INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.680

(12/97)

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

(Previously CCITT Recommendation)

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 SYSTEM 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.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 ITU-T List of Recommendations.

INTERNATIONAL STANDARD 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. It is particularly, but not exclusively, applicable to application layer protocols.

Source

The ITU-T Recommendation X.680 was approved on the 12th of December 1997. The identical text is also published as ISO/IEC International Standard 8824-1.

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The 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 Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 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

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The 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, the ITU 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 class of information (for example, numeric, textual, still image or video information). A *data value* (or *value* for short) is an instance of such a class. This Recommendation | International Standard defines several basic types and their corresponding values, and rules for combining them into more complex types and values.

Although this standard notation is defined within the OSI framework, it can be used for many other purposes. In the lower layers of the OSI Basic Reference Model (see ITU-T Rec. X.200 | ISO/IEC 7498-1) and in many other protocol architectures, each message is specified as the binary value of a sequence of octets. In the Presentation layer of OSI (see ITU-T Rec. X.216 | ISO/IEC 8822), the nature of user data parameters changes. However, Application layer standards need to define quite complex data types to carry their messages, without concern for their binary representation. In order to specify the data types, they require a notation which does not necessarily determine the representation of each value. Such notation has to be supplemented by the specification of one or more algorithms called **encoding rules** which determine the value of the lower layer octets that carry the Application data (called the **transfer syntax**). The Presentation layer protocol of OSI (see ITU-T Rec. X.226 | ISO/IEC 8823-1) can negotiate which transfer syntaxes (**encodings**) are to be used.

Outside the context of OSI there is increasing recognition of the notion of an abstract value of some class (e.g. a particular 256-colour picture) divorced from the details of any particular encoding where in order to correctly interpret the bit-pattern representation of the value, it is necessary to know (usually from the context), the type (class) of the value being represented, as well as the encoding mechanism being employed. Thus, the identification of a type is an important part of this Recommendation | International Standard.

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.

Four classes of tag are specified in the notation.

The first is the **universal** class. Universal class tags are only used as specified within this Recommendation | International Standard, and each tag is either:

a) assigned to a single type; or

b) assigned to a construction mechanism.

Users of this notation are not allowed to explicitly specify universal class tags in their ASN.1 specifications, for these tags are built-in and can be specified explicitly only in this Recommendation | International Standard.

The other three classes of tag are called **application** class tags, **private** class tags, and **context-specific** class tags. There is no formal difference between use of tags from these 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 C.2.12 on the way in which the classes are usually employed.

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.

NOTE – 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 the notation to be used for specifying values of types defined using ASN.1.

Clauses 32 to 33 (inclusive) define the types supported by ASN.1 for carrying within them the complete encoding of ASN.1 types.

Clauses 34 to 39 (inclusive) define the character string types.

Clauses 40 to 43 (inclusive) define certain types which are considered to be of general utility, but which require no additional encoding rules.

Clauses 44 and 48 define a notation which enables subtypes to be defined from the values of a parent type.

Annex A forms an integral part of this Recommendation | International Standard, and gives guidance on how users of this Recommendation | International Standard can refer to ASN.1 types and values defined using CCITT Rec. X.208 | ISO/IEC 8824.

Annex B forms an integral part of this Recommendation | International Standard, and records object identifier and object descriptor values assigned in this Recommendation | International Standard.

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

Annex E does not form an integral part of this Recommendation | International Standard, and describes features of the previous version of ASN.1 that have been superseded.

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

Annex G 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;
- defines a number of useful types (using ASN.1), which can be referenced by users of ASN.1;

The ASN.1 notation can be applied whenever it is necessary to define the abstract syntax of information. It is particularly, but not exclusively, applicable to application protocols.

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

- ITU-T Recommendation X.200 (1994) | ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model.*
- ITU-T Recommendation X.216 (1994) | ISO/IEC 8822:1994, *Information technology – Open Systems Interconnection – Presentation service definition.*
- ITU-T Recommendation X.226 (1994) | ISO/IEC 8823-1:1994, *Information technology – Open Systems Interconnection – Connection-oriented presentation protocol: Protocol specification.*
- ITU-T Recommendation X.660 (1992)/Amd.2(1997) | ISO/IEC 9834-1:1993/Amd.2:1998*, Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: General procedures (plus Amendments 1 and 2).*

- ITU-T Recommendation X.681 (1997) | ISO/IEC 8824-2:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.

- ITU-T Recommendation X.682 (1997) | ISO/IEC 8824-3:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification*.

- ITU-T Recommendation X.683 (1997) | ISO/IEC 8824-4:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.

- ITU-T Recommendation X.690 (1997) | ISO/IEC 8825-1:1998, *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 (1997) | ISO/IEC 8825-2:1998, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*.

2.2 Paired Recommendations | International Standards equivalent in technical content

- CCITT Recommendation X.208 (1988), *Specification of Abstract Syntax Notation One (ASN.1)*.

  ISO/IEC 8824:1990, *Information technology – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*.

2.3 Additional references

- CCITT Recommendation T.61 (1988), *Character repertoire and coded character sets for the international teletex service*.

- 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 6523:1984, *Data interchange – Structures for the identification of organizations*.

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

- ISO 8601:1988, *Data elements and interchange formats – Information interchange – Representation of dates and times*.

- ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS – Part 1: Architecture and Basic Multilingual Plane*.

- ISO/IEC 10646-1:1993/Amd.2:1996, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane – Amendment 2: UCS Transformation Format 8 (UTF-8)*.

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 Presentation service definition

This Recommendation | International Standard uses the following terms defined in ITU-T Rec. X.216 | ISO/IEC 8822:

a) (an) abstract syntax;

b) abstract syntax name;

c) defined context set;

d) presentation data value;

e) (a) transfer syntax;

f) transfer syntax name.

3.5 Presentation protocol specification

This Recommendation | International Standard uses the following term defined in ITU-T Rec. X.226 | ISO/IEC 8823-1:

– presentation context identifier.

3.6 Structure for identification of organizations

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

a) issuing organization;

b) organization code;

c) International Code Designator.

3.7 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.

3.8 Additional definitions

3.8.1 abstract character: 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; or

c) the definition of functions associated with the character when used in particular environments.

3.8.2 abstract value: A value whose definition is based only on the type, independent of how it is represented in any encoding rule.

NOTE – Use of the term "abstract value" is frequently an assertion that what is being said probably varies based upon the encoding rules used.

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

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

3.8.5 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.

3.8.6 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 the embedded-pdv type will in general provide a more flexible mechanism for announcement or agreement on the nature of the encodings than the bitstring type.

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

3.8.8 character: A member of a set of elements used for the organization, control or representation of data.

NOTE – For example, this implies that an accent combining character and lower case 'e' are two characters in the ISO 646 French Version, and not the single character é.

3.8.9 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.

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

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

3.8.12 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.

3.8.13 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.

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

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

3.8.16 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 – NEWLINE and TAB are examples of control characters that have been assigned a formatting function in a printing environment. DLE is an example of a control character that has been assigned a function in a communication environment.

3.8.17 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 Recommendation 460-2 of the Consultative Committee on International Radio (CCIR). CCIR has also defined the acronym for Coordinated Universal Time as UTC.

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

3.8.18 element: A member of an element class, distinguishable from all other elements of the same class.

3.8.19 element class: A type (whose elements are its values) or information object class (whose elements are all possible objects of that class).

3.8.20 element set: One or more elements of the same element class.

3.8.21 embedded-pdv type: A type whose set of values is the union of the sets of values in all possible abstract syntaxes. This type 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 as well as an identification of the encoding rule used to encode the value.

3.8.22 encoding: The bit-pattern resulting from the application of a set of encoding rules to a value of a specific abstract syntax.

3.8.23 (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.

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

3.8.25 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 a 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.)

3.8.26 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.

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

3.8.28 extensible constraint: A subtype constraint with an extension marker.

3.8.29 extension 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.

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

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

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

3.8.33 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.

3.8.34 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.

NOTE – Both nested and unnested types can be extended.

3.8.35 extensible type: A type with an extension marker.

3.8.36 external reference: A type reference, value reference, information object, etc., 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

3.8.37 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.

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

3.8.39 governing; governor: Relative to some object, object set, value set, value or subtype, the information object class or type which controls its interpretation by restricting the items(s) involved to be value notation of that class or type, respectively.

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

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

3.8.41 items: Named sequences of characters from the ASN.1 character set, specified in clause 11, which are used to form the ASN.1 notation.

3.8.42 module: One or more instances of the use of the ASN.1 notation for type, value, etc., encapsulated using the ASN.1 module notation (see clause 12).

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

3.8.44 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.

3.8.45 object descriptor type: A type whose distinguished values are human-readable text providing a brief description of an object.

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

3.8.46 object identifier: A value (distinguishable from all other such values) which is associated with an object.

3.8.47 object identifier type: A simple type whose distinguished values are the set of all object identifiers allocated in accordance with the rules of ITU-T Rec. X.660 | ISO/IEC 9834-1.

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.

3.8.48 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.

3.8.49 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, where the "FieldName" denotes either a type field or a variable-type value field. The "ANY" notation which was defined in CCITT Rec. X.208 | ISO/IEC 8824 was an open type notation.

3.8.50 parent type (of a subtype): The type that is being constrained when defining a subtype.

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

3.8.51 production: A part of the formal notation used to specify ASN.1.

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

3.8.53 recursive definitions: 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.

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

3.8.55 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.

3.8.56 sequence types: Types defined by referencing an 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.

3.8.57 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.

3.8.58 set types: Types defined by referencing a fixed, unordered, list of distinct 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 of the component types.

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

3.8.59 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.

3.8.60 simple types: Types defined by directly specifying the set of its values.

3.8.61 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 3.8.16).

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

3.8.62 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).

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

3.8.64 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.

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

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

3.8.67 type: A named set of values.

3.8.68 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.

3.8.69 unrestricted character string type: A type whose values are values from a character abstract syntax identified separately for each instance of use of that type.

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

3.8.71 value: A distinguished member of a set of values.

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

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

3.8.74 version brackets: A pair of adjacent left and right brackets ([[or]]) used to delineate the start and end of an extension addition group.

3.8.75 white-space: Any formatting action that yields a space on a printed page, such as the SPACE or TAB character, or multiple uses of such characters.

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
DCC	Data Country Code
DNIC	Data Network Identification Code
ICD	International Code Designator
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
PDV	Presentation Data Value
PER	Packed Encoding Rules of ASN.1
ROA	Recognized Operating Agency
UCS	Universal Multiple-Octet Coded Character Set

5 Notation

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

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

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

In order to specify these collections, this Recommendation | International Standard uses a formal notation defined in the following subclauses.

5.1 Productions

A new (more complex) collection of ASN.1 sequences is defined by means of a production. This uses the names of collections of production sequences defined in this Recommendation | International Standard and forms a new collection of production sequences by specifying either:

a) that the new collection of production sequences is to consist of any sequence contained in any of the original collections; or

b) that the new collection is to consist of any production sequence which can be generated by taking exactly one production sequence from each collection, and juxtaposing them in a specified order.

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

a) a name for the new collection of production sequences;

b) the characters

```
::=
```

c) one or more alternative collections of production sequences, defined as in 5.2, separated by the character

```
|
```

A production sequence is present in the new collection if it is present in one or more of the alternative collections. The new collection is referenced in this Recommendation | International Standard by the name in a) above.

NOTE – If the same production sequence appears in more than one alternative, any semantic ambiguity in the resulting notation is resolved by other parts of the complete ASN.1 production sequence.

5.2 The alternative collections

Each of the alternative collections of production sequences in "one or more alternative collections of" [see 5.1.c)] is specified by a list of names. Each name is either the name of an item, or is the name of a collection of production sequences defined by a production in this Recommendation | International Standard.

The collection of production sequences defined by the alternative consists of all production sequences obtained by taking any one of the production sequences (or the item) associated with the first name, in combination with (and followed by) any one of the production sequences (or item) associated with the second name, in combination with (and followed by) any one of the production sequences (or item) associated with the third name, and so on up to and including the last name (or item) in the alternative.

5.3 Example of a production

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

is a production which associates with the name "BitStringValue" the following production sequences:

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

b) any "hstring" (an item); or

c) any production sequence associated with "IdentifierList", preceded by a "{" and followed by a "}".

NOTE – "{" and "}" are the names of items containing the single characters { and } (see 11.17).

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

5.4 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.

5.5 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 unbounded collection of allowed sequences, some or all of which may themselves be unbounded. This is not an error.

5.6 References to a collection of sequences

This Recommendation | International Standard references a collection of sequences (part of the ASN.1 notation) by referencing the first name (that appears before the "::=") in a production; the name is surrounded by the double-quote character (") to distinguish it from natural language text, unless it appears as part of a production.

5.7 References to an item

This Recommendation | International Standard references an item by referencing the name of the item; the name is surrounded by the double-quote character (") to distinguish it from natural language text, unless it appears as part of a production and is not a single character item, "::=", "..", or "...".

5.8 Short-hand notations

In order to make productions more concise and more readable, the following short-hand notations are used in the definition of the collections of ASN.1 production sequences 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 (it is not used anywhere in this Recommendation | International Standard):

a) An asterisk (*) following two names, "A" and "B", denotes the empty item (see 11.7), or a production sequence associated with "A", or an alternating series of production sequences associated with "A" and "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 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 item (see 11.7) or a production sequence associated with "A". Thus:

   F ::= A ?

   is equivalent to:

   F ::= empty | A

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 a matter for the application layer designer to specify.

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

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 – All variants of the Basic Encoding Rules of ASN.1 and the Packed Encoding Rules of ASN.1 satisfy all these requirements.

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.

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

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 13-15	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

8.3 Some encoding rules require a canonical order for tags. To provide uniformity, a canonical order for tags is defined in 8.4.

8.4 The canonical order for tags 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 15.4), "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 An ASN.1 item shall consist of a sequence of the characters listed in Table 2 except as specified in 10.2 and 10.3.

Table 2 – ASN.1 characters

A to Z
a to z
0 to 9
: = , { } < . @ ( ) [ ] – ' " | & ^ * ; !

NOTE – Where equivalent derivative standards are developed by national standards bodies, additional characters may appear in the following 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 ASN.1 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 graphic symbols for the defined character set can appear in the ASN.1 notation, surrounded by the double quote characters (") (see 11.11).

10.3 Additional (arbitrary) graphic symbols may appear in the "comment" 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.

11 ASN.1 items

11.1 General rules

11.1.1 The following subclauses specify the characters in ASN.1 items. In each case the name of the item is given, together with the definition of the character sequences which form the item.

11.1.2 Each item specified in the following subclauses (except "bstring", "hstring" and "cstring") shall appear on a single line, and (except for the "comment", "bstring", "hstring" and "cstring" items) shall not contain white-space (see 11.9, 11.10 and 11.11).

11.1.3 The length of a line is not restricted.

11.1.4 The items in the production sequences specified by this Recommendation | International Standard (the ASN.1 notation) may appear on one line or may appear on several lines, and may be separated by white-space, empty lines or comments.

11.1.5 An item shall be separated from a following item by white-space, newline or comment, if the initial character (or characters) of the following item is a permitted character (or characters) for inclusion at the end of the characters in the earlier item.

11.2 Type references

Name of 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 11.18.

11.3 Identifiers

Name of 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 item – valuereference

A "valuereference" shall consist of the sequence of characters specified for an "identifier" in 11.2. 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 reference

Name of 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 Comment

Name of 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 ASN.1 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 A "comment" shall commence with a pair of adjacent hyphens and 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 opens it and the pair, if any, which ends it. It may include graphic symbols which are not in the character set specified in 10.1 (see 10.3).

11.7 Empty item

Name of 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 Number item

Name of 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" item is always mapped to an integer value by interpreting it as decimal notation.

11.9 Binary string item

Name of item – bstring

A "bstring" shall consist of an arbitrary number (possibly zero) of zeros, ones, white-space or newlines, preceded by a single quote (') and followed by the pair of characters:

'B

White-space and newlines that appear within a binary string item have no significance.

EXAMPLE – '01101100'B

11.10 Hexadecimal string item

Name of item – hstring

11.10.1 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

or white-space or newlines, preceded by a single quote (') and followed by the pair of characters:

'H

White-space and newlines that appear within a hexadecimal string item have no significance.

EXAMPLE – 'AB0196'H

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

11.11 Character string item

Name of item – cstring

11.11.1 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 double quotes ("). If the character set includes a double quote, this character (if present in the character string being represented by the "cstring") shall be represented in the "cstring" by a pair of double quotes 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". White-space that appears immediately prior to or following the end of line in the "cstring" have no significance.

NOTE 1 – The "cstring" can only be used to represent 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, alternative ASN.1 syntax is available. (See clause 34.)

NOTE 2 – The character string represented by a "cstring" consists of the characters associated with the printed 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 the character string denoted by "cstring" may be ambiguous.

EXAMPLE 1 – "屎 屍 市 弑"

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 if a proportional spacing font (such as is used above) is used in the specification.

11.11.2 When a character is a combining character it shall be denoted in 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.)

EXAMPLE – The accent combining character and lower case ’e’ are two characters in the ISO 646 French Version, and thus in a corresponding "cstring" is written as two characters and not as the single character é.

11.11.3 The "cstring" shall not be used to represent character string values which contain control characters. Only graphic and spacing characters are permitted in it.

11.12 Assignment 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).

11.13 Range separator

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).

11.14 Ellipsis

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).

11.15 Left version brackets

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).

11.16 Right version brackets

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).

11.17 Single character items

Names of items –

"{"
"}"
"<"
","
"."
"("
")"
"["
"]"
"-" (hyphen)
":"
";"
"@"
"|"
"!"
"^"

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

11.18 Reserved words

Names of reserved words –

ABSENT
ABSTRACT-SYNTAX
ALL
APPLICATION
AUTOMATIC
BEGIN
BIT
BMPString
BOOLEAN
BY
CHARACTER
CHOICE
CLASS
COMPONENT
COMPONENTS
CONSTRAINED
DEFAULT
DEFINITIONS
EMBEDDED
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
PDV
PLUS-INFINITY
PRESENT
PrintableString
PRIVATE
REAL
SEQUENCE
SET
SIZE
STRING
SYNTAX
T61String
TAGS
TeletexString
TRUE
TYPE-IDENTIFIER
UNION
UNIQUE
UNIVERSAL
UniversalString
UTCTime
UTF8String
VideotexString
VisibleString
WITH

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, 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 ";"           |
          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              |
        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" and "ValueAssignment" productions are specified in clause 15.

NOTE 4 – The grouping of ASN.1 data types into modules does not necessarily determine the formation of presentation data values into named abstract syntaxes for the purpose of presentation context definition.

NOTE 5 – 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 6 – 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 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.

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 items 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 "valuereferences" 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 "valuereferences".

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) only 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; and

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

12.13 When the "empty" alternative of "Exports" is selected, every "Symbol" defined in the module may be referenced from other modules.

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 exported 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
"ValueSetTypeAssignment"	15.4
"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 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.

13.3 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 an item 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.

13.4 An external reference shall be used in a module only to refer to an item 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.

13.5 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.

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 ::= "@" GlobalModuleReference
        "."
        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 "GlobalModuleReference" identifies an ASN.1 module (see 12.1).

14.5 The "typereference" references any ASN.1 type defined in the module identified by "GlobalModuleReference".

14.6 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.

14.7 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.

14.8 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, and called **the iteration count**) 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.

14.9 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 11.18).

15.2 A "valuereference" shall be assigned a value by the notation specified by the "ValueAssignment" production:

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

The "Value" being assigned to the "valuereference" shall be a notation for a value of the type defined by "Type" (as specified in 15.3).

15.3 "Value" is a notation for a value of a type if either:

a) "Value" is a "BuiltinValue" notation for the type (see 16.8); or

b) "Value" is a "DefinedValue" notation for a value of that type.

15.4 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 11.18), and may be referenced as a type. "ValueSet" is defined in 15.5.

15.5 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).

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 |
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	35
ChoiceType	28
EmbeddedPDVType	32
EnumeratedType	19
ExternalType	33
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
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	40.1
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 44.

16.5 This Recommendation | International Standard requires the use of the notation "NamedType" in specifying the components of the set type, sequence type 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 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":

```
Value ::= BuiltinValue | ReferencedValue | ObjectClassFieldValue
```

NOTE – ObjectClassFieldValue is defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.6.

16.8 Values of the built-in types of ASN.1 can be specified by the notation "BuiltinValue", defined as follows:

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

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

16.9 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**

16.10 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.

16.11 The value of a type referenced using the "NamedType" notation shall be defined by the notation "NamedValue":

```
NamedValue ::= identifier Value
```

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.

16.12 The implied or explicit presence of an extension marker 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.

17 Notation for the boolean type

17.1 The boolean type (see 3.8.7) 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 3.8.66 and 3.8.38) shall be defined by the notation "BooleanValue":

```
BooleanValue ::= TRUE | FALSE
```

18 Notation for the integer type

18.1 The integer type (see 3.8.40) 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" sequences 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":

```
IntegerValue ::=
                SignedNumber |
                identifier
```

18.10 The "identifier" in "IntegerValue" 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" should be preferred.

19 Notation for the enumerated type

19.1 The enumerated type (see 3.8.24) shall be referenced by the notation "EnumeratedType":

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

Enumerations ::= RootEnumeration |
            RootEnumeration  ","  "..." |
            RootEnumeration  ","  "..."  ","  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 this purpose, 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 "AdditionalEnumeration" shall be greater than all previously defined "AdditionalEnumeration"s in the type.

19.5 When a "NamedNumber" is used in defining an "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 "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":

```
EnumeratedValue ::= identifier
```

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

20 Notation for the real type

20.1 The real type (see 3.8.52) 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:

$$M \text{ x } B^{E}$$

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 – Values represented by "base" 2 and by "base" 10 are considered to be distinct abstract values even if they evaluate to the same real numbers 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":

```
RealValue ::=
        NumericRealValue | SpecialRealValue

NumericRealValue ::=  0         |
        SequenceValue           -- Value of the associated sequence type

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

The form "0" shall be used for zero values, the alternate form for "NumericRealValue" shall not be used for zero values.

21 Notation for the bitstring type

21.1 The bitstring type (see 3.8.6) 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 **bit zero**. 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.

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 bit 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 many 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":

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

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

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

21.11 The "BitStringValue" 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 – The "{" "}" production sequence is used to denote the bitstring which contains no one bits.

21.12 In specifying the encoding rules for a bitstring, the bits shall be referenced by the terms **first bit** and **trailing bit** where the first bit is bit zero (see 21.2).

21.13 When using the "bstring" notation, the **first bit** is on the left, and the **trailing bit** is on the right.

21.14 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.

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

EXAMPLE

'A8A'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.

22 Notation for the octetstring type

22.1 The octetstring type (see 3.8.48) 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":

```
OctetStringValue ::=
               bstring          |
               hstring
```

22.4 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**.

22.5 When using the "bstring" notation, the left-most bit shall be the most significant bit of the first octet. 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.

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

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

23 Notation for the null type

23.1 The null type (see 3.8.43) 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":

NullValue ::= NULL

24 Notation for sequence types

24.1 The notation for defining a sequence type (see 3.8.56) 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

RootComponentTypeList  ::=  ComponentTypeList

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

ExtensionAdditions  ::= ","  ExtensionAdditionList  |  empty

ExtensionAdditionList  ::=  ExtensionAddition  |
          ExtensionAdditionList  ","  ExtensionAddition

ExtensionAddition  ::= ComponentType  |  ExtensionAdditionGroup

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

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 components 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.)

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 if automatic tagging has been applied.

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

24.5.3 Subclause 24.5.1 shall apply to the complete series of "ComponentTypes" 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).

24.6 All "ComponentTypes" in extension additions shall have tags which are distinct from the tags of all textually following "ComponentTypes" that are in the root, up to and including the first such "ComponentType" that is not marked OPTIONAL or DEFAULT (if any).

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 tags is in effect and the "ComponentType"s in the extension root have no tags, then no "ComponentType" within the "ExtensionAdditionList" shall be a tagged type.

24.9 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" or "NamedType" in the "AlternativeTypeLists", 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 "TaggedTypes" 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 }
```

the tags attached to a, b and c are not impacted by the possible automatic tagging applied to components of E.

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 may result in changes to other components due to the side effect of modifying the tags.

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 presentation data 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 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).

24.16 The notation for defining a value of a sequence type shall be "SequenceValue":

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

ComponentValueList ::=

                    NamedValue                   |
                    ComponentValueList "," NamedValue
```

24.17 The "{" "}" 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{}".

24.18 There shall be one "NamedValue" 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 3.8.57) from another type shall be the "SequenceOfType".

SequenceOfType ::= SEQUENCE OF 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 24.15).

25.3 The notation for defining a value of a sequence-of type shall be the "SequenceOfValue":

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

```
ValueList ::=
        Value       |
        ValueList "," Value
```

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

25.4 Each "Value" in the "ValueList" shall be of the type specified in the "SequenceOfType".

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

26 Notation for set types

26.1 The notation for defining a set type (see 3.8.58) 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.)

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 "AdditionalComponentTypeList" shall be canonically greater (see 6.4) than those of the other components in the "AdditionalComponentTypeList".

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.

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":

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

"ComponentValueList" is specified in 24.16.

26.8 The "SetValue" shall only be "{" "}" 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" for each "NamedType" in the "SetType" which is not marked "OPTIONAL" or "DEFAULT".

NOTE – These "NamedValues" may appear in any order.

27 Notation for set-of types

27.1 The notation for defining a set-of type (see 3.8.59) from another type shall be the "SetOfType":

```
SetOfType ::=
        SET  OF  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":

```
SetOfValue ::= "{" ValueList "}"    |    "{"  "}"
```

"ValueList" is specified in 25.3.

The "{" "}" notation is used when the SetOfValue is an empty list.

27.4 Each "Value" sequence in the "ValueList" shall be the notation for a value of the "Type" specified in the "SetofType".

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.

28 Notation for choice types

28.1 The notation for defining a choice type (see 3.8.13) 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  ::=  ExtensionAdditionAlternatives | NamedType

ExtensionAdditionAlternatives  ::= "[["  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 The types defined in the "AlternativeTypeList" productions in an “AlternativeTypeLists” shall all have distinct tags (see clause 30). If automatic tags is in effect and the "NamedType"s in the extension root have no tags, then no "NamedType"within the "ExtensionAdditionAlternativesList"shall be tagged.

NOTE – Where the "TagDefault" for the module in which this notation appears is "AUTOMATIC TAGS", the tags are made distinct as a result of the application of 24.7.

28.3 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 it contain a "Type" which is an occurrence of "TaggedType", then automatic tagging transformation is selected for the entire "AlternativeTypeLists", otherwise it is not. When selected, the automatic tagging transformation of an "AlternativeTypeLists" is applied to each "NamedType" of the "AlternativeTypeLists" by replacing each "Type" originally in the "NamedType" production with a replacement "TaggedType" occurrence specified in 24.9.

28.4 The tag of each new “NamedType” added to the “ExtensionAdditionAlternativesList” shall be canonically greater (see 8.4) than those of the other alternatives in the “ExtensionAdditionAlternativesList”, and shall be the last “NamedType” in the "ExtensionAdditionAlternativesList".

28.5 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.)

28.6 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 24.5 to 24.6, 26.3 and 28.2), 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.

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

28.8 The notation for defining the value of a choice type shall be the "ChoiceValue":

ChoiceValue ::= identifier ":" Value

28.9 "Value" 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 3.8.55) 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 Where the "SelectionType" is used as a "NamedType", the "identifier" of the "NamedType" is present, as well as the "identifier" of the "SelectionType".

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

29.4 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 3.8.64) 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 28.2). 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 C.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 a choice type, open type, or a "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 28.2.

30.8 The "IMPLICIT" alternative shall not be used if the type defined by "Type" is a choice type or an open type or a "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":

```
TaggedValue ::= Value
```

where "Value" 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 3.8.47) 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":

```
ObjectIdentifierValue ::=
          "{" ObjIdComponentList "}"                   |
          "{" DefinedValue ObjIdComponentList "}"

ObjIdComponentList ::=
          ObjIdComponent                               |
          ObjIdComponent ObjIdComponentList

ObjIdComponent ::=     NameForm        |
                       NumberForm      |
                       NameAndNumberForm

NameForm ::= identifier

NumberForm ::= number | DefinedValue

NameAndNumberForm ::=
          identifier "(" NumberForm ")"
```

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 "NameForm" shall be used only for those object identifier components whose numeric value and identifier are specified in Annexes A to C of ITU-T Rec. X.660 | ISO/IEC 9834-1, and shall be one of the identifiers specified in Annexes A to C of ITU-T Rec. X.660 | ISO/IEC 9834-1. 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.

31.7 The "number" in the "NumberForm" shall be the numeric value assigned to the object identifier component.

31.8 The "identifier" in the "NameAndNumberForm" 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.

31.9 The semantics associated with an object identifier value are specified in ITU-T Rec. X.660 | ISO/IEC 9834-1.

31.10 The significant part of the object identifier component is the “NameForm” or “NumberForm” which it reduces to, and which provides the numeric value for the object identifier component. Except for the arcs specified in Annexes A to C of ITU-T Rec. X.660 | ISO/IEC 9834-1, the numeric value of the object identifier component is always present in an instance of object identifier value notation.

31.11 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.

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 embedded-pdv type

32.1 The embedded-pdv type (see 3.8.21) shall be referenced by the notation "EmbeddedPDVType":

```
EmbeddedPDVType ::= EMBEDDED PDV
```

32.2 This type has a tag which is universal class, number 11.

NOTE – Where presentation layer negotiation is in use, the same functionality as EXTERNAL is provided by EMBEDDED PDV (together with added functionality), but the bits on the line will be different. It is recommended in this case that further version changes to application protocols should incorporate the replacement of EXTERNAL by CHOICE{external EXTERNAL, embedded-pdv EMBEDDED PDV}. Additional replacements to use of EXTERNAL where Presentation layer negotiation is not in use are given in ITU-T Rec. X.681 | ISO/IEC 8824-2, Annex C.

32.3 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) a class of values containing that data value (an abstract syntax); and

   2) the encoding used (the transfer syntax) to distinguish that data value from other values in the same class.

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 references an OSI presentation context for identification of 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 may not be encoded in an instance of communication.

32.4 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.

32.5 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 class and encoding --,

        presentation-context-id         INTEGER
            -- (Applicable only to OSI environments)
            -- The negotiated presentation context identifies the class of the value and its encoding --,

        context-negotiation             SEQUENCE {
            presentation-context-id         INTEGER,
            transfer-syntax                 OBJECT IDENTIFIER }
            -- (Applicable only to OSI environments)
            -- Context-negotiation in progress for a context to identify the class of the value
            -- and its encoding --,

        transfer-syntax                 OBJECT IDENTIFIER
            -- The class 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.

32.6 For the "presentation-context-id" alternative, the integer value shall be a presentation context identifier in the defined context set. This alternative shall not be used on the P-CONNECT request nor on the P-ALTER-CONTEXT request for a presentation context that is being proposed for addition or deletion by those request primitives.

NOTE – Even if there is a single transfer syntax being proposed for a presentation context in the presentation context definition list, the "presentation-context-id" alternative cannot be used for that presentation context.

32.7 The "context-negotiation" alternative shall only be used on the P-CONNECT request or on the P-ALTER-CONTEXT request, and the integer value shall be a presentation context identifier proposed for addition to the defined context set. The object identifier "transfer-syntax" shall identify a proposed transfer syntax for that presentation context which is used to encode the value.

32.8 The notation for a value of the embedded-pdv type shall be the value notation for the associated type defined in 32.5, where the value of the “data-value” OCTET STRING represents an encoding using the transfer syntax specified in “identification”.

```
EmbeddedPdvValue ::= SequenceValue                  -- value of associated type defined in 32.5
```

32.9 EXAMPLE 1 – Where an application designer wishes the encoding to be independent of any presentation environment (and hence to be capable of being relayed or stored and retrieved without modification), it is necessary to forbid the use of the "presentation-context-id" and "context-negotiation" alternatives. This can be done by writing:

```
EMBEDDED PDV (WITH COMPONENTS {
            ... ,
            identification  (WITH COMPONENTS {
                                ... ,
                                presentation-context-id         ABSENT,
                                context-negotiation             ABSENT } ) } )
```

32.10 EXAMPLE 2 – 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 } ) } )
```

33 Notation for the external type

33.1 The external type (see 3.8.37) shall be referenced by the notation "ExternalType":

```
ExternalType ::= EXTERNAL
```

33.2 This type has a tag which is universal class, number 8.

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 of:

   1) a class of values containing that data value (an abstract syntax); and

   2) the encoding used (the transfer syntax) to distinguish that data value from other values in the same class; and

c) (optionally) an object descriptor which provides a human-readable description of the class 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 on 32.3 also applies to the external type.

33.4 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. In particular, the encoding in BER uses an equivalent sequence type identical to that which was present in the definition of the external type in CCITT Rec. X.208 | ISO/IEC 8824, and encodings of external values by BER are unchanged.

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 class and encoding --,

        presentation-context-id             INTEGER
            -- (Applicable only to OSI environments)
            -- The negotiated presentation context identifies the class of the value and its encoding --,

        context-negotiation                 SEQUENCE {
            presentation-context-id             INTEGER
            transfer-syntax                     OBJECT IDENTIFIER }
            -- (Applicable only to OSI environments)
            -- Context-negotiation in progress for a context to identify the class of the value
            -- and its encoding --,

        transfer-syntax                     OBJECT IDENTIFIER
            -- The class 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 – The external type does not allow the "syntaxes", "transfer-syntax" or "fixed" alternatives of "identification". These alternatives cannot be allowed for the external type because of the need to maintain backwards compatibility with the external type of CCITT Rec. X.208 | ISO/IEC 8824. 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.

33.6 The text of 32.6 and 32.7 also applies to the external type.

33.7 The notation for a value of the external type shall be the value notation for the associated type defined in 33.5, where the value of the "data-value" OCTET STRING represents an encoding using the transfer syntax specified in "identification".

```
ExternalValue ::= SequenceValue                    -- value of associated type defined in 33.5
```

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.

34 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).

The term **abstract character** denotes the totality of information associated with a cell in a character repertoire table. 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 value notation for character string types has three variants (which can be combined), specified formally below:

a) A printed representation of the characters in the string using the assigned graphic symbol, possibly including spacing characters; this is the "cstring" notation.

   NOTE 1 – Such a representation can be ambiguous when the same graphic symbol is used for more than one character in the repertoire.

   NOTE 2 – Such a representation can be ambiguous when spacing characters are used 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 37 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.

35 Notation for character string types

35.1 The notation for referencing a character string type (see 3.8.11) shall be:

CharacterStringType ::= RestrictedCharacterStringType | UnrestrictedCharacterStringType

"RestrictedCharacterStringType" is the notation for a restricted character string type and is defined in clause 36. "UnrestrictedCharacterStringType" is the notation for the unrestricted character string type and is defined in 39.1.

35.2 The tag of each restricted character string type is specified in 36.1. The tag of the unrestricted character string type is specified in 39.2.

35.3 The notation for a character string value shall be:

CharacterStringValue ::= RestrictedCharacterStringValue | UnrestrictedCharacterStringValue

"RestrictedCharacterStringValue" is defined in 36.7. "UnrestrictedCharacterStringValue" is notation for an unrestricted character string value and it is defined in 39.6.

36 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.

36.1 **Table 3** 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.

NOTE – The tag assigned to character string types unambiguously identifies the type. Note, however, that if ASN.1 is used to define new types from this type (particularly using IMPLICIT), it may be impossible to recognize these types without knowledge of the ASN.1 type definition.

Table 3 – 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 36.13	
NumericString	18	Table 4	Note 1
PrintableString	19	Table 5	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	Note 4
GeneralString	27	All G and all C sets + SPACE + DELETE	
UniversalString	28	See 36.6	
BMPString	30	See 36.12	

[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 – The entries corresponding to these registration numbers reference CCITT Rec. T.61 for rules concerning their use. 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.

NOTE 4 – The reference to register 6 of "ISO International Register of Coded Character Sets to be used with Escape Sequences" constitutes an indirect reference to ISO/IEC 646:1991. This is a change from CCITT Rec. X.208 | ISO/IEC 8824, which referenced the register 2 (indirect reference to ISO 646:1973). Applications wishing to reference register number 2 should use other means of doing so [e.g. use the unrestricted character string (see clause 39)] to carry the old definition of VisibleString or reference CCITT Rec. X.208 | ISO/IEC 8824.

36.2 Table 4 lists the characters which can appear in the NumericString type and NumericString character abstract syntax.

Table 4 – NumericString

Name	Graphic
Digits	0, 1, ... 9
Space	(space)

36.3 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 M.

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)}.

36.4 Table 5 lists the characters which can appear in the PrintableString type and PrintableString character abstract syntax.

Table 5 – PrintableString

Name	Graphic
Capital letters	A, B, ... Z
Small letters	a, b, ... z
Digits	0, 1, ... 9
Space	(space)
Apostrophe	'
Left Parenthesis	(
Right Parenthesis	)
Plus sign	+
Comma	,
Hyphen	-
Full stop	.
Solidus	/
Colon	:
Equal sign	=
Question mark	?

36.5 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 M.

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) }.

36.6 The characters which can appear in the UniversalString type are any of the characters allowed by ISO/IEC 10646-1, and use of this type invokes the conformance requirements specified in ISO/IEC 10646-1, especially with regard to the restricted use zone of ISO/IEC 10646-1.

NOTE 1 – Use of this type without a constraint is deprecated, as conformance will generally be impractical.

NOTE 2 – Clause 37 defines an ASN.1 module containing a number of subtypes of this type for the "Collections of graphics characters for subsets" defined in Annex A of ISO/IEC 10646-1.

36.7 The value notation for the restricted character string types shall be "cstring" (see 11.11), "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 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 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 37.1.2).

NOTE 2 – Clause 37 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 we wish 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"
```

36.8 The "DefinedValue" in "CharsDefn" shall be a reference to a value of that type.

36.9 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.

36.10 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.

36.11 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*).

36.12 BMPString is a subtype of UniversalString that has its own unique tag and models the Basic Multilingual Plane (the first 64K-2 cells) 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 37) 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.

36.13 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 is different from that of UniversalString, and for most text will be less verbose.

37 Naming characters and collections defined in ISO/IEC 10646

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 UniversalString.

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 37.1 are exported and must be imported by any module that uses them.

37.1 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.

37.1.1 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}
```

37.1.2 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 37.2;

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, 145}
      -- represents the character GREEK CAPITAL LETTER SIGMA, see ISO/IEC 10646-1
```

37.1.3 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 37.3;

c) <alternativelist> is formed by using the <namedcharacter>s as generated in 37.2 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 D.)

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.
```

37.1.4 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 CombiningCharactersB-2))

Level3 ::= BMPString
```

NOTE 1 – "CombiningCharacters" and "CombiningCharactersB-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 C.2.7.1 for an example.

NOTE 3 – See D.2.5 for more information on this topic.

37.1.5 The module is terminated by the statement:

END

37.1.6 A user-defined equivalent of the example in 37.1.3 is:

BasicLatin ::= BMPString (FROM (space..tilde))
-- represents the collection of characters BASIC LATIN, see ISO/IEC 10646-1.

37.2 A <namedcharacter> is the string obtained by taking an <iso10646name> (see 37.1.2) 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

37.3 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.

38 Canonical order of characters

38.1 For the purpose of "ValueRange" subtyping and for possible use by encoding rules, a canonical ordering of characters is specified for UniversalString, BMPString, NumericString, PrintableString, VisibleString, and IA5String.

38.2 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.

38.3 The canonical order of an abstract character is defined by the canonical order of its cell.

38.4 For UniversalString, the canonical order of the cells is defined (see ISO/IEC 10646-1) as:

256*(256*(128*(Group Number)+(Plane Number))+(Row Number))+(Cell Number)

The entire character set contains precisely 128*256*256*256 characters. 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) by giving the graphic symbol in a "cstring" (ASN1-CHARACTER-MODULE is defined in 37.1). It is not possible to specify a cell as an end-point of a range or to identify an individual character where there have been no names or graphic symbols assigned to that cell.

38.5 For BMPString, the canonical order of the cells is defined (see ISO/IEC 10646-1) as:

256*(Row Number)+(Cell Number)

The entire character set contains precisely 256*256 characters. 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) by giving the graphic symbol in a "cstring". It is not possible to specify a cell as an end-point of a range or to identify an individual character where there have been no names or graphic symbols assigned to that cell.

38.6 For NumericString, the canonical ordering, increasing from left to right, is defined (see Table 4 of 36.2) 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.

38.7 For PrintableString, the canonical ordering, increasing from left to right and top to bottom, is defined (see Table 5 of 36.4) as:

(Space) (Apostrophe) (Left Parenthesis) (Right Parenthesis) (Plus Sign) (Comma) (Hyphen) (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.

38.8 For VisibleString, the canonical order of the cells is defined from the ISO 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 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".

38.9 For IA5String, the canonical order of the cells is defined from the ISO 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 37.1.1.

39 Definition of unrestricted character string types

This clause defines a type whose values are the values of any character abstract syntax. This abstract syntax may be part of the defined context set in an instance of communication, or may be 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 IDENTIFIERs.

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 to include reference to supported syntaxes in a PICS proforma (Protocol Implementation Conformance Statement). Note that grouping of instances for the purpose of application layer specification can be achieved using different ASN.1 type references (all of which would be references for the CHARACTER STRING type).

39.1 The unrestricted character string type (see 3.8.69) shall be referenced by the notation "CharacterStringType":

UnrestrictedCharacterStringType ::= CHARACTER STRING

39.2 This type has a tag which is universal class, number 29.

39.3 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 class of values containing that character string value (a character abstract syntax); and

2) the encoding used (the character transfer syntax) to distinguish that character string value from other values in the same class.

39.4 The unrestricted character string type has an associated type. This associated type is used to support its value and subtype notations.

39.5 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 class and encoding --,
        presentation-context-id             INTEGER
                -- (Applicable only to OSI environments)
                -- The negotiated presentation context identifies the class of the value and its encoding  --,

        context-negotiation                 SEQUENCE {
            presentation-context-id             INTEGER,
            transfer-syntax                     OBJECT IDENTIFIER }
                -- (Applicable only to OSI environments)
                -- Context-negotiation in progress for a context to identify the class of the value
                -- and its encoding  --,

        transfer-syntax                         OBJECT IDENTIFIER
                -- The class 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.

39.6 The value notation shall be the value notation for the associated type, where the value of the “string-value” OCTET STRING represents an encoding using the transfer syntax specified in “identification”.

UnrestrictedCharacterStringValue ::= SequenceValue -- *value of associated type defined in 39.5*

39.7 An example of the unrestricted character string type is in C.2.8.

40 Notation for types defined in clauses 41 to 43

40.1 The notation for referencing a type defined in clauses 41 to 43 shall be:

UsefulType ::= typereference

where "typereference" is one of those defined in clauses 41 to 43 using the ASN.1 notation.

40.2 The tag of each "UsefulType" is specified in clauses 41 to 43.

41 Generalized time

41.1 This type shall be referenced by the name:

GeneralizedTime

41.2 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.

41.3 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.

41.4 The tag shall be as defined in 41.3.

41.5 The value notation shall be the value notation for the "VisibleString" defined in 41.3.

42 Universal time

42.1 This type shall be referenced by the name:

UTCTime

42.2 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.

42.3 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".

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 The object descriptor type

43.1 This type shall be referenced by the name:

ObjectDescriptor

43.2 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.

43.3 The type is defined, using ASN.1, as follows:

```
ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString
```

The "GraphicString" contains the text describing the object.

43.4 The tag shall be as defined in 43.3.

43.5 The value notation shall be the value notation for the "GraphicString" defined in 43.3.

44 Constrained Types

44.1 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 44.5. The second alternative is defined in 44.4.

44.2 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))
```

44.3 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.

44.4 As a consequence of the interpretation specified in 44.2, 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
```

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 first and third alternatives, the constraint is "Constraint" (see 44.5), while in the second and fourth it is "SizeConstraint" (see 48.5).

NOTE – Although the "Constraint" alternatives encompass the corresponding "SizeConstraint" alternatives, the latter, which have no enclosing brackets, are provided for backwards-compatibility with CCITT Rec. X.208 | ISO/IEC 8824.

44.5 A constraint is specified by the notation "Constraint":

```
Constraint ::= "(" ConstraintSpec  ExceptionSpec ")"

ConstraintSpec ::=
    SubtypeConstraint          |
    GeneralConstraint
```

"ExceptionSpec" is defined in clause 45. Unless it is used in conjunction with an "extension marker" (see clause 47), 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).

44.6 The notation "SubtypeConstraint" is the general-purpose "ElementSetSpec" 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.

45 The exception identifier

45.1 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).

45.2 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.

45.3 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 "!". 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.

45.4 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").

45.5 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.

45.6 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.

46 Element set specification

46.1 In some notations a set of elements of some identified element class (the governor) can be specified. In such cases, the notation "ElementSetSpec" is used:

```
ElementSetSpecs ::=
          RootElementSetSpec |
          RootElementSetSpec  ","   "..."  |
          "..."  ","  AdditionalElementSetSpec |
          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".

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 "Elements" notation is defined as follows:

```
Elements ::=
      SubtypeElements          |
      ObjectSetElements        |
      "(" ElementSetSpec ")"
```

The elements specified by this notation are:

a) As described in clause 48 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, 12.6, 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.

47 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.

47.1 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.

47.2 The joint use of the extension marker and an exception identifier is an indication that extension additions are expected and thus should not be treated as an error in the decoding process, and that the application standards prescribe specific 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 that any unrecognized extension additions are to be returned to the application for possible re-encoding and relaying.

47.3 Set arithmetic, if any, in the "ElementSetSpecs" notation shall be performed without consideration given to the presence of the extension marker.

NOTE – In other words, the presence of an extension marker has no effect on set arithmetic.

47.4 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. If the newly defined type is meant to be extensible, then an extension marker shall be explicitly added to its "ElementSetSpecs". 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.
```

47.5 If a type defined with an extensible constraint is further constrained with an "ElementSetSpecs" that does not contain an extension marker, the resulting type is one whose constraint is not extensible and which does not inherit any extension additions that may be present in the parent type. For example:

```
A ::= INTEGER (0..10, ...)            -- A is extensible.

B ::= A (2..5)                        -- B is inextensible.

C ::= A                               -- C is extensible.
```

47.6 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.
```

47.7 Where this Recommendation | International Standard requires distinct tags (see 24.5 to 24.6, 26.3 and 28.2), the following transformation shall conceptually be applied before performing the check for tag uniqueness:

47.7.1 A new element or alternative (called the conceptually-added element, see 47.7.2) 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.

47.7.2 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 28.2). It is conceptually-added *after* the application of automatic tagging (if applicable) and the expansion of COMPONENTS OF.

47.7.3 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 14.2, Note 2 of ITU-T Rec. X.681 | ISO/IEC 8824-2.

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 28.2) 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.

47.7.4 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 SEQUENCEs or SETs or CHOICEs). 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 items was different for different insertion points.

47.8 Examples

47.8.1 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".

47.8.2 Example 2

```
A ::= SET {
    a   A,
    b   CHOICE {
        c       C,
            ... ,
            ... ,
        d       D
    },
    ... ,
    e   E
}
```

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.

47.8.3 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".

47.8.4 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.

48 Subtype elements

48.1 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 6 summarizes which notations can be applied to which parent types.

```
SubtypeElements ::=
      SingleValue                 |
      ContainedSubtype            |
      ValueRange                  |
      PermittedAlphabet           |
      SizeConstraint              |
      TypeConstraint              |
      InnerTypeConstraints
```

Table 6 – Applicability of subtype value sets

Type	Single Value	Contained Subtype	Value Range	Size Constraint	Permitted Alphabet	Type constraint	Inner Subtyping
Bit String	Yes	Yes	No	Yes	No	No	No
Boolean	Yes	Yes	No	No	No	No	No
Choice	Yes	Yes	No	No	No	No	Yes
Embedded-pdv	Yes	No	No	No	No	No	Yes
Enumerated	Yes	Yes	No	No	No	No	No
External	Yes	No	No	No	No	No	Yes
Instance-of	Yes	Yes	No	No	No	No	Yes
Integer	Yes	Yes	Yes	No	No	No	No
Null	Yes	Yes	No	No	No	No	No
Object class field type	Yes	Yes	No	No	No	No	No
Object Identifier	Yes	Yes	No	No	No	No	No
Octet String	Yes	Yes	No	Yes	No	No	No
open type	No	No	No	No	No	Yes	No
Real	Yes	Yes	Yes	No	No	No	Yes
Restricted Character String Types	Yes	Yes	Yes[a)]	Yes	Yes	No	No
Sequence	Yes	Yes	No	No	No	No	Yes
Sequence-of	Yes	Yes	No	Yes	No	No	Yes
Set	Yes	Yes	No	No	No	No	Yes
Set-of	Yes	Yes	No	Yes	No	No	Yes
Unrestricted Character String Type	Yes	No	No	Yes	No	No	Yes

[a)] Allowed only within the "PermittedAlphabet" of BMPString, IA5String, NumericString, PrintableString, VisibleString and UniversalString

48.2 Single Value

48.2.1 The "SingleValue" notation shall be:

SingleValue ::= Value

where "Value" is the value notation for the parent type.

48.2.2 A "SingleValue" specifies the single value of the parent type specified by "Value".

48.3 Contained Subtype

48.3.1 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.

48.3.2 A "ContainedSubtype" specifies all of the values in the parent type resulting from the intersection of the parent type and "Type". "Type" is required to be derived from the same built-in type as the parent type.

48.4 Value Range

48.4.1 The "ValueRange" notation shall be:

ValueRange ::= LowerEndpoint ".." UpperEndpoint

48.4.2 A "ValueRange" specifies all 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 and UniversalString only) and real types.

NOTE – For the purpose of subtyping, "PLUS-INFINITY" exceeds all "NumericReal" values and "MINUS-INFINITY" is less than all "NumericReal" values.

48.4.3 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

48.4.4 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

48.5 Size Constraint

48.5.1 The "SizeConstraint" notation shall be:

SizeConstraint ::= SIZE Constraint

48.5.2 A "SizeConstraint" can only be applied to bit string types, octet string types, character string types, set-of types or sequence-of types, or types formed from any of those types by tagging.

48.5.3 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".

48.5.4 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.

48.6 Type Constraint

48.6.1 The "TypeConstraint" notation shall be:

TypeConstraint ::= Type

48.6.2 This notation is only applied to an open type notation and restricts the open type to values of "Type".

48.7 Permitted Alphabet

48.7.1 The "PermittedAlphabet" notation shall be:

PermittedAlphabet ::= FROM Constraint

48.7.2 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.

48.7.3 The "Constraint" is any which could be applied to the parent type (see Table 6), 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".

48.8 Inner Subtyping

48.8.1 The "InnerTypeConstraints" notation shall be:

```
InnerTypeConstraints ::=
    WITH COMPONENT SingleTypeConstraint |
    WITH COMPONENTS MultipleTypeConstraints
```

48.8.2 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 48.8.6). This notation can be applied to the set-of, sequence-of, set, sequence and choice types, or types formed from them by tagging.

48.8.3 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.

48.8.4 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
```

48.8.5 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".

48.8.6 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 48.8.9) 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.

48.8.7 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

48.8.8 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.

48.8.9 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 48.8.9.1 to 48.8.9.3.

48.8.9.1 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).

48.8.9.2 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 C.4.6 for a clarifying example.

48.8.9.3 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.

Annex A

Use of ASN.1-88/90 notation

(This annex forms an integral part of this Recommendation | International Standard)

A.1 Maintenance

The term **ASN.1-88/90 notation** is used to refer to that ASN.1 notation specified in CCITT Rec. X.208 | ISO/IEC 8824. The term **current ASN.1 notation** is used to refer to that specified in this Recommendation | International Standard.

At the time of publication of this Recommendation | International Standard, CCITT Rec. X.208 | ISO/IEC 8824 was still being maintained. This continued maintenance depends on an annual Resolution by ISO/IEC/JTC1/SC21, and cannot be expected to be indefinite. It is provided in order to give users of ASN.1 time to replace features (particularly ANY and use of the macro notation) of the ASN.1-88/90 notation with current ASN.1 notation. (This can be done with no change to bits on the line.)

A.2 Mixing ASN.1-88/90 and current ASN.1 notation

Both the ASN.1-88/90 and the current ASN.1 notation specify a top-level syntactic construct which is an ASN.1 module. A user of ASN.1 writes a collection of ASN.1 modules, and may import definitions from other ASN.1 modules.

For any given module, the notation used is required to conform (completely) to either the ASN.1-88/90 notation or to the current ASN.1 notation, and a user Specification should clearly identify which notation is being used (by reference to the appropriate Recommendation | International Standard) for each module textually included in the user Specification.

Note that it might happen that a user wishes to modify part of a module to use the new notation, but to leave other parts in the old notation. This can (only) be achieved by splitting the module into two modules.

Where a module conforms to the ASN.1-88/90 notation, type and value references may be imported from a module that was defined using the current notation. Such types and values must be associated with types that can be defined using only the ASN.1-88/90 notation. For example, a module written using the ASN.1-88/90 notation cannot import a value of type UniversalString, since this type is defined in the current notation but not in ASN.1-88/90; it can, however, import values whose types are, for example, INTEGER, IA5String, etc.

Where a module conforms to the current ASN.1 notation, type and value references may be imported from a module that was defined using the ASN.1-88/90 notation. No ASN.1 macro shall be imported. Value notation for an imported type shall only be used in the importing module if identifiers for SET and SEQUENCE and CHOICE values used in the value notation are present, and if there is no requirement in the value notation for a value of the ANY type. An inner type constraint shall not be applied to an imported type if the component being constrained does not have an identifier.

A.3 Migration to the current ASN.1 notation

When modifying a module (originally written to conform to the ASN.1-88/90 notation) to conform to the current notation, the following points should be noted:

a) All components of SET and SEQUENCE and CHOICE shall be given identifiers that are unambiguous within that SET, SEQUENCE or CHOICE, and such identifiers shall be included in the value notation.

NOTE 1 – The value notation for a CHOICE type contains a colon (":").

b) All uses of ANY and ANY DEFINED BY shall be supported by a suitable information object class definition, with the ANY and ANY DEFINED BY (and the referenced component) replaced by appropriate references to fields of that object class. In most cases the specification can be greatly improved by careful attention to the insertion of table and component relation constraints. In many cases the specification can be further improved if the table or component relation constraint is made a parameter of the type.

c) All macro definitions shall be replaced by either the definition of an information object class, a parameterized type or a parameterized value. If the WITH SYNTAX clause is carefully designed in the definition of an information object class, the notation used to define an object of that class can be made very similar to the notation defined by the old use of the macro notation.

d) All instances of use of a macro shall be replaced by either equivalent information object definitions, or by references to equivalent "ObjectClassFieldType"s, parameterized types or parameterized values. In most cases the specification of information objects can be greatly improved by grouping such definitions into information object sets, and by giving clear guidance on whether it is mandatory to support all information objects in the set, and on whether implementation-dependent extensions to that information object set are to be accommodated by receiving implementations, and if so, how they are to handle receipt of "unknown" values. It may also be desirable to consider the possibility that a later version of the user Specification may extend the information object set, and to give guidance to current implementors on how such extensions are to be treated.

e) All occurrences of EXTERNAL should be carefully examined; while such notation is still legal in the current ASN.1, a user Specification can probably be improved by doing the following:

   1) Consider the use of the INSTANCE OF notation (preferably with a table constraint that may be as a parameter of the type, as discussed above for ANY and ANY DEFINED BY) in place of the EXTERNAL notation; in many cases this will not change the bits on the line.

   2) Where EXTERNAL is retained, use of inner subtyping of the associated type (see 33.5) can help to give precision to the specification of whether use of presentation context identifiers is or is not permitted. Earlier comments (see clause 33) that give guidance about what values of EXTERNAL are to be supported, and what implementations should do if unsupported values are received also apply here.

   3) Consider a change to:

      CHOICE {external EXTERNAL, embedded-pdv EMBEDDED PDV}

      (again with inner subtyping if appropriate) to allow a phased migration of distributed peer applications to the current notation. This can affect the bits on the line, and would normally be done as part of a version change in the protocol. The use of EMBEDDED PDV (particularly for new specifications) will normally give more flexibility, as can be seen by comparison of the associated types; further, EMBEDDED PDV is encoded more efficiently than EXTERNAL by all the encoding rules specified in ITU-T Rec. X.690 | ISO/IEC 8825-1.

f) It may be possible to improve the readability of the notation in existing ASN.1 modules (with no change to bits on the line) by insertion of AUTOMATIC TAGS in the module header and deletion of some or all tags.

   NOTE 2 – This must be done with care, and with understanding of the way automatic tagging works, since if this is incorrectly applied, the bits on the line will change.

g) If AUTOMATIC TAGS is not applied to existing modules as described in f) above, it will normally be desirable not to add new type definitions to the existing module, but rather to create a new module (with automatic tagging) for new type definitions. This makes it possible for the benefits of automatic tagging to be enjoyed without affecting the bits on the line.

h) Attention should be given to fields that contain character strings to see whether the CHARACTER STRING, BMPString, or UniversalString notation should be employed. This would normally, however, change the bits on the line, and would be done as part of a version change.

i) The identifiers "mantissa", "base", and "exponent" need to be added to any real value notation that uses the "NumericRealValue" alternative of the "RealValue" production. Consideration should be given to restricting "base" to 2 or 10 in the type notation.

In general, there can be significant improvements in readability, efficiency, precision, and flexibility by use of the new ASN.1 notation (particularly if full advantage is taken of the use of table and component relation constraints and parameterization, and of the new character string types). All users of ASN.1-88/90 are urged to undertake migration whenever a Specification is revised, or as a separate activity if no revision is expected for some time.

It is generally regarded as a mistake to make additions to existing modules using notation which does not conform to the current ASN.1 specification, even if references to the ASN.1-88/90 specifications are retained for such modules. In particular, new uses of macros and ANY or ANY DEFINED BY, or new SET, SEQUENCE, or CHOICE constructs without unambiguous identifiers should be avoided.

Annex B

Assignment of object identifier values

(This annex forms an integral part of this Recommendation | International Standard)

The following values are assigned in this Recommendation | International Standard:

Subclause **Object Identifier Value**

36.3 { joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0) }

Object Descriptor Value

"NumericString ASN.1 type"

Subclause **Object Identifier Value**

36.5 { joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1) }

Object Descriptor Value

"PrintableString ASN.1 type"

Subclause **Object Identifier Value**

37.1 { joint-iso-itu-t asn1(1) specification(0) modules(0) iso10646(0) }

Object Descriptor Value

"ASN1 Character Module"

Annex C

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.

C.1 Example of a personnel record

The use of ASN.1 is illustrated by means of a simple, hypothetical personnel record.

C.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

C.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".

C.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" }
   }
}
```

C.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 41 to 43 are not dealt with here.

C.2.1 Boolean

C.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
```

C.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
```

C.2.2 Integer

C.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
```

C.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
```

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
```

C.2.3 Enumerated

C.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
```

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.

C.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
```

C.2.4 Real

C.2.4.1 Use a real type to model an approximate number.

EXAMPLE

```
AngleInRadians ::=  REAL

pi   REAL ::= {mantissa  3141592653589793238462643383279, base  10, exponent  –30}
```

C.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}
```

C.2.5 Bit string

C.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 Recommendation T.4.
```

```
image G3FacsimilePage ::= '100110100100001110110'B

trailer BIT STRING ::= '0123456789ABCDEF'H

body1 G3FacsimilePage ::= '1101'B

body2 G3FacsimilePage ::= '1101000'B
```

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).

C.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.
```

Note that "map1" and "map2" are the same abstract value, for the four trailing bits of "map2" are not significant.

C.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 - violates size constraint.
```

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.

C.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.

C.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
```

Note that "billClinton" and "hillaryClinton" have the same abstract values.

C.2.6 Octet string

C.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
-- Recommendations T.5 and T.6.

image G4FacsimilePage ::= '3FE2EBAD471005'H
```

C.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"
```

C.2.7 UniversalString and BMPString

Use the BMPString type to model any string of information which consists solely of characters from the ISO/IEC 10646-1 Basic Multilingual Plane (BMP), and UniversalString to model any string which consists of ISO/IEC 10646-1 characters not confined to the BMP.

C.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.
```

C.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))
```

C.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 packedBCDStringAbstractSyntax and
                          -- packedBCDStringTransferSyntax defined below.
                      } )

-- object identifier value for a character abstract syntax (character set) whose alphabet
-- is the digits 0 through 9.
packedBCDStringAbstractSyntaxId  OBJECT  IDENTIFIER ::=
    { joint-iso-itu-t xxx(999) yyy(999) zzz(999) packedBCD(999) charSet(0) }

-- object identifier value for a character transfer syntax that packs two
-- digits per octet, each digit encoded as 0000 to 1001, $1111_2$ used for padding.
packedBCDStringTransferSyntaxId  OBJECT IDENTIFIER ::=
    { joint-iso-itu-t xxx(999) yyy(999) zzz(999) packedBCD(999) 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.
```

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.

C.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
}
```

C.2.10 Sequence and sequence-of

C.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"}
```

C.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"}
```

C.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}
```

C.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 Record
      userName                VisibleString,
      password                VisibleString,
      accountNumber           INTEGER,
      ...,
      ...
}
```

in anticipation of:

```
Record ::= SEQUENCE {                                  -- Second version of Record
      userName                VisibleString,
      password                VisibleString,
      accountNumber           INTEGER,
      ...,
      [[                                               -- Extension addition added in version 2
        lastLoggedIn          GeneralizedTime OPTIONAL,
        minutesLastLoggedIn   INTEGER
      ]],
      ...
}
```

and later yet:

```
Record ::= SEQUENCE {                                  -- Third version of Record
      userName                VisibleString,
      password                VisibleString,
      accountNumber           INTEGER,
      ...,
      [[                                               -- Extension addition added in version 2
        lastLoggedIn          GeneralizedTime OPTIONAL,
        minutesLastLoggedIn   INTEGER
      ]],
      [[                                               -- Extension addition added in version 3

        certificate           Certificate,
        thumb                 ThumbPrint OPTIONAL
      ]],
      ...
}
```

C.2.11 Set and set-of

C.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."}
```

C.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  --  }
```

C.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" }
```

in anticipation of:

```
UserName ::= SET {              -- Second version of UserName
    personalName                VisibleString,
    organizationName            VisibleString OPTIONAL,
    countryName                 VisibleString OPTIONAL,
    ...,
    [[                                                      -- Extension addition added in version 2
      internetEmailAddress      VisbleString,
      faxNumber                 VisibleString OPTIONAL
    ]],
    …
}

user  UserName ::= {
    personalName                "Jonas Maruba",
    internetEmailAddress        "jonas@meteor.ngo.com"
}
```

and later yet:

```
UserName ::= SET {              -- Third version of UserName
    personalName                VisibleString,
    organizationName            VisibleString OPTIONAL,
    countryName                 VisibleString OPTIONAL,
    ...,
    [[                                                      -- Extension addition added in version 2
      internetEmailAddress      VisbleString,
      faxNumber                 VisibleString OPTIONAL
    ]],
    phoneNumber                 VisibleString OPTIONAL,     -- Extension addition added in version 3
    …
}

user  UserName ::= {
    personalName                "Jonas Maruba",
    internetEmailAddress        "jonas@meteor.ngo.com"
}
```

C.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"}
```

C.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.

C.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 Internationally Standardized Useful Types. It should not be used elsewhere.

C.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}
```

C.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  --}
```

C.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
```

C.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 --}
```

C.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.

C.2.13 Choice

C.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
```

C.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"
```

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
```

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
        vendorSpecific      VendorExt,
        unidentified   NULL
      ]],
      ...
}

fileId1  FileIdentifier ::= relativeName : "MarchProgressReport.doc"

fileId2  FileIdentifier ::= serialNumber : 214

fileId3  FileIdentifier ::= unidentified : NULL
```

C.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,
      …,
      [[                                                -- Extension addition added in version 2
        audio          Audio,
        video          Video
      ]],
      …
}
```

C.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
```

C.2.14 Selection type

C.2.14.1 Use a selection type to model a variable whose type is that of some particular alternatives of a previously defined CHOICE.

C.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" }
```

C.2.15 Object class field type

C.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 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.

C.2.16 Embedded-pdv

C.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 EMBEDDED PDV
```

C.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.

C.2.18 Instance-of

C.2.18.1 Use an instance-of to specify a type containing an object identifier field and an open type whose value is of a type determined by the object identifier. The instance-of type is restricted to carrying a value from the class TYPE-IDENTIFIER (see Annex A and Annex C of ITU-T Rec. X.681 | ISO/IEC 8824-2).

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 that Recommendation | International Standard for how to use an information object set to constrain an instance-of type. Note that the encoding of the INSTANCE OF ACCESS-CONTROL-CLASS is the same as that for an EXTERNAL value that has only an object identifier and a data value.

C.3 Identifying abstract syntaxes

C.3.1 Use of the presentation service ITU-T Rec. X.216 | ISO/IEC 8822 requires the specification of values called presentation data values and the grouping of those presentation data values into sets which are called abstract syntaxes. Each of these sets is given an abstract syntax name of ASN.1 type object identifier.

C.3.2 ASN.1 can be used as a general tool in the specification of presentation data values and their grouping into named abstract syntaxes.

C.3.3 In the simplest such use, there is a single ASN.1 type such that every presentation data value in the named abstract syntax is a value of that ASN.1 type. This type will normally be a choice type, and every presentation data value will be an alternative type from this choice type. In this case it is recommended that the ASN.1 module notation be used to contain this choice type as the first defined type, followed by the definition of those (non-universal) types referenced directly or indirectly by this choice type.

NOTE – This is not intended to exclude references to types defined in other modules.

C.3.4 It is recommended that the assignment of an object identifier and object descriptor to an abstract syntax be done using the useful information object class ABSTRACT-SYNTAX which is defined in ITU-T Rec. X.681 | ISO/IEC 8824-2. It is also recommended that all uses of ABSTRACT-SYNTAX be grouped into a single "root" module that identifies all abstract syntaxes used by an application standard.

C.3.5 The following is an example of text which might appear in an application standard:

EXAMPLE

```
ISOxxxx-yyyy {iso standard xxxx asn1-modules(...) yyyy-pdu(...)} DEFINITIONS ::=
BEGIN
      EXPORTS YYYY-PDU;

      YYYY-PDU ::= CHOICE {
            connect-pdu   ...... ,
            data-pdu   CHOICE {
                  ..... ,
                  .....
            },
            .....
      }
      ......
END

ISOxxxx-yyyy-Abstract-Syntax-Module {iso standard xxxx asn1-modules(...) } DEFINITIONS ::=
BEGIN
      IMPORTS YYYY-PDU FROM ISOxxxx-yyyy {iso standard xxxx asn1-modules(...) yyyy-pdu(...)};

-- This Recommendation | International Standard defines the following abstract syntax:

      YYYY-Abstract-Syntax ABSTRACT-SYNTAX ::=
            { YYYY-PDU  IDENTIFIED BY  yyyy-abstract-syntax-object-id }

      yyyy-abstract-syntax-object-id OBJECT IDENTIFIER ::= {iso standard yyyy(xxxx) abstract-syntax(...) }
```

```
-- The corresponding object descriptor is:

    yyyy-abstract-syntax-descriptor ObjectDescriptor ::= "..................."

-- 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,
-- ISOxxxx-yyyy-Abstract-Syntax.

END
```

C.3.6 In order to ensure interworking, the standard may additionally make mandatory the support of the transfer syntax obtained by applying the encoding rules mentioned in its abstract syntax module.

C.4 Subtypes

C.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)
```

C.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.

C.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.

C.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 } )
```

C.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)))
```

C.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 48.8.9.2).

X ::= Z (WITH COMPONENTS { a  PRESENT })                     -- only 'a' can be present (see 48.8.9.2).

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.

C.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 D

Tutorial annex on ASN.1 character strings

(This annex does not form an integral part of this Recommendation | International Standard)

D.1 Character string support in ASN.1

D.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, 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.

NOTE 1 – Use of these types unconstrained leads to violation of the conformance requirements for information interchange specified in ISO/IEC 10646-1, as no adopted subset has been specified.

NOTE 2 – Despite the above, use of this type with a simple subtype constraint which uses a parameter of the abstract syntax (restricted to a defined subtype of UniversalString) can provide a powerful and flexible provision for character handling, relying on profiles to determine the value of the parameter to meet the needs of particular communities of interest. In general, however, the use of CHARACTER STRING is to be preferred in Recommendations | International Standards (see below).

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 IDENTIFIERs 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).

D.2 The UniversalString, – UTF8String and BMPString types

D.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.

D.2.2 The BMPString type carries any character from the Basic Multilingual Plan of ISO/IEC 10646-1 (the first 62K characters). The Basic Multilingual Plane is normally subsetted to a combination of the standard collections of characters in Annex A of ISO/IEC 10646-1.

D.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 37). The "subtype constraint" mechanism allows new subtypes of UniversalString that are combinations of existing subtypes to be defined.

D.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

D.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 (cell references) 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.

D.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)))
```

D.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.

D.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.
```

The PICS 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 PICS

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

}
```

D.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 }
```

D.6 The CHARACTER STRING type

D.6.1 The CHARACTER STRING type gives complete flexibility in the choice of character set and encoding method. Where a single connection provides end-to-end data transfer (no application relaying), then negotiation of the character sets to be used and the encoding can be accomplished as part of the definition of the presentation contexts for character abstract syntaxes.

D.6.2 It is important to understand that 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 registration of such syntaxes, and negotiation of a presentation context, is performed in the normal way.

D.6.3 The encoding of CHARACTER STRING also permits announcement of the abstract and transfer syntax used, without negotiation, for environments where this is appropriate.

NOTE 1 – Application designers may forbid use of presentation negotiation for these fields, or may require it, or may leave the option to the sender.

NOTE 2 – Where announcement rather than negotiation is employed, the application designer should both consider how the sender can determine what character abstract syntaxes (and transfer syntaxes) might be acceptable to the receiver (for example by use of the Directory Service or as a result of profiling), and also consider the actions a receiver is to take if a CHARACTER STRING value is received from a Character Abstract Syntax that it does not support.

D.6.4 If negotiation is used, the application layer designer may control such negotiation, specifying when such presentation contexts are to be established, and specifying the user data parameter of the P-ALTER-CONTEXT primitives, or may simply assume that some profile will have determined which character abstract syntax to use, establishing a presentation context for it at the time of P-CONNECT.

D.6.5 The presentation service context management facilities enable an initiator (in a P-CONNECT or within an established connection using P-ALTER-CONTEXT) to propose a list of new abstract syntaxes (which can include character abstract syntaxes), or to remove abstract syntaxes from use, and for the responder to select from that list.

D.6.6 The initiator can express preference by the order of the abstract syntax in the list, or by use of the user-data parameter, which is available for use by the application designer in order to clarify the purpose of proposing the use of the new abstract syntax. It could, for example, indicate that all the (character) abstract syntaxes are being proposed for use for some single purpose, or that the intent is to allow the selection of a single (character) abstract syntax to be used for a number of purposes.

D.6.7 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.

D.6.8 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 E

Superseded features

(This annex does not form an integral part of this Recommendation | International Standard)

A number of features which were included in previous editions of this Recommendation | International Standard (namely CCITT Rec. X.208 | ISO/IEC 8824) have been replaced and do not now form a part of ASN.1. They may, however, be encountered in some existing ASN.1 modules. This annex describes these features, and how their capabilities can be achieved by the use of those which replace them.

E.1 Use of identifiers now mandatory

The notation for a NamedType and a NamedValue were originally:

NamedType ::= identifier Type | Type | SelectionType
NamedValue ::= identifier Value | Value

but this has been changed to:

NamedType ::= identifier Type
NamedValue ::= identifier Value

because the former could result in ambiguous grammar.

Identifiers can be added to "NamedType"s in old ASN.1 specifications without affecting the encoding of the type (although changes to the ASN.1 will be needed for any use of related value notation). Such change should be done either under a defect report or as part of a new revision of the Recommendation | International Standard being modified.

E.2 The choice value

The value notation for the choice type was originally:

ChoiceValue ::= NamedValue

NamedValue ::= identifier Value | Value

but this has been changed to:

ChoiceValue ::= identifier ":" Value

because the former could result in ambiguous grammar.

E.3 The any type

The any type was defined in some earlier versions of this Recommendation | International Standard.

The normal use of the any type was to leave a "hole" in the specification which would be filled in by some other specification. The notation was "AnyType", allowed as an alternative for "Type", and specified as:

AnyType ::= ANY | ANY DEFINED BY identifier

It was strongly recommended that the second alternative of the notation be used. In this alternative, only allowed where the any type was one of the component types of a set or sequence type, some other component of the set or sequence (that with the referenced "identifier") would indicate, by its integer or object identifier value (or a choice of these), the actual type governing the any component. The mapping from such values to particular ASN.1 types could be viewed as some sort of "table" which would form part of the abstract syntax. In the absence of the "DEFINED BY identifier" (the first notational alternative), there would be no indication within the notation of how the type of the field could be determined. This frequently led to specifications where the "hole" continued to exist even at the stage where implementations were expected.

The any type has now been superseded by the ability to specify information object classes and then to refer to the fields of information object classes from within type definitions (see ITU-T Rec. X.681 | ISO/IEC 8824-2). Since fields may be defined to allow an arbitrary ASN.1 type, the basic ability to leave "holes" is provided. However, the new feature also permits the specification of a "table constraint", wherein a particular "information object set" (a set of information objects of the appropriate information object class) is explicitly cited as constraining the type. This latter capability encompasses that offered by "**ANY DEFINED BY** identifier".

In addition, some pre-defined uses of the new capabilities are provided (see ITU-T Rec. X.681 | ISO/IEC 8824-2), which correspond to various commonly occurring patterns of use of the any type. For example, a sequence containing an object identifier and an any, often used previously to convey some arbitrary value together with an indication of its type, can now be described as:

```
INSTANCE OF MUMBLE
```

where **MUMBLE** is defined as an information object class (not as an ASN.1 type):

```
MUMBLE ::= TYPE-IDENTIFIER
```

This notation causes "INSTANCE OF MUMBLE" to be replaced by an object identifier for an object of class **MUMBLE**, together with the type identified by the object identifier. See C.2.18 for an example.

Particular pairings of object identifier and type are defined as information objects of class **MUMBLE**, and, if required, particular sets of these can also be defined and used to constrain the **INSTANCE OF** construct so that only those objects in the set can appear.

The macro capability was often used as a semi-formal way of defining tables of information objects to govern an associated use of an any type, and is also superseded by the new capabilities.

E.4 The macro capability

The macro capability allowed the user of ASN.1 to extend the notation by defining macros.

The predominant usage of the macro capability was to define notation for specifying information objects. This capability has now been included in ASN.1 directly (see ITU-T Rec. X.681 | ISO/IEC 8824-2) without the need for the full generality (and attendant dangers) of user-defined notation.

Besides this, the only other usage for macros seems to be in defining expressions which must be supplied with some parameters before they are fully-defined ASN.1 types. This is now provided through the more general parameterization capability (see ITU-T Rec. X.683 | ISO/IEC 8824-4).

Annex F

Tutorial annex on the ASN.1 model of type extension

(This annex does not form an integral part of this Recommendation | International Standard)

F.1 Overview

F.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**.

F.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.

F.1.3 The ASN.1 type definitions in this series are said to be **extension-related** (see 3.8.32 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.

F.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     1st extension           2nd extension           3rd extension

A ::= SEQUENCE {        A ::= SEQUENCE {        A ::= SEQUENCE {        A ::= SEQUENCE {
    a   INTEGER,            a   INTEGER,            a   INTEGER,            a   INTEGER,
    ...                     ...,                    ...,                    ...,
}                           b   BOOLEAN,            b   BOOLEAN,            b   BOOLEAN,
                            c   INTEGER             c   INTEGER,            c   INTEGER,
                        }                           d   SEQUENCE {          d   SEQUENCE {
                                                          e   INTEGER,            e   INTEGER,
                                                          ...,                    ...,
                                                          ...,                    g   BOOLEAN OPTIONAL,
                                                          f   IA5String           h   BMPString,
                                                    }                             ...,
                                                }                                 f   IA5String
                                                                            }
                                                                        }
```

F.1.5 All extension additions 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.

F.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.

F.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 1st 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 ::= SEQUENCE {            A ::= SEQUENCE {            A ::= SEQUENCE {
     a   INTEGER,               a    INTEGER,               a    INTEGER,               a    INTEGER,
     ...                        ...,                        ...,                        ...,
}                               [[                          [[                          [[
                                 b   BOOLEAN,                b   BOOLEAN,                b   BOOLEAN,
                                 c   INTEGER                 c   INTEGER                 c   INTEGER
                                ]]                          ]],                         ]],
                            }                               d    SEQUENCE {             d    SEQUENCE {
                                                                     e   INTEGER,                    e   INTEGER,
                                                                     ...,                        ...,
                                                                     ...,                        [[
                                                                     f   IA5String                g  BOOLEAN OPTIONAL,
                                                                }                                 h  BMPString
                                                            }                                    ]],
                                                                                                 ...,
                                                                                                 f   IA5String
                                                                                            }
                                                                                        }
```

F.1.8 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.

F.1.9 We start with a type and then decide whether we are going to want interworking with 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 without changing the bits on the line for earlier values, and with defined handling of 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) **will** in general change the bits on the line and will prevent interworking. Such a change generally requires a version number change in all affected protocols.

F.1.10 Table F.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).

Table F.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

F.2 Effects on version numbering, etc.

F.2.1 Where an ASN.1 specification is re-issued with type definitions changed in terms of extension-related type definitions, then for all purposes, these changes do not in themselves require a change in the object identifier of the module or the version number of the protocol.

F.2.2 It may be that for other reasons such changes might be accompanied by version number changes, but this is not a requirement.

F.2.3 By contrast, the addition of an extension marker to a type that previously had none, or the addition of components to a sequence or set type (for example) with no extension marker, creates a new type which is **not** extension-related to the old type, and the module containing it should be given a new object identifier, and a new version number would be appropriate in the associated protocol.

F.3 Requirements on encoding rules

F.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.

F.3.2 A set of well-formed encoding rules for an extension-related abstract syntax satisfies the additional requirements stated in F.3.3 to F.3.5.

NOTE – All ASN.1 encoding rules satisfy these requirements.

F.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.

F.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.

F.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.

Annex G

Summary of the ASN.1 notation

(This annex does not form an integral part of this Recommendation | International Standard)

The following items are defined in clause 11:

typereference

identifier

valuereference

modulereference

comment

empty

number

bstring

hstring

cstring

"::="

[[

]]

".."

"..."

"{"

"}"

"<"

","

"."

"("

")"

"["

"]"

"-"

":"

";"

"@"

"|"

"!"

"^"

ABSENT

ABSTRACT-SYNTAX

ALL

APPLICATION

AUTOMATIC

BEGIN

BIT

BMPString

BOOLEAN

BY

CHARACTER

CHOICE

CLASS

COMPONENT

COMPONENTS

CONSTRAINED

DEFAULT

DEFINITIONS

EMBEDDED

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

PDV

PLUS-INFINITY

PRESENT

PrintableString

PRIVATE

REAL

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 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 ";" |
                      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 | Symbol "," SymbolList

Symbol  ::= Reference | ParameterizedReference

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

```
AssignmentList ::=     Assignment  | AssignmentList Assignment

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

Externaltypereference ::=
                  modulereference
                  "."
                  typereference

Externalvaluereference ::=
                  modulereference
                  "."
                  valuereference

DefinedType ::=
            Externaltypereference |
            typereference |
            ParameterizedType |
            ParameterizedValueSetType

DefinedValue ::=
            Externalvaluereference |
            valuereference |
            ParameterizedValue

AbsoluteReference ::= "@" GlobalModuleReference
            "."
            ItemSpec

ItemSpec ::=
            typereference |
            ItemId "." ComponentId

ItemId ::= ItemSpec

ComponentId ::=
            identifier | number | "*"

TypeAssignment ::=        typereference
                          "::="
                          Type

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

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 |
        SequenceType |
        SequenceOfType |
        SetType |
        SetOfType |
        TaggedType

NamedType ::= identifier   Type

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

Value  ::=   BuiltinValue | ReferencedValue

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

ReferencedValue ::=
        DefinedValue |
        ValueFromObject

NamedValue ::= identifier Value

BooleanType  ::=BOOLEAN

BooleanValue::=      TRUE | FALSE

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

NamedNumberList ::=
                NamedNumber         |
                NamedNumberList "," NamedNumber

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

SignedNumber ::= number | "-" number

IntegerValue ::=      SignedNumber | identifier

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

Enumerations ::= RootEnumeration |
            RootEnumeration  ","   "..." |
            RootEnumeration  ","   "..."   ","   AdditionalEnumeration
RootEnumeration ::= Enumeration

AdditionalEnumeration ::= Enumeration

Enumeration ::=
            EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem ::=
            identifier | NamedNumber

EnumeratedValue  ::=
            identifier

RealType ::=    REAL

RealValue ::=
            NumericRealValue | SpecialRealValue

NumericRealValue ::= 0 |
            SequenceValue               -- Value of the associated sequence type

SpecialRealValue ::=
            PLUS-INFINITY | MINUS-INFINITY

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

NamedBitList::=            NamedBit | NamedBitList "," NamedBit

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

BitStringValue   ::=       bstring | hstring | "{" IdentifierList "}" | "{"  "}"

IdentifierList   ::=       identifier | IdentifierList "," identifier

OctetStringType  ::=       OCTET STRING

OctetStringValue ::=       bstring | hstring

NullType      ::=          NULL

NullValue    ::=NULL


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

RootComponentTypeList  ::=  ComponentTypeList

ExtensionEndMarker  ::=  “,”  “...”

ExtensionAdditions  ::=  “,”  ExtensionAdditionList  |  empty

ExtensionAdditionList  ::=  ExtensionAddition  |
            ExtensionAdditionList  “,”  ExtensionAddition

ExtensionAddition  ::= ComponentType  |  ExtensionAdditionGroup

ExtensionAdditionGroup  ::=  “[[”  ComponentTypeList  “]]”

ComponentTypeList  ::=             ComponentType   |
                                   ComponentTypeList "," ComponentType

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

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

ComponentValueList ::=        NamedValue    |
                              ComponentValueList "," NamedValue

SequenceOfType   ::=          SEQUENCE OF Type

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

ValueList    ::= Value | ValueList "," Value

      SetType ::= SET  “{”  “}” |
                  SET  “{”  ExtensionAndException  OptionalExtensionMarker  “}”  |
                  SET “{”  ComponentTypeLists  “}”

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

SetOfType    ::=       SET OF Type

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

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

AlternativeTypeLists ::=
            RootAlternativeTypeList |
            RootAlternativeTypeList  “,”
                       ExtensionAndException  ExtensionAdditionAlternatives  OptionalExtensionMarker

RootAlternativeTypeList  ::=  AlternativeTypeList

ExtensionAdditionAlternatives  ::=  “,”  ExtensionAdditionAlternativesList  |  empty

ExtensionAdditionAlternativesList  ::= ExtensionAdditionAlternative  |
            ExtensionAdditionAlternativesList  “,”  ExtensionAdditionAlternative

ExtensionAdditionAlternative  ::=  ExtensionAdditionAlternatives | NamedType

ExtensionAdditionAlternatives  ::=  “[[”  AlternativeTypeList  “]]”

AlternativeTypeList ::=             NamedType     |
                                    AlternativeTypeList "," NamedType

ChoiceValue  ::=              identifier ":" Value

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

EmbeddedPDVType ::=                 EMBEDDED PDV

EmbeddedPDVValue ::=                SequenceValue

ExternalType ::= EXTERNAL

ExternalValue ::= SequenceValue

ObjectIdentifierType  ::=           OBJECT IDENTIFIER

ObjectIdentifierValue ::=           "{" ObjIdComponentList "}"  |
                                    "{" DefinedValue ObjIdComponentList "}"

ObjIdComponentList  ::=             ObjIdComponent      |
                                    ObjIdComponent ObjIdComponentList
ObjIdComponent  ::=     NameForm             |
                        NumberForm   |
                        NameAndNumberForm

NameForm     ::=        identifier

NumberForm     ::=      number | DefinedValue

NameAndNumberForm ::=           identifier "(" NumberForm ")"

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

UnrestrictedCharacterStringType ::= CHARACTER STRING

CharacterStringValue ::= RestrictedCharacterStringValue | UnrestrictedCharacterStringValue

UnrestrictedCharacterStringValue ::= SequenceValue

UsefulType    ::= typereference
```

The following character string types are defined in 36.1:

NumericStringVisibleString

PrintableString   ISO646String

TeletexString   IA5String

T61String   GraphicString

VideotexString   GeneralString

UniversalString   BMPString

The following useful types are defined in clauses 41 to 43:

GeneralizedTime

UTCTime

ObjectDescriptor

The following productions are used in clauses 44 to 48:

```
ConstrainedType ::=
      Type   Constraint |
      TypeWithConstraint
```

```
TypeWithConstraint ::=
    SET Constraint OF Type |
    SET SizeConstraint OF Type |
    SEQUENCE Constraint OF Type |
    SEQUENCE SizeConstraint OF Type

Constraint ::= "(" ConstraintSpec  ExceptionSpec ")"

ConstraintSpec ::=
    SubtypeConstraint |
    GeneralConstraint

ExceptionSpec ::= "!"  ExceptionIdentification | empty

ExceptionIdentification ::= SignedNumber    |
            DefinedValue   |
            Type ":" Value

SubtypeConstraint ::= ElementSetSpecs

ElementSetSpecs ::=
        RootElementSetSpec |
        RootElementSetSpec  ","   "..."  |
        "..."  ","  AdditionalElementSetSpec |
        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

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
```

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the 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	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
Series Z	Programming languages