In alternative c) 2), the time (t_1) specified by a) and b) above is the local time; the time differential (t_2) specified by c) 2) above enables the coordinated universal time to be determined as follows:

Coordinated universal time is $t_1 - t_2$

EXAMPLE 1 – If local time is 7am on 2 January 1982 and coordinated universal time is 12 noon on 2 January 1982, the value of **UTCTime** is either of:

– **"8201021200Z"**; or
– **"8201020700-0500"**.

EXAMPLE 2 – If local time is 7am on 2 January 2001 and coordinated universal time is 12 noon on 2 January 2001, the value of **UTCTime** is either of:

– **"0101021200Z"**; or
– **"0101020700-0500"**.

43.4 The tag shall be as defined in 43.3.

43.5 The value notation shall be the value notation for the **VisibleString** defined in 43.3.

44 The object descriptor type

44.1 This type shall be referenced by the name:

ObjectDescriptor

44.2 The type consists of human-readable text which serves to describe an object. The text is not an unambiguous identification of the object, but identical text for different objects is intended to be uncommon.

NOTE – It is recommended that an authority assigning values of type **OBJECT IDENTIFIER** to an object should also assign values of type **ObjectDescriptor** to that object.

44.3 The type is defined, using ASN.1, as follows:

```
ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString
```

The **GraphicString** contains the text describing the object.

44.4 The tag shall be as defined in 44.3.

44.5 The value notation shall be the value notation for the **GraphicString** defined in 44.3.

45 Constrained types

45.1 The "ConstrainedType" notation allows a constraint to be applied to a (parent) type, either to restrict its set of values to some subtype of the parent or (within a set or sequence type) to specify that component relations apply to values of the parent type and to values of some other component in the same set or sequence value. It also allows an exception identifier to be associated with a constraint.

ConstrainedType ::=
Type Constraint
| **TypeWithConstraint**

In the first alternative, the parent type is "Type", and the constraint is specified by "Constraint" as defined in 45.6. The second alternative is defined in 45.5.

45.2 When the "Constraint" notation follows a set-of or sequence-of type notation, it applies to the "Type" in the (innermost) set-of or sequence-of notation, not to the set-of or sequence-of type.

NOTE – For example, in the following the constraint **(SIZE(1..64))** applies to the **VisibleString**, not the **SEQUENCE OF**:

```
NamesOfMemberNations ::= SEQUENCE OF VisibleString (SIZE(1..64))
```

45.3 When the "Constraint" notation follows the selection type notation, it applies to the choice type, and not to the type of the selected alternative. Such a constraint is ignored (see 29.2).

NOTE – In the following example, the constraint **(WITH COMPONENTS {..., a ABSENT})** applies to the **CHOICE** type **T**, not to the selected **SEQUENCE** type, and has no effect on the values of **V**.

```
T ::= CHOICE {
      a SEQUENCE {
            a INTEGER OPTIONAL,
            b BOOLEAN
      },
      b NULL
}

V ::= a < T (WITH COMPONENTS {..., a ABSENT})
```

45.4 When the "Constraint" notation follows a "TaggedType" notation, the interpretation of the overall notation is the same regardless of whether the "TaggedType" or the "Type" is considered as the parent type.

45.5 As a consequence of the interpretation specified in 45.2, special notation is provided to allow a constraint to be applied to a set-of or sequence-of type. This is "TypeWithConstraint":

TypeWithConstraint ::=
SET Constraint OF Type
| **SET SizeConstraint OF Type**
| **SEQUENCE Constraint OF Type**
| **SEQUENCE SizeConstraint OF Type**
| **SET Constraint OF NamedType**
| **SET SizeConstraint OF NamedType**
| **SEQUENCE Constraint OF NamedType**
| **SEQUENCE SizeConstraint OF NamedType**

In the first and second alternatives the parent type is "**SET OF** Type", while in the third and fourth it is "**SEQUENCE OF** Type". In the fifth and sixth alternatives the parent type is "**SET OF** NamedType", and in the seventh and eighth is "**SEQUENCE OF** NamedType". In the first, third, fifth and seventh alternatives, the constraint is "Constraint" (see 45.6), while in the second, fourth, sixth and eighth it is "SizeConstraint" (see 47.5).

NOTE – Although the "Constraint" alternatives encompass the corresponding "SizeConstraint" alternatives, the "SizeConstraint" alternatives are provided for historical reasons.

45.6 A constraint is specified by the notation "Constraint":

Constraint ::= "(" ConstraintSpec ExceptionSpec ")"

ConstraintSpec ::=
SubtypeConstraint
| **GeneralConstraint**

"ExceptionSpec" is defined in clause 49. Unless it is used in conjunction with an "extension marker" (see clause 48), it shall only be present if the "ConstraintSpec" includes an occurrence of "DummyReference" (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.3) or is a "UserDefinedConstraint" (see ITU-T Rec. X.682 | ISO/IEC 8824-3, clause 9). The "GeneralConstraint" is defined in ITU-T Rec. X.682 | ISO/IEC 8824-3, 8.1.

45.7 The notation "SubtypeConstraint" is the general-purpose "ElementSetSpecs" notation (see clause 46):

SubtypeConstraint ::= ElementSetSpecs

In this context, the elements are values of the parent type (the governor of the element set is the parent type). There shall be at least one element in the set.

46 Element set specification

46.1 In some notations a set of elements of some identified type or information object class (the governor) can be specified. In such cases, the notation "ElementSetSpec" is used:

ElementSetSpecs ::=
RootElementSetSpec
| **RootElementSetSpec "," "..."**
| **RootElementSetSpec "," "..." "," AdditionalElementSetSpec**

RootElementSetSpec ::= ElementSetSpec

AdditionalElementSetSpec ::= ElementSetSpec

ElementSetSpec ::= Unions
| **ALL Exclusions**

Unions ::= Intersections
| **UElems UnionMark Intersections**

UElems ::= Unions

Intersections ::= IntersectionElements
| **IElems IntersectionMark IntersectionElements**

IElems ::= Intersections

IntersectionElements ::= Elements | Elems Exclusions

Elems ::= Elements

Exclusions ::= EXCEPT Elements

UnionMark ::= "|" | UNION

IntersectionMark ::= "^" | INTERSECTION

NOTE 1 – The caret character "^" and the word **INTERSECTION** are synonymous. The character "|" and the word **UNION** are synonymous. It is recommended that, as a stylistic matter, either the characters or the words be used throughout a user Specification. **EXCEPT** can be used with either style.

NOTE 2 – The order of precedence from highest to lowest is: **EXCEPT**, "^", "|". Notice that **ALL EXCEPT** is specified so that it cannot be interspersed with the other constraints without the use of parentheses around "**ALL EXCEPT** xxx".

NOTE 3 – Anywhere that "Elements" occurs, either a constraint without parentheses [e.g., **INTEGER (1..4)**] or a parenthesized subtype constraint [e.g., **INTEGER ((1..4 | 9))**] can appear.

NOTE 4 – Note that two **EXCEPT** operators must have either "|", "^", "**(**" or "**)**" separating them, so **(A EXCEPT B EXCEPT C)** is not permitted. This must be changed to **((A EXCEPT B) EXCEPT C)** or **(A EXCEPT (B EXCEPT C))**.

NOTE 5 – Note that **((A EXCEPT B) EXCEPT C)** is the same as **(A EXCEPT (B | C))**.

NOTE 6 – The elements that are referenced by "ElementSetSpecs" is the union of the elements referenced by the "RootElementSetSpec" and "AdditionalElementSetSpec" (when present).

NOTE 7 – When the elements are information objects (i.e., the governor is an information object class), the notation "ObjectSetElements" as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.3 is used.

46.2 The elements forming the set are:

a) if the first alternative of the "ElementSetSpec" is selected, those specified in the "Unions" [see b)], otherwise all elements of the governor except those specified in the "Elements" notation of the "Exclusions";

b) if the first alternative of "Unions" is selected, then those specified in the "Intersections" [see c)], otherwise those specified at least once either in the "UElems" or "Intersections";

c) if the first alternative of "Intersections" is selected, those specified in the "IntersectionElements" [see d)], otherwise those specified by "IElems" which also are specified by "IntersectionElements";

d) if the first alternative of "IntersectionElements" is selected, those specified in the "Elements", otherwise those specified in the "Elems" except those specified in the "Exclusions".

46.3 The set of values is defined to be extensible if the following conditions hold:

a) for "Elements": there is an extension marker at the outer level;

NOTE – This applies even if all values of the parent are included in the root of the new constrained type.

b) for "Unions": at least one of the "UElems" is extensible;

c) for "Intersections": at least one of the "IElems" is extensible;

d) for "Exclusions": the set of elements preceding **EXCEPT** is extensible.

Otherwise, the set of values is not extensible (see also G.4).

46.4 If the set of values is extensible, the root values can be determined by performing the set arithmetic using only root values of the sets of values involved in the set arithmetic, as specified in 46.2. The extension additions can be determined by performing the set arithmetic using the root values augmented by the extension additions, for each set of values involved in the set arithmetic, and then excluding values that were determined to be root values.

46.5 The "Elements" notation is defined as follows:

```
Elements ::=
        SubtypeElements
    |   ObjectSetElements
    |   "(" ElementSetSpec ")"
```

The elements specified by this notation are:

a) As described in clause 47 below if the "SubtypeElements" alternative is used. This notation shall only be used when the governor is a type, and the actual type involved will further constrain the notational possibilities. In this context, the governor is referred to as the parent type.

b) As described in ITU-T Rec. X.681 | ISO/IEC 8824-2, 12.10, if the "ObjectSetElements" notation is used. This notation shall only be used when the governor is an information object class.

c) Those specified by the "ElementSetSpec" if the third alternative is used.

46.6 When performing set arithmetic within a subtype constraint or a value set when the governing type is extensible, only abstract values that are in the extension root of the governing type are used in the set arithmetic. In this case, all instances of value notation (including value references) used in set arithmetic are required to reference an abstract value of the extension root of the governing type. The end-points of a range constraint are required to reference values that are present in the extension root of the governing type, and the range specification as a whole references all (and only) those values in the range that are within the extension root of the governing type.

46.7 When performing set arithmetic involving information object sets, all information objects are used in the set arithmetic. If any of the information object sets contributing to the set arithmetic are extensible, or if there is an extension marker at the outermost level of an "ElementSetSpecs", the result of the set arithmetic is extensible.

46.8 If a subtype constraint is serially applied to a parent type which is extensible through the application of an extensible constraint, value notation used within it shall not reference values that are not in the extension root of the parent type. The result of the second (serially applied) constraint is defined to be the same as if the constraint had been applied to the parent type without its extension marker and possible extension additions.

EXAMPLE

```
Foo ::= INTEGER ( 1..6, ..., 73..80)
Bar ::= Foo (73) -- illegal
foo Foo ::= 73 -- legal since it is value notation for Foo, not part of a constraint
```

Bar is illegal since 73 is not in the extension root of **Foo**. If 73 had been in the extension root of **Foo**, the example would have been legal, and **Bar** would have contained the single value of 73.

47 Subtype elements

47.1 General

A number of different forms of notation for "SubtypeElements" are provided. They are identified below, and their syntax and semantics are defined in the following subclauses. Table 9 summarizes which notations can be applied to which parent types.

SubtypeElements ::=
SingleValue
| **ContainedSubtype**
| **ValueRange**
| **PermittedAlphabet**
| **SizeConstraint**
| **TypeConstraint**
| **InnerTypeConstraints**
| **PatternConstraint**

Table 9 – Applicability of subtype value sets

Type (or derived from such a type by tagging or subtyping)	Single value	Contained subtype	Value range	Size constraint	Permitted alphabet	Type constraint	Inner subtyping	Pattern constraint
Bit string	Yes	Yes	No	Yes	No	No	No	No
Boolean	Yes	Yes	No	No	No	No	No	No
Choice	Yes	Yes	No	No	No	No	Yes	No
Embedded-pdv	Yes	No	No	No	No	No	Yes	No
Enumerated	Yes	Yes	No	No	No	No	No	No
External	Yes	No	No	No	No	No	Yes	No
Instance-of	Yes	Yes	No	No	No	No	Yes	No
Integer	Yes	Yes	Yes	No	No	No	No	No
Null	Yes	Yes	No	No	No	No	No	No
Object class field type	Yes	Yes	No	No	No	No	No	No
Object descriptor	Yes	Yes	No	Yes	Yes	No	No	No
Object identifier	Yes	Yes	No	No	No	No	No	No
Octet string	Yes	Yes	No	Yes	No	No	No	No
open type	No	No	No	No	No	Yes	No	No
Real	Yes	Yes	Yes	No	No	No	Yes	No
Relative object identifier	Yes[b)]	Yes[b)]	No	No	No	No	No	No
Restricted character string types	Yes	Yes	Yes[a)]	Yes	Yes	No	No	Yes
Sequence	Yes	Yes	No	No	No	No	Yes	No
Sequence-of	Yes	Yes	No	Yes	No	No	Yes	No
Set	Yes	Yes	No	No	No	No	Yes	No
Set-of	Yes	Yes	No	Yes	No	No	Yes	No
Time types	Yes	Yes	No	No	No	No	No	No
Unrestricted character string type	Yes	No	No	Yes	No	No	Yes	No

a) Allowed only within the "PermittedAlphabet" of **`BMPString`**, **`IA5String`**, **`NumericString`**, **`PrintableString`**, **`VisibleString`**, **`UTF8String`** and **`UniversalString`**.

b) The starting node for all relative object identifier types or values in constraints or valuesets shall be the same as the starting node for the governor.

47.2 Single value

47.2.1 The "SingleValue" notation shall be:

SingleValue ::= Value

where "Value" is the value notation for the parent type.

47.2.2 A "SingleValue" specifies the single value of the parent type specified by "Value".

47.3 Contained subtype

47.3.1 The "ContainedSubtype" notation shall be:

ContainedSubtype ::= Includes Type

Includes ::= `INCLUDES` | empty

The "empty" alternative of the "Includes" production shall not be used when "Type" in "ContainedSubtype" is the notation for the null type.

47.3.2 A "ContainedSubtype" specifies all of the values in the root of the parent type that are also in the root of "Type". "Type" is required to be derived from the same built-in type as the parent type.

47.3.3 The set of values referenced by an extensible "Type" used in a contained subtype constraint does not inherit the extension marker from the "Type". Any values in "Type" that are not in the extension root of that type are ignored, and do not contribute to the values of the constrained type.

NOTE – The use of an extensible "Type" does not in itself make the constrained type extensible.

47.4 Value range

47.4.1 The "ValueRange" notation shall be:

ValueRange ::= LowerEndpoint " . . " UpperEndpoint

47.4.2 A "ValueRange" specifies the values in a range of values which are designated by specifying the values of the endpoints of the range. This notation can only be applied to integer types, the "PermittedAlphabet" of certain restricted character string types (**IA5String**, **NumericString**, **PrintableString**, **VisibleString**, **BMPString**, **UniversalString** and **UTF8String** only) and real types. All values specified in the "ValueRange" are required to be in the root of the parent type.

NOTE – For the purpose of subtyping, **PLUS-INFINITY** exceeds all real values and **MINUS-INFINITY** is less than all real values.

47.4.3 Each endpoint of the range is either closed (in which case that endpoint is specified) or open (in which case the endpoint is not specified). When open, the specification of the endpoint includes a less-than symbol ("<"):

LowerEndpoint ::= LowerEndValue | LowerEndValue "<"

UpperEndpoint ::= UpperEndValue | "<" UpperEndValue

47.4.4 An endpoint may also be unspecified, in which case the range extends in that direction as far as the parent type allows:

LowerEndValue ::= Value | MIN

UpperEndValue ::= Value | MAX

NOTE – When a "ValueRange" is used as a "PermittedAlphabet" constraint, "LowerEndValue" and "UpperEndValue" shall be of size 1.

47.5 Size constraint

47.5.1 The "SizeConstraint" notation shall be:

SizeConstraint ::= SIZE Constraint

47.5.2 A "SizeConstraint" can only be applied to bit string types, octet string types, character string types, set-of types or sequence-of types.

47.5.3 The "Constraint" specifies the permitted integer values for the length of the specified values, and takes the form of any constraint which can be applied to the following parent type:

```
INTEGER (0 .. MAX)
```

The "Constraint" shall use the "SubtypeConstraint" alternative of "ConstraintSpec".

47.5.4 The unit of measure depends on the parent type, as follows:

Type	*Unit of measure*
bit string	bit
octet string	octet
character string	character
set-of	component value
sequence-of	component value

NOTE – The count of the number of characters specified in this subclause for determining the size of a character string value shall be clearly distinguished from a count of octets. The count of characters shall be interpreted according to the definition of the collection of characters used in the type, in particular, in relation to references to the standards, tables or registration numbers in a register which can appear in such a definition.

47.6 Type constraint

47.6.1 The "TypeConstraint" notation shall be:

TypeConstraint ::= Type

47.6.2 This notation is only applied to an open type notation and restricts the open type to values of "Type".

47.7 Permitted alphabet

47.7.1 The "PermittedAlphabet" notation shall be:

PermittedAlphabet ::= `FROM` Constraint

47.7.2 A "PermittedAlphabet" specifies all values which can be constructed using a sub-alphabet of the parent string. This notation can only be applied to restricted character string types.

47.7.3 The "Constraint" is any which could be applied to the parent type (see Table 9), except that it shall use the "SubtypeConstraint" alternative of "ConstraintSpec". The sub-alphabet includes precisely those characters which appear in one or more of the values of the parent string type which are allowed by the "Constraint".

47.7.4 If "Constraint" is extensible, then the set of values selected by the permitted alphabet constraint is extensible. The set of values in the root are those permitted by the root of "Constraint", and the extension additions are those values permitted by the root together with the extension-additions of "Constraint", excluding those values already in the root.

47.8 Inner subtyping

47.8.1 The "InnerTypeConstraints" notation shall be:

InnerTypeConstraints ::=
`WITH COMPONENT` SingleTypeConstraint
| `WITH COMPONENTS` MultipleTypeConstraints

47.8.2 An "InnerTypeConstraints" specifies only those values which satisfy a collection of constraints on the presence and/or values of the components of the parent type. A value of the parent type is not specified unless it satisfies all of the constraints expressed or implied (see 47.8.6). This notation can be applied to the set-of, sequence-of, set, sequence and choice types.

NOTE – An "InnerTypeConstraints" applied to a set or sequence type is ignored by the **`COMPONENTS OF`** transformation (see 24.4 and 26.2).

47.8.3 For the types which are defined in terms of a single other (inner) type (set-of and sequence-of), a constraint taking the form of a subtype value specification is provided. The notation for this is "SingleTypeConstraint":

SingleTypeConstraint ::= Constraint

The "Constraint" defines a subtype of the single other (inner) type. A value of the parent type is specified if and only if each inner value belongs to the subtype obtained by applying the "Constraint" to the inner type.

47.8.4 For the types which are defined in terms of multiple other (inner) types (choice, set, and sequence), a number of constraints on these inner types can be provided. The notation for this is "MultipleTypeConstraints":

MultipleTypeConstraints ::=
FullSpecification
| PartialSpecification

FullSpecification ::= "{" TypeConstraints "}"

PartialSpecification ::= "{" "..." "," TypeConstraints "}"

TypeConstraints ::=
NamedConstraint
| NamedConstraint "," TypeConstraints

NamedConstraint ::=
identifier ComponentConstraint

47.8.5 The "TypeConstraints" contains a list of constraints on the component types of the parent type. For a sequence type, the constraints must appear in order. The inner type to which the constraint applies is identified by means of its identifier. For a given component, there shall be at most one "NamedConstraint".

47.8.6 The "MultipleTypeConstraints" comprises either a "FullSpecification" or a "PartialSpecification". When "FullSpecification" is used, there is an implied presence constraint of **ABSENT** on all inner types which can be constrained to be absent (see 47.8.9) and which is not explicitly listed. Where "PartialSpecification" is employed, there are no implied constraints, and any inner type can be omitted from the list.

47.8.7 A particular inner type may be constrained in terms of its presence (in values of the parent type), its value, or both. The notation is "ComponentConstraint":

ComponentConstraint ::= ValueConstraint PresenceConstraint

47.8.8 A constraint on the value of an inner type is expressed by the notation "ValueConstraint":

ValueConstraint ::= Constraint | empty

The constraint is satisfied by a value of the parent type if and only if the inner value belongs to the subtype specified by the "Constraint" applied to the inner type.

47.8.9 A constraint on the presence of an inner type shall be expressed by the notation "PresenceConstraint":

PresenceConstraint ::= PRESENT | ABSENT | OPTIONAL | empty

The meaning of these alternatives, and the situations in which they are permitted are defined in 47.8.9.1 to 47.8.9.3.

47.8.9.1 If the parent type is a sequence or set, a component type marked **OPTIONAL** may be constrained to be **PRESENT** (in which case the constraint is satisfied if and only if the corresponding component value is present) or to be **ABSENT** (in which case the constraint is satisfied if and only if the corresponding component value is absent) or to be **OPTIONAL** (in which case no constraint is placed upon the presence of the corresponding component value).

47.8.9.2 If the parent type is a choice, a component type can be constrained to be **ABSENT** (in which case the constraint is satisfied if and only if the corresponding component type is not used in the value), or **PRESENT** (in which case the constraint is satisfied if and only if the corresponding component type is used in the value); there shall be at most one **PRESENT** keyword in a "MultipleTypeConstraints".

NOTE – See E.4.6 for a clarifying example.

47.8.9.3 The meaning of an empty "PresenceConstraint" depends on whether a "FullSpecification" or a "PartialSpecification" is being employed:

a) in a "FullSpecification", this is equivalent to a constraint of **PRESENT** for a set or sequence component marked **OPTIONAL** and imposes no further constraint otherwise;

b) in a "PartialSpecification", no constraint is imposed.

47.9 Pattern constraint

47.9.1 The "PatternConstraint" notation shall be:

PatternConstraint ::= PATTERN Value

47.9.2 "Value" shall be a "cstring" of type **UniversalString** (or a reference to such a character string) which contains an ASN.1 regular expression as defined in Annex A. The "PatternConstraint" selects those values of the parent type that satisfy the ASN.1 regular expression. The entire value shall satisfy the entire ASN.1 regular expression, i.e., the "PatternConstraint" does not select values whose leading characters match the (entire) ASN.1 regular expression but which contain further trailing characters.

NOTE – "Value" is formally defined as a value of type **UniversalString**, but the sets of values of type **UniversalString** and **UTF8String** are the same (see 37.16). Thus a totally equivalent definition could have been to say that "Value" is a value of type **UTF8String**.

48 The extension marker

NOTE – Like the constraint notation in general, the extension marker has no effect on some encoding rules of ASN.1, such as the Basic Encoding Rules, but does on others, such as the Packed Encoding Rules. Its effect on encodings defined using ECN is determined by the ECN specification.

48.1 The extension marker, ellipsis, is an indication that extension additions are expected. It makes no statement as to how such additions should be handled other than that they shall not be treated as an error during the decoding process.

48.2 The joint use of the extension marker and an exception identifier (see clause 49) is both an indication that extension additions are expected and also provides a means for identifying the action to be taken by the application if there is a constraint violation. It is recommended that this notation be used in those situations where store and forward or

any other form of relaying is in use, so as to indicate (for example) that any unrecognized extension additions are to be returned to the application for possible re-encoding and relaying.

48.3 The result of set arithmetic involving subtype constraints, value sets or information object sets that are extensible is specified in clause 46.

48.4 If a type defined with an extensible constraint is referenced in a "ContainedSubtype", the newly defined type does not inherit the extension marker or any of its extension additions (see 47.3.3). The newly defined type can be made extensible by including an extension marker at the outermost level in its "ElementSetSpecs" (see also 46.3). For example:

```
A ::= INTEGER (0..10, ..., 12) -- A is extensible.
B ::= INTEGER (A)              -- B is inextensible and is constrained to 0-10.
C ::= INTEGER (A, …)           -- C is extensible and is constrained to 0-10.
```

48.5 If a type defined with an extensible constraint is further constrained with an "ElementSetSpecs", the resulting type does not inherit the extension marker nor any extension additions that may be present in the former constraint (see 46.8). For example:

```
A ::= INTEGER (0..10, ...)   -- A is extensible.
B ::= A (2..5)               -- B is inextensible.
C ::= A                      -- C is extensible.
```

48.6 Components of a set, sequence or choice type that are constrained to be absent shall not be present, regardless of whether the set, sequence or choice type is an extensible type.

NOTE – Inner type constraints have no effect on extensibility.

For example:

```
A ::= SEQUENCE   {
a     INTEGER
b     BOOLEAN OPTIONAL,
...
}

B ::= A (WITH COMPONENTS {b ABSENT})
                             -- B is extensible, but 'b' shall not be
                             -- present in any of its values.
```

48.7 Where this Recommendation | International Standard requires distinct tags (see 24.5 to 24.6, 26.3 and 28.3), the following transformation shall conceptually be applied before performing the check for tag uniqueness:

48.7.1 A new element or alternative (called the conceptually-added element, see 48.7.2) is conceptually added at the extension insertion point if:

a) there are no extension markers but extensibility is implied in the module heading, and then an extension marker is added and the new element is added as the first addition after that extension marker; or

b) there is a single extension marker in a **CHOICE** or **SEQUENCE** or **SET**, and then the new element is added at the end of the **CHOICE** or **SEQUENCE** or **SET** immediately prior to the closing brace; or

c) there are two extension markers in a **CHOICE** or **SEQUENCE** or **SET**, and then the new element is added immediately before the second extension marker.

48.7.2 This conceptually-added element is solely for the purposes of checking legality through the application of rules requiring distinct tags (see 24.5 to 24.6, 26.3 and 28.3). It is conceptually-added *after* the application of automatic tagging (if applicable) and the expansion of **COMPONENTS OF**.

48.7.3 The conceptually-added element is defined to have a tag which is distinct from the tag of all normal ASN.1 types, but which matches the tag of all such conceptually-added elements and matches the indeterminate tag of the open type, as specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.2, Note 2.

NOTE – The rules concerning tag uniqueness relating to the conceptually added element and to the open type, together with the rules requiring distinct tags (see 24.5 to 24.6, 26.3 and 28.3) are necessary and sufficient to ensure that:

a) any unknown extension addition can be unambiguously attributed to a single insertion point when a BER encoding is decoded; and

b) unknown extension additions can never be confused with **OPTIONAL** elements.

In PER the above rules are sufficient but are not necessary to ensure these properties. They are nonetheless imposed as rules of ASN.1 to ensure independence of the notation from encoding rules.

48.7.4 If, with these conceptually-added elements, the rules requiring distinct types are violated, then the specification has made illegal use of the extensibility notation.

NOTE – The purpose of the above rules is to make precise restrictions arising from the use of insertion points (particularly those which are not at the end of **SEQUENCE**s or **SET**s or **CHOICE**s). The restrictions are designed to ensure that in BER, DER and CER it is possible to attribute an unknown element received by a version 1 system unambiguously to a specific insertion point. This would be important if the exception handling of such added elements was different for different insertion points.

48.8 Examples

48.8.1 Example 1

```
A ::= SET {
      a     A,
      b     CHOICE {
            c     C,
            d     D,
            ...
      }
}
```

is legal, for there is no ambiguity as any added material must be part of **b**.

48.8.2 Example 2

```
A ::= SET {
      a     A,
      b     CHOICE {
            c     C,
            d     D,
            ...
      },
      ... ,
      d     D
}
```

is illegal, for added material may be part of **b**, or may be at the outer level of **A**, and a version 1 system cannot tell which.

48.8.3 Example 3

```
A ::= SET {
      a     A,
      b     CHOICE {
            c     C,
            ...
      } ,
      d     CHOICE {
            e     E,
            ...
      }
}
```

is also illegal, for added material may be part of **b** or **d**.

48.8.4 More complex examples can be constructed, with extensible choices inside extensible choices, or extensible choices within elements of a sequence marked **OPTIONAL** or **DEFAULT**, but the above rules are necessary and sufficient to ensure that an element not present in version 1 can be unambiguously attributed by a version 1 system to precisely one insertion point.

49 The exception identifier

49.1 In a complex ASN.1 specification, there are a number of places where it is specifically recognized that decoders have to handle material that is not completely specified in it. These cases arise in particular from use of a constraint that is defined using a parameter of the abstract syntax (see ITU-T Rec. X.683 | ISO/IEC 8824-4, clause 10).

49.2 In such cases, the application designer needs to identify the actions to be taken when some implementation-dependent constraint is violated. The exception identifier is provided as an unambiguous means of referring to parts of an ASN.1 specification in order to indicate the actions to be taken. The identifier consists of a "**!**" character, followed by an optional ASN.1 type and a value of that type. In the absence of the type, **INTEGER** is assumed as the type of the value.

49.3 If an "ExceptionSpec" is present, it indicates that there is text in the body of the standard saying how to handle the constraint violation associated with the "**!**" character. If it is absent, then the implementors will either need to identify text that describes the action that they are to take, or will take implementation-dependent action when a constraint violation occurs.

49.4 The "ExceptionSpec" notation is defined as follows:

ExceptionSpec ::= "!" ExceptionIdentification | empty

ExceptionIdentification ::=
SignedNumber
| DefinedValue
| Type ":" Value

The first two alternatives denote exception identifiers of type integer. The third alternative denotes an exception identifier ("Value") of arbitrary type ("Type").

49.5 Where a type is constrained by multiple constraints, more than one of which has an exception identifier, the exception identifier in the outermost constraint shall be regarded as the exception identifier for that type.

49.6 Where an exception marker is present on types that are used in set arithmetic, the exception identifier is ignored and is not inherited by the type being constrained as a result of the set arithmetic.

Annex A

ASN.1 regular expressions

(This annex forms an integral part of this Recommendation | International Standard)

A.1 Definition

A.1.1 An ASN.1 regular expression is a pattern that describes a set of strings whose format conforms to this pattern. A regular expression is itself a string; it is constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. The smallest expressions, which are (usually) made of one or two characters, are placeholders that stand for a set of characters.

The regular expressions presented here are very similar to those of scripting languages like Perl and to those of XML Schema, where some other examples of use can be found.

A.1.2 Most characters, including all letters and digits, are regular expressions that match themselves.

EXAMPLE

The regular expression **"fred"** matches only the string **"fred"**.

A.1.3 Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

A.2 Metacharacters

A.2.1 A metacharacter sequence (or metacharacter) is a set of one or more contiguous characters that have a special meaning in the context of a regular expression. The following list contains all of the metacharacter sequences. Their meaning is explained in the following clauses.

[ ]		Match any character in the set where ranges are denoted by "**-**". A "**^**" after the opening bracket complements the set which follows it.
{g,p,r,c}		Quadruple which identifies a character of ISO/IEC 10646-1 (see 37.8)
\N{name}		Match the named character (or any character of the named character set) 38.1
.		Match any character (unless it is one of the newline characters defined in 11.1.6)
\d		Match any digit (equivalent to "**[0-9]**")
\w		Match any alphanumeric character (equivalent to "**[a-zA-Z0-9]**")
\t		Match the HORIZONTAL TABULATION (9) character (see 11.1.6)
\n		Match any one of the newline characters defined in 11.1.6
\r		Match the CARRIAGE RETURN (13) character (see 11.1.6)
\s		Match any one of the white-space characters (see 11.1.6)
\b		atch a word boundary
****	(prefix)	Quote the next metacharacter and cause it to be interpreted literally
****		Match the REVERSE SOLIDUS (92) character "****"
""		Match the QUOTATION MARK (34) character (**"**)
|	(infix)	Alternative between two expressions
( )		Grouping of the enclosed expression
*****	(postfix)	Match the previous expression zero, one or several times
+	(postfix)	Match the previous expression one or several times
?	(postfix)	Match the previous expression once or not at all
#n	(postfix)	Match the previous expression exactly n times (where n is a single digit)
#(n)	(postfix)	Match the previous expression exactly n times
#(n,)	(postfix)	Match the previous expression at least n times
#(n,m)	(postfix)	Match the previous expression at least n but not more than m times
#(,m)	(postfix)	Match the previous expression not more than m times

NOTE 1 – The characters CIRCUMFLEX ACCENT (94) "**^**" and HYPHEN-MINUS (45) "**-**" are additional metacharacters in certain positions of the string defined in A.2.2.

NOTE 2 – The value in round brackets after a character name in this annex is the decimal value of the character in ISO/IEC 10646-1.

NOTE 3 – This notation does not provide the metacharacters "**^**" and "**$**" to match the beginning and the end of a string respectively. Hence a string shall match a regular expression in its entirety except if the latter includes "**.***" at its beginning, at its end or at both sides.

NOTE 4 – The following metacharacter sequences cannot contain white-space (see 11.1.6) unless the white-space appears immediately prior to or following a newline:

{g,p,r,c**}**
\N{name**}**
#n
#(n**)**
#(n,**)**
#(n,m**)**
#(,m**)**

If a regular expression contains a newline, any spacing characters that appear immediately prior to or following the newline have no significance and match nothing (see 11.14.1).

A.2.2 A list of characters enclosed by "**[**" and "**]**" matches any single character in that list. If the first character of the list is the caret "**^**", then it matches any character which is not in the list. A range of characters may be specified by giving the first and last characters, separated by a hyphen (according to the order relation defined in 39.3). All metacharacter sequences, except "**]**" and "\", lose their special meaning inside a list. To include a literal CIRCUMFLEX ACCENT (94) "**^**", place it anywhere except in the first position or precede it with a backslash. To include a literal HYPHEN-MINUS (45) "**-**", place it first or last in the list, or precede it with a backslash. To include a literal CLOSING SQUARE BRACKET (93) "**]**", place it first. If the first character in the list is the caret "**^**", then the characters "**-**" and "**]**" also match themselves when they immediately follow that caret. The metacharacter sequences defined in A.2.3, A.2.4, A.2.6 and A.2.7 can be used between the square brackets where they keep their meaning.

EXAMPLES

The regular expression **"[0123456789]"**, or equivalently **"[0-9]"**, matches any single digit.

The regular expression **"[^0]"** matches any single character except **0**.

The regular expression **"[\d^.-]"** matches any single digit, a caret, a hyphen or a period.

A.2.3 To avoid any ambiguity between two ISO/IEC 10646-1 characters which have the same glyph, two notations are provided. A notation of the form **"{**group**,**plane**,**row**,**cell**}"** references a (single) character according to the "Quadruple" production defined in 37.8.

A.2.4 A notation of the form **"\N{**valuereference**}"** matches the referenced character if "valuereference" is a reference to a restricted character string value of size 1 (see clause 37) which is defined or imported in the current module. A notation of the form **"\N{**typereference**}"** matches any character of the referenced character set if "typereference" is a reference to a subtype of a "RestrictedCharacterStringType" which is defined in the current module, or is one of the "RestrictedCharacterStringType"s defined in clause 37.

NOTE – In particular, "valuereference" or "typereference" can be one of the references defined in the module **ASN1-CHARACTER-MODULE** (see 38.1) and imported into the current module (see 37.8).

EXAMPLES

The regular expression **"\N{greekCapitalLetterSigma}"** matches GREEK CAPITAL LETTER SIGMA.

The regular expression **"\N{BasicLatin}"** matches any (single) character of the BASIC LATIN character set.

"[\N{BasicLatin}\N{Cyrillic}\N{BasicGreek}]+", or equivalently **"(\N{BasicLatin} | \N{Cyrillic} | \N{BasicGreek})+"**, are regular expressions that match a string made of any (non null) number of characters from the three character sets specified.

A.2.5 The period "**.**" matches any single character, unless it is one of the newline characters defined in 11.1.6.

A.2.6 The symbol "**\d**" is a synonym for "**[0-9]**", i.e., it matches any single digit. The symbol "**\t**" matches the HORIZONTAL TABULATION (9) character. The symbol "**\w**" is a synonym for "**[a-zA-Z0-9]**", i.e., it matches any single (lower-case or upper-case) character or any single digit.

EXAMPLE

The regular expression **"\w+(\s\w+)*\."** matches a sentence made of at least one (alphanumeric) word. The words are separated by one white-space character as defined in 11.1.6. There is no white-space character before the ending period.

A.2.7 The symbol "**\r**" matches the CARRIAGE RETURN (13) character. The symbol "**\n**" matches any one of the newline characters defined in 11.1.6. The symbol "**\s**" matches any one of the white-space characters defined in 11.1.6. The symbol "**\b**" matches the empty string at the beginning or at the end of a word.

EXAMPLE

The regular expression **".*\bfred\b.*"** matches any string which includes the word **"fred"** (this word is not only a series of four characters; it is delimited). Hence it matches strings like **"fred"** or **"I am fred the first"**, but not strings like **"My name is freddy"** or **"I am afred I don't know how to spell 'afraid'!"**.

A.2.8 A character that normally functions as a metacharacter can be interpreted literally by prefixing it with a "\". If the regular expression includes a QUOTATION MARK (34), this character shall be represented by a pair of QUOTATION MARK characters.

EXAMPLES

The regular expression **"\."** matches the (single) string **"."**, but not any string of any single character.

The regular expression **""""** matches the string which contains a single QUOTATION MARK.

The regular expression **"\)"** matches the string **")"**.

The regular expression **"\a"** matches the character **"a"**.

NOTE – The fourth example shows that the backslash is allowed to precede characters that are not metacharacters, but this use is deprecated (because other metacharacters could be allowed in future versions of this Recommendation | International Standard).

A.2.9 Two or more regular expressions may be joined by the infix operator "**|**". The resulting regular expression matches any string matching either subexpression.

A.2.10 A regular expression may be followed by a repetition operator. If the operator is "**?**", the preceding item is optional and matched at most once. If the operator is "*****", the preceding item will be matched zero or more times. If the operator is "**+**", the preceding item will be matched one or more times. If the operator is of the form "**#(**n**)**", the preceding item is matched exactly n times; in this particular case, the parentheses can be omitted if n consists of one digit. If it is of the form "**#(n,)**", the item is matched n or more times. If it is of the form "**#(,**m**)**", the item is optional and is matched at most m times. Finally, if it is of the form "**#(n,m)**", the item is matched at least n times, but not more than m times.

NOTE – It is illegal to use the metacharacters "*****", "**+**", "**?**" or "**#**" as the first character of a regular expression. It is also illegal to use the metacharacters "**#**" or "**|**" as the last character of a regular expression.

EXAMPLES

A phone number like **"555-1212"** is matched by the regular expression **"\d#3-\d#4"**, or equivalently **"\d#(3)-\d#(4)"**.

A price in dollars like **"$12345.90"** is matched by the regular expression **"$\d#(1,)(\.\d#(1,2))?"**. Note that parentheses are requested after the "**#**" symbol when it is followed by a range.

A social security number like **"123-45-5678"** is matched by the regular expression **"\d#3-?\d#2-?\d#4"**.

A.2.11 Repetition (see A.2.10) takes precedence over concatenation (see A.1.3), which in turn takes precedence over alternation (see A.2.9). A whole subexpression may be enclosed in parentheses to override these precedence rules.

A.2.12 When a regular expression contains subexpressions in parentheses, each (non-quoted) opening parenthesis is successively assigned a distinct (strictly positive) integer from the left to the right of the regular expression. Each subexpression can then be referenced inside a comment with a notation like "*\1*", "*\2*" which uses the associated integer. The empty subexpression "**()**" is not permitted.

EXAMPLE

```
"((\d#2)(\d#2)(\d#4))" -- \1 is a date in which \2 is the month, \3 the day
                       -- and \4 the year.
```

NOTE – There is a requirement for formal reference to subexpressions of a regular expression for many purposes. One such instance is the need to write text to document the regular expression within the ASN.1 module. This is a notation which can be used to provide such references. This notation is not used elsewhere in this Recommendation | International Standard.

Annex B

Rules for type and value Compatibility

(This annex forms an integral part of this Recommendation | International Standard)

This annex is expected to be mainly of use to tool builders to ensure that they interpret the language identically. It is present in order to clearly specify what is legal ASN.1 and what is not, and to be able to specify the precise value that any value reference name identifies, and the precise set of values that any type or value set reference name identifies. It is not intended to provide a definition of valid transformations of ASN.1 notations for any purpose other than those stated above.

B.1 The need for the value mapping concept (tutorial introduction)

B.1.1 Consider the following ASN.1 definitions:

```
A ::= INTEGER

B ::= [1] INTEGER

C ::= [2] INTEGER (0..6,...)

D ::= [2] INTEGER (0..6,...,7)

E ::= INTEGER (7..20)

F ::= INTEGER {red(0), white(1), blue(2), green(3), purple(4)}

a A ::= 3

b B ::= 4

c C ::= 5

d D ::= 6

e E ::= 7

f F ::= green
```

B.1.2 It is clear that the value references **a**, **b**, **c**, **d**, **e**, and **f** can be used in value notation governed by **A**, **B**, **C**, **D**, **E**, and **F**, respectively. For example:

```
W ::= SEQUENCE {w1  A  DEFAULT a}
```

and:

```
x A ::= a
```

and:

```
Y ::= A(1..a)
```

are all valid given the definitions in B.1.1. If, however, **A** above were replaced by **B**, or **C**, or **D**, or **E**, or **F**, would the resulting statements be illegal? Similarly, if the value reference **a** above were replaced in each of these cases by **b**, or **c**, or **d**, or **e**, or **f**, are the resulting statements legal?

B.1.3 A more sophisticated question would be to consider in each case replacement of the type reference by the explicit text to the right of its assignment. Consider for example:

```
f INTEGER {red(0), white(1), blue(2), green(3), purple(4)} ::= green

W ::= SEQUENCE {
      w1    INTEGER {red(0), white(1), blue(2), green(3),  purple(4)}
      DEFAULT f}

x INTEGER {red(0), white(1), blue(2), green(3),  purple(4)} ::= f

Y ::= INTEGER {red(0), white(1), blue(2), green(3),  purple(4)}(1..f)
```

Would the above be legal ASN.1?

B.1.4 Some of the above examples are cases which, even if legal (as most of them are – see later text), users would be ill-advised to write similar text, as they are at the least obscure and at worst confusing. However, there are frequent uses of a value reference to a value of some type (not necessarily just an **INTEGER** type) as the default value for that type

with tagging or subtyping applied in the governor. The *value mapping* concept is introduced in order to provide a clear and precise means of determining which constructs such as the above are legal.

B.1.5 Again, consider:

```
C ::= [2] INTEGER (0..6,...)

E ::= INTEGER (7..20)

F ::= INTEGER {red(0), white(1), blue(2), green(3), purple(4)}
```

In each case a new type is being created. For **F** we can clearly identify a 1-1 correspondence between the values in it and the values in the universal type **INTEGER**. In the case of **C** and **E**, we can clearly identify a 1-1 correspondence between the values in them and a subset of the values in the universal type **INTEGER**. We call this relationship a *value mapping* between values in the two types. Moreover, because values in **F**, **C**, and **E** all have (1-1) mappings to values of **INTEGER**, we can use these mappings to provide mappings between the values of **F**, **C**, and **E** themselves. This is illustrated for **F** and **C** in Figure B.1.

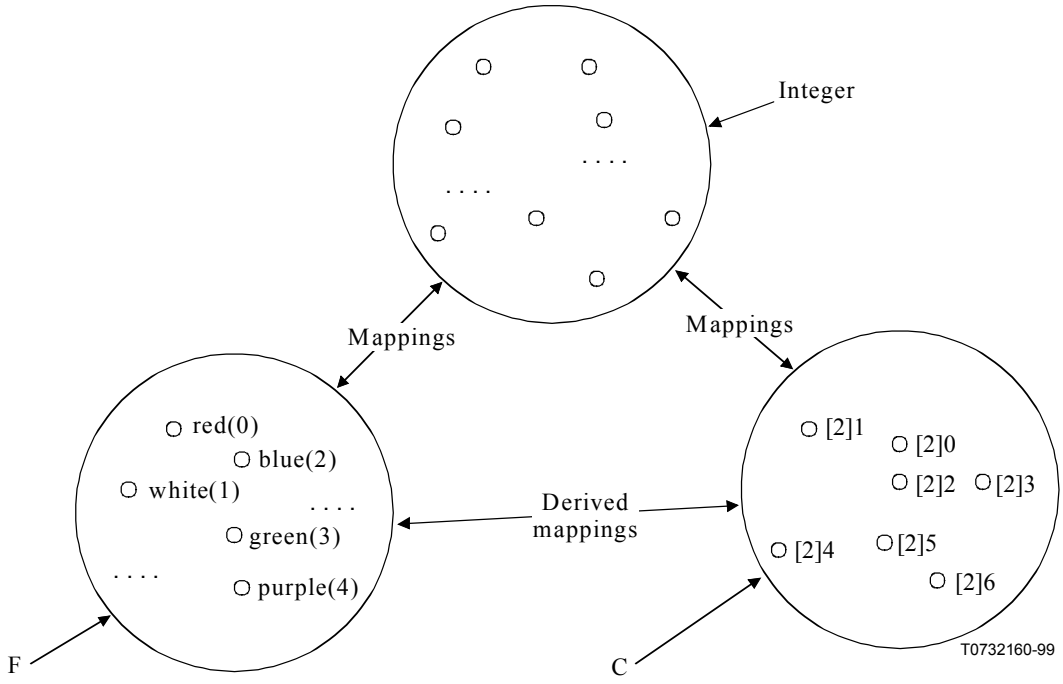

Figure B.1

B.1.6 Now when we have a value reference such as:

```
c C ::= 5
```

to a value in **C** which is required in some context to identify a value in **F**, then, provided a value mapping exists between that value in **C** and a (single) value in **F**, we can (and do) define **c** to be a legal reference to the value in **F**. This is illustrated in Figure B.2, where the value reference **c** is used to identify a value in **F**, and can be used in place of a direct reference **f1** where we would otherwise have to define:

```
f1 F ::= 5
```

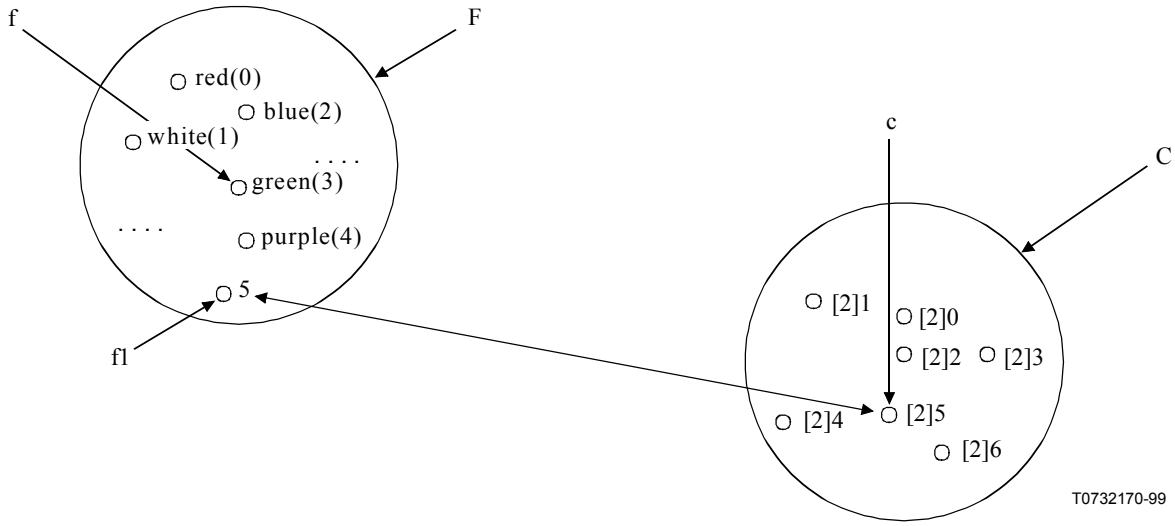

Figure B.2

B.1.7 It should be noted that in some cases there will be values in one type (7 to 20 in **A** of B.1.1 for example) that have value mappings to values in another type (7 to 20 in **E** of B.1.1 for example), but other values (21 upwards of **A**) that have no such mapping. A reference to such values in **A** would not provide a valid reference to a value in **E**. (In this example, the whole of **E** has a value mapping to a subset of **A**. In the general case, there may be a subset of values in both types that have mappings, with other values in both types that are unmapped.)

B.1.8 In the body of the ASN.1 standards, normal English text is used to specify legality in the above and similar cases. Subclause B.6 gives the precise requirements for legality and should be referenced whenever there is doubt about a complex construction.

NOTE – The fact that value mappings are defined to exist between two occurrences of the "Type" construct permits the use of value references established using one "Type" construct to identify values in another "Type" construct which is sufficiently similar. It allows dummy and actual parameters to be typed using two textually separate "Type" constructs without violating the rules for compatibility of dummy and actual parameters. It also allows fields of information object classes to be specified using one "Type" construct and the corresponding value in an information object to be specified using a distinct "Type" construct which is sufficiently similar. (These examples are not intended to be exhaustive.) It is, however, recommended that advantage be taken of this freedom only for simple cases such as **SEQUENCE OF INTEGER**, or **CHOICE {int INTEGER, id OBJECT IDENTIFIER}**, and not for more complex "Type" constructs.

B.2 Value mappings

B.2.1 The underlying model is of types, as non-overlapping containers, that contain values, with every occurrence of the ASN.1 "Type" construct defining a distinct new type (see Figures B.1 and B.2). This annex specifies when *value mappings* exist between such types, enabling a reference to a value in one type to be used where a reference to a value in some other type is needed.

EXAMPLE: Consider:

```
X ::= INTEGER

Y ::= INTEGER
```

X and **Y** are type reference names (pointers) to two distinct types, but value mappings exist between these types, so any value reference to a value of **X** can be used when governed by **Y** (for example, following **DEFAULT**).

B.2.2 In the set of all possible ASN.1 values, a value mapping relates a pair of values. The whole set of value mappings is a mathematical relation. This relation possesses the following properties: it is reflexive (each ASN.1 value is related to itself), it is symmetric (if a value mapping is defined to exist from a value **x1** to a value **x2**, then there automatically exists a value mapping from **x2** to **x1**), and it is transitive (if there is a value mapping from a value **x1** to **x2**, and a value mapping from **x2** to **x3**, then there automatically exists a value mapping from **x1** to **x3**).

B.2.3 Furthermore, given any two types **X1** and **X2**, seen as sets of values, the set of value mappings from values in **X1** to values in **X2** is a one-to-one relation, that is, for all values **x1** in **X1**, and **x2** in **X2**, if there is a value mapping from **x1** to **x2**, then:

a) there is no value mapping from **x1** to another value in **X2** different from **x2**; and

b) there is no value mapping from any value in **X1** (other than **x1**) to **x2**.

B.2.4 Where a value mapping exists between a value **x1** and a value **x2**, a value reference to either one can automatically be used to reference the other if so required by some governing type.

NOTE – The fact that value mappings are defined to exist between values in some "Type" constructs is solely for the purpose of providing flexibility in the use of the ASN.1 notation. The existence of such mappings carries no implications whatsoever that the two types carry the same application semantics, but it is recommended that ASN.1 constructs which would be illegal without value mappings are used only if the corresponding types do indeed carry the same application semantics. Note that value mappings will frequently exist in any large specification between two types that are identical ASN.1 constructs, but which carry totally different application semantics, and where the existence of these value mappings is never used in determining the legality of the total specification.

B.3 Identical type definitions

B.3.1 The concept of identical type definitions is used to enable value mappings to be defined between two instances of "Type" which are either identical or sufficiently similar that one would normally expect their use to be interchangeable. In order to give precision to the meaning of "sufficiently similar", this subclause specifies a series of transformations which are applied to each of the instances of "Type" to produce a *normal form* for those instances of "Type". The two instances of "Type" are defined to be identical type definitions if, and only if, their normal forms are identical ordered lists of the same lexical items (see clause 11).

B.3.2 Each occurrence of "Type" in an ASN.1 specification is an ordered list of the lexical items defined in clause 11. The normal form is obtained by applying the transformations defined in B.3.2.1 to B.3.2.6 in that order.

B.3.2.1 All the comments (see 11.6) are removed.

B.3.2.2 The following transformations are not recursive and hence need only to be applied once, in any order:

a) For a type defined by a "ValueSetTypeAssignment", its definition is replaced by a "TypeAssignment" using the same "Type" and a subtype constraint which is the contents of the "ValueSet" as specified in 15.6.

b) For each integer type: the "NamedNumberList" (see 18.1), if any, is reordered so that the "identifier"s are in alphabetical order ("a" first, "z" last).

c) For each enumerated type: numbers are added, as specified in 19.3, to any "EnumerationItem" (see 19.1) that is an "identifier" (without a number); then the "RootEnumeration" is reordered so that the "identifiers" are in alphabetical order ("a" first, "z" last).

d) For each bitstring type: the "NamedBitList" (see 21.1), if any, is reordered so that the "identifiers" are in alphabetical order ("a" first, "z" last).

e) For each object identifier value: each "ObjIdComponents" is transformed into its corresponding "NumberForm" in accordance with the semantics of clause 31 (see the example in 31.12).

f) For each relative object identifier value (see 32.3): each "RelativeOIDComponents" is transformed into its corresponding "NumberForm" in accordance with the semantics of clause 32.

g) For sequence types (see clause 24) and set types (see clause 26): any extension of the form "ExtensionAndException", "ExtensionAdditions", is cut and pasted to the end of the "ComponentTypeLists"; "OptionalExtensionMarker", if present, is removed.

If "TagDefault" is **IMPLICIT TAGS**, the keyword **IMPLICIT** is added to all instances of "Tag" (see clause 30) unless either:

- it is already present; or
- the reserved word **EXPLICIT** is present; or
- the type being tagged is a **CHOICE** type or;
- it is an open type.

If "TagDefault" is **AUTOMATIC TAGS**, the decision on whether to apply automatic tagging is taken according to 24.2 (the automatic tagging will be performed later on).

NOTE – Subclauses 24.3 and 26.2 specify that the presence of a "Tag" in a "ComponentType" which was inserted as a result of the replacement of "Components of Type" does not in itself prevent the automatic tagging transformation.

If "ExtensionDefault" is **EXTENSIBILITY IMPLIED**, an ellipsis ("**...**") is added after the "ComponentTypeLists" if it is not present.

h) For choice type (see clause 28): "RootAlternativeTypeList" is reordered so that the identifiers of the "NameType"s are in alphabetical order ("a" first, "z" last). "OptionalExtensionMarker", if present, is

removed. If "TagDefault" is **IMPLICIT TAGS**, the keyword **IMPLICIT** is added to all instances of "Tags" (see clause 30) unless either:

- it is already present; or
- the reserved word **EXPLICIT** is present; or
- the type being tagged is a **CHOICE** type; or
- it is an open type.

If "TagDefault" is **AUTOMATIC TAGS**, the decision on whether to apply automatic tagging is taken according to 28.5 (the automatic tagging will be performed later on). If "ExtensionDefault" is **EXTENSIBILITY IMPLIED**, an ellipsis ("**...**") is added after the "AlternativeTypeLists" if it is not present.

B.3.2.3 The following transformations shall be applied recursively in the specified order, until a fix-point is reached:

a) For each object identifier value (see 31.3): if the value definition begins with a "DefinedValue", the "DefinedValue" is replaced by its definition.

b) For each relative object identifier value (see 32.3): if the value definition contains "DefinedValue"s, the "DefinedValue"s are replaced by their definition.

c) For sequence types and set types: all instances of "**COMPONENTS OF** Type" (see clause 24) are transformed according to clauses 24 and 26.

d) For sequence, set and choice types: if it has earlier been decided to tag automatically (see B.3.2.2 g) and h)), the automatic tagging is applied according to clauses 24, 26 and 28.

e) For selection type: the construction is replaced by the selected alternative according to clause 29.

f) All type references are replaced by their definitions according to the following rules:

- If the replacing type is a reference to the type being transformed, the type reference is replaced by a special item that matches no other item than itself.
- If the replacing type is a sequence-of type or a set-of type, the constraints following the replaced type, if any, are moved in front of the keyword **OF**.
- If the replaced type is a parameterized type or a parameterized value set (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2), every "DummyReference" is replaced by the corresponding "ActualParameter".

g) All value references are replaced by their definitions; if the replaced value is a parameterized value (see ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2), every "DummyReference" is replaced by the corresponding "ActualParameter".

NOTE – Before replacing any value reference, the procedures of this annex shall be applied to ensure that the value reference identifies, through value mappings or directly, a value in its governing type.

B.3.2.4 For set type: the "RootComponentTypeList" is reordered so that the "ComponentType"s are in alphabetical order ("a" first, "z" last).

B.3.2.5 The following transformations shall be applied to value definitions:

a) If an integer value is defined with an identifier, that identifier is replaced by the associated number.

b) If a bitstring value is defined using identifiers, it is replaced by the corresponding "bstring" with all trailing zero bits removed.

c) All white-space immediately before and after each newline (including the newline) in a "cstring" is removed.

d) All white-space in "bstring" and "hstring" is removed.

e) Each real value defined with base 2 is normalized so that the mantissa is odd, and each real value defined with base 10 is normalized so that the last digit of the mantissa is not 0.

f) Each **GeneralizedTime** and **UTCTime** value is replaced by a string which conforms to the rules used when encoding in DER and CER (see ITU-T Rec. X.690 | ISO/IEC 8825-1, 11.7 and 11.8).

g) After applying c), each **UTF8String**, **NumericString**, **PrintableString**, **IA5String**, **VisibleString** (**ISO646String**), **BMPString** and **UniversalString** value is replaced by the equivalent value of type **UniversalString** written using the "Quadruple" notation (see clause 37.8).

B.3.2.6 Any occurrence of "realnumber" shall be transformed to a "base" 10 associated "SequenceValue". Any occurrence of the "RealValue" associated with "SequenceValue" shall be transformed to the associated "SequenceValue" of the same "base", such that the last digit of the mantissa is not zero.

B.3.3 If two instances of "Type", when transformed to their normal form, are identical lists of lexical items (see clause 11), then the two instances of "Type" are defined to be identical type definitions with the following exception: if an "objectclassreference" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.1), an "objectreference" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.2) or an "objectsetreference" (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.3) appears within the normalized form of the "Type", then the two types are not defined to be identical type definitions, and value mappings (see B.4 below) will not exist between them.

NOTE – This exception was inserted to avoid the need to provide transformation rules to normal form for elements of syntax concerned with information object class, information object, and information object set notation. Similarly, specification for the normalization of all value notation and of set arithmetic notation has not been included at this time. Should there prove to be a requirement for such specification, this could be provided in a future version of this Recommendation | International Standard. The concept of identical type definitions and of value mappings was introduced to ensure that simple ASN.1 constructs could be used either by using reference names or by copying text. It was felt unnecessary to provide this functionality for more complex instances of "Type" that included information object classes, etc.

B.4 Specification of value mappings

B.4.1 If two occurrences of "Type" are identical type definitions under the rules of B.3, then value mappings exist between every value of one type and the corresponding value of the other type.

B.4.2 For a type, **X1**, created from any type, **X2**, by tagging (see clause 30), value mappings are defined to exist between all the members of **X1** and the corresponding members of **X2**.

NOTE – Whilst value mappings are defined to exist between the values of **X1** and **X2** in B.4.2 above, and between the values of **X3** and **X4** in B.4.3, if such types are embedded in otherwise identical but distinct type definitions (such as **SEQUENCE** or **CHOICE** type definitions), the resulting type definitions (the **SEQUENCE** or **CHOICE** types) will not be identical type definitions, and there will be no value mappings between them.

B.4.3 For a type, **X3**, created by selecting values from any governing type, **X4**, by the element set construct or by subtyping, value mappings are defined to exist between the members of the new type and those members of the governing type that were selected by the element set or subtyping construct. The presence or absence of an extension marker has no effect on this rule.

B.4.4 Additional value mappings are specified in B.5 between some of the character string types.

B.4.5 A value mapping is defined to exist between all the values of any type defined as an integer type with named values and any integer type defined without named values, or with different named values, or with different names for named values, or both.

NOTE – The existence of the value mapping does not affect any scope rule requirements on the use of the names of named values. They can only be used in a scope governed by the type in which they are defined, or by a typereference name to that type.

B.4.6 A value mapping is defined to exist between all the values of any type defined as a bit string type with named bits and any bit string type defined without named bits, or with different named bits, or with different names for named bits, or both.

NOTE – The existence of the value mapping does not affect any scope rule requirements on the use of the names of named bits. They can only be used in a scope governed by the type in which they are defined, or by a typereference name to that type.

B.5 Additional value mappings defined for the character string types

B.5.1 There are two groups of restricted character string types, group A (see B.5.2) and group B (see B.5.3). Value mappings are defined to exist between all types in group A, and value references to values of these types can be used when governed by one of the other types. For the types in group B, value mappings never exist between these different types, nor between any type in group A and any type in group B.

B.5.2 Group A consists of:

UTF8String
NumericString
PrintableString
IA5String
VisibleString (**ISO646String**)
UniversalString
BMPString

B.5.3 Group B consists of:

TeletexString (**T61String**)
VideotexString
GraphicString
GeneralString

B.5.4 The value mappings in group A are specified by mapping the character string values of each type to **UniversalString**, then using the transitivity property of value mappings. To map values from one of the group A types to **UniversalString**, the string is replaced by a **UniversalString** of the same length with each character mapped as specified below.

B.5.5 Formally, the set of abstract values in **UTF8String** is the same set of abstract values that occur in **UniversalString** but with a different tag (see 37.16), and each abstract value in **UTF8String** is defined to map to the corresponding abstract value in **UniversalString**.

B.5.6 The glyphs (printed character shapes) for characters used to form the types **NumericString** and **PrintableString** have recognizable and unambiguous mappings to a subset of the glyphs assigned to the first 128 characters of ISO/IEC 10646-1. The mapping for these types is defined using this mapping of glyphs.

B.5.7 **IA5String** and **VisibleString** are mapped into **UniversalString** by mapping each character into the **UniversalString** character that has the identical (32-bit) value in the BER encoding of **UniversalString** as the (8-bit) value of the BER encoding of **IA5String** and **VisibleString**.

B.5.8 **BMPString** is formally a subset of **UniversalString**, and corresponding abstract values have value mappings.

B.6 Specific type and value compatibility requirements

This subclause uses the value mapping concept to provide precise text for the legality of certain ASN.1 constructs.

B.6.1 Any "Value" occurrence, *x-notation*, with a governing type, **Y**, identifies the value, *y-val*, in the governing type **Y** that has a value mapping to the value *x-val* specified by *x-notation*. It is a requirement that such a value exists.

For example, consider the occurrence of **x** in the last line of the following:

```
X   ::= [0] INTEGER (0..30)

x X ::= 29

Y   ::= [1] INTEGER (25..35)

Z1  ::= Y (x | 30)
```

These ASN.1 constructs are legal, and in the last assignment the *x-notation* **x** is referencing the *x-val* 29 in **X** and, through value mapping, identifies the *y-val* 29 in **Y**. The *x-notation* 30 is referencing the *y-val* 30 in **Y**, and **Z1** is the set of values 29 and 30. On the other hand, the assignment:

```
Z2 ::= Y (x | 20)
```

is illegal because there is no *y-val* to which the *x-notation* 20 can refer.

B.6.2 Any "Type" occurrence, *t-notation*, that has a governing type, **V**, identifies the complete set of values in the root of the governing type **V** that have value mappings to any of the values in the root of the "Type" *t-notation*. This set is required to contain at least one value.

For example, consider the occurrence of **W** in the last line of the following:

```
V  ::= [0] INTEGER (0..30)

W  ::= [1] INTEGER (25..35)

Y  ::= [2] INTEGER (31..35)

Z1 ::= V (W | 24)
```

W contributes values 25-30 to the set arithmetic resulting in **Z1** having the values 24-30. On the other hand, the assignment:

```
Z2 ::= V (Y | 24)
```

is illegal because there are no values in **Y** which map to a value in **V**.

B.6.3 The type of any value supplied as an actual parameter is required to have a value mapping from that value to one of the values in the type governing the dummy parameter, and it is a value of that governing type which is identified.

B.6.4 If a "Type" is supplied as an actual parameter for a dummy parameter which is a value set dummy parameter, then all values of that "Type" are required to have value mappings to values in the governor of the value set dummy parameter. The actual parameter selects the total set of values in the governor which have mappings to the "Type".

B.6.5 In specifying the type, **A**, of a dummy parameter that is a value or a value set parameter, it is an illegal specification unless for all values of **A**, and for every instance of use of **A** on the right-hand side of the assignment, that value of **A** can legally be applied in place of the dummy parameter.

B.7 Examples

B.7.1 This subclause provides examples to illustrate B.3 and B.4.

B.7.2 Example 1

```
X ::= SEQUENCE                  X1 ::=  SEQUENCE
      {name VisibleString,          {name VisibleString,
      age  INTEGER}                 -- comment --
                                          age  INTEGER}

X2 ::= [8] SEQUENCE             X3 ::=  SEQUENCE
      {name VisibleString,          {name VisibleString,
      age  INTEGER}                 age  AgeType}

                                AgeType ::= INTEGER
```

X, **X1**, **X2**, and **X3** are all identical type definitions. Differences of white-space and comment are not visible, nor does the use of the **AgeType** type reference in **X3** affect the type definition. Note, however, that if any of the identifiers for the elements of the sequence were changed, the types would cease to be identical definitions, and there would be no value mappings between them.

B.7.3 Example 2

```
B ::= SET                       B1 ::= SET
      {name VisibleString,              {age INTEGER,
      age  INTEGER}                     name VisibleString}
```

are identical type definitions provided neither is in a module with **AUTOMATIC TAGS** in the module header, otherwise they are not identical type definitions, and value mappings will not exist between them. Similar examples can be written using **CHOICE** and **ENUMERATED** (using the "identifier" form of "EnumerationItem").

B.7.4 Example 3

```
C ::= SET                       C1 ::= SET
      {name [0]VisibleString,           {name VisibleString,
      age INTEGER}                      age INTEGER (1..64)}
```

are not identical type definitions, nor are either of them identical type definitions to either of **B** or **B1**, and there are no value mappings between any of the values of **C** and **C1**, nor between either of them and either of **B** or **B1**.

B.7.5 Example 4

```
x  INTEGER { y (2) } ::= 3

z  INTEGER ::= x
```

is legal, and assigns the value 3 to **z** through the value mapping defined in B.4.5.

B.7.6 Example 5

```
b1 BIT STRING ::= '101'B

b2 BIT STRING {version1(0), version2(1), version3(2)} ::= b1
```

is legal, and assigns the value **{version1, version3}** to **b2**.

B.7.7 Example 6

With the definitions of B.1.1, **SEQUENCE** elements of the form:

```
X DEFAULT y
```

are legal, where **X** is any of **A**, **B**, **C**, **D**, **E**, or **F**, or any of the text to the right of the type assignments to these names, and **y** is any of **a**, **b**, **c**, **d**, **e**, or **f**, with the following exceptions: **E DEFAULT y** is illegal for all of **a**, **b**, **c**, **d**, **f**, and **C DEFAULT e** is illegal, because in these cases there are no value mappings available from the defaulting value reference into the type being defaulted.

Annex C

Assigned object identifier values

(This annex forms an integral part of this Recommendation | International Standard)

This annex records object identifier and object descriptor values assigned in the ASN.1 series of Recommendations | International Standards, and provides an ASN.1 module for use in referencing those object identifier values.

C.1 Object identifiers assigned in this Recommendation | International Standard

The following values are assigned in this Recommendation | International Standard:

Subclause 37.3

Object Identifier Value:
{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0) }

Object Descriptor Value: **"NumericString ASN.1 type"**

Subclause37.5

Object Identifier Value:
{ joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1) }

Object Descriptor Value: **"PrintableString ASN.1 type"**

Subclause 38.1

Object Identifier Value:
{ joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) }

Object Descriptor Value: **"ASN.1 Character Module"**

Subclause C.2

Object Identifier Value:
{ joint-iso-itu-t asn1(1) specification(0) modules(0) object-identifiers(1) }

Object Descriptor Value: **"ASN.1 Object Identifier Module"**

C.2 Object identifiers in the ASN.1 and encoding rules standards

This clause specifies an ASN.1 module which contains the definition of a value reference name for each object identifier value defined in the ASN.1 standards (ITU-T Rec. X.680 | ISO/IEC 8824-1 to ITU-T Rec. X.693 | ISO/IEC 8825-4).

NOTE – These values are available for use in the value notation of the OBJECT IDENTIFIER type and types derived from it. All of the value references defined in the module specified in this clause are exported and have to be imported by any module that wishes to use them.

```
ASN1-Object-Identifier-Module { joint-iso-itu-t asn1(1) specification(0) modules(0)
object-identifiers(1) }
      DEFINITIONS ::= BEGIN

      -- NumericString ASN.1 type (see 37.3) --
      numericString OBJECT IDENTIFIER ::=
            { joint-iso-itu-t asn1(1) specification(0) characterStrings(1)
      numericString(0) }

      -- PrintableString ASN.1 type (see 37.5) --
      printableString OBJECT IDENTIFIER ::=
            { joint-iso-itu-t asn1(1) specification(0) characterStrings(1)
      printableString(1) }

      -- ASN.1 Character Module (see 38.1) --
      asn1CharacterModule OBJECT IDENTIFIER ::=
            { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) }

      -- ASN.1 Object Identifier Module (this module) --
      asn1ObjectIdentifierModule OBJECT IDENTIFIER ::=
            { joint-iso-itu-t asn1(1) specification(0) modules(0)
      object-identifiers(1) }
```

```
        -- BER encoding of a single ASN.1 type --
        ber OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) basic-encoding(1) }

        -- CER encoding of a single ASN.1 type --
        cer OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) ber-derived(2) canonical-encoding(0) }

        -- DER encoding of a single ASN.1 type --
        der OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1) }

        -- PER encoding of a single ASN.1 type (basic aligned) --
        perBasicAligned OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) packed-encoding(3) basic(0) aligned(0) }

        -- PER encoding of a single ASN.1 type (basic unaligned) --
        perBasicUnaligned OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) packed-encoding(3) basic(0) unaligned(1) }

        -- PER encoding of a single ASN.1 type (canonical aligned) --
        perCanonicalAligned OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) packed-encoding(3) canonical(1) aligned(0) }

        -- PER encoding of a single ASN.1 type (canonical unaligned) --
        perCanonicalUnaligned OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) packed-encoding(3) canonical(1) unaligned(1) }

        -- XER encoding of a single ASN.1 type (basic) --
        xerBasic OBJECT IDENTIFIER ::=
              {joint-iso-itu-t asn1(1) xml-encoding(5) basic(0) }

        -- XER encoding of a single ASN.1 type (canonical) --
        xerCanonical OBJECT IDENTIFIER ::=
              {joint-iso-itu-t asn1(1) xml-encoding(5) canonical(1) }

    END -- ASN1-Object-Identifier-Module --
```

Annex D

Assignment of object identifier component values

(This annex does not form an integral part of this Recommendation | International Standard)

This annex describes the top-level arcs of the registration tree for object identifiers. No explanation is given on how new arcs are added, nor on the rules that registration authorities should follow. These are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

D.1 Root assignment of object identifier component values

D.1.1 Three arcs are specified from the root node. The assignment of values and identifiers, and the authority for assignment of subsequent component values, are as follows:

Value	*Identifier*	*Authority for subsequent assignments*
0	**itu-t**	ITU-T (See D.2)
1	**iso**	ISO (See D.3)
2	**joint-iso-itu-t**	See D.4

D.1.2 The identifiers **itu-t**, **iso** and **joint-iso-itu-t**, assigned above, may each be used as a "NameForm" (see 31.3).

D.1.3 The identifiers **ccitt** and **joint-iso-ccitt** are synonyms for **itu-t** and **joint-iso-itu-t**, respectively, and thus may appear in object identifier values.

D.2 ITU-T assignment of object identifier component values

D.2.1 Five arcs are specified from the node identified by **itu-t**. The assignment of values and identifiers is:

Value	*Identifier*	*Authority for subsequent assignments*
0	**recommendation**	See D.2.2
1	**question**	See D.2.3
2	**administration**	See D.2.4
3	**network-operator**	See D.2.5
4	**identified-organization**	See D.2.6

These identifiers may be used as a "NameForm" (see 31.3).

D.2.2 The arcs below **recommendation** have the value **1** to **26** with assigned identifiers of **a** to **z**. Arcs below these have the numbers of ITU-T (and CCITT) Recommendations in the series identified by the letter. Arcs below this are determined as necessary by the ITU-T (and CCITT) Recommendations. The identifiers **a** to **z** may be used as a "NameForm".

D.2.3 The arcs below **question** have values corresponding to ITU-T Study Groups, qualified by the study period. The value is computed by the formula:

study group number + (period * 32)

where "period" has the value 0 for 1984-1988, 1 for 1988-1992, etc., and the multiplier is 32 decimal.

The arcs below each study group have the values corresponding to the questions assigned to that study group. Arcs below this are determined as necessary by the group (e.g., working party or special rapporteur group) assigned to study the question.

D.2.4 The arcs below **administration** have the values of X.121 DCCs. Arcs below this are determined as necessary by the Administration of the country identified by the X.121 DCC.

D.2.5 The arcs below **network-operator** have the value of X.121 DNICs. Arcs below this are determined as necessary by the Administration or ROA identified by the DNIC.

D.2.6 The arcs below **identified-organization** are assigned values by the ITU Telecommunication Standardization Bureau (TSB). Arcs below this are determined as necessary by the identified organizations.

NOTE – Organizations which may find this arc useful include:
- recognized operating agencies not operating a public data network;
- scientific and industrial organizations;
- regional standards organizations; and
- multi-national organizations.

D.3 ISO assignment of object identifier component values

D.3.1 Three arcs are specified from the node identified **`iso(1)`**. The assignment of values and identifiers is:

Value	*Identifier*	*Authority for subsequent assignments*
`0`	**`standard`**	See D.3.2
`2`	**`member-body`**	See D.3.3
`3`	**`identified-organization`**	See D.3.4

These identifiers may be used as a "NameForm".

NOTE – The use of arc **`registration-authority(1)`** has been withdrawn.

D.3.2 The arcs below **`standard`** shall each have the value of the number of an International Standard. Where the International Standard is multi-part, there shall be an additional arc for the part number, unless this is specifically excluded in the text of the International Standard. Further arcs shall have values as defined in that International Standard.

D.3.3 The arcs immediately below **`member-body`** shall have values of a three digit numeric country code, as specified in ISO 3166, that identifies the ISO National Body in that country. The "NameForm" of object identifier component is not permitted with these identifiers.

D.3.4 The arcs immediately below **`identified-organization`** shall have values of an International Code Designator (ICD) allocated by the Registration Authority for ISO/IEC 6523 that identify an issuing organization specifically registered by that authority as allocating object identifier components. The arcs immediately below the ICD shall have values of an "organization code" allocated by the issuing organization in accordance with ISO/IEC 6523.

D.4 Joint assignment of object identifier component values

D.4.1 The arcs below **`joint-iso-itu-t`** have values which are assigned and agreed from time to time by a Registration Authority established by ISO/IEC and ITU-T to identify areas of joint ISO/IEC | ITU-T standardization activity, in accordance with ITU-T Rec. X.662 | ISO/IEC 9834-3.

Annex E

Examples and hints

(This annex does not form an integral part of this Recommendation | International Standard)

This annex contains examples of the use of ASN.1 in the description of (hypothetical) data structures. It also contains hints, or guidelines, for the use of the various features of ASN.1. Unless otherwise stated, an environment of **AUTOMATIC TAGS** is assumed.

E.1 Example of a personnel record

The use of ASN.1 is illustrated by means of a simple, hypothetical personnel record.

E.1.1 Informal description of Personnel Record

The structure of the personnel record and its value for a particular individual are shown below.

Name:	John P Smith
Title:	Director
Employee Number:	51
Date of Hire:	17 September 1971
Name of Spouse:	Mary T Smith
Number of Children:	2
Child Information	
Name:	Ralph T Smith
Date of Birth	11 November 1957
Child Information	
Name:	Susan B Jones
Date of Birth	17 July 1959

E.1.2 ASN.1 description of the record structure

The structure of every personnel record is formally described below using the standard notation for data types.

```
PersonnelRecord ::= [APPLICATION 0] SET
{     name              Name,
      title             VisibleString,
      number            EmployeeNumber,
      dateOfHire        Date,
      nameOfSpouse      Name,
      children          SEQUENCE OF ChildInformation DEFAULT {}
}

ChildInformation ::= SET
{     name              Name,
      dateOfBirth       Date
}

Name ::= [APPLICATION 1] SEQUENCE
{     givenName         VisibleString,
      initial           VisibleString,
      familyName        VisibleString
}
```

```
EmployeeNumber ::= [APPLICATION 2] INTEGER

Date ::= [APPLICATION 3] VisibleString  --  YYYY MMDD
```

This example illustrates an aspect of the parsing of the ASN.1 syntax. The syntactic construct **DEFAULT** can only be applied to a component of a **SEQUENCE** or a **SET**, it cannot be applied to an element of a **SEQUENCE OF**. Thus, the **DEFAULT { }** in **PersonnelRecord** applies to **children**, not to **ChildInformation**.

E.1.3 ASN.1 description of a record value

The value of John Smith's personnel record is formally described below using the standard notation for data values.

```
{     name             {givenName "John", initial "P", familyName "Smith"},
      title            "Director",
      number           51,
      dateOfHire       "19710917",
      nameOfSpouse     {givenName "Mary", initial "T", familyName "Smith"},
      children
      {  {name  {givenName "Ralph", initial "T", familyName "Smith"} ,
          dateOfBirth "19571111"},
         {name  {givenName "Susan", initial "B", familyName "Jones"} ,
          dateOfBirth "19590717" }
      }
}
```

or in XML value notation:

```
person ::=
      <PersonnelRecord>
         <name>
            <givenName>John</givenName>
            <initial>P</initial>
            <familyName>Smith</familyName>
         </name>
         <title>Director</title>
         <number>51</number>
         <dateOfHire>19710917</dateOfHire>
         <nameOfSpouse>
            <givenName>Mary</givenName>
            <initial>T</initial>
            <familyName>Smith</familyName>
         </nameOfSpouse>
         <children>
            <ChildInformation>
               <name>
                  <givenName>Ralph</givenName>
                  <initial>T</initial>
                  <familyName>Smith</familyName>
               </name>
               <dateOfBirth>19571111</dateOfBirth>
            </ChildInformation>
            <ChildInformation>
               <name>
                  <givenName>Susan</givenName>
                  <initial>B</initial>
                  <familyName>Jones</familyName>
               </name>
               <dateOfBirth>19590717</dateOfBirth>
            </ChildInformation>
         </children>
      </PersonnelRecord>
```

E.2 Guidelines for use of the notation

The data types and formal notation defined by this Recommendation | International Standard are flexible, allowing a wide range of protocols to be designed using them. This flexibility, however, can sometimes lead to confusion, especially when the notation is approached for the first time. This annex attempts to minimize confusion by giving guidelines for, and examples of, the use of the notation. For each of the built-in data types, one or more usage guidelines are offered. The character string types (for example, **VisibleString**) and the types defined in clauses 42 to 44 are not dealt with here.

E.2.1 Boolean

E.2.1.1 Use a boolean type to model the values of a logical (that is, two-state) variable, for example, the answer to a yes-or-no question.

EXAMPLE

```
Employed ::= BOOLEAN
```

E.2.1.2 When assigning a reference name to a boolean type, choose one that describes the *true* state.

EXAMPLE

```
Married ::= BOOLEAN
```

not

```
MaritalStatus ::= BOOLEAN
```

E.2.2 Integer

E.2.2.1 Use an integer type to model the values (for all practical purposes, unlimited in magnitude) of a cardinal or integer variable.

EXAMPLE

```
CheckingAccountBalance ::= INTEGER -- in cents; negative means overdrawn.
balance CheckingAccountBalance ::= 0
```

or using XML value notation:

```
balance ::= <CheckingAccountBalance>0</CheckingAccountBalance>
```

E.2.2.2 Define the minimum and maximum allowed values of an integer type as named numbers.

EXAMPLE

```
DayOfTheMonth ::= INTEGER {first(1), last(31)}
today DayOfTheMonth ::= first
unknown DayOfTheMonth ::= 0
```

or using XML value notation:

```
today ::= <DayOfTheMonth><first/></DayOfTheMonth>
unknown ::= <DayOfTheMonth>0</DayOfTheMonth>
```

Note that the named numbers **first** and **last** were chosen because of their semantic significance to the reader, and does not exclude the possibility of **DayOfTheMonth** having other values which may be less than 1, greater than 31 or between 1 and 31.

To restrict the value of **DayOfTheMonth** to just **first** and **last**, one would write:

```
DayOfTheMonth ::= INTEGER {first(1), last(31)} (first | last)
```

and to restrict the value of the **DayOfTheMonth** to all values between 1 and 31, inclusive, one would write:

```
DayOfTheMonth ::= INTEGER {first(1), last(31)} (first .. last)
dayOfTheMonth DayOfTheMonth ::= 4
```

or using XML value notation:

```
dayOfTheMonth ::= <DayOfTheMonth>4</DayOfTheMonth>
```

E.2.3 Enumerated

E.2.3.1 Use an enumerated type to model the values of a variable with three or more states. Assign values starting with zero if their only constraint is distinctness.

EXAMPLE

```
DayOfTheWeek ::= ENUMERATED {sunday(0), monday(1), tuesday(2),
                             wednesday(3), thursday(4), friday(5), saturday(6)}

firstDay DayOfTheWeek ::= sunday
```

or using XML value notation:

```
firstDay ::= <DayOfTheWeek><sunday/></DayOfTheWeek>
```

Note that while the enumerations **sunday**, **monday**, etc., were chosen because of their semantic significance to the reader, **DayOfTheWeek** is restricted to assuming one of these values and no other. Further, only the name **sunday**, **monday**, etc., can be assigned to a value; the equivalent integer values are not allowed.

E.2.3.2 Use an extensible enumerated type to model the values of a variable that has just two states now, but that may have additional states in a future version of the protocol.

EXAMPLE

```
MaritalStatus ::= ENUMERATED {single, married}
         -- First version of MaritalStatus
```

in anticipation of:

```
MaritalStatus ::= ENUMERATED {single, married, …, widowed}
         -- Second version of MaritalStatus
```

and later yet:

```
MaritalStatus ::= ENUMERATED {single, married, …, widowed, divorced}
         -- Third version of MaritalStatus
```

E.2.4 Real

E.2.4.1 Use a real type to model an approximate number.

EXAMPLE

```
AngleInRadians ::=  REAL

pi REAL ::= {mantissa 31415926535897932384626433832 79, base 10, exponent –30}
```

or using the alternate value notation for **REAL**:

```
pi   REAL ::= 3.14159265358979323846264338327
```

or using XML value notation:

```
pi ::=
     <REAL>
          3.14159265358979323846264338327
     </REAL>
```

E.2.4.2 Application designers may wish to ensure full interworking with real values despite differences in floating point hardware, and in implementation decisions to use (for example) single or double length floating point for an application. This can be achieved by the following:

```
App-X-Real ::= REAL (WITH COMPONENTS {
                     mantissa (–16777215..16777215),
                     base (2),
                     exponent (–125..128) } )
/*
  Senders shall not transmit values outside these ranges
  and conforming receivers shall be capable of receiving
  and processing all values in these ranges.
*/

girth App-X-Real ::= {mantissa 16, base 2, exponent 1}
```

or using XML value notation:

```
girth ::=
     <App-X-Real>
        32
     </App-X-Real>
```

E.2.5 Bit string

E.2.5.1 Use a bit string type to model binary data whose format and length are unspecified, or specified elsewhere, and whose length in bits is not necessarily a multiple of eight.

EXAMPLE

```
G3FacsimilePage ::= BIT STRING
--  a sequence of bits conforming to ITU-T Rec. T.4.

image G3FacsimilePage ::= '10011010010000l110110'B

trailer BIT STRING ::= '0123456789ABCDEF'H

body1 G3FacsimilePage ::= '1101'B

body2 G3FacsimilePage ::= '1101000'B
```

or using XML value notation:

```
image ::= <G3FacSimile>10011010010000l110110</G3FacSimile>

trailer ::=
           <BIT_STRING>
                 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011
                 1100 1101 1110 1111
           </BIT_STRING>

body1 ::= <G3FacSimile>1101</G3FacSimile>

body2 ::= <G3FacSimile>1101000</G3FacSimile>
```

Note that **body1** and **body2** are distinct abstract values because trailing 0 bits are significant (due to there being no "NamedBitList" in the definition of **G3FacsimilePage**).

E.2.5.2 Use a bit string type with a size constraint to model the values of a fixed sized bit field.

EXAMPLE

```
BitField ::= BIT STRING (SIZE (12))

map1 BitField ::= '100110100100'B

map2 BitField ::= '9A4'H

map3 BitField ::= '1001101001'B   --  Illegal - violates size constraint.
```

or using XML value notation:

```
map1 ::= <BitField>100110100100</BitField>
```

Note that **map1** and **map2** are the same abstract value, for the four trailing bits of **map2** are not significant.

E.2.5.3 Use a bit string type to model the values of a **bit map**, an ordered collection of logical variables indicating whether a particular condition holds for each of a correspondingly ordered collection of objects.

```
DaysOfTheWeek ::= BIT STRING {
           sunday(0), monday (1), tuesday(2),
           wednesday(3), thursday(4), friday(5),
           saturday(6) } (SIZE (0..7))

sunnyDaysLastWeek1 DaysOfTheWeek ::= {sunday, monday, wednesday}
     sunnyDaysLastWeek2 DaysOfTheWeek ::= '1101'B
     sunnyDaysLastWeek3 DaysOfTheWeek ::= '1101000'B

sunnyDaysLastWeek4 DaysOfTheWeek ::= '11010000'B -- Illegal
```

or using XML value notation:

```
sunnyDaysLastWeek1 ::=
     <DaysOfTheWeek>
         <sunday/><monday/><wednesday/>
     </DaysOfTheWeek>
```

```
sunnyDaysLastWeek2 ::= <DaysOfTheWeek>1101</DaysOfTheWeek>

sunnyDaysLastWeek3 ::= <DaysOfTheWeek>1101000</DaysOfTheWeek>
```

Note that if the bit string value is less than 7 bits long, then the missing bits indicate a cloudy day for those days, hence the first three values above have the same abstract value.

E.2.5.4 Use a bit string type to model the values of a *bit map*, a fixed-size ordered collection of logical variables indicating whether a particular condition holds for each of a correspondingly ordered collection of objects.

```
DaysOfTheWeek ::= BIT STRING {
            sunday(0), monday (1), tuesday(2),
            wednesday(3), thursday(4), friday(5),
            saturday(6) } (SIZE (7))
sunnyDaysLastWeek1 DaysOfTheWeek ::= {sunday, monday, wednesday}
sunnyDaysLastWeek2 DaysOfTheWeek ::= '1101'B -- Illegal
                                     -- violates size constraint.
sunnyDaysLastWeek3 DaysOfTheWeek ::= '1101000'B

sunnyDaysLastWeek4 DaysOfTheWeek ::= '11010000'B --  Illegal
                                         -- violates size constraint.
```

Note that the first and third values have the same abstract value.

E.2.5.5 Use a bit string type with named bits to model the values of a collection of related logical variables.

EXAMPLE

```
PersonalStatus ::= BIT STRING
            {married(0), employed(1), veteran(2), collegeGraduate(3)}

billClinton PersonalStatus ::= {married, employed, collegeGraduate}

hillaryClinton PersonalStatus ::= '110100'B
```

or using XML value notation:

```
billClinton ::=
      <PersonalStatus>
         <married/>
         <employed/>
         <collegeGraduate/>
      </PersonalStatus>

hillaryClinton ::= <PersonalStatus>110100</PersonalStatus>
```

Note that **billClinton** and **hillaryClinton** have the same abstract values.

E.2.6 Octet string

E.2.6.1 Use an octet string type to model binary data whose format and length are unspecified, or specified elsewhere, and whose length in bits is a multiple of eight.

EXAMPLE

```
G4FacsimileImage ::= OCTET STRING
--  a sequence of octets conforming to ITU-T Rec. T.5 and CCITT Rec. T.6

image G4FacsimilePage ::= '3FE2EBAD471005'H
```

or using XML value notation:

```
image ::= <G4FacSimileImage>3FE2EBAD471005</G4FacSimileImage>
```

E.2.6.2 Use a restricted character string type in preference to an octet string type, where an appropriate one is available.

EXAMPLE

```
Surname ::= PrintableString

president Surname ::= "Clinton"
```

or using XML value notation:

```
president ::= <Surname>Clinton</Surname>
```

E.2.7 UniversalString, BMPString and UTF8String

Use the **BMPString** type or the **UTF8String** type to model any string of information which consists solely of characters from the ISO/IEC 10646-1 Basic Multilingual Plane (BMP), and **UniversalString** or **UTF8String** to model any string which consists of ISO/IEC 10646-1 characters not confined to the BMP.

E.2.7.1 Use **Level1** or **Level2** to denote that the implementation level places restrictions on the use of combining characters.

EXAMPLE

```
RussianName ::= Cyrillic (Level1)
      -- RussianName uses no combining characters.

SaudiName ::= BasicArabic (SIZE (1..100) ^ Level2)
      --  SaudiName uses a subset of combining characters.
```

Representation of letter Σ:

```
greekCapitalLetterSigma BMPString ::= {0, 0, 3, 163}
```

or using XML value notation:

```
greekCapitalLetterSigma ::= <BMPString>&#x03a3;</BMPString>
```

Representation of string "f $\rightarrow \propto$":

```
rightwardsArrow UTF8String ::= {0, 0, 33, 146}

infinity UTF8String ::= {0, 0, 34, 30}

property UTF8String ::= {"f  ", rightwardsArrow, " ", infinity}
```

or using XML value notation:

```
property ::= <UTF8String>f &#x2192; &#x221E;</UTF8String>
```

E.2.7.2 A collection can be expanded to be a selected subset (i.e., include all characters in the BASIC LATIN collection) by use of the "UnionMark" (see clause 46).

EXAMPLE

```
KatakanaAndBasicLatin ::= UniversalString (FROM (Katakana | BasicLatin))
```

E.2.8 CHARACTER STRING

Use the unrestricted character string type to model any string of information which cannot be modelled using one of the restricted character string types. Be sure to specify the repertoire of characters and their coding into octets.

EXAMPLE

```
PackedBCDString ::= CHARACTER STRING (WITH COMPONENTS {
                                                identification  (WITH
COMPONENTS {
                                                  fixed  PRESENT })
      /* The abstract and transfer syntaxes shall be
         packedBCDString-AbstractSyntaxId and
         packedBCDString-TransferSyntaxId defined below.
      */
                          } )

      /* object identifier value for a character abstract syntax
         (character set) whose alphabet
         is the digits 0 through 9.
      */
      PackedBCDString-AbstractSyntaxId  OBJECT  IDENTIFIER ::=
            { joint-iso-itu-t asn1(1) examples(123) packedBCD(2) charSet(0) }
```

```
        /* object identifier value for a character transfer syntax that
           packs two digits per octet, each digit encoded as 0000 to
           1001, 1111₂ used for padding.
        */
        PackedBCDString-TransferSyntaxId  OBJECT IDENTIFIER ::=
              { joint-iso-itu-t asn1(1) examples(123) packedBCD(2)
                    characterTransferSyntax(1) }
/* The encoding of PackedBCDString will contain only the defined
   encoding of the characters, with any necessary length field, and in
   the case of BER with a field carrying the tag. The object
   identifier values are not carried, as "fixed" has been specified.
*/
```

or using XML value notation:

```
packedBCDString-AbstractSyntaxId ::=
      <OBJECT_IDENTIFIER>
            joint-iso-itu-t.asn1(1).examples(123).packedBCD(2).charSet(0)
      </OBJECT_IDENTIFIER>


packedBCDString-TransferSyntaxId ::=
 <OBJECT_IDENTIFIER>
 joint-iso-itu-t.asn1(1).examples(123).packedBCD(2).characterTransferSyntax(1)
 </OBJECT_IDENTIFIER>
```

or:

```
packedBCDString-AbstractSyntaxId ::=
      <OBJECT_IDENTIFIER>2.1.123.2.0</OBJECT_IDENTIFIER>

PackedBCDString-TransferSyntaxId ::=
      <OBJECT_IDENTIFIER>2.1.123.2.1</OBJECT_IDENTIFIER>
```

NOTE – Encoding rules do not necessarily encode values of the type **CHARACTER STRING** in a form that always includes the object identifier values, although they do guarantee that the abstract value is preserved in the encoding.

E.2.9 Null

Use a null type to indicate the effective absence of a component of a sequence.

EXAMPLE

```
PatientIdentifier ::= SEQUENCE {
      name              VisibleString,
      roomNumber        CHOICE {
            room        INTEGER,
            outPatient  NULL  --  if an out-patient  --
      }
}

lastPatient PatientIdentifier ::= {
      name  "Jane Doe",
      roomNumber  outPatient : NULL
}
```

or using XML value notation:

```
lastPatient ::=
      <PatientIdentifier>
         <name>Jane Doe</name>
         <roomNumber><outPatient/></roomNumber>
      </PatientIdentifier>
```

E.2.10 Sequence and sequence-of

E.2.10.1 Use a sequence-of type to model a collection of variables whose types are the same, whose number is large or unpredictable, and whose order is significant.

EXAMPLE

```
NamesOfMemberNations ::= SEQUENCE OF VisibleString
--  in alphabetical order

firstTwo  NamesOfMemberNations ::= {"Australia", "Austria"}
```

or, using the optional identifier:

```
NamesOfMemberNations2 ::= SEQUENCE OF memberNation VisibleString
--  in alphabetical order

firstTwo2  NamesOfMemberNations2 ::=
      {memberNation "Australia", memberNation "Austria"}
```

Using XML value notation, the above two values are as follows:

```
firstTwo ::=
      <NamesOfMemberNations>
          <VisibleString>Australia</VisibleString>
          <VisibleString>Austria</VisibleString>
      </NamesOfMemberNations>

firstTwo2 ::=
      <NamesOfMemberNations2>
          <memberNation>Australia</memberNation>
          <memberNation>Austria</memberNation>
      </NamesOfMemberNations2>
```

E.2.10.2 Use a sequence type to model a collection of variables whose types are the same, whose number is known and modest, and whose order is significant, provided that the make-up of the collection is unlikely to change from one version of the protocol to the next.

EXAMPLE

```
NamesOfOfficers ::= SEQUENCE {
            president             VisibleString,
            vicePresident         VisibleString,
            secretary             VisibleString}

acmeCorp NamesOfOfficers ::= {
            president             "Jane Doe",
            vicePresident         "John Doe",
            secretary             "Joe Doe"}
```

or using XML value notation:

```
acmeCorp ::=
      <NamesOfOfficers>
          <president>Jane Doe</president>
           <vicePresident>John Doe</vicePresident>
           <secretary>Joe Doe</secretary>
      </NamesOfOfficers>
```

E.2.10.3 Use an inextensible sequence type to model a collection of variables whose types differ, whose number is known and modest, and whose order is significant, provided that the make-up of the collection is unlikely to change from one version of the protocol to the next.

EXAMPLE

```
Credentials ::= SEQUENCE {
            userName              VisibleString,
            password              VisibleString,
            accountNumber         INTEGER}
```

E.2.10.4 Use an extensible sequence type to model a collection of variables whose order is significant, whose number currently is known and is modest, but which is expected to be increased:

EXAMPLE

```
Record ::= SEQUENCE {  -- First version of protocol containing "Record"
      userName                    VisibleString,
      password                    VisibleString,
      accountNumber               INTEGER,
      ...,
      ...
}
```

in anticipation of:

```
Record ::= SEQUENCE {  -- Second version of protocol containing "Record"
      userName                    VisibleString,
      password                    VisibleString,
      accountNumber               INTEGER,
      ...,
```

```
      [[2:              -- Extension addition added in protocol version 2
         lastLoggedIn              GeneralizedTime OPTIONAL,
         minutesLastLoggedIn       INTEGER
      ]],
      ...
}
```

and later yet (version 3 of the protocol made no additions to **Record**):

```
Record ::= SEQUENCE {  -- Third version of protocol containing "Record"
      userName                   VisibleString,
      password                   VisibleString,
      accountNumber              INTEGER,
      ...,
      [[2:              -- Extension addition added in protocol version 2
         lastLoggedIn              GeneralizedTime OPTIONAL,
         minutesLastLoggedIn       INTEGER
      ]],
      [[4:              -- Extension addition added in protocol version 3
         certificate               Certificate,
         thumb                     ThumbPrint OPTIONAL
      ]],
      ...
      }
```

E.2.11 Set and set-of

E.2.11.1 Use a set type to model a collection of variables whose number is known and modest and whose order is insignificant. If automatic tagging is not in effect, identify each variable by context-specifically tagging it as shown below. (With automatic tagging, the tags are not needed.)

EXAMPLE

```
UserName ::= SET {
            personalName                [0] VisibleString,
            organizationName            [1] VisibleString,
            countryName                 [2] VisibleString}

user UserName ::= {
            countryName                 "Nigeria",
            personalName                   "Jonas Maruba",
            organizationName            "Meteorology, Ltd."}
```

or using XML value notation:

```
user ::=
      <UserName>
         <countryName>Nigeria</countryName>
         <personalName>Jonas Maruba</personalName>
         <organizationName>Meteorology, Ltd.</organizationName>
      </UserName>
```

E.2.11.2 Use a set type with **OPTIONAL** to model a collection of variables that is a (proper or improper) subset of another collection of variables whose number is known and reasonably small and whose order is insignificant. If automatic tagging is not in effect, identify each variable by context-specifically tagging it as shown below. (With automatic tagging, the tags are not needed.)

EXAMPLE

```
UserName ::= SET {
      personalName                    [0] VisibleString,
      organizationName                [1] VisibleString OPTIONAL
         --  defaults to that of the local organization  -- ,
      countryName                     [2] VisibleString OPTIONAL
         --  defaults to that of the local country  --  }
```

E.2.11.3 Use an extensible set type to model a collection of variables whose make-up is likely to change from one version of the protocol to the next. The following assumes **AUTOMATIC TAGS** was specified in the module definition.

EXAMPLE

```
UserName ::= SET {
      personalName           VisibleString,        -- First version of
"UserName"
      organizationName       VisibleString OPTIONAL ,
      countryName            VisibleString OPTIONAL,
      ...,
      ...
}

user UserName ::= { personalName  "Jonas Maruba" }
```

or using XML value notation:

```
user ::=
      <UserName>
         <personalName>Jonas Maruba</personalName>
      </UserName>
```

in anticipation of:

```
UserName ::= SET {           -- Second version of "UserName"
      personalName                VisibleString,
      organizationName            VisibleString OPTIONAL,
      countryName                 VisibleString OPTIONAL,
      ...,
      [[2:                   -- Extension addition added in protocol version 2
         internetEmailAddress     VisbleString,
         faxNumber                VisibleString OPTIONAL
      ]],
      ...
}

user  UserName ::= {
      personalName                "Jonas Maruba",
      internetEmailAddress        "jonas@meteor.ngo.com"
}
```

or using XML value notation:

```
 user ::=
<UserName>
   <personalName>Jonas Maruba</personalName>
   <internetEmailAddress>jonas@meteor.ngo.com</internetEmailAddress>
</UserName>
```

and later yet (versions 3 and 4 of the protocol made no additions to **UserName**):

```
UserName ::= SET {     -- Fifth version of protocol containing "UserName"
      personalName                VisibleString,
      organizationName            VisibleString OPTIONAL,
      countryName                 VisibleString OPTIONAL,
      ...,
      [[2:                        -- Extension addition added in version 2
         internetEmailAddress     VisbleString,
         faxNumber                VisibleString OPTIONAL
      ]],
      [[5:                        -- Extension addition added in version 5
      phoneNumber                 VisibleString OPTIONAL
      ]],
      ...
}

user  UserName ::= {
      personalName                "Jonas Maruba",
      internetEmailAddress        "jonas@meteor.ngo.com"
}
```

or using XML value notation:

```
user ::=
      <UserName>
         <personalName>Jonas Maruba</personalName>
         <internetEmailAddress>jonas@meteor.ngo.com</internetEmailAddress>
      </UserName>
```

E.2.11.4 Use a set-of type to model a collection of variables whose types are the same and whose order is insignificant.

EXAMPLE

```
Keywords ::= SET OF VisibleString  --  in arbitrary order

someASN1Keywords Keywords ::= {"INTEGER", "BOOLEAN", "REAL"}
```

or, using the optional identifier:

```
Keywords2 ::= SET OF keyword VisibleString  --  in arbitrary order

someASN1Keywords2 Keywords2 ::= {keyword "INTEGER", keyword "BOOLEAN",
     keyword "REAL"}
```

Using XML value notation, the above two values are as follows:

```
someASN1Keywords ::=
     <Keywords>
        <VisibleString>INTEGER</VisibleString>
        <VisibleString>BOOLEAN</VisibleString>
        <VisibleString>REAL</VisibleString>
     </Keywords>

someASN1Keywords2 ::=
     <Keywords2>
        <keyword>INTEGER</keyword>
        <keyword>BOOLEAN</keyword>
        <keyword>REAL</keyword>
     </Keywords2>
```

E.2.12 Tagged

Prior to the introduction of the **AUTOMATIC TAGS** construct, ASN.1 specifications frequently contained tags. The following subclauses describe the way in which tagging was typically applied. With the introduction of **AUTOMATIC TAGS**, new ASN.1 specifications need make no use of the tag notation, although those modifying old notation may have to concern themselves with tags. New users of the ASN.1 notation are encouraged to use **AUTOMATIC TAGS** as this makes the notation more readable.

E.2.12.1 Universal class tags are used only within this Recommendation | International Standard. The notation **[UNIVERSAL 30]** (for example) is provided solely to enable precision in the definition of the "UsefulTypes" (see 41.1). It should not be used elsewhere.

E.2.12.2 A frequently encountered style for the use of tags is to assign an application class tag precisely once in the entire specification, using it to identify a type that finds wide, scattered, use within the specification. An application class tag is also frequently used (once only) to tag the types in the outermost **CHOICE** of an application, providing identification of individual messages by the application class tag. The following is an example use in the former case:

EXAMPLE

```
FileName ::= [APPLICATION 8] SEQUENCE {
           directoryName               VisibleString,
           directoryRelativeFileName   VisibleString}
```

E.2.12.3 Context-specific tagging is frequently applied in an algorithmic manner to all components of a **SET**, **SEQUENCE**, or **CHOICE**. Note, however, that the **AUTOMATIC TAGS** facility does this easily for you.

EXAMPLE

```
CustomerRecord ::= SET {
           name                        [0] VisibleString,
           mailingAddress              [1] VisibleString,
           accountNumber               [2] INTEGER,
           balanceDue                  [3] INTEGER  --  in cents  --}

CustomerAttribute ::= CHOICE {
           name                        [0] VisibleString,
           mailingAddress              [1] VisibleString,
           accountNumber               [2] INTEGER,
           balanceDue                  [3] INTEGER  --  in cents  --}
```

E.2.12.4 Private class tagging should normally not be used in internationally standardized specifications (although this cannot be prohibited). Applications produced by an enterprise will normally use application and context-specific tag classes. There may be occasional cases, however, where an enterprise-specific specification seeks to extend an

internationally standardized specification, and in this case use of private class tags may give some benefits in partially protecting the enterprise-specific specification from changes to the internationally standardized specification.

EXAMPLE

```
AcmeBadgeNumber ::= [PRIVATE 2] INTEGER

badgeNumber AcmeBadgeNumber ::= 2345
```

or using XML value notation:

```
badgeNumber ::= <AcmeBadgeNumber>2345</AcmeBadgeNumber>
```

E.2.12.5 Textual use of **IMPLICIT** with every tag is generally found only in older specifications. BER produces a less compact representation when explicit tagging is used than when implicit tagging is used. PER produces the same compact encoding in both cases. With BER and explicit tagging, there is more visibility of the underlying type (**INTEGER**, **REAL**, **BOOLEAN**, etc.) in the encoded data. These guidelines use implicit tagging in the examples whenever it is legal to do so. This may, depending on the encoding rules, result in a compact representation, which is highly desirable in some applications. In other applications, compactness may be less important than, for example, the ability to carry out strong type-checking. In the latter case, explicit tagging can be used.

EXAMPLE

```
CustomerRecord ::= SET {
            name                        [0] IMPLICIT VisibleString,
            mailingAddress              [1] IMPLICIT VisibleString,
            accountNumber               [2] IMPLICIT INTEGER,
            balanceDue                  [3] IMPLICIT INTEGER  --  in cents  --
}

CustomerAttribute ::= CHOICE {
            name                        [0] IMPLICIT VisibleString,
            mailingAddress              [1] IMPLICIT VisibleString,
            accountNumber               [2] IMPLICIT INTEGER,
            balanceDue                  [3] IMPLICIT INTEGER  --  in cents  --
}
```

E.2.12.6 Guidance on use of tags in new ASN.1 specifications referencing this Recommendation | International Standard is quite simple: DON'T USE TAGS. Put **AUTOMATIC TAGS** in the module header, then forget about tags. If you need to add new components to the **SET**, **SEQUENCE** or **CHOICE** in a later version, add them to the end.

E.2.13 Choice

E.2.13.1 Use a **CHOICE** to model a variable that is selected from a collection of variables whose number are known and modest.

EXAMPLE

```
FileIdentifier ::= CHOICE {
            relativeName     VisibleString,
          --  name of file (for example, "MarchProgressReport")
            absoluteName     VisibleString,
          --  name of file and containing directory
          --  (for example, "<Williams>MarchProgressReport")
            serialNumber     INTEGER
          --  system-assigned identifier for file  --}

file FileIdentifier ::= serialNumber : 106448503
```

or using XML value notation:

```
fileIdentifier ::=
           <FileIdentifier>
              <serialNumber>106448503</serialNumber>
           </FileIdentifier>
```

E.2.13.2 Use an extensible **CHOICE** to model a variable that is selected from a collection of variables whose make-up is likely to change from one version of the protocol to the next.

EXAMPLE

```
     FileIdentifier ::= CHOICE {                        -- First version of
FileIdentifier
                relativeName    VisibleString,
```

```
                absoluteName    VisibleString,

                ...,   ...
        }

        fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"
```

or using XML value notation:

```
    fileId1 ::=
        <FileIdentifier>
           <relativeName>MarchProgressReport.doc</relativeName>
        </FileIdentifier>
```

in anticipation of:

```
        FileIdentifier ::= CHOICE {       -- Second version of FileIdentifier
            relativeName     VisibleString,
            absoluteName     VisibleString,
            ...,
            serialNumber     INTEGER,    -- Extension addition added in version 2
            ...
        }

        fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"

        fileId2  FileIdentifier ::= serialNumber : 214
```

or using XML value notation:

```
    fileId1 ::=
        <FileIdentifier>
           <relativeName>MarchProgressReport.doc</relativeName>
        </FileIdentifier>

    fileId2 ::=
        <FileIdentifier>
           <serialNumber>214</serialNumber>
        </FileIdentifier>
```

and later yet:

```
        FileIdentifier ::= CHOICE {       -- Third version of FileIdentifier
            relativeName     VisibleString,
            absoluteName     VisibleString,
            ...,
            serialNumber     INTEGER,    -- Extension addition added in version 2
            [[                           -- Extension addition added in version 3
               vendorSpecificVendorExt,
               unidentified  NULL
            ]],
            ...
        }

        fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"

        fileId2  FileIdentifier ::= serialNumber : 214

        fileId3  FileIdentifier ::= unidentified : NULL
```

or using XML value notation:

```
        fileId1 ::=
            <FileIdentifier>
               <relativeName>MarchProgressReport.doc</relativeName>
            </FileIdentifier>

        fileId2 ::=
            <FileIdentifier>
               <serialNumber>214</serialNumber>
            </FileIdentifier>

        fileId3 ::=
            <FileIdentifier>
               <unidentified/>
```

```
        </FileIdentifier>
```

E.2.13.3 Use an extensible **CHOICE** of only one type where the possibility is envisaged of more than one type being permitted in the future.

EXAMPLE

```
Greeting ::= CHOICE {                -- First version of "Greeting"
      postCard         VisibleString,
      ...,
      ...
}
```

in anticipation of:

```
Greeting ::= CHOICE {                -- Second version of "Greeting"
      postCard    VisibleString,
      ...,
      [[2:                           -- Extension addition added in version 2
         audio          Audio,
         video          Video
      ]],
      ...
}
```

E.2.13.4 Multiple colons are required when a choice value is nested within another choice value.

EXAMPLE

```
Greeting ::= [APPLICATION 12] CHOICE {
            postCard         VisibleString,
            recording        Voice }

Voice ::= CHOICE {
            english          OCTET STRING,
            swahili          OCTET STRING }

myGreeting Greeting ::= recording : english : '019838547E0'H
```

or using XML value notation:

```
myGreeting ::=
      <Greeting>
         <recording><english>019838547E0</english></recording>
      </Greeting>
```

E.2.14 Selection type

E.2.14.1 Use a selection type to model a variable whose type is that of some particular alternatives of a previously defined **CHOICE**.

E.2.14.2 Consider the definition:

```
FileAttribute ::= CHOICE {
            date-last-used   INTEGER,
            file-name        VisibleString}
```

then the following definition is possible:

```
AttributeList ::= SEQUENCE {
            first-attribute  date-last-used  < FileAttribute,
            second-attribute file-name < FileAttribute }
```

with a possible value notation of:

```
listOfAttributes AttributeList ::=  {
            first-attribute  27,
            second-attribute "PROGRAM" }
```

or using XML value notation:

```
listOfAttributes ::=
      <AttributeList>
         <first-attribute>27</first-attribute>
         <second-attribute>PROGRAM</second-attribute>
      </AttributeList>
```

E.2.15 Object class field type

E.2.15.1 Use an object class field type to identify a type defined by means of an information object class (see ITU-T Rec. X.681 | ISO/IEC 8824-2). For example, fields of the information object class **ATTRIBUTE** may be used in defining a type, **Attribute**.

EXAMPLE

```
ATTRIBUTE ::= CLASS {
      &AttributeType,
      &attributeId          OBJECT IDENTIFIER UNIQUE
            }

Attribute ::= SEQUENCE {
      attributeID       ATTRIBUTE.&attributeId,   --  this is normally constrained.
      attributeValue    ATTRIBUTE.&AttributeType     --  this is normally
      constrained.
            }
```

Both **ATTRIBUTE.&attributeId** and **ATTRIBUTE.&AttributeType** are object class field types, in that they are types defined by reference to an information object class (**ATTRIBUTE**). The type **ATTRIBUTE.&attributeId** is fixed because it is explicitly defined in **ATTRIBUTE** as an **OBJECT IDENTIFIER**. However, the type **ATTRIBUTE.&AttributeType** can carry a value of any type defined using ASN.1, since its type is not fixed in the definition of the information object class **ATTRIBUTE**. Notations that possess this property of being able to carry a value of any type are termed "open type notation", hence **ATTRIBUTE.&AttributeType** is an open type.

E.2.16 Embedded-pdv

E.2.16.1 Use an embedded-pdv type to model a variable whose type is unspecified, or specified elsewhere with no restriction on the notation used to specify the type.

EXAMPLE

```
FileContents ::= EMBEDDED PDV

DocumentList ::= SEQUENCE OF document EMBEDDED PDV
```

E.2.17 External

The external type is similar to the embedded-pdv type, but has fewer identification options. New specifications will generally prefer to use embedded-pdv because of its greater flexibility and the fact that some encoding rules encode its values more efficiently.

E.2.18 Instance-of

E.2.18.1 Use an instance-of to specify a type containing an object identifier field and an open type value whose type is determined by the object identifier. The instance-of type can only be used if the association between the object identifier value and the type is specified using an information object of a class derived from **TYPE-IDENTIFIER** (see ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex A and Annex C).

EXAMPLE

```
ACCESS-CONTROL-CLASS ::= TYPE-IDENTIFIER

Get-Invoke ::= SEQUENCE {
      objectClass       ObjectClass,
      objectInstance    ObjectInstance,
      accessControl     INSTANCE OF ACCESS-CONTROL-CLASS, --  this is normally
                                                          --  constrained.
      attributeID       ATTRIBUTE.&attributeId
      }
```

Get-Invoke is then equivalent to:

```
Get-Invoke ::= SEQUENCE {
      objectClass       ObjectClass,
      objectInstance    ObjectInstance,
      accessControl     [UNIVERSAL 8] IMPLICIT SEQUENCE {
            type-id     ACCESS-CONTROL-CLASS.&id,    --  this is normally
                                                     --  constrained.
            value       [0] ACCESS-CONTROL-CLASS.&Type --  this is normally
                                                       --  constrained.
      },
```

```
        attributeID ATTRIBUTE.&attributeId
}
```

The true utility of the instance-of type is not seen until it is constrained using an information object set, but such an example goes beyond the scope of this Recommendation | International Standard. See ITU-T Rec. X.682 | ISO/IEC 8824-3 for the definition of information object set, and Annex A of ITU-T Rec. X.682 | ISO/IEC 8824-3 for how to use an information object set to constrain an instance-of type.

E.2.19 Relative object identifier

E.2.19.1 Use a relative object identifier type to transmit object identifier values in a more compact form in contexts where the early part of the object identifier value is known. There are three situations that can arise:

a) The early part of the object identifier value is fixed for a given specification (it is an industry-specific standard, and all OIDs are relative to an OID allocated to the standardizing body. In this case, use:

```
RELATIVE-OID      --  The relative object identifier value is
                  --  relative to {iso identified-organization  set(22)}
```

b) The early part of the object identifier value is frequently a value that is known at specification time, but may occasionally be a more general value. In this case, use:

```
CHOICE
      {a    RELATIVE-OID    --  The value is relative to {1  3  22}--,
       b    OBJECT IDENTIFIER  -- Any object identifier value--}
```

c) The early part of the object identifier value is not known until communications time, but will frequently be common to many values that need to be sent, and quite often will be a value known at specification time. In this case use (for example):

```
SEQUENCE
      {oid-root   OBJECT IDENTIFIER DEFAULT {1  3  22},
       reloids    SEQUENCE OF RELATIVE-OID --relative to oid-root--}
```

E.3 Identifying abstract syntaxes

E.3.1 It is common for protocols to be defined by associating semantics with each of the values of a single ASN.1 type, typically a choice type. (This ASN.1 type is sometimes referred to informally as "the top-level type for the application".) This set of abstract values is formally called the abstract syntax for the application. An abstract syntax can be identified by giving it an abstract syntax name of ASN.1 type object identifier.

E.3.2 The assignment of an object identifier to an abstract syntax can be done using the built-in information object class **ABSTRACT-SYNTAX** which is defined in ITU-T Rec. X.681 | ISO/IEC 8824-2. This also serves to clearly identify the top-level type for the application.

E.3.3 The following is an example of text which might appear in an application specification:

EXAMPLE

```
Application-ASN1 DEFINITIONS ::=
      BEGIN
      EXPORTS Application-PDU;

      Application-PDU ::= CHOICE {
                  connect-pdu    ..... ,
                  data-pdu    CHOICE {
                        ..... ,
                        .....
                  },
                  .....
            }
            ......
      END

Abstract-Syntax-Module DEFINITIONS ::=
      BEGIN
      IMPORTS Application-PDU FROM Application-ASN1;

--  This application defines the following abstract syntax:

      Abstract-Syntax ABSTRACT-SYNTAX ::=
                  { Application-PDU  IDENTIFIED BY
                        application-abstract-syntax-object-id }
```

```
     application-abstract-syntax-object-id OBJECT IDENTIFIER ::=
                {joint-iso-itu-t asn1(1) examples(123)
                     application-abstract-syntax(3) }
--  The corresponding object descriptor is:

     application-abstract-syntax-descriptor ObjectDescriptor ::=
                     "Example Application Abstract Syntax"

--  The ASN.1 object identifier and object descriptor values:
                --  encoding rule object identifier
                --  encoding rule object descriptor
--  assigned to encoding rules in ITU-T Rec. X.690 | ISO/IEC 8825-1
-- and ITU-T Rec. X.691 | ISO/IEC 8825-2 can be used as the transfer
-- syntax identifier in conjunction with this transfer syntax.

END
```

E.3.4 In order to ensure interworking, the standard may additionally identify a mandatory transfer syntax (typically one of those defined in the encoding rules of ITU-T Rec. X.690 | ISO/IEC 8825-1 or ITU-T Rec. X.691 | ISO/IEC 8825-2 or ITU-T Rec. X.692 | ISO/IEC 8825-3).

E.4 Subtypes

E.4.1 Use subtypes to limit the values of an existing type which are to be permitted in a particular situation.

EXAMPLES

```
AtomicNumber    ::=   INTEGER (1..104)

TouchToneString ::=  IA5String
                 (FROM ("0123456789" | "*" | "#")) (SIZE (1..63))

ParameterList   ::=  SET SIZE (1..63) OF Parameter

SmallPrime   ::=  INTEGER (2|3|5|7|11|13|17|19|23|29)
```

E.4.2 Use an extensible subtype constraint to model an **INTEGER** type whose set of permitted values is small and well defined, but which is expected to increase.

EXAMPLE

```
SmallPrime ::= INTEGER (2 | 3, ...)                    -- First version of
SmallPrime
```

in anticipation of:

```
SmallPrime ::= INTEGER (2 | 3, ..., 5 | 7 | 11)
-- Second version of SmallPrime
```

and later yet:

```
SmallPrime ::= INTEGER (2 | 3, ..., 5 | 7 | 11 | 13 | 17 | 19)
-- Third version of SmallPrime
```

NOTE – For certain types, some encoding rules (e.g., PER) provide a highly optimized encoding for subtype constraint extension root values (i.e., values appearing before the "**...**") and a less optimized encoding for subtype constraint extension addition values (i.e., values appearing after the "**...**"), while in some other encoding rules (e.g., BER) subtype constraints have no effect on the encoding.

E.4.3 Where two or more related types have significant commonality, consider explicitly defining their common parent as a type and use subtyping for the individual types. This approach makes clear the relationship and the commonality, and encourages (though does not force) this to continue as the types evolve. It thus facilitates the use of common implementation approaches to the handling of values of these types.

EXAMPLE

```
Envelope    ::=  SET {
                        typeA  TypeA,
                        typeB  TypeB   OPTIONAL,
                        typeC  TypeC   OPTIONAL}
               --  the common parent

ABEnvelope  ::=  Envelope (WITH  COMPONENTS
                        {... ,
                             typeB   PRESENT, typeC   ABSENT})
```

```
                -- where typeB must always appear and typeC must not
        ACEnvelope  ::=  Envelope (WITH  COMPONENTS
                                  {... ,
                                       typeB   ABSENT, typeC   PRESENT})
                -- where typeC must always appear and typeB must not
```

The latter definitions could alternatively be expressed as:

```
        ABEnvelope  ::= Envelope (WITH COMPONENTS {typeA, typeB})

        ACEnvelope  ::= Envelope (WITH COMPONENTS {typeA, typeC})
```

The choice between the alternatives would be made upon such factors as the number of components in the parent type, and the number of those which are optional, the extent of the difference between the individual types, and the likely evolution strategy.

E.4.4 Use subtyping to partially define a value, for example, a protocol data unit to be tested for in a conformance test, where the test is concerned only with some components of the PDU.

EXAMPLE

Given:

```
        PDU  ::=  SET
                 {alpha          INTEGER,
                  beta           IA5String  OPTIONAL,
                  gamma          SEQUENCE OF Parameter,
                  delta          BOOLEAN}
```

then in composing a test which requires the Boolean to be false and the integer to be negative, write:

```
        TestPDU   ::=  PDU (WITH COMPONENTS
                        {... ,
                         delta (FALSE),
                         alpha (MIN..<0)})
```

and if, further, the **IA5String**, **beta**, is to be present and either 5 or 12 characters in length, write:

```
        FurtherTestPDU ::= TestPDU (WITH COMPONENTS {... , beta (SIZE (5|12)) PRESENT
        } )
```

E.4.5 If a general-purpose data type has been defined as a **SEQUENCE OF**, use subtyping to define a restricted subtype of the general type.

EXAMPLE

```
        Text-block ::=  SEQUENCE OF VisibleString

        Address   ::=  Text-block (SIZE (1..6)) (WITH COMPONENT (SIZE (1..32)))
```

E.4.6 If a general-purpose data type had been defined as a **CHOICE**, use subtyping to define a restricted subtype of the general type.

EXAMPLE

```
        Z ::= CHOICE {
                    a          A,
                    b          B,
                    c          C,
                    d          D,
                    e          E
              }

        V ::= Z (WITH COMPONENTS { ..., a  ABSENT,  b  ABSENT })   --  'a' and 'b'
        must
                                       -- be absent, either 'c',
                                       --  'd' or 'e' may be present in a value.
```

```
     W ::= Z (WITH COMPONENTS { ..., a  PRESENT })         --  only 'a' can be
present
                                          -- (see 47.8.9.2).

     X ::= Z (WITH COMPONENTS { a  PRESENT })         --  only 'a' can be present
                                       -- (see 47.8.9.2).

     Y ::= Z (WITH COMPONENTS { a  ABSENT,  b,  c })          --  'a', 'd' and 'e'
must be
                                       -- absent, either 'b' or 'c' may
                                       -- be present in a value.
```

NOTE – **W** and **X** are semantically identical.

E.4.7 Use contained subtypes to form new subtypes from existing subtypes.

EXAMPLE

```
     Months      ::= ENUMERATED {
                             january          (1),
                             february         (2),
                             march            (3),
                             april            (4),
                             may              (5),
                             june             (6),
                             july             (7),
                             august           (8),
                             september        (9),
                             october          (10),
                             november         (11),
                             december         (12) }

     First-quarter ::= Months ( january | february  | march )

     Second-quarter ::= Months ( april | may | june )

     Third-quarter ::= Months ( july | august | september )

     Fourth-quarter ::= Months ( october | november | december )

     First-half ::= Months ( First-quarter | Second-quarter )

     Second-half ::= Months ( Third-quarter | Fourth-quarter )
```

Annex F

Tutorial annex on ASN.1 character strings

(This annex does not form an integral part of this Recommendation | International Standard)

F.1 Character string support in ASN.1

F.1.1 There are four groups of character string support in ASN.1. The four groups are:

a) Character string types based on *ISO International Register of Coded Character Sets to be used with Escape Sequences* (that is, based on the structure of ISO/IEC 646) and the associated International Register of Coded Character Sets, and provided by the types **VisibleString**, **IA5String**, **TeletexString**, **VideotexString**, **GraphicString**, and **GeneralString**.

b) Character string types based on ISO/IEC 10646-1, and provided by subsetting the type **UniversalString**, **UTF8String** or **BMPString** with subsets defined in ISO/IEC 10646-1 or by using named characters.

c) Character string types providing a simple small collection of characters specified in this Recommendation | International Standard, and intended for specialized use; these are the **NumericString** and **PrintableString** types.

d) Use of the type **CHARACTER STRING**, with negotiation of the character set to be used (or announcement of the set being used); this permits an implementation to use any collection of characters and encodings for which **OBJECT IDENTIFIER**s have been assigned, including those of *ISO International Register of Coded Character Sets to be used with Escape Sequences*, ISO/IEC 7350, ISO/IEC 10646-1, and private collections of characters and encodings (profiles may impose requirements or restrictions on the character sets – the character abstract syntaxes – to be used).

F.2 The UniversalString, UTF8String and BMPString types

F.2.1 The **UniversalString** and **UTF8String** types carry any character from ISO/IEC 10646-1. The set of characters in ISO/IEC 10646-1 is generally too large for meaningful conformance to be required, and should normally be subsetted to a combination of the standard collections of characters in Annex A of ISO/IEC 10646-1.

F.2.2 The **BMPString** type carries any character from the Basic Multilingual Plan of ISO/IEC 10646-1. The Basic Multilingual Plane is normally subsetted to a combination of the standard collections of characters in Annex A of ISO/IEC 10646-1.

F.2.3 For the collections defined in Annex A of ISO/IEC 10646-1, there are type references defined in the built-in ASN.1 module **ASN1-CHARACTER-MODULE** (see clause 38). The "subtype constraint" mechanism allows new subtypes of **UniversalString** that are combinations of existing subtypes to be defined.

F.2.4 Examples of type references defined in **ASN1-CHARACTER-MODULE** and their corresponding ISO/IEC 10646-1 collection names are:

BasicLatin	BASIC LATIN
Latin-1Supplement	LATIN-1 SUPPLEMENT
LatinExtended-a	LATIN EXTENDED-A
LatinExtended-b	LATIN EXTENDED-B
IpaExtensions	IPA EXTENSIONS
SpacingModifierLetters	SPACING MODIFIER LETTERS
CombiningDiacriticalMarks	COMBINING DIACRITICAL MARKS

F.2.5 ISO/IEC 10646-1 specifies three "levels of implementation", and requires that all uses of ISO/IEC 10646-1 specify the implementation level.

The implementation level relates to the extent to which support is given for *combining characters* in the character repertoire, and hence, in ASN.1 terms, defines a subset of the **UniversalString** and **BMPString** restricted character string types.

In implementation level 1, combining characters are not allowed, and there is normally a one-to-one correspondence between abstract characters in ASN.1 character strings and printed characters in a physical rendition of the string.

In implementation level 2, certain combining characters (listed in ISO/IEC 10646-1, Annex B) are available for use, but there are others whose use is prohibited.

In implementation level 3, there are no restrictions on the use of combining characters.

F.2.6 A **BMPString** or **UniversalString** can be restricted to exclude all control functions by use of the subtype notation as follows:

```
VanillaBMPString ::= BMPString (FROM (ALL EXCEPT ({0,0,0,0}..{0,0,0,31} |
                                        {0,0,0,128}..{0,0,0,159})))
```

or equivalently:

```
C0 ::= BMPString (FROM ({0,0,0,0} .. {0,0,0,31}))   -- C0 control functions
C1 ::= BMPString (FROM ({0,0,0,128} .. {0,0,0,159}))  -- C1 control functions
VanillaBMPString ::= BMPString (FROM (ALL EXCEPT (C0 | C1)))
```

F.3 On ISO/IEC 10646-1 conformance requirements

Use of **UniversalString**, **BMPString** or **UTF8String** (or subtypes of these) in an ASN.1 type definition requires that the conformance requirements of ISO/IEC 10646-1 be addressed.

These conformance requirements demand that implementors of a standard (X say) using such ASN.1 types provide (in the Protocol Implementation Conformance Statement) a statement of the adopted subset of ISO/IEC 10646-1 for their implementation of standard X, and of the level (support for combining characters) of the implementation.

The use of an ASN.1 subtype of **UniversalString**, **UTF8String** or **BMPString** in a specification requires that an implementation support all the ISO/IEC 10646-1 characters that are included in that ASN.1 subtype, and hence that (at least) those characters be present in the adopted subset for the implementation. It is also a requirement that the stated level be supported for all such ASN.1 subtypes.

NOTE – An ASN.1 specification (in the absence of parameters of the abstract syntax and exception specifications) determines both the (maximum) set of characters that can be transmitted and the (minimum) set of characters that have to be handled on receipt. The adopted set of ISO/IEC 10646-1 requires that characters beyond this set not be transmitted, and that all characters within this set be supported on receipt. The adopted set therefore needs to be precisely the set of all characters permitted by the ASN.1 specification. The case where a parameter of the abstract syntax is present is discussed below.

F.4 Recommendations for ASN.1 users on ISO/IEC 10646-1 conformance

Users of ASN.1 should make clear the set of ISO/IEC 10646-1 characters that will form the adopted subset of implementations (and the required implementation level) if the requirements of their standard are to be met.

This can conveniently be done by defining an ASN.1 subtype of **UniversalString**, **UTF8String** or **BMPString** that contains all the characters needed for the standard, and by restricting it to **Level1** or **Level2** if appropriate. A convenient name for this type might be **ISO-10646-String**.

EXAMPLE

```
ISO-10646-String ::= BMPString
     (FROM (Level2 INTERSECTION (BasicLatin UNION HebrewExtended UNION Hiragana)))
     --  This is the type that defines the minimum set of characters in
     --  the adopted subset for an implementation of this standard. The
     --  implementation level is required to be at least level 2.
```

In an OSI environment, the OSI Protocol Implementation Conformance Statement would then contain a simple statement that the adopted subset of ISO/IEC 10646-1 is the limited subset (and the level) defined by **ISO-10646-String**, and **ISO-10646-String** (possibly subtyped) would be used throughout the standard where ISO/IEC 10646-1 strings were to be included.

EXAMPLE CONFORMANCE STATEMENT

The adopted subset of ISO/IEC 10646-1 is the limited subset consisting of all the characters in the ASN.1 type **ISO-10646-String** defined in module <your module name goes here>, with an implementation level of 2.

EXAMPLE USE IN PROTOCOL

```
Message ::=  SEQUENCE {
     first-field ISO-10646-String,--  all characters in the adopted
                                  -- subset can appear
     second-field     ISO-10646-String
          (FROM (latinSmallLetterA .. latinSmallLetterZ)),   --  lower case
                                                  -- latin letters only
```

```
        third-field ISO-10646-String
              (FROM (digitZero .. digitNine)) --  digits only

    }
```

F.5 Adopted subsets as parameters of the abstract syntax

ISO/IEC 10646-1 requires that the adopted subset and level of an implementation be *explicitly* defined. Where an ASN.1 user does not wish to constrain the range of ISO/IEC 10646-1 characters in some part of the standard being defined, this can be expressed by defining **ISO-10646-String** (for example) as a subtype of **UniversalString**, **BMPString** or **UTF8String** with a subtype constraint consisting of (or including) **ImplementorsSubset** which is left as a parameter of the abstract syntax.

Users of ASN.1 are warned that in this case a conforming sender may transmit to a conforming receiver characters that cannot be handled by the receiver because they fall outside the (implementation-dependent) adopted subset or level of the receiver, and it is recommended that an exception-handling specification be included in the definition of **ISO-10646-String** in this case.

EXAMPLE

```
ISO-10646-String {UniversalString : ImplementorsSubset, ImplementationLevel} ::=
           UniversalString (FROM((ImplementorsSubset UNION BasicLatin)
                      INTERSECTION ImplementationLevel) !characterSetProblem)
-- The adopted subset of ISO/IEC 10646-1 shall include "BasicLatin", but
-- may also include any additional characters specified in
-- "ImplementorsSubset", which is a parameter of the abstract syntax.
-- "ImplementationLevel", which is a parameter of the abstract
-- syntax defines the implementation level. A conforming receiver must be
-- prepared to recieve characters outside of its adopted subset and
-- implementation level.  In this case the exception handling specified in
-- clause <add your clause number here> for "characterSetProblem" is
-- invoked. Note that this can never be invoked by a conforming
-- receiver if the actual characters used in an instance of communication
-- are restricted to "BasicLatin".

My-Level2-String ::= ISO-10646-String { { HebrewExtended UNION Hiragana }, Level2 }
```

F.6 The CHARACTER STRING type

F.6.1 The **CHARACTER STRING** type gives complete flexibility in the choice of character set and encoding method.

NOTE – Where a single connection provides end-to-end data transfer (no relaying), and the OSI protocols are in use, then negotiation of the character sets to be used and their encoding can be accomplished as part of the definition of the OSI presentation contexts for character abstract syntaxes. Otherwise, the abstract and transfer character syntaxes (character repertoire and encodings) are announced by a pair of object identifier values.

F.6.2 In formal terms, a character abstract syntax is an ordinary abstract syntax with some restrictions on the possible values (they are all character strings, and indeed are all the character strings formed from some collection of characters). Thus allocation of object identifier values for character abstract and transfer syntaxes is performed in the normal way.

F.6.3 The encoding of **CHARACTER STRING** announces the abstract and transfer syntax of the character repertoire in use (that is, character set and encoding). In OSI environments, negotiation of both these syntaxes is possible.

F.6.4 Character abstract syntaxes (and corresponding character transfer syntaxes) have been defined in a number of ITU-T Recommendations and International Standards, and additional character abstract syntaxes (and/or character transfer syntaxes) can be defined by any organization able to allocate object identifiers.

F.6.5 In ISO/IEC 10646-1, there is a character abstract syntax defined (and object identifiers assigned) for the entire collection of characters, for each of the defined collection of characters for subsets (BASIC LATIN, BASIC SYMBOLS, etc.), and for every possible combination of the defined collections of characters. There are also two character transfer syntaxes defined to identify the various options (particularly 16-bit and 32-bit) in ISO/IEC 10646-1.

Annex G

Tutorial annex on the ASN.1 model of type extension

(This annex does not form an integral part of this Recommendation | International Standard)

G.1 Overview

G.1.1 It can happen that an ASN.1 type evolves over time from an **extension root** type by means of a series of extensions called **extension additions**.

G.1.2 An ASN.1 type available to a particular implementation may be the extension root type, or may be the extension root type plus one or more extension additions. Each such ASN.1 type that contains an extension addition also contains all previously defined extension additions.

G.1.3 The ASN.1 type definitions in this series are said to be **extension-related** (see 3.6.32 for a more precise definition of "extension-related"), and encoding rules are required to encode extension-related types in a such a way that if two systems are using two different types which are extension-related, transmissions between the two systems will successfully transfer the information content of those parts of the extension-related types that are common to the two systems. It is also required that those parts that are not common to both systems can be delimited and retransmitted (perhaps to a third party) on a subsequent transmission, provided the same transfer syntax is used.

NOTE – The sender may be using a type that is either earlier or later in the series of extension additions.

G.1.4 The series of types obtained by progressively adding to a root type is called an **extension series**. In order for encoding rules to make appropriate provision for transmissions of extension-related types (which may require more bits on the line), such types (including the extension root type) need to be syntactically flagged. The flag is an ellipsis (**...**), and is called an **extension marker**.

EXAMPLE

Extension root type

```
A ::= SEQUENCE {
      a INTEGER,
        ...
      }
```

1st extension

```
A ::= SEQUENCE {
        a INTEGER,
          ...,
        b BOOLEAN,
        c INTEGER
      }
```

2nd extension

```
A ::= SEQUENCE {
        a INTEGER,
          ...,
        b BOOLEAN,
        c INTEGER,
        d SEQUENCE {
          e INTEGER,
            ...,
            ...,
          f IA5String
        }
      }
```

3rd extension

```
A ::= SEQUENCE {
        a INTEGER,
          ...,
        b BOOLEAN,
        c INTEGER,
        d SEQUENCE {
          e INTEGER,
            ...,
          g BOOLEAN OPTIONAL,
          h BMPString,
            ...,
          f IA5String
        }
      }
```

G.1.5 All extension additions in sequence, set, and choice types are inserted between pairs of extension markers. A single extension marker is allowed if (in the extension root type) it appears as the last item in the type, in which case a matching extension marker is assumed to exist just before the closing brace of the type; in such cases all extension additions are inserted at the end of the type.

G.1.6 A type that has an extension marker can be nested inside a type that has none, or it can be nested within a type in an extension root, or it can be nested in an extension addition type. In such cases the extension series are treated independently, and the nested type with the extension marker has no impact on the type within which it is nested. Only one **extension insertion point** (the end of the type if a single extension marker is used, or just before the second extension marker if a pair of extension markers is used) can appear in any specific construct.

G.1.7 A new extension addition in the extension series is defined in terms of a single **extension addition group** (one or more types nested within "**[[**" "**]]**") or a single type added at the extension insertion point. In the following example the first extension defines an extension addition group where **b** and **c** must either be both present or both absent in a value of type **A**. The second extension defines a single component type, **d**, which may be absent in a value of type **A**. The third extension defines an extension addition group in which **h** must be present in a value of type **A** whenever the newly added extension addition group is present in a value.

EXAMPLE

Extension root type	1st extension	2nd extension	3rd extension
A ::= SEQUENCE { **a INTEGER,** **...** **}**	**A ::= SEQUENCE {** **a INTEGER,** **...,** **[[** **b BOOLEAN,** **c INTEGER** **]]** **}**	**A ::= SEQUENCE {** **a INTEGER,** **...,** **[[** **b BOOLEAN,** **c INTEGER** **]],** **d SEQUENCE {** **e INTEGER,** **...,** **...,** **f IA5String** **}** **}**	**A ::= SEQUENCE {** **a INTEGER,** **...,** **[[** **b BOOLEAN,** **c INTEGER** **]],** **d SEQUENCE {** **e INTEGER,** **...,** **[[** **g BOOLEAN OPTIONAL,** **h BMPString** **]],** **...,** **f IA5String** **}** **}**

G.1.8 It is also possible to add the version number to version brackets, but only if it is present on all brackets within a module, and only if all extensions in the module are within version brackets. It is recommended that version numbers be used. The ability to omit numbers and version brackets is for historical reasons. (Version brackets and version numbers were not allowed in earlier versions of this Recommendation | International Standard.) (See also G.3.)

G.1.9 While the normal practice will be for extension additions to be added over time, the underlying ASN.1 model and specification does not involve time. Two types are extension-related if one can be "grown" from the other by extension additions. That is, one contains all the components of the other. There may be types that have to be "grown" in the opposite direction (although this is unlikely). It could even be that, over time, a type *starts* with a lot of extension additions which were progressively removed! All that ASN.1 and its encoding rules care about is whether a pair of type specifications are extension-related or not. If they are, then *all* ASN.1 encoding rules will ensure interworking between their users.

G.1.10 We start with a type and then decide whether we are going to want interworking with implementations of earlier versions if we later have to extend it. If so, we include the extension marker *now*. We can then add later extension additions to the type with defined handling of the extended values by earlier systems. It is, however, important to note that adding an extension marker to a type that was previously without one (or removing an extension marker) may prevent interworking.

NOTE – When ECN is used, it can be possible to add extensions in version 2 at places that did not have extension markers in version 1, and still retain interworking between versions 1 and 2.

G.1.11 Table G.1 shows the ASN.1 types that can form the extension root type of an ASN.1 extension series, and the nature of the single extension addition that is permitted for that type (multiple extension additions can of course be made in succession, or together as an extension group).

Table G.1 – Extension additions

Extension root type	Nature of extension addition
ENUMERATED	Addition of a single further enumeration at the end of the "AdditionalEnumeration"s, with an enumeration value greater than that of any enumeration already added.
SEQUENCE and **SET**	Addition of a single type or extension addition group to the end of the "ExtensionAdditionList". "ComponentType"s that are extension additions (not contained in an extension addition group) are not required to be marked **OPTIONAL** or **DEFAULT**, although this will often be the case.
CHOICE	Addition of a single "NamedType" to the end of the "ExtensionAdditionAlternativesList".
Constraint notation	Addition of a single "AdditionalElementSetSpec" to the "ElementSetSpecs" notation.

G.2 Meaning of version numbers

G.2.1 Version numbers are not used in BER or PER encodings. Their use (if any) in ECN encodings is determined by the ECN specification.

G.2.2 Version numbers are most useful when they relate to the means of decoding a complete PDU, not to an individual type. Where a type which is used as a component of several protocols and hence contributes to different

complete PDUs, an addition to that type will normally require that the version number for all the PDUs to which it contributes be incremented.

G.2.3 When used to provide interworking between deployed systems, version numbers should be used on extension addition groups in such a way that deployed systems have knowledge of the syntax and semantics for all extension addition groups with a given version number (no matter where they appear within the protocol), and of all extension addition groups with an earlier version number. ECN specifiers will normally assume that version numbers have been allocated (to all parts of types to which ECN is applied) in accordance with this principle.

G.3 Requirements on encoding rules

G.3.1 An abstract syntax can be defined as the values of a single ASN.1 type that is an extensible type. It then contains all the values that can be obtained by the addition or removal of extension-additions. Such an abstract syntax is called an extension-related abstract syntax.

G.3.2 A set of well-formed encoding rules for an extension-related abstract syntax satisfies the additional requirements stated in G.3.3 to G.3.5.

NOTE – All ASN.1 encoding rules satisfy these requirements.

G.3.3 The definition of the procedures for transforming an abstract value into an encoding for transfer, and for transforming a received encoding into an abstract value shall recognize the possibility that the sender and receiver are using abstract syntaxes that are not identical, but are extension-related.

G.3.4 In this case, the encoding rules shall ensure that where the sender has a type specification that is earlier in the extension series than that of the receiver, values of the sender shall be transferred in such a way that the receiver can determine that extension additions are not present.

G.3.5 The encoding rules shall ensure that where the sender has a type specification that is later in the extension series than that of the receiver, transfer of values of that type to the receiver shall be possible.

G.4 Combination of (possibly extensible) constraints

G.4.1 Model

G.4.1.1 The basic ASN.1 model for applying constraints is simple: A type is a set of abstract values, and a constraint applied to it selects a subset of those abstract values. If the unconstrained type was not extensible, then the resulting type is defined to be extensible if and only if the applied constraint is defined to be extensible.

G.4.1.2 Even in this simple case, there is one feature to clarify: A type may be formally extensible, even though there can never be any extension additions. Consider:

```
A ::= INTEGER (MIN .. MAX, ... , 1..10)
```

As with many examples in this annex, this is something that nobody would ever write, but which tool vendors have to write code for because the ASN.1 standard has been left simple and general, and this example is therefore legal ASN.1. In this example, A is formally an extensible **INTEGER**, with the full range of integer values in the root.

G.4.1.3 Complexities arise from three main sources:

- The application of a constraint to a type that has already had an extensible constraint applied to it (serial application of constraints – see G.4.2).
- The combination of extensible constraints using **UNION** and **INTERSECTION** and **EXCEPT** (set arithmetic – see G.4.3).
- The use of a typereference (a contained subtype) in the set arithmetic of a constraint, when the typereference de-references to an extensible type (perhaps with actual extension additions – see G.4.4).

G.4.2 Serial application of constraints

G.4.2.1 Serial application of constraints occurs when a type is constrained (in an assignment to a typereference) and the typereference is subsequently used with a further constraint applied to it.

G.4.2.2 It can also, but less commonly, occur when a type has multiple constraints directly applied to it in a serial fashion. This latter form is used for many of the examples in this annex (for simplicity of exposition), but the case where a typereference links the two (or more) constraints is the form in which serial application normally occurs in real specifications.

G.4.2.3 There are two key points in the serial application of constraints:

– If a constrained type is extensible (and perhaps extended), the "extensible" flag and all extension additions are discarded if a further constraint is subsequently serially applied. The extensibility of a constrained type (and any extension additions) depends solely on the last constraint that is applied, which can reference only values in the root of the type that is being further constrained (the parent type). Values included in the root or the extension additions of the resulting type can only be values that are in the root of the parent type.

– The serial application of constraints is (for complex cases) not the same as a set arithmetic intersection, even when there is no extensibility involved. Firstly, the environment in which **MIN** and **MAX** are interpreted, and secondly the abstract values that can be referenced in the second constraint are very different in serial application from the situation where the two constraints are specified as an intersection of values from a common parent.

NOTE – Use of a range such as **20..28** in a constraint on an integer type is legal if (and only if) both **20** and **28** are in the (root of the) parent type, but the values referenced by this range specification are only those in (the root of) the parent. So if the parent has already been constrained to exclude the values **24** and **25**, the range **20..28** is referencing only **20** to **23** and **26** to **28**.

Here are some examples:

```
A1 ::= INTEGER (1..32, ... , 33..128)
      -- A1 is extensible, and contains values 1 to 128 with 1 to 32
      -- in the root and 33 to 128 as extension additions.

B1 ::= A1 (1..128)
      -- or equivalently

B1 ::= INTEGER (1..32, ... , 33..128) (1..128)
      -- These are illegal, as 128 is not in the parent, which
      -- lost its extension additions when it was further constrained

B2 ::= A1 (1..16)
      -- This is legal. B2 is not extensible, and contains 1 to 16.

A2 ::= INTEGER (1..32) (MIN .. 63)
      -- MIN is 1, and 63 is illegal

A3 ::= INTEGER ( (1..32) INTERSECTION (MIN..63) )
      -- This is legal.  MIN is minus infinity and A3 contains 1 to 32
```

G.4.3 Use of set arithmetic

G.4.3.1 The results are largely intuitive, and obey the normal mathematical rules for intersection, union and set difference (**EXCEPT**). In particular, both intersection and union are commutative, that is:

(<*some set 1 of values*> INTERSECTION <*some set 2 of values*>)

is the same as

(<*some set 2 of values*> INTERSECTION <*some set 1 of values*>)

similarly for **UNION**.

G.4.3.2 The commutativity is true, no matter what sets of values are extensible, and no matter what extension additions are present.

G.4.3.3 Misunderstandings can arise if an intersection makes it impossible for extension addition values ever to occur. This is similar to the case of **INTEGER (MIN..MAX, ...)**.

G.4.3.4 For example:

```
A ::= INTEGER ((1..256, ... , B) INTERSECTION (1..256))
-- A always contains (only) the values 1..256, no matter what values
-- B contains, but is nonetheless formally extensible
```

G.4.3.5 It is also important to remember that while parents lose their extensibility and extension additions when further constrained, and contained subtypes lose their extensibility and extension additions, sets of values directly specified in set arithmetic lose neither their extensibility nor their extension additions.

G.4.3.6 The rules for extensibility of sets of values produced by set arithmetic are clearly stated in 46.3 and 46.4, and do not depend on whether the set arithmetic makes actual extension additions possible or not.

G.4.3.7 The rules are summarized here for completeness, using **E** to denote a set of values with the "extensible" flag set and **N** to denote a set values which are formally non-extensible. The values in the root of each set are denoted by **R**, and the extension additions (if any) by **X**, and the contents of the result are shown for each case.

NOTE 1 – For the purposes of this annex and for simplicity of exposition, if a set of values is not extensible, we describe all its values as root values.

NOTE 2 – It is an illegal specification if the root of any resulting set of values used in a serially applied constraint is empty.

NOTE 3 – To avoid verbosity below, "Extensions" is used in place of the more correct "Extension additions".

G.4.3.8 The rules are:

```
N1 INTERSECTION N2 => N
        Root: R1 INTERSECTION R2
N1 INTERSECTION E2 => E
        Root: R1 INTERSECTION R2, Extensions: R1 INTERSECTION X2
E1 INTERSECTION E2 => E
        Root: R1 INTERSECTION R2, Extensions: ((R1 UNION X1)
                                              INTERSECTION
                                              (R2 UNION X2))
                                              EXCEPT
                                              (R1 INTERSECTION R2)
N1 UNION N2 => N
        Root: R1 UNION R2
N1 UNION E2 => E
        Root: R1 UNION R2, Extensions: X2
E1 UNION E2 => E
        Root: R1 UNION R2, Extensions: (R1 UNION X1 UNION R2 UNION X2)
                                       EXCEPT
                                       (R1 UNION R2)
N1 EXCEPT N2 => N
        Root: R1 EXCEPT R2
N1 EXCEPT E2 => N
        Root: R1 EXCEPT R2
E1 EXCEPT N2 => E
        Root: R1 EXCEPT R2, Extensions: (X1 EXCEPT R2)
                                        EXCEPT
                                        (R1 EXCEPT R2)
E1 EXCEPT E2 => E
        Root: R1 EXCEPT R2, Extensions: (X1 EXCEPT (R2 UNION X2) )
                                        EXCEPT
                                        (R1 EXCEPT R2)
N1 ... N2 => E
        Root: R1, Extensions: R2 EXCEPT R1
E1 ... N2 => E
        Root: R1, Extensions: X1 UNION R2
                              EXCEPT
                              R1
N1 ... E2 => E
        Root: R1, Extensions: R2 UNION X2
                              EXCEPT
                              R1
E1 ... E2 => E
        Root: R1, Extensions: X1 UNION R2 UNION E2
                              EXCEPT
                              R1
```

NOTE – If the result of set arithmetic on extensible sets of values does not have actual extension additions, or even can never have actual extension additions (no matter what extension additions are added to the extensible inputs), the result is still formally defined to be extensible for results **E** above.

G.4.4 Use of the Contained Subtype notation

A contained subtype may or may not be extensible, but when it is used in set arithmetic it is always treated as not extensible, and all its extension additions are discarded.

Annex H

Summary of the ASN.1 notation

(This annex does not form an integral part of this Recommendation | International Standard)

The following lexical items are defined in clause 11:

typereference
identifier
valuereference
modulereference
comment
empty
number
realnumber
bstring
hstring
cstring
xmlbstring
xmlhstring
xmlcstring
xmlasn1typename
"true"
"false"
"::="
"**[[**"
"**]]**"
"**..**"
"**...**"
"</"
"/>"
"**{**"
"**}**"
"<"
">"
","
"**.**"
"**(**"
"**)**"
"["
"]"
"**-**"
"**:**"
"="
""" (QUOTATION MARK)
"'" (APOSTROPHE)
" " (SPACE)
";"
"@"
"|"
"**!**"
"**^**"
ABSENT
ABSTRACT-SYNTAX
ALL
APPLICATION
AUTOMATIC
BEGIN
BIT
BMPString
BOOLEAN
BY
CHARACTER
CHOICE
CLASS
COMPONENT
COMPONENTS
CONSTRAINED
CONTAINING
DEFAULT
DEFINITIONS
EMBEDDED
ENCODED
END
ENUMERATED
EXCEPT
EXPLICIT
EXPORTS
EXTENSIBILITY
EXTERNAL
FALSE
FROM
GeneralizedTime
GeneralString
GraphicString
IA5String
IDENTIFIER
IMPLICIT
IMPLIED
IMPORTS
INCLUDES
INSTANCE
INTEGER
INTERSECTION
ISO646String
MAX
MIN
MINUS-INFINITY
NULL
NumericString
OBJECT
ObjectDescriptor
OCTET
OF
OPTIONAL
PATTERN
PDV
PLUS-INFINITY
PRESENT
PrintableString
PRIVATE
REAL
RELATIVE-OID
SEQUENCE
SET
SIZE
STRING
SYNTAX

T61String

TAGS

TeletexString

TRUE

TYPE-IDENTIFIER

UNION

UNIQUE

UNIVERSAL

UniversalString

UTCTime

UTF8String

VideotexString

VisibleString

WITH

The following productions are used in this Recommendation | International Standard, with the above lexical items as terminal symbols:

```
ModuleDefinition ::= ModuleIdentifier
                     DEFINITIONS
                     TagDefault
                     ExtensionDefault
                     "::="
                     BEGIN
                     ModuleBody
                     END

ModuleIdentifier ::=  modulereference
                          DefinitiveIdentifier

DefinitiveIdentifier ::= "{" DefinitiveObjIdComponentList "}"
    |                 empty

DefinitiveObjIdComponentList ::=
          DefinitiveObjIdComponent
    |     DefinitiveObjIdComponent DefinitiveObjIdComponentList

DefinitiveObjIdComponent ::=
          NameForm
    |     DefinitiveNumberForm
    |     DefinitiveNameAndNumberForm

DefinitiveNumberForm     ::= number

DefinitiveNameAndNumberForm ::= identifier "(" DefinitiveNumberForm ")"

TagDefault ::=
          EXPLICIT TAGS
    |     IMPLICIT TAGS
    |     AUTOMATIC TAGS
    |     empty

ExtensionDefault ::=
          EXTENSIBILITY IMPLIED | empty

ModuleBody ::=  Exports Imports AssignmentList
    |           empty

Exports ::=     EXPORTS SymbolsExported ";"
    |           EXPORTS ALL ";"
    |           empty

SymbolsExported ::=  SymbolList
    |                empty

Imports ::=     IMPORTS SymbolsImported ";"
    |           empty

SymbolsImported ::=  SymbolsFromModuleList
    |                empty

SymbolsFromModuleList ::=
                     SymbolsFromModule
    |                SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=    SymbolList FROM GlobalModuleReference

GlobalModuleReference ::= modulereference AssignedIdentifier

AssignedIdentifier ::=  ObjectIdentifierValue
    |                   DefinedValue
    |                   empty

SymbolList   ::= Symbol | SymbolList "," Symbol
```

```
Symbol  ::= Reference | ParameterizedReference

Reference ::=
          typereference
     |    valuereference
     |    objectclassreference
     |    objectreference
     |    objectsetreference

AssignmentList ::=          Assignment  | AssignmentList Assignment

Assignment ::=
          TypeAssignment
     |    ValueAssignment
     |    XMLValueAssignment
     |    ValueSetTypeAssignment
     |    ObjectClassAssignment
     |    ObjectAssignment
     |    ObjectSetAssignment
     |    ParameterizedAssignment

DefinedType ::=
          ExternalTypeReference
     |    typereference
     |    ParameterizedType
     |    ParameterizedValueSetType

ExternalTypeReference ::=
                    modulereference
                    "."
                    typereference

NonParameterizedTypeName ::=
          ExternalTypeReference
     |    typereference
     |    xmlasn1typename

DefinedValue ::=
          ExternalValueReference
     |    valuereference
     |    ParameterizedValue

ExternalValueReference ::=
                    modulereference
                    "."
                    valuereference

AbsoluteReference ::=
                    "@" ModuleIdentifier
                    "."
                    ItemSpec

ItemSpec ::=
          typereference
     |    ItemId "." ComponentId

ItemId ::= ItemSpec

ComponentId ::=
          identifier | number | "*"

TypeAssignment ::=    typereference
                           "::="
                           Type

ValueAssignment ::=   valuereference
                           Type
```

```
                    ":: ="
                    Value

XMLValueAssignment ::=
        valuereference
        ":: ="
        XMLTypedValue

XMLTypedValue ::=
        "<" & NonParameterizedTypeName ">"
        XMLValue
        "</" & NonParameterizedTypeName ">"
    |   "<" & NonParameterizedTypeName "/>"

ValueSetTypeAssignment ::=  typereference
                    Type
                    ":: ="
                    ValueSet

ValueSet ::= "{" ElementSetSpecs "}"

Type ::= BuiltinType | ReferencedType | ConstrainedType

BuiltinType ::=
        BitStringType
    |   BooleanType
    |   CharacterStringType
    |   ChoiceType
    |   EmbeddedPDVType
    |   EnumeratedType
    |   ExternalType
    |   InstanceOfType
    |   IntegerType
    |   NullType
    |   ObjectClassFieldType
    |   ObjectIdentifierType
    |   OctetStringType
    |   RealType
    |   RelativeOIDType
    |   SequenceType
    |   SequenceOfType
    |   SetType
    |   SetOfType
    |   TaggedType

NamedType ::= identifier  Type

ReferencedType ::=
        DefinedType
    |   UsefulType
    |   SelectionType
    |   TypeFromObject
    |   ValueSetFromObjects

Value  ::=  BuiltinValue | ReferencedValue | ObjectClassFieldValue

XMLValue ::= XMLBuiltinValue | XMLObjectClassFieldValue

BuiltinValue ::=
        BitStringValue
    |   BooleanValue
    |   CharacterStringValue
    |   ChoiceValue
    |   EmbeddedPDVValue
    |   EnumeratedValue
    |   ExternalValue
```

```
        |   InstanceOfValue
        |   IntegerValue
        |   NullValue
        |   ObjectIdentifierValue
        |   OctetStringValue
        |   RealValue
        |   RelativeOIDValue
        |   SequenceValue
        |   SequenceOfValue
        |   SetValue
        |   SetOfValue
        |   TaggedValue

XMLBuiltinValue ::=
            XMLBitStringValue
        |   XMLBooleanValue
        |   XMLCharacterStringValue
        |   XMLChoiceValue
        |   XMLEmbeddedPDVValue
        |   XMLEnumeratedValue
        |   XMLExternalValue
        |   XMLInstanceOfValue
        |   XMLIntegerValue
        |   XMLNullValue
        |   XMLObjectIdentifierValue
        |   XMLOctetStringValue
        |   XMLRealValue
        |   XMLRelativeOIDValue
        |   XMLSequenceValue
        |   XMLSequenceOfValue
        |   XMLSetValue
        |   XMLSetOfValue
        |   XMLTaggedValue

ReferencedValue ::=
            DefinedValue
        |   ValueFromObject

NamedValue ::= identifier Value

XMLNamedValue ::=
            "<" & identifier ">" XMLValue "</" & identifier ">"

BooleanType ::=BOOLEAN

BooleanValue::= TRUE | FALSE

XMLBooleanValue ::=
            "<" & "true" "/>"
        |   "<" & "false" "/>"

IntegerType ::=
            INTEGER
        |   INTEGER "{" NamedNumberList "}"

NamedNumberList ::=
            NamedNumber
        |   NamedNumberList "," NamedNumber

NamedNumber ::=
            identifier "(" SignedNumber ")"
        |   identifier "(" DefinedValue ")"

SignedNumber ::= number | "-" number

IntegerValue ::=  SignedNumber | identifier
```

```
XMLIntegerValue ::=
        SignedNumber
    |   "<" & identifier "/>"

EnumeratedType ::=
        ENUMERATED "{" Enumerations "}"

Enumerations ::= RootEnumeration
    |   RootEnumeration "," "..." ExceptionSpec
    |   RootEnumeration "," "..." ExceptionSpec "," AdditionalEnumeration

RootEnumeration ::= Enumeration

AdditionalEnumeration ::= Enumeration

Enumeration ::= EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem ::= identifier | NamedNumber

EnumeratedValue ::= identifier

XMLEnumeratedValue ::= "<" & identifier "/>"

RealType ::= REAL

RealValue ::=
        NumericRealValue | SpecialRealValue

NumericRealValue ::=
        realnumber
    |   "-" realnumber
    |   SequenceValue          -- Value of the associated sequence type

SpecialRealValue ::=
        PLUS-INFINITY | MINUS-INFINITY

XMLRealValue ::=
        XMLNumericRealValue | XMLSpecialRealValue

XMLNumericRealValue ::=
        realnumber
    |   "-" realnumber

XMLSpecialRealValue ::=
        "<" & PLUS-INFINITY "/>" | "<" & MINUS-INFINITY "/>"

BitStringType   ::=     BIT STRING | BIT STRING "{" NamedBitList "}"

NamedBitList::=     NamedBit | NamedBitList "," NamedBit

NamedBit    ::= identifier "(" number ")"
    |           identifier "(" DefinedValue ")"

BitStringValue  ::=  bstring | hstring | "{" IdentifierList "}" | "{" "}" | CONTAINING Value

IdentifierList  ::=  identifier | IdentifierList "," identifier

XMLBitStringValue ::=
        XMLTypedValue
    |   xmlbstring
    |   XMLIdentifierList
    |   empty

XMLIdentifierList ::=
        "<" & identifier "/>"
    |   XMLIdentifierList "<" & identifier "/>"

OctetStringType ::= OCTET STRING

OctetStringValue ::= bstring | hstring | CONTAINING Value
```

```
XMLOctetStringValue ::=
        XMLTypedValue
    |   xmlhstring

NullType    ::=  NULL

NullValue   ::=  NULL

XMLNullValue ::= empty

SequenceType ::=
        SEQUENCE "{" "}"
    |   SEQUENCE "{" ExtensionAndException OptionalExtensionMarker "}"
    |   SEQUENCE "{" ComponentTypeLists "}"

ExtensionAndException ::= "..." | "..." ExceptionSpec

OptionalExtensionMarker ::= "," "..." | empty

ComponentTypeLists ::=
        RootComponentTypeList
    |   RootComponentTypeList "," ExtensionAndException ExtensionAdditions
            OptionalExtensionMarker
    |   RootComponentTypeList "," ExtensionAndException ExtensionAdditions
            ExtensionEndMarker "," RootComponentTypeList
    |   ExtensionAndException ExtensionAdditions ExensionEndMarker ","
            RootComponentTypeList
    |   ExtensionAndException ExtensionAdditions OptionalExtensionMarker

RootComponentTypeList ::= ComponentTypeList

ExtensionEndMarker ::= "," "..."

ExtensionAdditions ::= "," ExtensionAdditionList | empty

ExtensionAdditionList ::= ExtensionAddition
    |   ExtensionAdditionList "," ExtensionAddition

ExtensionAddition ::= ComponentType | ExtensionAdditionGroup

ExtensionAdditionGroup ::= "[[" VersionNumber ComponentTypeList "]]"

VersionNumber ::= empty | number ":"

ComponentTypeList ::=   ComponentType
    |                   ComponentTypeList "," ComponentType

ComponentType ::=
        NamedType
    |   NamedType OPTIONAL
    |   NamedType DEFAULT Value
    |   COMPONENTS OF Type

SequenceValue   ::=  "{" ComponentValueList "}" | "{" "}"

ComponentValueList ::=
        NamedValue
    |   ComponentValueList "," NamedValue

XMLSequenceValue ::=
        XMLComponentValueList
    |   empty

XMLComponentValueList ::=
        XMLNamedValue
    |   XMLComponentValueList XMLNamedValue

SequenceOfType ::=  SEQUENCE OF Type | SEQUENCE OF NamedType

SequenceOfValue ::=  "{" ValueList "}" | "{" NamedValueList "}" | "{" "}"
```

```
ValueList   ::=     Value | ValueList "," Value

XMLSequenceOfValue ::=
            XMLValueList
      |     XMLDelimitedItemList
      |     XMLSpaceSeparatedList
      |     empty

XMLValueList ::=
            XMLValueOrEmpty
      |     XMLValueOrEmpty XMLValueList

XMLValueOrEmpty ::=
            XMLValue
      |     "<" & NonParameterizedTypeName "/>"

XMLSpaceSeparatedList ::=
            XMLValueOrEmpty
      |     XMLValueOrEmpty " " XMLSpaceSeparatedList

XMLDelimitedItemList ::=
            XMLDelimitedItem
      |     XMLDelimitedItem XMLDelimitedItemList

XMLDelimitedItem ::=
            "<" & NonParameterizedTypeName ">" XMLValue
                  "</" & NonParameterizedTypeName ">"
      |     "<" & identifier ">" XMLValue "</" & identifier ">"

SetType ::=   SET "{" "}"
      |       SET "{" ExtensionAndException OptionalExtensionMarker "}"
      |       SET "{" ComponentTypeLists "}"

SetValue    ::=   "{" ComponentValueList "}" | "{" "}"

XMLSetValue ::= XMLComponentValueList   |   empty

SetOfType   ::=   SET OF Type | SET OF NamedType

SetOfValue  ::=   "{" ValueList "}" | "{" NamedValueList "}" | "{" "}"

XMLSetOfValue ::=
            XMLValueList
      |     XMLDelimitedItemList
      |     XMLSpaceSeparatedList
      |     empty

ChoiceType   ::= CHOICE "{" AlternativeTypeLists "}"

AlternativeTypeLists ::=
            RootAlternativeTypeList
      |     RootAlternativeTypeList ","
                  ExtensionAndException ExtensionAdditionAlternatives OptionalExtensionMarker

RootAlternativeTypeList ::= AlternativeTypeList

ExtensionAdditionAlternatives ::= "," ExtensionAdditionAlternativesList | empty

ExtensionAdditionAlternativesList ::= ExtensionAdditionAlternative
      |       ExtensionAdditionAlternativesList "," ExtensionAdditionAlternative

ExtensionAdditionAlternative ::= ExtensionAdditionAlternativesGroup | NamedType

ExtensionAdditionAlternativesGroup ::= "[[" VersionNumber AlternativeTypeList "]]"

AlternativeTypeList ::=             NamedType
      |                             AlternativeTypeList "," NamedType

ChoiceValue ::=         identifier ":" Value

XMLChoiceValue ::= "<" & identifier ">" XMLValue "</" & identifier ">"
```

```
SelectionType   ::=         identifier "<" Type

TaggedType   ::= Tag Type
     |                  Tag IMPLICIT Type
     |                  Tag EXPLICIT Type

Tag     ::=             "[" Class ClassNumber "]"

ClassNumber  ::= number | DefinedValue

Class   ::=   UNIVERSAL
     |        APPLICATION
     |        PRIVATE
     |        empty

TaggedValue  ::=  Value

XMLTaggedValue ::= XMLValue

EmbeddedPDVType ::=             EMBEDDED PDV

EmbeddedPDVValue ::=            SequenceValue

XMLEmbeddedPDVValue ::= XMLSequenceValue

ExternalType ::= EXTERNAL

ExternalValue ::= SequenceValue

XMLExternalValue ::= XMLSequenceValue

ObjectIdentifierType  ::=  OBJECT IDENTIFIER

ObjectIdentifierValue ::= "{" ObjIdComponentsList "}"
     |                    "{" DefinedValue ObjIdComponentsList "}"

ObjIdComponentsList  ::=  ObjIdComponents
     |                    ObjIdComponents ObjIdComponentsList

ObjIdComponents  ::=NameForm
     |              NumberForm
     |              NameAndNumberForm
     |              DefinedValue

NameForm     ::=    identifier

NumberForm    ::=       number | DefinedValue

NameAndNumberForm ::=     identifier "(" NumberForm ")"

XMLObjectIdentifierValue ::=
          XMLObjIdComponentList

XMLObjIdComponentList ::=
          XMLObjIdComponent
     |    XMLObjIdComponent & "." & XMLObjIdComponentList

XMLObjIdComponent ::=
          NameForm
     |    XMLNumberForm
     |    XMLNameAndNumberForm

XMLNumberForm ::= number

XMLNameAndNumberForm ::=
          identifier & "(" & XMLNumberForm & ")"

RelativeOIDType ::=
               RELATIVE-OID

RelativeOIDValue ::=
               "{" RelativeOIDComponentsList "}"
```

```
RelativeOIDComponentsList ::=
        RelativeOIDComponents
    |   RelativeOIDComponents RelativeOIDComponentsList

RelativeOIDComponents ::=   NumberForm
    |                       NameAndNumberForm
    |                       DefinedValue

XMLRelativeOIDValue ::=
        XMLRelativeOIDComponentList

XMLRelativeOIDComponentList ::=
        XMLRelativeOIDComponent
    |   XMLRelativeOIDComponent & "." & XMLRelativeOIDComponentList

XMLRelativeOIDComponent ::=
        XMLNumberForm
    |   XMLNameAndNumberForm

CharacterStringType ::= RestrictedCharacterStringType | UnrestrictedCharacterStringType

RestrictedCharacterStringType ::=
        BMPString
    |   GeneralString
    |   GraphicString
    |   IA5String
    |   ISO646String
    |   NumericString
    |   PrintableString
    |   TeletexString
    |   T61String
    |   UniversalString
    |   UTF8String
    |   VideotexString
    |   VisibleString

RestrictedCharacterStringValue ::= cstring | CharacterStringList | Quadruple | Tuple

CharacterStringList  ::=  "{" CharSyms "}"

CharSyms  ::=  CharsDefn | CharSyms "," CharsDefn

CharsDefn  ::=  cstring | Quadruple | Tuple | DefinedValue

Quadruple    ::= "{" Group "," Plane "," Row "," Cell "}"

Group        ::= number

Plane        ::= number

Row          ::= number

Cell         ::= number

Tuple ::= "{" TableColumn "," TableRow "}"

TableColumn ::= number

TableRow ::= number

XMLRestrictedCharacterStringValue ::= xmlcstring

UnrestrictedCharacterStringType ::= CHARACTER STRING

CharacterStringValue ::= RestrictedCharacterStringValue | UnrestrictedCharacterStringValue

XMLCharacterStringValue ::=
    XMLRestrictedCharacterStringValue
    |XMLUnrestrictedCharacterStringValue

UnrestrictedCharacterStringValue ::= SequenceValue
```

XMLUnrestrictedCharacterStringValue ::= XMLSequenceValue

UsefulType ::= typereference

The following character string types are defined in 37.1:

```
NumericString      VisibleString
PrintableString    ISO646String
TeletexString      IA5String
T61String          GraphicString
VideotexString     GeneralString
UniversalString    BMPString
```

The following useful types are defined in clauses 42 to 44:

```
GeneralizedTime

UTCTime

ObjectDescriptor
```

The following productions are used in clauses 45 to 47:

```
ConstrainedType ::=
        Type   Constraint
    |   TypeWithConstraint

TypeWithConstraint ::=
        SET Constraint OF Type
    |   SET SizeConstraint OF Type
    |   SEQUENCE Constraint OF Type
    |   SEQUENCE SizeConstraint OF Type
    |   SET Constraint OF NamedType
    |   SET SizeConstraint OF NamedType
    |   SEQUENCE Constraint OF NamedType
    |   SEQUENCE SizeConstraint OF NamedType

Constraint ::= "(" ConstraintSpec  ExceptionSpec ")"

ConstraintSpec ::=      SubtypeConstraint
    |                   GeneralConstraint

ExceptionSpec ::= "!"  ExceptionIdentification | empty

ExceptionIdentification ::= SignedNumber
    |                       DefinedValue
    |                       Type ":" Value

SubtypeConstraint ::= ElementSetSpecs

ElementSetSpecs ::=
        RootElementSetSpec
    |   RootElementSetSpec  ","  "..."
    |   RootElementSetSpec  ","  "..."  ","  AdditionalElementSetSpec

RootElementSetSpec ::= ElementSetSpec

AdditionalElementSetSpec ::= ElementSetSpec

ElementSetSpec ::= Unions | ALL  Exclusions

Unions ::=      Intersections
    |           UElems UnionMark Intersections

UElems ::= Unions

Intersections ::=   IntersectionElements
    |               IElems IntersectionMark IntersectionElements
```

```
IElems ::= Intersections

IntersectionElements ::= Elements | Elems Exclusions

Elems ::= Elements

Exclusions ::= EXCEPT Elements

UnionMark  ::=  "|"   |    UNION

IntersectionMark  ::=  "^" |     INTERSECTION

Elements  ::= SubtypeElements
     |          ObjectSetElements
     |          "(" ElementSetSpec ")"

SubtypeElements ::=
          SingleValue
     |    ContainedSubtype
     |    ValueRange
     |    PermittedAlphabet
     |    SizeConstraint
     |    TypeConstraint
     |    InnerTypeConstraints
     |    PatternConstraint

SingleValue   ::=  Value

ContainedSubtype ::= Includes Type

Includes ::= INCLUDES | empty

ValueRange ::= LowerEndpoint ".." UpperEndpoint

LowerEndpoint  ::=  LowerEndValue | LowerEndValue "<"

UpperEndpoint  ::=  UpperEndValue | "<" UpperEndValue

LowerEndValue ::= Value | MIN

UpperEndValue ::= Value | MAX

SizeConstraint ::= SIZE Constraint

PermittedAlphabet ::= FROM Constraint

TypeConstraint ::= Type

InnerTypeConstraints ::=
          WITH COMPONENT SingleTypeConstraint
     |    WITH COMPONENTS MultipleTypeConstraints

SingleTypeConstraint::= Constraint

MultipleTypeConstraints ::= FullSpecification | PartialSpecification

FullSpecification   ::= "{" TypeConstraints "}"

PartialSpecification ::= "{"  "..."  ","  TypeConstraints "}"

TypeConstraints ::=    NamedConstraint
     |                 NamedConstraint "," TypeConstraints

NamedConstraint ::=  identifier ComponentConstraint

ComponentConstraint ::= ValueConstraint PresenceConstraint

ValueConstraint ::=  Constraint | empty

PresenceConstraint  ::= PRESENT | ABSENT | OPTIONAL | empty

PatternConstraint ::= PATTERN Value
```

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	**Data networks and open system communications**
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2003