INTERNATIONAL TELECOMMUNICATION UNION

ITU-T X.208

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

OPEN SYSTEMS INTERCONNECTION
MODEL AND NOTATION

SPECIFICATION OF ABSTRACT SYNTAX NOTATION ONE (ASN.1)

ITU-T Recommendation X.208

(Extract from the *Blue Book*)

NOTES

1 ITU-T Recommendation X.208 was published in Fascicle VIII.4 of the *Blue Book*. This file is an extract from the *Blue Book*. While the presentation and layout of the text might be slightly different from the *Blue Book* version, the contents of the file are identical to the *Blue Book* version and copyright conditions remain unchanged (see below).

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

Recommendation X.208

SPECIFICATION OF ABSTRACT SYNTAX NOTATION ONE (ASN.1)[1]

(Melbourne, 1988)

The CCITT,

considering

(a) the variety and complexity of information objects conveyed within the application layer;

(b) the need for a high-level notation for specifying such information objects;

(c) the value of isolating and standardizing the rules for encoding such information objects,

unanimously recommends

(1) that the notation for defining the abstract syntax of information objects is defined in Section 1;

(2) that character string types are defined in Section 2;

(3) that other useful types are defined in Section 3;

(4) that subtypes are defined in Section 4.

CONTENTS

[1] Recommendation X.208 and ISO 8824 [Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)] as extended by Addendum 1 to ISO 8824, were developed in close cooperation and are technically aligned.

0 Introduction

In the lower layers of the Basic Reference Model (see Recommendation X.200), each user data parameter of a service primitive is specified as the binary value of a sequence octets.

In the presentation layer, the nature of user data parameters changes. Application layer specifications require the presentation service user data (see Recommendation X.216) to carry the value of quite complex types, possibly

including strings of characters from a variety of character sets. In order to specify the value which is carried, they require a defined notation which does not determine the representation of the value. This is supplemented by the specification of one or more algorithms called **encoding rules** which determine the value of the session layer octets carrying such application layer values (called the **transfer syntax**). The presentation layer protocol (see Recommendation X.226) can negotiate which transfer syntaxes are to be used.

The purpose of specifying a value is to distinguish it from other possible values. The collection of the value together with the values from which it is distinguished is called a **type**, and one specific instance is a value of that type. More generally, a value or type can often be considered as composed of several simpler values or types, together with the relationships between them. The term datatype is often used as a synonym for type.

In order to correctly interpret the representation of a value (whether by marks on paper or bits on communication line), it is necessary to know (usually from the context), the type of the value being represented. Thus the identification of a type is an important part of this Recommendation.

A very general technique for defining a complicated type 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 a list 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 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 (ordered) sequence or (unordered) set of zero, one or more values of the existing type; the (infinite) collection of all possible 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;

Types which are defined in this way are called **structured types**.

Every type defined using the notation specified in this Recommendation is assigned a **tag**. The tag is defined either by this Recommendation or by the user of the notation.

It is common for the same tag to be assigned to many different types, the particular type being identified by the context in which the tag is used.

The user of the notation may choose to assign distinct tags to two occurrences of a single type, thereby creating two distinct types. This can be necessary when it is required to distinguish which choice has been made in situations such as d) above.

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, and each tag is either

a) assigned to a single type; or

b) assigned to a construction mechanism.

The second class of tag is the **application** class. Application class tags are assigned to types by other standards or Recommendations. Within a particular standard or Recommendation, an application class tag is assigned to only one type.

The third class is the **private** class. Private class tags are never assigned by ISO Standards or CCITT Recommendations. Their use is enterprise specific.

The final class of tag is the **context-specific** class. This is freely assigned within any use of this notation, and is interpreted according to the context in which it is used.

Tags are mainly intended for machine use, and are not essential for the human notation defined in this Recommendation. 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 1 – All types which can be defined in the notation of this Recommendation have a tag. Given any type, the user of the notation can define a new type with a different tag.

Note 2 – Encoding rules always carry the tag of a type, explicitly or implicitly, with any representation of a value of the type. The restrictions placed on the use of the notation are designed to ensure that the tag is sufficient to unambiguously determine the actual type, provided the applicable type of definitions are available.

This Recommendation specifies a notation which both enables complicated types to be defined and also enables values of these types to be specified. This is done without determining the way an instance of this type is to be represented (by a sequence of octets) during transfer. A notation which provides this facility is called a **notation for abstract syntax definition**.

The purpose of this Recommendation is to specify a notation for abstract syntax definition called **Abstract Syntax Notation One**, or ASN.1. Abstract Syntax Notation One is used as a semi-formal tool to define protocols. The use of the notation does not necessarily preclude ambiguous specifications. It is the responsibility of the users of the notation to ensure that their specifications are not ambiguous.

This Recommendation is supported by other standards and Recommendations which specify encoding rules. The application of encoding rules to the value of a type defined by ASN.1 results in a complete specification of the representation of values of that type during transfer (a transfer syntax).

This Recommendation is technically and editorially aligned with ISO 8824 plus Addendum 1 to ISO 8824.

Section one of this Recommendation defines the simple types supported by ASN.1, and specifies the notation to be used for referencing simple types and defining structured types. Section one also specifies the notation to be used for specifying values of types defined using ASN.1.

Section two of this Recommendation defines additional types (character string types) which, by the application of encoding rules for character sets, can be equated with the octetstring type.

Section three of this Recommendation defines certain structured types which are considered to be of general utility, but which require no additional encoding rules.

Section four of this Recommendation defines a notation which enables subtypes to be defined from the values of a parent type.

Annex A is part of this Recommendation, and specifies a notation for extending the basic ASN.1 notation. This is called the macro facility.

Annex B is part of this Recommendation, and defines the object identifier tree for authorities supported by ISO.

Annex C is part of this Recommendation and defines the object identifier tree for authorities supported by CCITT.

Annex D is part of this Recommendation and defines the object identifier tree for joint use by ISO and CCITT.

Appendix I is not part of this Recommendation, and provides examples and hints on the use of the ASN.1 notation.

Appendix II is not part of this Recommendation, and provides a summary of ASN.1 using the notation of § 5.

The text of this Recommendation, and in particular the annexes B to D, are the subject of joint ISO-CCITT agreement.

1 Scope and field of application

This Recommendation specifies a notation for abstract syntax definition called ςAbstract Syntax Notation One (ASN.1).

This Recommendation defines a number of simple types, with their tags, and specifies a notation for referencing these types and for specifying values of these types.

This Recommendation defines mechanisms for constructing new types from more basic types, and specifies a notation for defining such structured types and assigning them tags, and for specifying values of these types.

This Recommendation defines character sets for use within ASN.1.

This Recommendation 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 also referenced by other presentation layer standards and Recommendations which define encoding rules for the simple types, the structured types, the character string types and the useful types defined in ASN.1.

2 References

[1] Recommendation X.200, *Reference Model of Open Systems Interconnection for CCITT Applications* (see also ISO 7498).

[2] Recommendation X.209, *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)* (see also ISO 8825).

[3] Recommendation X.216, (see also ISO 8822), *Presentation Service Definition for Open Systems Interconnection for CCITT Applications*.

[4] Recommendation X.226, *Presentation Protocol Specification for Open Systems Interconnection for CCITT Applications* (see also ISO 8823).

[5] ISO 2014, *Writing of calendar dates in all-numeric form*.

[6] ISO 2375, *Data processing – Procedure for registration of escape sequences*.

[7] ISO 3166, *Codes for the representation of names of countries.*

[8] ISO 3307, *Information interchange – Representations of time of the day*.

[9] ISO 4031, Information interchange – *Representation of local time differentials*.

[10] ISO 6523, *Data interchange – Structure for identification of organizations.*

[11] Recommendation X.121, *International numbering plan for public data networks*.

3 Definitions

The definitions in Recommendation X.200 are used in this Recommendation.

3.1 **value**

A distinguished member of a set of values.

3.2 **type**

A named set of values.

3.3 **simple type**

A type defined by directly specifying the set of its values.

3.4 **structured type**

A type defined by reference to one or more other types.

3.5 **component type**

One of the types referenced when defining a structured type.

3.6 **tag**

A type denotation which is associated with every ASN.1 type.

3.7 **tagging**

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

3.8 **ASN.1 character set**

The set of characters, specified in § 7, used in the ASN.1 notation.

3.9 **items**

Named sequences of characters from the ASN.1 character set, specified in § 8, which are used to form the ASN.1 notation.

3.10 **type (or value) reference name**

A name associated uniquely with a type (or value) within some context.

Note – Reference names are assigned to the types defined in this Recommendation; these are universally available within ASN.1. Other reference names are defined in other standards and Recommendations, and are applicable only in the context of the standard or Recommendation.

3.11 **ASN.1 encoding rules**

Rules which specify the representation during transfer of the value of any ASN.1 type; ASN.1 encoding rules enable information being transferred to be identified by the recipient as a specific value of a specific ASN.1 type.

3.12 **character string type**

A type whose values are strings of characters from some defined character set.

3.13 **Boolean type**

A simple type with two distinguished values.

3.14 **true**

One of the distinguished values of the Boolean type.

3.15 **false**

The other distinguished value of the Boolean type.

3.16 **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.

3.17 **enumerated type**

A simple type whose values are given distinct identifiers as part of the type notation.

3.18 **real type**

A simple type whose distinguished values (specified in § 16.2) are members of the set of real numbers.

3.19 **bitstring type**

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

Note – Encoding rules do not limit the number of bits in a bit-string.

3.20 **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.

Note – Encoding rules do not limit the number of octets in an octet string.

3.21 **null type**

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

Note – The null value is commonly used where several alternatives are possible, but none of them apply.

3.22 **sequence type**

A structured type, defined by referencing a fixed, ordered, list of types (some of which may be declared to be optional); each value of the new 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 new type need not contain a value of that component type.

3.23 **sequence-of type**

A structured type, defined by referencing a single existing type; each value in the new type is an ordered list of zero, one or more values of the existing type.

Note – Encoding rules do not limit the number of values in a sequence-of value.

3.24 **set type**

A structured type, defined by referencing a fixed, unordered, list of distinct types (some of which may be declared to be optional); each value in the new 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.

3.25 **set-of type**

A structured type, defined by referencing a single existing type; each value in the new type is an unordered list zero, one or more values of the existing type.

Note – Encoding rules do not limit the number of values in a set-of value.

3.26 **tagged type**

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.

3.27 **choice type**

A structured type, defined by referencing a fixed, unordered, list of distinct types; each value of the new type is a value of one of the component types.

3.28 **selection type**

A structured type, defined by reference to a component type of a choice type.

3.29 **any type**

A choice type whose component types are unspecified, but are restricted to the set of types which can be defined using ASN.1.

3.30 **external type**

A type whose distinguished values cannot be deduced from their characterisation as external, but which can be deduced from the encoding of such a value; the value may, but need not, be describable using ASN.1, and thus their encoding may, but need not, conform to ASN.1 encoding rules.

3.31 **information 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.

3.32 **object identifier**

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

3.33 **object identifier type**

A type whose distinguished values are the set of all object identifiers allocated in accordance with the rules of this Recommendation.

Note – The rules of this Recommendation permit a wide range of authorities to independently associate object identifiers with information objects.

3.34 **object descriptor type**

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

Note – An object descriptor value is usually, but not always associated with a single information object. Only an object identifier value unambiguously identifies an information object.

3.35 **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 of ensuring that those values (of the resulting types) which are used have a finite representation.

3.36 **module**

One or more instances of the use of the ASN.1 notation for type and value definition, encapsulated using the ASN.1 module notation (see § 9).

3.37 **production**

A part of the formal notation used to specify ASN.1, in which allowed sequences of items are associated with a name which can be used to reference those sequences in the definition of new sets of allowed sequences.

3.38 **Coordinated Universal Time (UTC)**

The time scale maintained by the Bureau Internationale 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 is also referred to as Greenwich Mean Time and appropriate time signals are regularly broadcast.

3.39 **user (of ASN.1)**

The individual or organization that defines the abstract syntax of a particular piece of information using ASN.1.

3.40 **subtype (of a parent type)**

A type whose values are specified as a subset of the values of some other type (the parent type).

3.41 **parent type (of a subtype)**

Type used to define a subtype.

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

3.42 **subtype specification**

A notation which can be used in association with the notation for a type, to define a subtype of that type.

3.43 **subtype value set**

A notation forming part of a subtype specification, specifying a set of values of the parent type which are to be included in the subtype.

3.44 This Recommendation uses the following terms defined in Recommendation X.216:

a) presentation data value; and

b) (an) abstract syntax; and

c) abstract syntax name; and

d) transfer syntax name.

3.45 This Recommendation also uses the following terms defined in ISO 6523;

a) issuing organization; and

b) organization code; and

c) International Code Designator.

3.46 This Recommendation uses the following term defined in Recommendation X.226:

a) presentation context identifier.

4 Abbreviations

ASN.1 Abstract Syntax Notation One.

UTC Coordinated Universal Time.

ICD International Code Designator.

DCC Data Country Code

DNIC Data Network Identification Code.

5 Notation used in this Recommendation

The ASN.1 notation consists of a sequence of characters from the ASN.1 character set specified in § 7.

Each use of the ASN.1 notation contains characters from the ASN.1 character set grouped into items. Clause 8 specifies all the sequences of characters forming ASN.1 items, and names each item.

The ASN.1 notation is specified in § 9 (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 uses a formal notation defined in the following sub-clauses.

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 sequences defined in this Recommendation and forms a new collection of sequences by specifying either

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

b) that the new collection is to consist of any sequence which can be generated by taking exactly one 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 sequences;

b) the characters

::=

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

|

A 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 by the name in a) above.

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

5.2 *The alternative collections*

Each of the alternative collections of sequences in "one or more alternative collections of" is specified by a list of names. Each name is either the name of an item, or is the name of a collection of sequences defined by a production in this Recommendation.

The collection of sequences defined by the alternative consists of all sequences obtained by taking any one of the sequences (or the item) associated with the first name, in combination with (and followed by) any one of the sequences (or item) associated with the second name, in combination with (and followed by) any one of the 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 sequences:

a) any bstring (an item); and

b) any hstring (an item); and

c) any sequence associated with **IdentifierList**, preceded by a { and followed by a }

Note – { and } are the names of items containing the single characters { and } (see § 8).

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

5.4 *Layout*

Each production used in this Recommendation 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 are frequently recursive. In this case the productions are to be continously 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 references a collection of sequences (part of the ASN.1 notation) by referencing the first name (before the ::=) in a production; the name is surrounded by " to distinguish it from natural language text, unless it appears as part of a production.

5.7 *References to an item*

This Recommendation references an item by referencing the name of the item; the name is surrounded by " to distinguish it from natural language text, unless it appears as part of a production.

TABLE 1/X.208

Universal class tag assignments

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
UNIVERSAL 9	Real type
UNIVERSAL 10	Enumerated type
UNIVERSAL 12 to 15	Reserved forfuture versions of this Recommendation
UNIVERSAL 16	Sequence and Sequence-of types
UNIVERSAL 17	Set and Set-of types
UNIVERSAL 18 to 22, 25 to 27	Character string types
UNIVERSAL 23, 24	Time types
UNIVERSAL 28- . . .	Reserved for future versions of this Recommendation

5.8 *Tags*

A tag is specified by giving its class and the number within the class. The class is one of

universal
application

private
context-specific

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

Restrictions on tags assigned by the user of ASN.1 are specified in § 26.

Tags in the universal class are assigned in such a way that, for structured types, the top-level structure can be deduced from the tag, and for simple types, the type can be deduced from the tag. Table 1/X.208 summarises the assignment of tags in the universal class which are specified in this Recommendation.

Note – Additional tags in the universal class are reserved for assignment by future versions of this Recommendation.

6 Use of the ASN.1 notation

6.1 The ASN.1 notation for a type definition shall be “Type” (see § 12.1).

6.2 The ASN.1 notation for a value of a type shall be “Value” (see § 12.7).

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

6.3 The ASN.1 notation for assigning a type to a type reference name shall be “Typeassignment” (see § 11.1).

6.4 The ASN.1 notation for assigning a value reference name shall be “Valueassignment” (see § 11.2).

6.5 The notation “Typeassignment” and “Valueassignment” shall only be used within the notation “ModuleDefinition” (but see § 9.1).

SECTION 1 - SPECIFICATION OF ASN.1 NOTATION

7 The ASN.1 character set

7.1 An ASN.1 item shall consist of a sequence of the characters listed in Table 2/X.208, except as specified in § 7.2 and § 7.3.

TABLE 2/X.208

ASN.1 characters

A to Z
a to z
0 to 9
: = , { } < .
( ) [ ] - ' ”

Note 1 – The additional characters > and | are used in the macro-notation (see Annex A).

Note 2 – Where equivalent derivative standards are develped by national standards bodies, additional characters may appear in the following items (the last five of which are defined in Annex A):

typereference (§ 8.2.1)
identifier (§ 8.3)

Valuereference	(§ 8.4)
modulereference	(§ 8.5)
macroreference	(§ A.2.1)
productionreference	(§ A.2.2)
localtypereference	(§ A.2.3)
localvaluereference	(§ A.2.4)
astring	(§ A.2.7)

Note 3 – 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.

7.2 Where the notation used to specify the value of a character string type, all characters of the defined character set can appear in the ASN.1 notation, surrounded by the characters ". (See § 8.11.)

7.3 Additional characters may appear in the "comment" item. (See § 8.6.)

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

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

8 ASN.1 items

8.1 *General rules*

8.1.1 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.

Note – Annex A specifies additional items used in the macro notation.

8.1.2 Each item specified in the following subclauses shall appear on a single line, and (except for the "comment" item) shall not contain spaces.

8.1.3 The length of a line is not restricted.

8.1.4 The items in the sequences specified by this Recommendation (the ASN.1 notation) may appear on one line or may appear on several lines, and may be separated by one or more spaces or empty lines.

8.1.5 An item shall be separated from a following item by a space, or by being placed on a separate line, 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.

8.2 *Type references*

Name of item-typereference.

8.2.1 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.

8.2.2 A "typereference" shall not be one of the reserved character sequences listed in Table 3/X.208.

Note – § A.2.9 specifies additional reserved character sequences when within a macro definition.

8.3 *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.

TABLE 3/X.208

Reserved character sequences

BOOLEAN	BEGIN
INTEGER	END
BIT	DEFINITIONS
STRING	EXPLICIT
OCTET	ENUMERATED
NULL	EXPORTS
SEQUENCE	IMPORTS
OF	REAL
SET	INCLUDES
IMPLICIT	MIN
CHOICE	MAX
ANY	SIZE
EXTERNAL	FROM
OBJECT	WITH
IDENTIFIER	COMPONENT
OPTIONAL	PRESENT
DEFAULT	ABSENT
COMPONENTS	DEFINED
UNIVERSAL	BY
APPLICATION	PLUS-INFINITY
PRIVATE	MINUS-INFINITY
TRUE	TAGS
FALSE	

8.4 *Value references*

Name of item-valuereference.

A "valuereference" shall consist of the sequence of characters specified for an "identifier" in § 8.3. In analysing an instance of use of this notation, a "valuereference" is distinguished from an "identifier" by the context in which it appears.

8.5 *Module reference*

Name of item-modulereference.

A "modulereference" shall consist of the sequence of characters specified for a "typereference" in § 8.2. In analysing an instance of use of this notation, a "modulereference" is distinguished from a "typereference" by the context in which it appears.

8.6 *Comment*

Name of item-comment.

8.6.1 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 significance.

8.6.2 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 characters which are not in the character set specified in § 7.1 (see § 7.3).

8.7 *Empty item*

Name of item-empty.

The “empty” item contains no characters. It is used in the notation of § 5 when alternative sets of sequences are specified, to indicate that absence of all aternatives is possible.

8.8 *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.

8.9 *Binary string item*

Name of item-bstring.

A “bstring” shall consist of an arbitrary number (possibly zero) of zeros and ones, preceded by a single ' and followed by the pair of characters:

'B

Example: '01101100'B

8.10 *Hexadecimal string item*

Name of item-hstring.

8.10.1 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

preceded by a single ' and followed by the pair of characters

'H

Example: 'AB0196'H

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

8.11 *Character string item*

Name of item-cstring.

A “cstring” shall consist of an arbitrary number (possibly zero) of characters from the character set referenced by a character string type, preceded and followed by ". If the character set includes the character ", this character shall be represented in the “cstring” by a pair of ". The character set involved is not limited to the character set listed in Table 2/X.208, but is determined by the type for which the “cstring” is a value (see § 7.2).

Example: "宛 姐 虻 飴 鵠"

8.12 *Assignment item*

Name of item-”::=”

This item shall consist of the sequence of characters

::=

Note – This sequence does not contain any space characters (see § 8.1.2).

8.13 *Single character items*

Names of items-

{
}
<
,
.
(
)
[
]
- (hyphen)
;

An item with any of the names listed above shall consist of the single character forming the name.

8.14 *Keyword items*

Names of items -

BOOLEAN
INTEGER
BIT
STRING
OCTET
NULL
SEQUENCE
OF
SET
IMPLICIT
CHOICE
ANY
EXTERNAL
OBJECT
IDENTIFIER
OPTIONAL
DEFAULT
COMPONENTS
UNIVERSAL
APPLICATION
PRIVATE
TRUE
FALSE
BEGIN
END
DEFINITIONS
EXPLICIT
ENUMERATED
EXPORTS
IMPORTS
REAL
INCLUDES
MIN
MAX
SIZE
FROM
WITH
COMPONENT
PRESENT
ABSENT
DEFINED

BY
PLUS-INFINITY
MINUS-INFINITY
TAGS

Items with the above names shall consist of the sequence of characters in the name.

Note 1 – Spaces do not occur in these sequences.

Note 2 – Where these sequences are not listed as reserved sequences in § 8.2.2, they are distinguished from other items containing the same characters by the context in which they appear.

9 Module definition

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

ModuleDefinition ::=

ModuleIdentifier
DEFINITIONS
TagDefault
"::="
BEGIN
ModuleBody
END

TagDefault ::=

EXPLICIT TAGS |
IMPLICIT TAGS |
empty

ModuleIdentifier ::=

modulereference
AssignedIdentifier

AssignedIdentifier ::=

ObjectIdentifierValue |
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 ModuleIdentifier

SymbolList ::= Symbol, SymbolList | Symbol

Symbol ::=typereference | valuereference

AssignmentList ::=

Assignment AssignmentList |
Assignment

Assignment ::=

TypeAssignment | ValueAssignment

Note 1 – Annex A specifies a "MacroDefinition" sequence which can also appear in the "AssignmentList". Notations defined by a macro definition may appear before or after the macro definition, within the same module.

Note 2 – In individual (but deprecated) cases, and for examples and for the definition of types with universal class tags, the "ModuleBody" can be used outside of a "ModuleDefinition".

Note 3 – "Typeassignment" and "Valueassignment" productions are specified in § 11.

Note 4 – The grouping of ASN.1 datatypes 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 – A "macroreference" (see Annex A), can also appear as a "Symbol".

9.2 The "TagDefault" is taken as "EXPLICIT TAGS" if it is "empty".

Note – § 26 gives the meaning of both "EXPLICIT TAGS" and "IMPLICIT TAGS".

9.3 The "modulereference" appearing in the "ModuleDefinition" production is called the module name. Module names are chosen so as to ensure consistency and completeness of all "Assignment" sequences appearing within the "ModuleBody" of all "ModuleDefinition" sequences with this module name. A set of "Assignment" sequences is consistent and complete if, for every "typereference" or "valuereference" appearing within it, there is exactly one "Typeassignment" or "Valueassignment" (respectively) associating the name with a type or value (respectively), or exactly one "SymbolsFromModule" in which the "typereference" or "valuereference" (respectively) appears as a "Symbol".

9.4 Module names should be used only once (except as specified in § 9.10) within the sphere of interest of the definition of the module.

Note – It is recommended that modules defined in ISO Standards should have module names of the form

ISOxxxx-yyyy

where xxxx is the number of the Standard, and yyyy is a suitable acronym for the Standard (e.g. JTM, FTAM, or CCR). A similar convention can be applied by other standards-making bodies.

9.5 If the "AssignedIdentifier" includes an "ObjectIdentifierValue", the latter unambiguously and uniquely identifies the module.

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

9.6 The "ModuleIdentifier" in a "SymbolsFromModule" shall appear in the "ModuleDefinition" of another module, except that if it includes an "ObjectIdentifierValue", the "modulereference" may differ in the two cases.

Note 1 – 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 modules (the modules being named in disregard of clause 9.4). The use of alternative distinct names makes these names available for use in the body of the module (see 9.8).

Note 2 – When both a "modulereference" and an "ObjectIdentifierValue" are used in referencing a module, the latter shall be considered definitive.

9.7 Each "SymbolsExported" shall be defined in the module being constructed.

Note – It is recommended that every symbol to which reference from outside the module is appropriate be included in the "SymbolsExported". If there are no such symbols, then the "empty" alternative of "SymbolsExported" (not of "Exports") should be selected.

9.8 Each "SymbolsFromModule" shall be defined in the module denoted by the "ModuleIdentifier" in "SymbolsFromModule". If "Exports" is used in the definition of that module, the "Symbol" shall appear in its "SymbolsExported".

9.9 A "Symbol" in a "SymbolsFromModule" may appear in "ModuleBody" in a "DefinedType" (if it is a "typereference") or "DefinedValue" (if it is a "valuereference"). The meaning associated with "Symbol" also appears in an "AssignmentList" (deprecated), or appears in one or more instances of "SymbolsFromModule", it shall only be used in a "ExternalTypeReference" or "ExternalValueReference" whose "modulereference" is that in "SymbolsFromModule" (see § 9.10). Where it does not so appear, it may be used in a "DefinedType" or "DefinedValue" directly.

9.10 Except as specified in § 9.9, a "typereference" or "valuereference" shall be referenced in a module different from that in which it is defined by using an "Externaltypereference" or "Externalvaluereference", specified by the following productions:

Externaltypereference ::=

modulereference
typereference

Externalvaluereference ::=

modulereference
valuereference

10 Referencing type and value definitions

10.1 The productions

DefinedType ::=

Externaltypereference |
typereference

DefinedValue ::=

Externalvaluereference |
valuereference

specify the sequences which shall be used to reference type and value definitions.

10.2 Except as specified in § 9.10, the "typereference" and "valuereference" alternatives shall not be used unless the reference is within the module in which a type is assigned (see § 11.1 and § 11.2) to the typereference or valuereference.

10.3 The "Externaltypereference" and "Externalvaluereference" shall not be used unless the corresponding "typereference" or "valuereference" has been assigned a type or value respectively (see § 11.1 and § 11.2) within the corresponding "modulereference".

11 Assigning types and values

11.1 A "typereference" shall be assigned a type by the notation specified by the "Typeassignment" production:

Typeassignment ::= typereference

"::="
Type

The "typereference" shall not be one of the names used to reference the character string types defined in section two, and shall not be one of the names used to reference the types defined in section three.

11.2 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 valid notation (see § 12.7) for a value of the type defined by “Type”.

12 Definition of types and values

12.1 A type shall be referenced by one of the sequences “Type”:

Type ::= BuiltinType | DefinedType | Subtype

(see § 10.1) (see § 37)

BuiltinType ::=

BooleanType |
IntegerType |
BitStringType |
OctetStringType |
NullType |
SequenceType |
SequenceOfType |
SetType |
SetOfType |
ChoiceType |
SelectionType |
TaggedType |
AnyType |
ObjectIdentifierType |
CharacterStringType |
UsefulType |
EnumeratedType |
RealType |

Note 1 – A type notation defined in a macro can also be used as a sequence for “Type” (see Annex A).

Note 2 – Additional built-in types may be defined by future versions of this Recommendation.

12.2 The “BuiltinType” notation is specified in the following clauses.

12.3 The “Subtype” notation is specified in section four.

12.4 The type being referenced is the type defined by the “BuiltinType” or “Subtype” assigned to the “DefinedType”.

12.5 In some notations within which a type is referenced, the type may be named. In such cases, this Recommendation specifies the use of the notation “NamedType”:

NamedType ::=

identifier Type |
Type |
SelectionType

The notation “SelectionType” and the corresponding value notation is specified in § 25.

Note – The notation “SelectionType” contains an “identifier” which may form part of the value notation when “SelectionType” is used as a “NamedType” (see § 25.1).

12.6 The “identifier” is not part of the type, and has no effect on the type. The type referenced by a “NamedType” sequence is that referenced by the contained “Type” sequence.

12.7 The value of a type shall be specified by one of the sequences "Value":

Value ::= BuiltinValue | DefinedValue

BuiltinValue ::=

BooleanValue |
IntegerValue |
BitStringValue |
OctetStringValue |
NullValue |
SequenceValue |
SequenceOfValue |
SetValue |
SetOfValue |
ChoiceValue |
SelectionValue |
TaggedValue |
AnyValue |
ObjectIdentifierValue |
CharacterStringValue |
EnumeratedValue |
RealValue |

Note – A value notation defined in a macro may also be used as a sequence for "Value" (see Annex A).

12.8 If the type is defined using one of the notations shown on the left below, then the value shall be specified using the notation shown on the right below:

Type notation	**Value notation**
BooleanType	BooleanValue
IntegerType	IntegerValue
BitStringType	BitStringValue
OctetStringType	OctetStringValue
NullType	NullValue
SequenceType	SequenceValue
SequenceOfType	SequenceOfValue
SetType	SetValue
SetOfType	SetOfValue
ChoiceType	ChoiceValue
TaggedType	TaggedValue
AnyType	AnyValue
ObjectIdentifierType	ObjectIdentifierValue
CharacterStringType	CharacterStringValue
EnumeratedType	EnumeratedValue
RealType	RealValue

Note – Additional value notations may be defined by future versions of this Recommendation.

Where the type is a DefinedType, the value notation shall be the notation for a type used in producing the DefinedType.

12.9 The value notation for a type defined by the "UsefulType" notation is specified in section three.

12.10 The "BuiltinValue" notation is specified in the following clauses.

12.11 The value of a type referenced using the “NamedType” notation shall be defined by the notation “NamedValue”:

NamedValue ::=

identifier Value |
Value

where the “identifier” (if any) is the same as that used in the “NamedType” notation. § 25.2 specifies further restrictions on the “NamedValue” when the “NamedType” was a “SelectionType”.

Note – The “identifier” is part of the notation, it does not form part of the value itself.

12.12 The “identifier” shall be present in the “NamedValue” if and only if it was present in the “NamedType”.

Note – An “identifier” is always present in the case of a “SelectionType”.

13 Notation for the Boolean type

13.1 The Boolean type (see § 3.13) shall be referenced by the notation “BooleanType”:

BooleanType ::= BOOLEAN

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

13.3 The value of a Boolean type (see § 3.14 and § 3.15) shall be defined by the notation “BooleanValue”:

BooleanValue ::= TRUE | FALSE

14 Notation for the integer type

14.1 The integer type (see § 3.16) shall be referenced by the notation “IntegerType”:

IntegerType ::=

INTEGER |
INTEGER{NamedNumberList}

NamedNumberList ::=

NamedNumber |
NamedNumberList,NamedNumber

NamedNumber ::=

identifier(SignedNumber) |
identifier(DefinedValue)

SignedNumber ::= number | -number

14.2 The second alternative of “SignedNumber” shall not be used if the “number” is zero.

14.3 The “NamedNumberList” is not significant in the definition of a type. It is used solely in the value notation specified in § 14.9.

14.4 The “DefinedValue” shall be a reference to a value of type integer, or of a type derived from integer by tagging or subtyping.

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

14.6 Each “identifier” appearing in the “NamedNumberList” shall be different.

14.7 The order of the “NamedNumber” sequences in the “NamedNumberList” is not significant.

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

14.9 The value of an integer type shall be defined by the notation “IntegerValue”:

IntegerValue ::=

SignedNumber |
identifier

14.10 The “identifier” in “IntegerValue” shall be equal to that of an “identifier” in the “IntegerType” sequence with which the value is associated, and shall represent the corresponding number.

Note – When defining an integer value for which an “identifier” has been defined, use of the “identifier” form of “IntegerValue” should be preferred.

15 Notation for the enumerated type

15.1 The enumerated type (see § 3.17) shall be referenced by the notation “EnumeratedType”:

EnumeratedType ::= ENUMERATED {Enumeration}

Enumeration ::=

NamedNumber |
NamedNumber, Enumeration

Note 1 – Each value has an identifier which is associated, in this notation, with a distinct integer. This provides control of the representation of the value in order to facilitate compatible extensions, but the values themselves are not expected to have any integer semantics.

Note 2 – The numeric values inside the “NamedNumber” in the “Enumeration” are not necessarily ordered or contiguous.

15.2 For each “NamedNumber”, the “identifier” and the “SignedNumber” shall be distinct from all other “identifier”s and “SignedNumber”s in the “Enumeration”.

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

15.4 The value of an enumerated type shall be defined by the notation “EnumeratedValue”:

EnumeratedValue ::= identifier

16 Notation for the real type

16.1 The real type (see § 3.18) shall be referenced by the notation “RealType”:

RealType ::= REAL

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

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

where M is called the mantissa, B the base, and E the exponent. M and E may take any integer values, positive or negative, while B can take the values 2 or 10. All combinations of M, B and E are permitted.

Note 1 – This type is capable of carrying an exact representation of any number which can be stored in typical floating point hardware, and of any number with a finite character decimal representation.

Note 2 – The encoding (of this type) which is specified in Recommendation X.209 allows use of base 2, 8 or 16 with a binary representation of real values, and base 10 with a character representation. The choice is a sender's option.

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

16.4 The notation for defining a value of a real type shall be “RealValue”:

RealValue ::= NumericRealValue | SpecialRealValue

NumericRealValue ::=

{Mantissa, Base, Exponent} | 0

Mantissa ::= SignedNumber

Base ::= 2 | 10

Exponent ::= SignedNumber

SpecialRealValue ::= PLUS-INFINITY | MINUS-INFINITY

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

17 Notation for the bitstring type

17.1 The bitstring type (see § 3.19) shall be referenced by the notation "BitStringType":

BitStringType ::=

BIT STRING |
BIT STRING{NamedBitList}

NamedBitList ::=

NamedBit |
NamedBitList,NamedBit

NamedBit ::=

identifier(number) |
identifier(DefinedValue)

17.2 The "NamedBitList" is not significant in the definition of a type. It is used solely in the value notation specified in § 17.8.

17.3 The first bit in a bitstring has the number **zero**. The final bit in a bit string is called the **trailing bit**.

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

17.4 The "DefinedValue" shall be a reference to a non-negative value of type integer or enumerated, or of a type derived from those by tagging or subtyping.

17.5 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.

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

Note – The order of the "NamedBit" sequences in the "NamedBitList" is not significant.

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

17.8 The value of a bitstring type shall be defined by the notation "BitStringValue":

BitStringValue ::=

bstring |
hstring |
{IdentifierList} |
{ }

IdentifierList ::=

identifier |
IdentifierList,identifier

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

17.10 The use of the notation determines, and can indicate by comment, whether or not the presence or absence of trailing zero bits is significant.

Note – Encoding rules enable the transfer of an arbitrary pattern, arbitrary length, string of bits.

17.11 The "{IdentifierList}" and "{ }" notations for "BitStringValue" shall not be used if the presence or absence of trailing zero bits is significant. This notation denotes a bitstring value with ones in the bit positions specified by the numbers corresponding to the "identifier" sequences, and with all other bits zero.

Note – The "{ }" sequence is used to denote a bitstring value which contains no one bits.

17.12 In specifying the encoding rules for a bitstring, the bits shall be referenced by the terms **first bit** and **trailing bit**, as defined above.

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

17.14 When using the "hstring" notation, the most significant bit of each hexadecimal digit corresponds to the earlier (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.

17.15 The "hstring" notation shall not be used unless either:

a) the bitstring value consists of a multiple of four bits; or

b) the presence or absence of trailing zero bits is not significant.

Example:

'A98A'H

and

'1010100110001010'B

are alternative notations for the same bitstring value.

18 Notation for the octetstring type

18.1 The octetstring type (see § 3.20) shall be referenced by the notation "OctetStringType":

OctetStringType ::= OCTET STRING

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

18.3 The value of an octetstring type shall be defined by the notation "OctetStringValue":

OctetStringValue ::=

bstring |
hstring

18.4 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** and **least significant bit**.

18.5 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.

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

19 Notation for the null type

19.1 The null type (see § 3.21) shall be referenced by the notation "NullType":

NullType ::= NULL

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

19.3 The value of the null type shall be referenced by the notation "NullValue":

NullValue ::= NULL

20 Notation for sequence types

20.1 The notation for defining a sequence type (see § 3.22) from other types shall be the "SequenceType":

SequenceType ::=

SEQUENCE{ElementTypeList} |
SEQUENCE{ }

ElementTypeList ::=

ElementType |
ElementTypeList,ElementType

ElementType ::=

NamedType |
NamedType OPTIONAL |
NamedType DEFAULT Value |
COMPONENTS OF Type

20.2 The "Type" in the fourth alternative of the "ElementType" shall be a sequence type. The "COMPONENTS OF Type" notation shall be used to define the inclusion, at this point in the "ElementTypeList", of all the "ElementType" sequences appearing in the referenced type.

Note – This transformation is logically completed prior to the satisfaction of the requirements in the following clauses.

20.3 For each series of one or more consecutive "ElementTypes" marked as OPTIONAL or DEFAULT, the tags of those "ElementTypes" and of any immediately following "ElementType" shall be distinct (see § 26).

20.4 If "OPTIONAL" or "DEFAULT" are present, the corresponding value may be omitted from a value of the new type, and from the information transferred by encoding rules.

Note 1 – The value notation may be ambiguous in this case, unless "identifier" sequences are present in each NamedType.

Note 2 – Encoding rules ensure that the encoding for a sequence value in which a "DEFAULT" or "OPTIONAL" element value is omitted is the same as that for a sequence value of a type in whose type definition the corresponding element was omitted. This feature can be useful in defining subsets.

20.5 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 signification that is valid for the type defined by "Type" in the "NamedType" sequence.

20.6 The "identifier" (if any) in all "NamedType" sequences of the "ElementTypeList" shall be distinct.

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

Note – Sequence–of types have the same tag (see § 21.3).

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

SequenceValue ::=

{ElementValueList} |
{ }

ElementValueList ::=

NamedValue |
ElementValueList,NamedValue

20.9 The "{ }" notation shall only be used if:

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

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

20.10 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.

Note – The use of "NamedType" sequences which do not contain an identifier is not prohibited, but can render the value notation ambiguous if "OPTIONAL" or "DEFAULT" is used.

21 Notation for sequence-of types

21.1 The notation for defining a sequence-of type (see § 3.23) from another type shall be the "SequenceOf-Type".

SequenceOfType ::=

SEQUENCE OF Type |

SEQUENCE

21.2 The notation "SEQUENCE" is synonymous with the notation "SEQUENCE OF ANY" (see § 27).

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

Note – Sequence types have the same tag (see § 20.7).

21.4 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 there are no component values in the sequence-of value.

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

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

22 Notation for set types

22.1 The notation for defining a set type (see § 3.24) from other types shall be the "SetType".

SetType ::=

SET{ElementTypeList} |

SET{ }

"ElementTypeList" is specified in § 20.1

22.2 The "Type" in the fourth alternative of the "ElementType" (see § 20.1) shall be a set type. The "COMPONENTS OF Type" notation shall be used to define the inclusion of all the "ElementType" sequences appearing in the referenced type.

Note – This transformation is logically completed prior to the satisfaction of the requirements in the following clauses.

22.3 The "ElementType" types in a set type shall all have different tags. (See § 26.)

22.4 Sub-clauses § 20.4, § 20.5 and § 20.6 also apply to set types.

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

Note – Set-of types have the same tag (see § 23.3).

22.6 There shall be no more semantic associated with the order of values in a set type.

22.7 The notation for defining the value of a set type shall be "SetValue":

SetValue ::= {ElementValueList} | { }

"ElementValueList" is specified in § 20.8.

22.8 The "SetValue" shall only be "{ }" if:

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

b) the type notation was “SET”{ }”.

22.9 There shall be one “NamedValue” for each “NamedType” in the “SetType” which is not marked “OPTIONAL” or “DEFAULT”.

Note 1 – These “NamedValues” may appear in any order.

Note 2 – The use of “NamedType” sequences which do not contain an identifier is not prohibited, but can render the value notation ambiguous.

23 Notation for set-of types

23.1 The notation for defining a set-of type (see § 3.25) from another type shall be the “Set OfType”:

SetOfType ::= SET OF Type |

SET

23.2 The notation “SET” is synonymous with the notation “SET OF ANY” (see § 27).

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

Note – Set types have the same tag (see § 22.5).

23.4 The notation for defining a value of a set-of type shall be the “SetOfValue”:

SetOfValue ::= {ValueList} | { }

“ValueList” is specified in § 21.4

The “{ }” notation is used when there are no component values in the set-of values.

23.5 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.

24 Notation for choice types

24.1 The notation for defining a choice type (see § 3.27) from other types shall be the “ChoiceType”:

ChoiceType ::= CHOICE{AlternativeTypeList}

AlternativeTypeList ::=

NamedType |
AlternativeTypeList,NamedType

Note 1 – The encoding rules encode the chosen alternative in a way which is indistinguishable from a “Type” consisting only of the “Type” contained in that alternative.

Note 2 – Specifying a “ChoiceType” with a single “NamedType” in the “AlternativeTypeList” cannot be distinguished in any encoding of a value from direct use of the “Type” in the “NamedType”.

24.2 The types defined in the “AlternativeTypeList” shall all have distinct tags. (See § 26.)

24.3 The tag of the choice type shall be considered to be variable. When a value is selected, the tag becomes equal to the tag of the “Type” in the “NamedType” in the “AlternativeTypeList” from which the value is taken.

24.4 Where this type is used in a place where this Recommendation requires the use of types with distinct tags (see § 20.3, § 22.3 and § 24.2), the tags of all types defined in the “AlternativeTypeList” shall differ from those of the other types. (See § 26.) The following examples illustrate this requirement. Examples 1 and 2 are correct uses of the notation. Example 3 is incorrect, as the tags for types d and f, and e and g are identical.

Example 1:

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

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

Example 2:

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

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

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

Example 3:

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

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

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

24.5 The "identifier" (if any) in all "NamedType" sequences of the "AlternativeTypeList" shall be distinct.

24.6 Where this type is used in a place where this Recommendation requires the use of "NamedTypes" with distinct "identifiers", the "identifiers" (if any) of all "NamedTypes" in the "AlternativeTypeList" shall differ from those (if any) of the other "NamedTypes".

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

ChoiceValue ::= NamedValue

24.8 If the "NamedValue" contains an "identifier", it shall be a notation for a value of that type in the "AlternativeTypeList" that is named by the same "identifier". If the "NamedValue" does not contain an "identifier", it shall be a notation for a value of one of those types in the "AlternativeTypeList" that are not named by an "identifier".

Note – Failure to use an "identifier" in the "NamedType" can make the value notation ambiguous.

25 Notation for selection types

25.1 A "NamedType" appearing in the "AlternativeTypeList" of a "ChoiceType" can be referenced by the notation "SelectionType":

SelectionType ::= identifier<Type

where "Type" is a notation referencing the "ChoiceType", and "identifier" is the "identifier" in the "NamedType".

Note – "SelectionType" can be used either as a "NamedType", in which case the "identifier" is used in the value notation, or as a "Type" within a "NamedType", in which case its "identifier" is not used.

25.2 The notation for a value of a selection type shall be "SelectionValue":

SelectionValue ::= NamedValue

where the "NamedValue" contains the identifier that appears in the corresponding "SelectionType" if the "SelectionType" is used as a "NamedType", but not otherwise.

26 Notation for tagged types

A tagged type (see § 3.26) is a new type which is isomorphic with an old type, but which has a different tag. In all encoding schemes, a value of the new type can be distinguished from a value of the old type. The tagged type is mainly of use where this Recommendation requires the use of types with distinct tags. (See § 20.3, § 22.3, § 24.2, § 24.4, and § 27.6.)

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

26.1 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

26.2 The "DefinedValue" shall be a reference to a non-negative value of type integer, or of a type derived from type integer by tagging.

26.3 The new type is isomorphic with the old type, but has a tag with "Class" class and number "ClassNumber", unless the "Class" is "empty", when the tag is context-specific class, number "ClassNumber".

26.4 The "Class" shall not be "UNIVERSAL" except for types defined in this Recommendation.

Note – Use of universal class tags is agreed from time to time by ISO and CCITT.

26.5 If the "Class" is "APPLICATION", the same "Tag" shall not be used again in the same module.

26.6 If the "Class" is "PRIVATE" the "Tag" is available for use on an enterprise-specific basis.

26.7 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 "EXPLICIT TAGS";

c) the "Tag Type" alternative is used and the value of "TagDefault" for the module is "IMPLICIT TAGS", but the type defined by "Type" is a choice type or on any type.

The tagging construction specifies implicit tagging otherwise.

26.8 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 § 20.3, § 22.3, and § 24.2.

26.9 Implicit tagging indicates, for those encoding rules which provide the option, that explicit identification of the 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 encoding using IMPLICIT is given in Recommendation X.209.

26.10 The "IMPLICIT" alternative shall not be used if the type defined by "Type" is a choice type or an any type.

26.11 The notation for a value of a "TaggedType" shall be "TaggedValue":

TaggedValue ::= Value

where “value” is the notation for a value of the “Type” in the “TaggedType”.

Note – The “Tag” does not appear in this notation.

27 Notation for the any type

27.1 The notation for any type (see § 3.29) is “AnyType”:

AnyType ::=

ANY |
ANY DEFINED BY identifier

Note – The use of “ANY” in an ISO Standard or CCITT Recommendation produces an incomplete specification unless it is supplemented by additional specification. The “ANY DEFINED BY” construct provides the means of specifying in an instance of communication the type which fills the ANY, and a pointer to its semantics. If the following rules for its use are followed, it can provide a complete specification. Use of ANY without the DEFINED BY construct is deprecated.

27.2 The “DEFINED BY” alternative shall be used only when the any type, or a type derived from it by tagging, is one of the component types of a sequence type or set type (the containing type).

27.3 The “identifier” in the “DEFINED BY” alternative shall also appear in a “NamedType” that specifies another, non-optional, component of the containing type. The “NamedType” shall be an integer type or an enumerated type or an object identifier type or a type derived from those by tagging or subtyping.

27.4 When the “NamedType” is an integer or enumerated type, or of a type derived from those types by tagging ot subtyping the document employing the “DEFINED BY” notation shall contain, or explicitly reference, a single list which specifies the ASN.1 type to be carried by the ANY for each permitted value of the integer type. There shall be precisely any one such list in all instances of communication of the containing type.

27.5 When the “NamedType” is an object identifier type, or a type derived from object identifier by tagging, there is a need for registers which, for each allocated object identifier value, associate a single ASN.1 type (which may be a CHOICE type) which is to be carried by the ANY.

Note 1 – There may be an arbitrary number of registers associating an object identifier value with an ASN.1 type for this purpose.

Note 2 – Registration of values for open interconnection is expected to occur within ISO Standards and CCITT Recommendations using the notation. Where a separate International Registration Authority is intended for any instance of “ANY DEFINED BY”, this should be identified in the document using the notation.

Note 3 – The main difference between the integer and object identifier definers is that the use of integer references a single list, contained in the using standard or Recommendation, whilst the use of object identifier allows an open-ended set of types determined by any authority able to allocate object identifiers.

27.6 This type has an indeterminate tag, and shall not be used where this Recommendation requires distinct tags. (See § 20.3, § 22.3, § 24.2 and § 24.4.)

27.7 The notation for the value of an any type shall be defined using ASN.1 and is “AnyValue”:

AnyValue ::= Type Value

where “Type” is the notation for the chosen type, and “Value” is the notation for a value of this type.

28 Notation for the object identifier type

28.1 The object identifier type (see § 3.34) shall be referenced by the notation “ObjectIdentifierType”:

ObjectIdentifierType ::=

OBJECT IDENTIFIER

28.2 This type has a tag which is universal class, number 6.

28.3 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)

28.4 The “DefinedValue” in “NumberForm” shall be a reference to a value of type integer or enumerated, or of a type derived from those by tagging or subtyping.

28.5 The “DefinedValue” in “ObjectIdentifierValue” shall be a reference to a value of type object identifier, or of a type derived from object identifier by tagging.

28.6 The “NameForm” shall be used only for those object identifier components whose numeric value and identifier are specified in annexes B to D, and shall be one of the identifiers specified in annexes B to D.

28.7 The “number” in the “NumberForm” shall be the numeric value assigned to the object identifier component.

28.8 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 the annexes to this Recommendation.

28.9 The semantics of an object identifier value are defined by reference to an **object identifier tree**. An object identifier tree is a tree whose root corresponds to this Recommendation and whose vertices correspond to administrative authorities responsible for allocating arcs from that vertex. Each arc of the tree is labelled by an object identifier component which is a numeric value. Each information object to be identified is allocated precisely one vertex (normally a leaf), and no other information object (of the same or a different type) is allocated to that same vertex. Thus an information object is uniquely and unambiguously identified by the sequence of numeric values (object identifier components) labelling the arcs in a path from the root to the vertex allocated to the information object.

Note – Object identifier values contain at least two object identifier components, as specified in annexes B to D.

28.10 An object identifier value is semantically an ordered list of object identifier component values. Starting with the root of the object identifier tree, each object identifier component value identifies an arc in the object identifier tree. The last object identifier component value identifies an arc leading to a vertex which an information object has been assigned. It is this information object which is identified by the object identifier value. 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.

Note – In general, an information object is a class of information (for example, a file format), rather than an instance of such a class (for example, an individual file). It is thus the class of information (defined by some referencable specification), rather than the piece of information itself, that is assigned a place in the tree.

28.11 Where the “ObjectIdentifierValue” includes a “DefinedValue”, the list of object identifier components to which it refers is prefixed to the components explicitly present in the value.

Examples: With identifiers is assigned as specified in Annex B, 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 also equivalent to those above:

{ftam pci (1)}

Note – It is recommended that, whenever a CCITT Recommendation, ISO standard or other document assigns values of type OBJECT IDENTIFIER to information objects there should be an appendix or annex which summarizes the assignments made therein. It is also recommended that an authority assigning values of type OBJECT IDENTIFIER to an information object should also assign values of type ObjectDescriptor to that information object.

29 Notation for character string types

29.1 The notation for referencing a character string type (see § 3.12 and section two) shall be:

CharacterStringType ::= typereference

where “typereference” is one of the character string type names listed in section two.

29.2 The tag of each character string type is specified in section two.

29.3 The notation for a character string value shall be:

CharacterStringValue ::= cstring

The definition of the character string type determines the characters appearing in the “cstring”.

30 Notation for types defined in section three

30.1 The notation for referencing a type defined in section three of this Recommendation shall be:

UsefulType ::= typereference

where “typereference” is one of those defined in section three using the ASN.1 notation.

30.2 The tag of each “UsefulType” is specified in section three.

30.3 The notation for a value of a “UsefulType” is specified in section three.

SECTION 2 – CHARACTER STRING TYPES

31 Definition of character string types

This clause defines types whose distinguished values are sequences of zero, one or more characters from some character set.

31.1 The type is defined by specifying:

a) the tag assigned to the type; and

b) a name by which the type definition can be referenced; and

c) the characters in the character set used in defining the type, either 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 to be used with Escape Sequences.

The name in b) above may be used as a “typereference” in the ASN.1 notation (see § 29).

31.2 Table 6/X.208 lists the name by which each of these type definitions can be referenced, the number of the universal class tag assigned to the type, the defining registration numbers or following table, 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.

31.3 Table 4/X.208 lists the characters which can appear in the NumericString type.

TABLE 4/X.208

NumericString

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

Name	Graphics
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	?

TABLE 6/X.208

List of character string types

Name for referencing the type	Universal class number	Defining registration numbers (see ISO 2375) or table number	Notes
NumericString	18	Table 4	1
PrintableString	19	Table 5	1
TeletexString (T61String)	20	87, 102, 103, 106, 107 + SPACE + DELETE	2
VideotexString	21	1, 72, 73, 102, 108, 128, 129 + SPACE + DELETE	3
VisibleString (ISO646String)	26	2 + SPACE	
IA5String	22	1, 2 + SPACE + DELETE	
GraphicString	25	All G sets + SPACE	
GeneralString	27	All G and all C sets + SPACE + DELETE	

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 Recommendation T.61 for rules concerning their use.

Note 3 – The entries corresponding to these registration numbers provide the functionality of CCITT Recommendations T.100 and T.101.

31.4 Table 5/X.208 lists the characters which can appear in the PrintableString type.

31.5 The notation for these types shall be “cstring”.

Note – This notation can only be used on a medium capable of displaying the characters which are present in the value. The notation for the value in other cases is not defined.

31.6 In all cases, the range of permitted characters may be restricted by a comment, but shall not be extended.

SECTION 3 – USEFUL DEFINITIONS

This section contains definitions which are expected to be useful in a number of applications.

Note – It is expected that this section will be added to, to encompass other common datatypes such as diagnostics, authentication information, accounting information, security parameters and so on.

The value notation and semantic definition for types defined in this section are derived from a definition of the type using the ASN.1 notation. This type definition can be referenced by standards or Recommendations defining encoding rules in order to specify encodings for these types.

32 Generalized time

32.1 This type shall be referenced by the name:

GeneralizedTime

32.2 The type consists of values representing:

a) a calendar date, as defined in ISO 2014 (Writing of calendar dates in all-numeric form); and

b) a time of day, to any of the precisions defined in clause 2 of ISO 3307 (Representations of time of day); and

c) the local differential factor as defined in ISO 4031 (Representation of local time differentials).

32.3 The type can be 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 2014, 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 3307, without separators other than decimal comma or decimal period (as provided for in clauses 2.3, 2.4 and 2.5 of ISO 3307), and with no terminating Z (as provided for in clause 3 of ISO 3307); 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 4031, without separators.

In case a), the time shall represent the local time. In case b), the time shall represent UTC time. In case c), the part of the string formed in case a) represents the local time (t_1), and the time differential (t_2) enables UTC time to be determined as follows:

UTC 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 UTC time as above.
Case c)	19851106210627.3-0500 Local time as in example a), with local time 5 hours retarded in relation to UTC time.

32.4 The tag shall be as defined in § 32.3.

32.5 The value notation shall be the value notation for the "VisibleString" defined in § 32.3.

33 Universal time

33.1 This type shall be referenced by the name:

UTCTime

33.2 The type consists of values representing:

a) a calendar date; and

b) a time to a precision of one minute or one second; and

c) (optionally) a local time differential from coordinated universal time.

33.3 The type can be 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 UTC 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 UTC time to be determined as follows:

UTC Time is $t_1 - t_2$

Example: If local time is 7 AM on 2 January and coordinated universal time is 12 noon on 2 January, the value is either of:

UTCTime "8201021200Z"
UTCTime "8201020700-0500"

33.4 The tag shall be as defined in § 33.3.

33.5 The value notation shall be the value notation for the "VisibleString" defined in § 33.3.

34 The external type

34.1 The notation for an external type (see § 3.30) is "ExternalType":

ExternalType ::= EXTERNAL

34.2 The type consists of values representing:

a) an encoding of a single data value that may, but need not, be the value of a single ASN.1 datatype; and

b) identification information which determines the semantics and encoding rules; and

c) (optionally) an object descriptor which describes the object.

The optional object descriptor shall not be present unless explicitly permitted by comment associated with the use of the EXTERNAL notation.

34.3 Type EXTERNAL permits the inclusion of any data value from an identified set of data values.

Note 1 – The specification of this set of data values, their semantics, the assignment of an object identifier and (optionally) an object descriptor, and the dissemination of this information to all communicating parties is called **registering an abstract syntax**. This operation can be performed by any authority entitled to allocate an OBJECT IDENTIFIER value, as specified in Annexes B to D.

Note 2 – A set of data values registered as an abstract syntax (with associated encoding rules) is not well-formed unless the encoding of each data value is self-identifying within the set of data value encodings. When ASN.1 is used to define an abstract syntax, tagging is used to provide self-identification. Where an abstract syntax is not well-formed, use of the communications channel is either context-sensitive or leads to ambiguity.

34.4 The EXTERNAL type can be defined, using ASN.1, as follows:

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE
{direct-reference       OBJECT IDENTIFIER OPTIONAL,
indirect-reference      INTEGER OPTIONAL,
data-value-descriptor   ObjectDescriptor OPTIONAL,
```

encoding CHOICE
{single-ASN1-type [0] ANY,
octet-aligned [1] IMPLICIT OCTET STRING,
arbitrary [2] IMPLICIT BIT STRING}}

34.5 When presentation layer negotiation of encoding rules is not in use (prior agreement of transfer syntax) for the value of this EXTERNAL, the "direct-reference OBJECT IDENTIFIER" shall be present. In this case the identifier of the set of data values is an object identifier which directly references an abstract syntax and fills the "direct-reference OBJECT IDENTIFIER" field of the "EXTERNAL". In this case, the abstract syntax registration also defines the encoding rules (transfer syntax) for the data value and the "indirect-reference INTEGER" shall not be included.

34.6 When presentation layer negotiation is in use for the value of this EXTERNAL, the "indirect-reference INTEGER" shall be present. In this case the identifier of the set of data values is an integer which references an instance of use of an abstract syntax. The integer is called a **presentation context identifier** and fills the "indirect-reference INTEGER" field of the "EXTERNAL". If presentation layer negotiation has been completed, the presentation context identifier also identifies the encoding rules (transfer syntax) for the data value and the "direct-reference OBJECT IDENTIFIER" shall not be included. If presentation layer negotiation is not complete, an object identifier value is also needed which identifies the encoding rules (transfer syntax) used for the encoding. Where presentation layer negotiation is in use, and where the "direct-reference OBJECT IDENTIFIER" element is allowed or required to carry such a value, this shall be identified by comment associated with the use of the "EXTERNAL" notation, otherwise the field shall be absent.

Note 1 – The effect of § 34.5 and § 34.6 is to make the presence of at least one of the "direct-reference" and the "indirect-reference" mandatory.

Note 2 – Both references are present when presentation layer negotiation is in use but incomplete.

34.7 If the data value is the value of a single ASN.1 datatype, and if the encoding rules for this data value are the same as those for the complete "EXTERNAL" datatype, then the sending implementation shall use any of the "Encoding" choices:

single-ASN1-type
octet-aligned
arbitrary

as an implementation option.

34.8 If the encoding of the data value, using the agreed or negotiated encoding, is an integral number of octets, then the sending implementation shall use any of the "Encoding" choices:

octet-aligned
arbitrary

as an implementation option.

Note – A data value which is a series of ASN.1 types, and for which the transfer syntax specifies simple concatenation of the octet strings produced by applying the ASN.1 Basic Encoding Rules to each ASN.1 type, falls into this category, not that of § 34.7.

34.9 If the encoding of the data value, using the agreed or negotiated encoding, is not an integral number of octets, the "Encoding" choice shall be:

arbitrary

34.10 If the "Encoding" choice is chosen as "single-ASN1-type", then the ASN.1 type shall replace the "ANY", with a value equal to the data value to be encoded.

Note – The range of values which might occur in the "ANY" is determined by the registration of the object identifier value associated with the "direct-reference", and/or the integer value associated with the "indirect-reference".

34.11 If the "Encoding" choice is chosen as "octet-aligned", then the data value shall be encoded according to the agreed or negotiated transfer syntax, and the result shall form the value of the bitstring.

34.12 If the "Encoding" choice is chosen as "arbitrary", then the data value shall be encoded according to the agreed or negotiated transfer syntax, and the result shall form the value of the bitstring.

34.13 The tag shall be as defined in § 34.4.

34.14 The value notation shall be the value of the type defined in § 34.4.

35 The object descriptor type

35.1 This type shall be referenced by the name:

ObjectDescriptor

35.2 The type consists of human-readable text which serves to describe an information object. The text is not an unambiguous identification of the information object, but identical text for different information objects is intended to be uncommon.

Note – It is recommended that an authority assigning values of type "OBJECT IDENTIFIER" to an information object should also assign values of type "ObjectDescriptor" to that information object.

35.3 The type can be defined, using the ASN.1 notation, as follows:

ObjectDescriptor ::=

[UNIVERSAL 7] IMPLICIT
GraphicString

The "GraphicString" contains the text describing the information object.

35.4 The tag shall be as defined in § 35.3.

35.5 The value notation shall be the value notation for the "GraphicString" defined in § 37.3.

SECTION 4 – SUBTYPES

36 Subtype notation

36.1 A subtype is defined by the notation for a parent type followed by an appropriate subtype specification. The subtype specification notation is made up of subtype value sets. The values in the subtype are determined as specified in § 36.7 by taking the union of all the subtype value sets.

36.2 The subtype notation shall not be used so as to produce a subtype with no values.

36.3 The notation for a subtype shall be "Subtype":

Subtype ::=

ParentType SubtypeSpec |
SET SizeConstraint OF Type |
SEQUENCE SizeConstraint OF Type

ParentType ::= Type

36.4 When the "SubtypeSpec" notation follows the "SelectionType" notation, the parent type is the "SelectionType", not the "Type" in the "SelectionType" notation.

36.5 When the "SubtypeSpec" notation follows a set-of or sequence-of type notation, it applies to the "Type" in the set-of or sequence-of notation, not to the set-of or sequence-of type.

Note – The special notation "SET SizeConstraint OF" and "SEQUENCE Size Constraint OF" is used to provide an alternative mechanism (which is more readable than the general case notation) for simple cases. More complex cases require the general mechanism.

36.6 The subtype specification notation shall be "SubtypeSpec":

SubtypeSpec ::=

(SubtypeValueSet SubtypeValueSetList)

SubtypeList ::=

" | "
SubtypeValueSet
SubtypeValueSetList |
empty

36.7 Each “SubtypeValueSet” specifies a number (possibly zero) of values of the parent type, which are then included in the subtype. A value of the parent type is a value of the subtype if and only if it is included by one or more of the subtype value sets. The subtype is thus formed from the set union of the values included by the subtype value sets.

36.8 A number of different forms of notation for “SubtypeValueSet” are provided. They are identified below, and their syntax and semantics is defined in § 37. As specified in § 37, and summarized in Table 7/X.208, some notations can only be applied to particular parent types.

TABLE 7/X.208

Applicability of subtype value sets

Type (or derived from such a type by tagging)	Single Value	Cont'd Subtype	Value Range	Size Range	Alphabet Limitation	Inner Subtyping
Boolean	/	/	x	x	x	x
Integer	/	/	/	x	x	x
Enumerated	/	/	x	x	x	x
Real	/	/	/	x	x	x
Object Identifier	/	/	x	x	x	x
Bit String	/	/	x	/	x	x
Octet String	/	/	x	/	x	x
Character String Types	/	/	x	/	/	x
Sequence	/	/	x	x	x	/
Sequence-of	/	/	x	/	x	/
set	/	/	x	x	x	/
set-of	/	/	x	/	x	/
Any	/	/	x	x	x	x
Choice	/	/	x	x	x	/

Subtype ValueSet ::=

SingleValue |
ContainedSubtype |
ValueRange |
PermittedAlphabet | SizeConstraint | InnerTypeConstraints

37 Subtype Value Sets

37.1 *Single Value*

37.1.1 The “SingleValue” notation shall be:

SingleValue ::= Value

where “Value” is the value notation for the parent type.

37.1.2 A “SingleValue” set is the single value of the parent type specified by “Value”. This notation can be applied to all parent types.

37.2 *Contained Subtype*

37.2.1 The “ContainedSubtype” notation shall be:

ContainedSubtype ::= INCLUDES Type

37.2.2 A “ContainedSubType” value consists of all the values of the “Type”, which is itself required to be a subtype of the parent type. This notation can be applied to all parent types.

37.3 *Value Range*

37.3.1 The “ValueRange” notation shall be:

ValueRange ::= LowerEndpoint .. Upper Endpoint

37.3.2 A “ValueRange” value set consists of all the values in a range which are designated by specifying the numerical values of the endpoints of the range. This notation can only be applied to integer types, real types and types derived from those types by tagging or subtyping.

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

37.3.3 Each endpoint of the range is either closed (in which case that endpoint is included in the value set) or open (in which case the endpoint is not included). When open, the specification of the endpoint includes a less-than symbol (“<”):

LowerEndpoint ::= LowerEndValue | LowerEndValue <

UpperEndpoint ::= UpperEndValue | < UpperEndValue

37.3.4 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

37.4 *Size Constraint*

37.4.1 The “SizeConstraint” notation shall be:

SizeConstraint ::= SIZE SubtypeSpec

37.4.2 A “SizeConstraint” can only be applied to bitstring types, octetstring types, character string types, set-of types or sequence-of types, or types formed from any of those types by tagging or subtyping.

37.4.3 The “SubtypeSpec” specifies the permitted integer values for the length of the numbers of the value set, and takes the form of any subtype specifications which can be applied to the following parent type:

INTEGER (0. .MAX)

37.4.4 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

37.5 *Permitted Alphabet*

37.5.1 The “PermittedAlphabet” notation shall be:

PermittedAlphabet ::= FROM SubtypeSpec

37.5.2 A “PermittedAlphabet” value consists of all values which can be constructed using a sub-alphabet of the parent string. This notation can only be applied to character string types, or to types formed from them by tagging or subtyping.

37.5.3 The “SubtypeSpec” specifies the characters which may appear in the character string, and is any subtype specification which can be applied to the subtype obtained by applying the subtype specification “SIZE(1)” to the parent type.

37.6 *Inner Subtyping*

37.6.1 The “InnerTypeConstraints” notation shall be:

InnerTypeConstraints ::=

WITH COMPONENT SingleTypeConstraint |
WITH COMPONENTS MultipleTypeConstraints

37.6.2 An “InnerTypeConstraints” includes in the value set 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 included in the subtype unless it satisfies all of the constraints expressed or implied (see § 37.6.6). This notation can be applied to the set-of, sequence-of, set, sequence and choice types, or types formed from them by tagging or subtyping.

37.6.3 For the types which are defined in terms of a single other (inner) type (set-of, sequence-of and types derived from them by tagging), a constraint taking the form of a subtype value specification is provided. The notation for this is “SingleTypeConstraint”:

SingleTypeConstraint ::= SubtypeSpec

The “SubtypeSpec” defines a subtype of the single other (inner) type. A value of the parent type is a member of the subtype value set if and only if each inner value belongs to the subtype obtained by applying the “SubtypeSpec” to the inner type.

37.6.4 For the types which are defined in terms of multiple other (inner) types (choice, set, sequence, and types derived from them by tagging or subtyping), 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 Constraint | Constraint

37.6.5 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, if it has one, or by its position, in the case of sequence types.

Note – Where the inner type has no identifier, the notation can be ambiguous.

37.6.6 The "MultipleTypeConstraints" comprises either a "FullSpecification" or a "PartialSpecification". Where "FullSpecification" is used, there is an implied presence constraint of "ABSENT" on all inner types not explicitly listed (see § 37.6.8), and each inner type which is not marked "OPTIONAL" or "DEFAULT" in the parent type shall be explicitly listed. Where "PartialSpecification" is employed, there are no implied constraints, and any inner type can be omitted from the list.

37.6.7 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 "Constraint":

Constraint ::= ValueConstraint PresenceConstraint

37.6.8 A constraint on the value of an inner type is expressed by the notation "ValueConstraint":

ValueConstraint ::= SubtypeSpec | 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 "SubtypeSpec" applied to the inner type.

37.6.9 A constraint on the presence of an inner type shall be expressed by the notation "PresenceConstraint":

PresenceConstraint ::= PRESENT | ABSENT | empty | OPTIONAL

The meaning of these alternatives, and the situations in which they are permitted, are defined in § 37.6.9.1 to § 37.6.9.3.

37.6.9.1 If the parent type is a sequence or set, an element type marked "OPTIONAL" may be constrained to be "PRESENT" (in which case the constraint is satisfied if and only if the corresponding element value is present) or to be "ABSENT" (in which case the constraint is satisfied if and only if the corresponding element value is absent or to be "OPTIONAL" (in which case the constraint is placed upon the presence of the corresponding element value).

37.6.9.2 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.

37.6.9.3 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";

b) in a "PartialSpecification", no constraint is imposed.

ANNEX A

(to Recommendation X.208)

The macro notation

A.1 *Introduction*

A mechanism is provided within ASN.1 for the user of ASN.1 to define a new notation with which he can then construct and reference ASN.1 types or specify values of types. The new notation is defined using the notation "MacroDefinition". A "MacroDefinition" simultaneously specifies a new notation for constructing and referencing a type and also a new notation for specifying a value. (See § I.3 for an illustration of the use of the macro notation.)

With a "MacroDefinition" the ASN.1 user specifies the new notation by means of a set of productions in a manner similar to that of this Recommendation. The writer of the macro definition;

a) specifies the complete syntax to be used for defining all types supported by the macro (this syntax specification is invoked for syntax analysis by any occurrence of the macro name in the ASN.1 type notation); and

b) specifies the complete syntax to be used for a value of one of these types (this syntax specification is invoked for syntax analysis whenever a value of the macro type is expected); and

c) specifies, as the value of a standard ASN.1 type (of arbitrary complexity), the resulting type and value for all instances of the macro value notation.

An instance of the syntax defined by the macro definition can contain instances of types or values (using the standard ASN.1 notation). These types or values (appearing in the use of macro notation) can be associated, for the duration of the syntax analysis, with a **local type reference** or a **local value reference** by appropriate statements in the macro definition. It is also possible to embed, within the macro definition, standard ASN.1 type assignments. These assignments become active when the associated syntactic category is matched against an item or items in the instance of the new notation being analysed. Their lifetime is limited to that of the analysis.

When analysing a value in the new notation, assignments made during analysis of the corresponding type notation are available. Such analysis is considered to logically precede analysis of every instance of the value notation.

The resulting type and value of an instance of use of the new value notation is determined by the value (and the type of the value) finally assigned to the distinguished local value reference identified by the keyword item VALUE, according to the processing of the macrodefinition for the new type notation followed by that for the new value notation.

Each “MacroDefinition” defines a notation (a syntax) for type definition and a notation (a syntax) for value definition. The ASN.1 type which is defined by an instance of the new type notation may, but need not, depend on the instance of the value notation with which the type is associated. To this extent, the use of the new type notation is similar to a CHOICE - the tag is indeterminate. Thus the new notation cannot in this case be used in places where a known tag is required, nor can it be implicitly tagged.

A.2 *Extensions to the ASN.1 character set and items*

The characters | and > are used in the macro notation.

The items specified in the following subclauses are also used.

A.2.1 *Macroreference*

Name of item – macroreference

A «macroreference» shall consist of the sequence of characters specified for a “typereference” in § 8.2, except that all characters shall be in upper-case. Within a single module, the same sequence of characters shall not be used for both a typereference and a macroreference.

A.2.2 *Productionreference*

Name of item – productionreference

A «productionreference» shall consist of the sequence of characters specified for a “typereference” in § 8.2.

A.2.3 *Localtypereference*

Name of item – localtypereference

A “localtypereference” shall consist of the sequence of characters specified for a “typereference” in § 8.2. A “localtypereference” is used as an identifier for types which are recognized during syntax analysis of an instance of the new type or value notation.

A.2.4 *Localvaluereference*

Name of item – localvaluereference

A “localvaluereference” shall consist of the sequence of characters specified for a “valuereference” in § 8.2. A “localvaluereference” is used as an identifier for values which are recognized during syntax analysis of an instance of the new type or value notation.

A.2.5 *Alternation item*

Name of item – “ | ”

This item shall consist of the single character |.

A.2.6 *Definition terminator item*

Name of item – >

This item shall consist of the single character >.

Note – The item < for the start of definitions is defined in § 8.13.

A.2.7 *Syntactic terminal item*

Name of item – astring

An "astring" shall consist of an arbitrary number (possibly zero) of characters from the ASN.1 character set (see § 7), surrounded by ". The character " shall be represented in an "astring" by a pair of ".

Note – Use of "astring" in the macronotation specifies the occurrence, at the corresponding point in the syntax being analysed, of the characters enclosed in quotation marks (").

TABLE 8/X.208

Sequence specified by items

Item name	Defining clause
"string"	any sequence of characters
"identifier"	8.3 – Identifiers
"number"	8.8 – Numbers
"empty"	8.7 – Empty

A.2.8 *Syntactic category keywork items*

Names of items: "string"
"identifier"
"number"
"empty"

Items with the above names shall consist (in the macronotation) of the sequences of characters in the name, excluding the quotation symbols ("). These items are used in the macro notation to specify the occurrence, in an instance of the new notation, of certain sequences of characters. The sequences in the new notation specified by each item are given in Table 8/X.208 by reference to a clause in this Recommendation which defines the sequence of characters appearing in the new notation.

Note – The macro notation does not support the distinction between identifiers and references based on the case of the initial letter. This is for historical reasons.

A.2.9 *Additional keyword items*

Names of items: MACRO
TYPE
NOTATION
VALUE
value
type

Items with the above names shall consist of the sequence of characters in the name.

The items specified in clauses § A.2.2 to § A.2.4 inclusive shall not be one of the § A.2.9 sequences, except when used as specified below.

The keyword “MACRO” shall be used to introduce a macro definition. The keyword “TYPE NOTATION” shall be used as the name of the production which defines the new type notation. The keyword “VALUE NOTATION” shall be used as the name of the production which defines the new value notation. The keyword “VALUE” shall be used as the “localvaluereference” to which the resulting value is assigned. The keyword “value” shall be used to specify that each instance of the new notation contains at this point, using standard ASN.1 notation, some value of a type (specified in the macro definition). The keyword “type” shall be used to specify that each instance of the new notation contains at this point, using standard ASN.1 notation, some “Type”.

A.3 *Macro definition notation*

A.3.1 A macro shall be defined using the notation “MacroDefinition”:

```
MacroDefinition ::=
        macroreference
        MACRO
        “::=”
        MacroSubstance

MacroSubstance ::=
        BEGIN MacroBody END |
        macroreference |
        Externalmacroreference

MacroBody ::=
        TypeProduction
        ValueProduction
        SupportingProductions

TypeProduction ::=
        TYPE NOTATION
        “::=”
        MacroAlternativeList

ValueProduction ::=
        VALUE NOTATION
        “::=”
        MacroAlternativeList

SupportingProduction ::=
        ProductionList |
        empty

ProductionList ::=
        Production |
        ProductionList Production

Production ::=
        productionreference
        “::=”
        MacroAlternativeList

Externalmacroreference ::=
        modulereference . macroreference
```

Note – It is intended that one macro definition be permitted to reference (i.e., use) other macros. Ensuring that the notation permits this is for further study.

A.3.2 If the “macroreference” alternative of “MacroSubstance” is chosen, then the module containing the macro definition shall either:

a) contain another macro definition defining that “macroreference”; or

b) contain the “macroreference” in its “SymbolsImported”.

A.3.3 If the “Externalmacroreference” alternative of “MacroSubstance” is chosen, then the module denoted by “modulereference” shall contain a macro definition defining the “macroreference”. The associated definition is then also associated with the “macroreference” being defined.

A.3.4 The chain of definitions which can arise from repeated applications of the rules of § A.3.2 to § A.3.3 shall terminate with a “MacroDefinition” which uses the “BEGIN MacroBody END” alternative and it is that “MacroBody” which defines the type and value notation for the macro being defined.

A.3.5 Each “productionreference” which occurs in a “SymbolDefn” (see § A.3.9) shall occur exactly once as the first item in a “Production”.

A.3.6 Each instance of the new type notation shall commence with the sequence of characters in the “macroreference”, followed by one of the sequences of characters referenced by “TYPE NOTATION” after applying the productions specified in the macro definition.

A.3.7 Each instance of the new value notation shall consist of one of the sequences of characters referenced by “VALUE NOTATION” after applying the productions specified in the macro definition.

A.3.8 The “MacroAlternativeList” in a production specifies the possible sets of character sequences referenced by the production. It is specified by:

MacroAlternative List ::=
    MacroAlternative |
    MacroAlternativeList “ | ” MacroAlternative

The set of character sequences referenced by the “MacroAlternativeList” consists of all the character sequences which are referenced by any of the “MacroAlternative” productions in the “MacroAlternativeList”.

A.3.9 The notation for a “MacroAlternative” shall be:

MacroAlternative ::= SymbolList

SymbolList ::=
    SymbolElement |
    SymbolList SymbolElement

SymbolElement ::=
    SymbolDefn |
    EmbeddedDefinitions

SymbolDefn ::=
    astring |
    productionreference |
    “string” |
    “identifier” |
    “number” |
    “empty” |
    type |
    type(localtypereference) |
    value(MacroType) |
    value(localvaluereference MacroType) |
    value(VALUE MacroType)

MacroType ::= localtypereference |
    Type

Note – When in a macro, any “MacroType” defined in that macro can appear at any point in which ASN.1 specifies a “Type”.

A “MacroAlternative” references all characters strings which are formed by taking any of the character strings referenced by the first “SymbolDefn” in the “SymbolList”, followed by any one of the character strings referenced by the second “SymbolDefn” in the “SymbolList”, and so on, up to and including the last “SymbolDefn” in the “SymbolList”.

Note – The “EmbeddedDefinitions” (if any) play no direct part in determining these strings.

A.3.10 An “astring” references the sequence of characters in the “astring” without the enclosing pair of ”.

A.3.11 A “productionreference” references any sequence of characters specified by the “Production” it identifies.

A.3.12 The sequences of characters referenced by the next four alternatives for “SymbolDefn” are specified in Table 8/X.208.

Note – The sequences of characters referenced by the “string” should be terminated in an instance of the macro notation by the appearance of a sequence referenced by the next “SymbolDefn” in the “SymbolList”.

A.3.13 A “type” references any sequence of symbols which forms a “Type” notation as specified in § 12.1.

Note – The “DefinedType” of § 12.1 may in this case contain a “localtypereference” referencing a type defined in the macro notation.

A.3.14 A “type(localtypereference)” references any sequence of symbols which forms a “Type” as specified in § 12.1, but in addition assigns that type to the “localtypereference”. A later assignment may occur to the same “localtypereference”.

A.3.15 A “value(MacroType)” references any sequence of symbols which forms a “Value” notation (as specified in § 12.7) for the type specified by the “Macro Type”.

A.3.16 A “value(localvaluereference MacroType)” references any sequence of symbols which forms a “Value” notation (as specified in § 12.7) for the type specified by “MacroType”, but in addition assigns the value specified by the value notation to the “localvaluereference”. A later assignment may occur to the “localvaluereference”.

A.3.17 A “value(VALUE MacroType)” references any sequence of symbols which forms a “Value” notation (as specified in § 12.7) for the type specified by “MacroType”, but in addition returns the value as the value specified by the value notation. The type of the value returned is the type referenced by MacroType.

A.3.18 Precisely one assignment to VALUE (as specified in § A.3.17 or in § A.3.19) occurs in the analysis of any correct instance of the new value notation.

A.3.19 The notation for an “EmbeddedDefinitions” shall be:

```
EmbeddedDefinitions ::=
        <EmbeddedDefinitionList>

EmbeddedDefinitionList ::=
        EmbeddedDefinition |
        EmbeddedDefinitionList
        EmbeddedDefinition

EmbeddedDefinition ::=
        LocalTypeassignment |
        LocalValueassignment

LocalTypeassignment ::=
        localtypereference
        “::=”
        MacroType

LocalValueassignment ::=
        localvaluereference
        MacroType
        “::=”
        MacroValue

MacroValue ::=
        Value |
        localvaluereference
```

The assignment of a “MacroType” to a “localtypereference” (or of a “MacroValue” to a “localvaluereference”) within an “EmbeddedDefinitions” takes effect during the syntax analysis of an instance of the new notation at the time when the “EmbeddedDefinitions” is encountered, and persists until redefinition of the “localtypereference” or “localvaluereference” occurs.

Note 1 – The use of the associated “localtypereference” or “localvaluereference” elsewhere in the “Alternative” implies assumptions on the nature of the parsing algorithm. Such assumptions should be indicated by comment. For example, use of the “localtypereference” textually following the “EmbeddedDefinitions” implies a left to right parse.

Note 2 – The "localvaluereference" "VALUE" may be assigned a value either by the "value (VALUE MacroType)" construct or by an "EmbeddedDefinition". In both cases, the value is returned, as specified in § A.3.17.

A.4 *Use of the new notation*

Whenever a "Type" (or "Value") notation is called for by this Recommendation, an instance of the type notation (or value notation) defined by a macro may be used, provided that the macro is either:

a) defined within the same module; or

b) imported into the module, by means of the appearance of the "macroreference" in the "SymbolsImported" of the module.

To allow the latter possibility, a "macroreference" can appear as a "Symbol" in § 9.1.

Note 1 – This extension to the standard ASN.1 notation is not shown in the body of this Recommendation.

Note 2 – It is possible to construct modules including sequences of type assignment and macro definitions which make parsing of the value syntax in DEFAULT values arbitrarily complex.

ANNEX B

(to Recommendation X.208)

ISO assignment of OBJECT IDENTIFIER component values

B.1 Three arcs are specified from the root node. The assignment of values and identifiers, and the authority for assignment of subsequent component values, are as follows:

Value	Identifier	Authority for subsequent assignments
0	ccitt	CCITT
1	iso	ISO
2	joint-iso-ccitt	See Annex D

Note – The remainder of this annex concerns itself only with ISO assignment of values.

B.2 The identifiers "ccitt", "iso" and "joint-iso-ccitt", assigned above, may each be used as a "NameForm".

B.3 Four arcs are specified from the node identified by "iso". The assignment of values and identifiers is

Value	Identifier	Authority for subsequent assignments
0	standard	See § B.4
1	registration-authority	See § B.5
2	member-body	See § B.6
3	identified-organization	See § B.7

These identifiers may be used as a "NameForm".

B.4 The arcs below "standard" shall each have the value of the number of an International Standard. Where the International Standard is multi-part, there shall be an additional arc for the part number, unless this is specifically excluded in the text of the International Standard. Further arcs shall have values as defined in that International Standard.

Note – If a non-multipart International Standard allocates object identifiers, and subsequently becomes a multipart International Standard, it shall continue to allocate object identifiers as if it were a single part International Standard.

B.5 The arcs below "registration authority" are reserved for an addendum to this Recommendation which will be progressed alongide the establishment of procedures for the identification of specific OSI Registration Authorities.

B.6 The arcs immediately below "member-body" shall have values of three digit numeric country code, as specified in ISO 3166, that identifies the ISO Member Body in that country (see Note). The "NameForm" of object identifier component is not permitted with these identifiers. Arcs below the "country code" are not defined in this Recommendation.

Note – The existence of a country code in ISO 3166 does not necessarily imply that there is an ISO Member Body representing that country or that the ISO Member Body for that country administers a scheme for the allocation of object identifier components.

B.7 The arcs immediately below "identified-organization" shall have values of an International Code Designator (ICD) allocated by the Registration Authority for ISO 6523 that identify an issuing organization specifically registered by the authority as allocating object identifier components (see Notes 1 and 2). The arcs immediately below the ICD shall have values of an "organization code" allocated by the issuing organization in accordance with ISO 6523. Arcs below "organization code" are not defined by this Recommendation (see Note 3).

Note 1 – The requirement that issuing organizations are recorded by the Registration Authority for ISO 6523 as allocating organization codes for the purpose of object identifier components ensures that only numerical values in accordance with this Recommendation are allocated.

Note 2 – The declaration that an issuing organization allocates organization codes for the purpose of object identifier components does not preclude the use of these codes for other purposes.

Note 3 – It is assumed that the organizations identified by the "organization code" will define further arcs in such a way as to ensure allocation of unique values.

Note 4 – The effect of B.7 is that any organization can obtain an organization code from an appropriate issuing organization, and can then assign OBJECT IDENTIFIER values for its own purposes, with the assurance that those values will not conflict with values assigned by other organizations. By this means, a manufacturer could, for example, assign an OBJECT IDENTIFIER to its own proprietary information formats.

ANNEX C

(to Recommendation X.208)

CCITT assignment of OBJECT IDENTIFIER component values

C.1 Three arcs are specified from the root node. The assignment of values and identifiers, and the authority for assignment of subsequent component values, are as follows:

Value	Identifier	Authority for subsequent assignments
0	ccitt	CCITT
1	iso	ISO
2	joint-iso-ccitt	See Annex D

Note – The remainder of this annex concerns itself only with CCITT assignment of values.

C.2 The identifiers "ccitt", "iso" and "joint-iso-ccitt", assigned above, may each be used as a "NameForm".

C.3 Four arcs are specified from the node identified by "ccitt". The assignment of values and identifiers is

Value	Identifier	Authority for subsequent assignments
0	recommendation	See § C.4
1	question	See § C.5
2	administration	See § C.6
3	network-operator	See § C.7

These identifiers may be used as a "NameForm".

C.4 The arcs below "recommendation" have the value 1 to 26 with assigned identifiers of a to z. Arcs below these have the numbers of CCITT Recommendations in the series identified by the letter. Arcs below this are determined as necessary by the CCITT Recommendation. The identifiers a to z may be used as a "NameForm".

C.5 The arcs below "question" have values corresponding to CCITT Study Groups, qualified by the Study Period. The value is computed by the formula:

study group number + (Period * 32)

where "Period" has the value 0 for 1984-1988, 1 for 1988-1992, etc., and the multiplier is 32 decimal.

The arcs below each study group have the values corresponding to the questions assigned to that study group. Arcs below this are determined as necessary by the group (e.g. working party or special rapporteur group) assigned to study the question.

C.6 The arcs below administration have the values of X.121 DCCs. Arcs below this are determined as necessary by the Administration of the country identified by the X.121 DCC.

C.7 The arcs below "network-operator" have the value of X.121 DNICs. Arcs below this are determined as necessary by the Administration or RPOA identified by the DNIC.

ANNEX D

(to Recommendation X.208)

Joint assignment of OBJECT IDENTIFIER component values

D.1 Three arcs are specified from the root node. The assignment of values and identifiers, and the authority for assignment of subsequent component values, are as follows:

Value	Identifier	Authority for subsequent assignments
0	ccitt	CCITT
1	iso	ISO
2	joint-iso-ccitt	See below

Note – The remainder of this annex concerns itself only with ISO-CCITT assignment of values.

D.2 The identifiers "ccitt", "iso" and "joint-iso-ccitt", assigned above, may each be used as a "NameForm".

D.3 The arcs below "joint-iso-ccitt" have values which are assigned and agreed from time to time by ISO and CCITT to identify areas of joint ISO-CCITT standardisation activity, in accordance with the "Procedures for assignment of object identifier component values for joint ISO-CCITT use"[1].

D.4 The arcs beneath each arc identified by the mechanisms of § D.3 shall be allocated in accordance with mechanisms established when the arc is allocated.

Note – It is expected that this will involve delegation of authority to the joint agreement of CCITT and ISO Rapporteurs for the joint area of work.

D.5 Initial ISO Standards and CCITT Recommendations in areas of joint ISO-CCITT activity require to allocate OBJECT IDENTIFIERS in advance of the establishment of the procedures of D.3, and hence allocate in accordance with annexes B or C. Information objects identified in this way by ISO Standards or CCITT Recommendations shall not have their OBJECT IDENTIFIERS changed when the procedures of D.3 are established.

APPENDIX I

(to Recommendation X.208)

Exemples and hints

This appendix 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.

I.1 *Example of a personnel record*

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

---

[1] The Registration Authority for assignment of object identifier component values for joint ISO-CCITT use is the American National Standards Institute (ANSI), 1430 Broadway, New York, NY 10018, USA.

I.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

I.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] IMPLICIT SET
{                               Name,
title                           [0] VisibleString,
number                          EmployeeNumber,
dateOfHire                      [1] Date,
 nameOfSpouse                   [2] Name,
 children                       [3] IMPLICIT
                                SEQUENCE OF
                                ChildInformation
                                DEFAULT { } }

ChildInformation ::= SET
  {                             Name,
   dateOfBirth                  [0] Date}

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE
{givenName                      VisibleString,
 initial                        VisibleString,
 familyName                     VisibleString}

EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER

Date ::= [APPLICATION 3] IMPLICIT VisibleString-- YYYYMMDD
```

This example illustrates an aspect of the parsing of the ASN.1 syntax. The sintactic construct “DEFAULT” can only be applied to an element 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”.

I.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.

```
{                               {givenName “John”,initial “P”,familyName “Smith”}
 title                          “Director”
 number                         51
 dataOfHire                     “19710917”
 nameOfSpouse                   {givenName “Mary”,initial “T”,familyName “Smith”}
 children
    {{{ givenName “Ralph”,initial “T”,familyName “Smith”}
              dateOfBirth “19571111”}
              {{givenName ”Susan”,initial “B”,familyName “Jones”}
              dateOfBirth “19590717”}}}
```

I.2 *Guidelines for use of the notation*

The data types and formal notation defined by this Recommendation 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 minimise 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 defines in section three are not dealt with here.

I.2.1 *Boolean*

I.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
```

I.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
```

See also § I.2.3.2

I.2.2 *Integer*

I.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
```

I.2.2.2 Define the minimum and maximum allowed values of an integer type as distinguished values.

Example:

```
DayOfTheMonth ::= INTEGER {first(1),last(31)}
```

I.2.3 *Enumerated*

I.2.3.1 Use an enumerated type with distinguished values 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)}
```

I.2.3.2 Use an 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(0),married(1)}
```

in anticipation of

```
MaritalStatus ::= ENUMERATED {single(0),married(1),widowed(2)}
```

I.2.4 *Real*

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

Example:

```
AngleInRadians ::= REAL
pi REAL ::= {31415926535897932384626433832 79, 10, -30}
```

I.2.5 *Bit string*

I.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 CCITT
   -- Recommendation T.4.
```

I.2.5.2 Define the first and last meaningful bits of a fixed-length bit string as distinguished bits.

Example:

```
Nibble ::= BIT STRING {first(0),last(3)}
```

I.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.

Example:

```
SunnyDaysOfTheMonth ::= BIT STRING {first(1),last(31)}
   -- Day i was sunny if and only if bit i is one
```

I.2.5.4 Use a bit string type with distinguished values to model the values of a collection of related logical variables.

Example:

```
PersonalStatus ::= BIT STRING
   {married(0),employed(1),veteran(2),collegeGraduate(3)}
```

I.2.6 *Octet string*

I.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
   -- CCITT Recommendations T.5 and T.6
```

I.2.6.2 Use a character string type in preference to an octet string type, where an appropriate one is available.

Example:

```
Surname ::= PrintableString
```

I.2.6.3 Use an octet string type to model any string of information which cannot be modelled using one of the character string types. Be sure to specify the repertoire of characters and their coding into octets.

Example:

```
PackedBCDString ::= OCTET STRING
   -- the digits 0 through 9, two digits per octet,
   -- each digit encoded as 0000 to 1001,
   -- 1111₂ used for padding.
```

I.2.7 *Null*

Use a null type to indicate the effective absence of an element of a sequence.

Example:

```
PatientIdentifier ::= SEQUENCE
  {name                 VisibleString,
    roomNumber          CHOICE
            {INTEGER,
            NULL -- if an out-patient --}}
```

Note – The use of “OPTIONAL” provides an equivalent facility.

I.2.8 *Sequence and sequence-of*

I.2.8.1 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 the order in which they joined
```

I.2.8.2 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 makeup 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}
```

I.2.8.3 Use a 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 makeup 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}
```

I.2.8.4 If the elements of a sequence type are fixed in number but of several types, a reference name should be assigned to every element whose purpose is not fully evident from its type.

Example:

```
File ::= SEQUENCE
  {                     ContentType,
  other                 FileAttributes,
  content               ANY}
```

See also § I.2.5.3, § I.2.5.4, and § I.2.7.

I.2.9 *Set*

I.2.9.1 Use a set type to model a collection of variables whose number is known and modest and whose order is insignificant. Identify each variable by context-specifically tagging it.

Example:

```
UserName ::= SET
  {personalName         [0] IMPLICIT VisibleString,
    organisationName    [1] IMPLICIT VisibleString,
    countryName         [2] IMPLICIT VisibleString}
```

I.2.9.2 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. Identify each variable by context-specifically tagging it.

Example:

```
UserName ::= SET
  {personalName         [0] IMPLICIT VisibleString,
    organisationName    [1] IMPLICIT VisibleString OPTIONAL
      -- defaults to that of the local organisation --,
```

```
    countryName             [2] IMPLICIT VisibleString OPTIONAL
      -- defaults to that of the local country --}
```

I.2.9.3 Use a set type to model a collection of variables whose makeup is likely to change from one version of the protocol to the next. Identify each variable by context-specifically tagging it.

Example:

```
UserName ::= SET
  {personalName             [0] IMPLICIT VisibleString,
    organisationName        [1] IMPLICIT VisibleString OPTIONAL
      -- defaults to that of the local organisation --,

    countryName             [2] IMPLICIT VisibleString OPTIONAL
      -- defaults to that of the local country
      -- other optional attributes are for further study --}
```

I.2.9.4 If the members of a set type are fixed in number, a reference name should be assigned to every member whose purpose is not fully evident from its type.

Example:

```
FileAttributes ::= SET
  {owner                    [0] IMPLICIT UserName,
   sizeOfContentInOctets    [1] IMPLICIT INTEGER,
                            [2] IMPLICIT AccessControls,
    . . .}
```

I.2.9.5 Use a set 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
```

See also § I.2.5.4 and § I.2.10.3.

I.2.10 *Tagged*

I.2.10.1 Use a universal tagged type to define - in this Recommendation only - a generally useful, application independent data type that must be distinguishable (by means of its representation) from all other data types.

Example:

```
EncryptionKey ::= [UNIVERSAL 30] IMPLICIT OCTET STRING
-- seven octets
```

I.2.10.2 Use an application-wide tagged type to define a data type that finds wide, scattered use within a particular presentation context and that must be dintinguisable (by means of its representation) from all other data types used in the presentation context.

Example:

```
FileName ::= [APPLICATION 8] IMPLICIT SEQUENCE
  {directoryName                VisibleString,
    directoryRelativeFileName   VisibleString}
```

I.2.10.3 Use context-specific tagged types to distinguish the members of a set. Assign numeric tags starting with zero if their only constraint is distinctness.

Example:

```
CustomerRecord ::= SET
  {name                     [0] IMPLICIT VisibleString,
   mailingAddress           [1] IMPLICIT VisibleString,
   accountNumber            [2] IMPLICIT INTEGER,
   balanceDue               [3] IMPLICIT INTEGER -- in cents --}
```

I.2.10.4 Where a particular set member has been application-wide tagged, a further context-specific tag need not be used, unless it is (or may be in the future) needed for distinctness. Where the set member has been universally tagged, a further context-specific tag should be used.

Example:

```
ProductRecord ::= SET
  {                         UniformCode,
   description              [0] IMPLICIT VisibleString,
   inventoryNo              [1] IMPLICIT INTEGER,
   inventoryLevel           [2] IMPLICIT INTEGER}

UniformCode ::= [APPLICATION 13] IMPLICIT INTEGER
```

I.2.10.5 Use context-specific tagged types to distinguish the alternatives of a CHOICE. Assign numeric tags starting with zero if their only constraint is distinctness.

Example:

```
CustomerAttribute ::= CHOICE
  {name                     [0] IMPLICIT VisibleString,
     mailingAddress         [1] IMPLICIT VisibleString,
     accountNumber          [2] IMPLICIT INTEGER,
     balanceDue             [3] IMPLICIT INTEGER -- in cents --}
```

I.2.10.6 Where a particular CHOICE alternative has been defined using an application-wide tagged type, a further context-specific tag need not be used, unless it is (or maybe in the future) needed for distinctness.

Example:

```
ProductDesignator ::= CHOICE
  {                         UniformCode,
   description              [0] IMPLICIT VisibleString,
   inventoryNo              [1] IMPLICIT INTEGER}

UniformCode ::= [APPLICATION 13] IMPLICIT INTEGER
```

I.2.10.7 Where a particular CHOICE alternative has been universally tagged, a further context-specific tag should be used, unless the provision of more than one universal type is the purpose of the choice.

Example:

```
CustomerIdentifier ::= CHOICE
  {name                     VisibleString,
   number                   INTEGER}
```

I.2.10.8 Use a private-use tagged type to define a data type that finds use within a particular organisation or country and that must be distinguishable (by means of its representation) from all other data types used by that organisation or country.

Example:

```
AcmeBadgeNumber ::= [PRIVATE 2] IMPLICIT INTEGER
```

I.2.10.9 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.

See also § I.2.9.1, § I.2.9.2, § I.2.11.1 and § I.2.11.2.

I.2.11 *Choice*

I.2.11.1 Use a CHOICE to model a variable that is selected from a collection of variables whose number is known and modest. Identify each candidate variable by context-specifically tagging it.

Example:

```
FileIdentifier ::= CHOICE
  {relativeName             [0] IMPLICIT VisibleString,
     -- name of file (for example, “MarchProgressReport”)
```

```
    absoluteName            [1] IMPLICIT VisibleString,
      -- name of file and containing directory
      -- (for example, "<Williams>MarchProgressReport")

    serialNumber            [2] IMPLICIT INTEGER
      -- system-assigned identifier for file --}
```

I.2.11.2 Use a CHOICE to model a variable that is selected from a collection of variables whose makeup is likely to change from one version of the protocol to the next. Identify each candidate variable by context-specifically tagging it.

Example:

```
FileIdentifier ::= CHOICE
  {relativeName             [0] IMPLICIT VisibleString,
     -- name of file (for example, "MarchProgressReport")

   absoluteName             [1] IMPLICIT VisibleString,
     -- name of file and containing directory
     -- (for example, "<Williams>MarchProgressReport")
    -- other forms of file identifiers are for further study --}
```

I.2.11.3 A reference name should be assigned to each alternative whose purpose is not fully evident from its type.

Example:

```
FileIdentifier ::= CHOICE
  {relativeName             [0] IMPLICIT VisibleString,
       -- name of file (for example, "MarchProgressReport")

    absoluteName            [1] IMPLICIT VisibleString,
       -- names of file and containing directory
       -- (for example, "<Williams>MarchProgressReport")

                            [2] IMPLICIT SerialNumber
       -- system-assigned identifier for file --}
```

I.2.11.4 Where implicit tagging is the norm in a particular application of this Recommendation, use a CHOICE of only one type where the possibility is envisaged or more than one type being permitted in the future. This precludes the possibility of implicit tagging taking place, and thus aids transition.

Example:

```
Greeting ::= [APPLICATION 12] CHOICE
                            {VisibleString}
```

in anticipation of

```
Greeting ::= [APPLICATION 12] CHOICE
                            {VisibleString,
                            Voice}
```

I.2.12 *Selection type*

I.2.12.1 Use a selection type to model a variable whose type is that of some particular alternatives of a previously defined CHOICE.

I.2.12.2 Consider the definition:

```
FileAttribute ::= CHOICE
{date-last-used             INTEGER,
   file-name                VisibleString}
```

then the following definition is possible:

```
CurrentAttributes ::= SEQUENCE
{date-last-used             <FileAttribute,
   file-name                <FileAttribute}
```

with a possible value notation of

{date-last-used 27
file-name “PROGRAM”}

The following definition is also possible:

AttributeList ::= SEQUENCE
{first-attribute date-last-used <FileAttribute,
second-attribute file-name <FileAttribute}

with a possible value notation of

{first-attribute 27,
second-attribute “PROGRAM”}

I.2.13 *Any*

I.2.13.1 Use an any type to model a variable whose type is unspecified, or specified elsewhere using ASN.1.

Example:

MessageContents ::= ANY
-- a data element whose type is specified
-- outside this Recommendation using the ASN.1 notation.

I.2.14 *External*

I.2.14.1 Use an external 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 ::= EXTERNAL

DocumentList ::= SEQUENCE OF EXTERNAL

I.3 *An example of the use of the macro notation*

Suppose it is desired to have a notation for type definition of the form

PAIR TYPEX = ... TYPEY = ...

with a corresponding value notation allowing

(X = ----, Y = ----)

The . . . . and the ---- refer to any ASN.1 type and corresponding value respectively.

This macro type notation could be used to define types and values as follows:

T1 ::= PAIR

TYPEX = INTEGER
TYPEY = BOOLEAN

T2 ::= PAIR

TYPEX = VisibleString
TYPEY = T1

Then a value of type T1 might be:

(X = 3, Y = TRUE)

and a value of type T2 might be:

(X = “Name”, Y = (X = 4, Y = FALSE))

The following macro definition could be used to establish this new notation, as an extension of the basic ASN.1:

PAIR
MACRO ::= BEGIN

```
TYPE NOTATION ::=
        "TYPEX"
        "="
        type (Local-type-1)
          -- Expects any ASN.1 type and assigns it
          -- to the variable Local-type-1;
        "TYPEY"
        "="
        type (Local-type-2)
          -- Expects a second ASN.1 type and assigns
          -- it to the variable Local-type-2;

VALUE NOTATION ::=
        "("
        "X"
        "="
        value (Local-value-1 Local-type-1)
          -- Expects a value for the type in
          -- Local-type-1, and assigns it
          -- to the variable Local-value-1;
        ","
        "Y"
        "="
        value (Local-value-2 Local-type-2)
          -- Expects a value for the type in
          -- Local-type-2 and assigns it
          -- to the variable Local-value-2;
        <VALUE SEQUENCE {Local-type-1, Local-type-2}
          ::= {Local-value-1, Local-value-2}>
        -- This "embedded definition" returns
        -- the final value as the value
        -- of a sequence of the two types.
        ")"

END
```

In this example, the basic ASN.1 type of the returned value is independent of the actual instance of the value notation, but does depend on the instance of the type notation that is used. In other cases, it may be fully determined by the macro definition, or it may depend on the instance of the value notation that is used. Note, however, that in all cases it is the "VALUE NOTATION" production that needs to be examined in order to determine the type of the returned value. The "TYPE NOTATION" production simply defines the syntax for type definition, and establishes the initial value of local variables for use when analysing an instance of the value notation.

I.4 *Use in identifying abstract syntaxes*

I.4.1 Use of the presentation service (Recommendation X.216) 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.

I.4.2 ASN.1 can be used as a general tool in the specification of presentation data values and their grouping into named abstract syntaxes.

I.4.3 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.

I.4.4 The following is an example of text which might appear in an application standard. The end of the example is identified by the phrase "END OF EXAMPLE" in order to avoid confusion.

Example:

```
ISOxxxx-yyyy DEFINITIONS ::=
BEGIN
PDU ::= CHOICE
      {connect-pdu  . . .,
        data-pdu  CHOICE
                  { . . .,
                . . .}     ,
      . . .            }
. . .
END
```

This International Standard assigns the ASN.1 object identifier value

{iso standard xxxx abstract-syntax (1)}

as an abstract syntax name for the set of presentation data values, each of which is a value of the ASN.1 type "ISOxxxx-yyyy.PDU".

The corresponding ASN.1 object descriptor value shall be

"....."

The ASN.1 object identifier and object descriptor values

{joint-iso-ccitt asn1 (1) basic-encoding (1)}

and

"Basic Encoding of a single ASN.1 type"

(assigned to an information object in Recommendation X.209) can be used as a transfer syntax name with this abstract syntax name.

END OF EXAMPLE

I.4.5 The standard may additionally make support of the transfer syntax obtained by applying

{joint-iso-ccitt asn1 (1) basic-encoding (1)}

mandatory for this abstract syntax

I.5 *Subtypes*

I.5.1 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
                                ("0"|"1"|"2"|"3"|"4"|"5"|"6"|
                                "7"|"8"|"9"|"*"|"#")|
                                SIZE (1..63))

ParameterList ::= SET SIZE (0..63) OF Parameter

SmallPrime ::= INTEGER (2|3|5|7|11|13|17|19|23|29)
```

I.5.2 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.

I.5.3 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   [0]    INTEGER,
                         beta    [1]    IA5String OPTIONAL,
                         gamma   [2]    SEQUENCE OF Parameter,
                         delta   [3]    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})
```

I.5.4 If a general-purpose datatype 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)))
```

I.5.5 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 (
                                INCLUDES First-quarter |
                                INCLUDES Second-quarter)

        Second-half ::= Months (
                                INCLUDES Third-quarter |
                                INCLUDES Fourth-quarter )
```

APPENDIX II

(to Recommendation X.208)

Summary of the ASN.1 notation

The following items are defined in clause 8:

typereference	INTEGER	BEGIN
identifier	BIT	END
valuereference	STRING	DEFINITIONS
modulereference	OCTET	EXPLICIT
comment	NULL	ENUMERATED
empty	SEQUENCE	EXPORTS
number	OF	IMPORTS
bstring	SET	
hstring	IMPLICIT	REAL
cstring	CHOICE	INCLUDES
“::=”	ANY	MIN
{	EXTERNAL	MAX
}	OBJECT	SIZE
<	IDENTIFIER	FROM
,	OPTIONAL	WITH
.	DEFAULT	COMPONENT
(	COMPONENTS	PRESENT
)	UNIVERSAL	ABSENT
[	APPLICATION	DEFINED

]	PRIVATE	BY
-	TRUE	PLUS-INFINITY
BOOLEAN	FALSE	MINUS-INFINITY

The following productions are used in this Recommendation, with the above items as terminal symbols:

ModuleDefinition ::=

ModuleIdentifier
DEFINITIONS
TagDefault
"::="
BEGIN
ModuleBody
END

TagDefault ::=
EXPLICIT TAGS |
IMPLICIT TAGS |
empty

ModuleIdentifier ::=
modulereference
AssignedIdentifier

AssignedIdentifier ::=
ObjectIdentifier Value |
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 ModuleIdentifier

SymbolList ::= Symbol, SymbolList | Symbol

Symbol ::= typereference | valuereference

AssignmentList ::=
Assignment AssignmentList |
Assignment

Assignment ::= Typeassignment | Valueassignment

Externaltypereference ::=
modulereference

.
typereference

Externalvaluereference ::=
modulereference
.
valuereference

DefinedType ::= Externaltypereference | typereference

DefinedValue ::= Externalvaluereference | valuereference

Typeassignment ::=
typereference
“::=”
Type

Valueassignment ::=
valuereference
Type
“::=”
Value

Type ::= BuiltinType | DefinedType | Subtype

BuiltinType ::=

BooleanType		
IntegerType		
BitStringType		
OctetStringType		
NullType		
SequenceType		
SequenceOfType		
SetType		
SetOfType		
ChoiceType		
SelectionType		
TaggedType		
AnyType		
ObjectIdentifierType		
CharacterStringType		
UsefulType		
EnumeratedType		
RealType		

NamedType ::= identifier Type | Type | SelectionType

Value ::= BuiltinValue | DefinedValue
BuiltinValue ::=

BooleanValue		
IntegerValue		
BitStringValue		
OctetStringValue		
NullValue		
SequenceValue		
SequenceOfValue		
SetValue		
SetOfValue		
ChoiceValue		
SelectionValue		
TaggedValue		
AnyValue		
ObjectIdentifierValue		
CharacterStringValue		

```
        EnumeratedValue         |
        RealValue               |

NamedValue ::= identifier Value | 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 {Enumeration}

Enumeration ::=
        NamedNumber |
        NamedNumber, Enumeration

EnumeratedValue ::= identifier

RealType ::= REAL

RealValue ::= NumericRealValue | SpecialRealValue

NumericRealValue ::= {Mantissa, Base, Exponent} | 0

Mantissa ::= SignedNumber

Base ::= 2 | 10

Exponent ::= SignedNumber

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 {ElementTypeList} |
        SEQUENCE { }

ElementTypeList ::=
        ElementType |
        ElementTypeList,ElementType
```

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

SequenceValue ::= {ElementValueList} | { }

ElementValueList ::=
        NamedValue |
        ElementValueList,NamedValue

SequenceOfType ::= SEQUENCE OF Type | SEQUENCE

SequenceOfValue ::= {ValueList} | { }

ValueList ::= Value | ValueList,Value

SetType ::= SET{ElementTypeList} | SET { }

SetValue ::= {ElementValueList} | { }

SetOfType ::= SET OF Type | SET

SetOfValue ::= {ValueList} | { }

ChoiceType ::= CHOICE{AlternativeTypeList}

AlternativeTypeList ::=
        NamedType |
        AlternativeTypeList,NamedType

ChoiceValue ::= NamedValue

SelectionType ::= identifier < Type

SelectionValue ::= NamedValue

TaggedType ::=
        Tag Type |
        Tag IMPLICIT Type |
        Tag EXPLICIT Type

Tag ::= [Class ClassNumber]

ClassNumber ::= number | DefinedValue

Class ::=
        UNIVERSAL |
        APPLICATION |
        PRIVATE |
        empty

TaggedValue ::= Value

AnyType ::=
        ANY |
        ANY DEFINED BY identifier

AnyValue ::= Type Value

ObjectIdentifierType ::= OBJECT IDENTIFIER

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

ObjIdComponentList ::=
        ObjIdComponent |
        ObjIdComponent ObjIdComponentList
```

ObjIdComponent ::=
NameForm |
NumberForm |
NameAndNumberForm

NameForm ::= identifier

NumberForm ::= number | DefinedValue

NameAndNumberForm ::= identifier(NumberForm)

CharacterStringType ::= typereference

CharacterStringValue ::= cstring

UsefulType ::= typereference

The following characterstring types are defined in section two:

NumericString	VisibleString
PrintableString	ISO646String
TeletexString	IA5String
T61String	GraphicString
VideotexString	GeneralString

The following useful types are defined in section three:

GeneralizedTime	EXTERNAL
UTCTime	ObjectDescriptor

The following productions are used in section four:

Subtype ::=
ParentType SubtypeSpec |
SET SizeConstraint OF Type |
SEQUENCE SizeConstraint OF Type

ParentType ::= Type

SubtypeSpec ::=
(SubtypeValueSet SubtypeValueSetList)

SubtypeValueSetList ::=
“|”
SubtypeValueSet
SubtypeValueSetList |
empty

SubtypeValueSet ::=
SingleValue |
ContainedSubtype |
ValueRange |
PermittedAlphabet | SizeConstraint | InnerTypeConstraint
SingleValue ::= Value
ContainedSubtype ::= INCLUDES Type
ValueRange ::= LowerEndPoint . . UpperEndPoint

LowerEndpoint ::= LowerEndValue | LowerEndValue <

UpperEndpoint ::= UpperEndValue | <UpperEndValue

LowerEndValue ::= Value | MIN

UpperEndValue ::= Value | MAX

SizeConstraint ::= SIZE SubtypeSpec

PermittedAlphabet ::= FROM SubtypeSpec
InnerTypeConstraints ::=

WITH COMPONENT SingleTypeConstraint |
WITH COMPONENTS MultipleTypeConstraints

SingleTypeConstraint ::= SubtypeSpec

MultipleTypeConstraints ::=
FullSpecification | PartialSpecification

FullSpecification ::= {TypeConstraints}

PartialSpecification ::= {. . ., TypeConstraints}

TypeConstraints ::=
NamedConstraint |
NamedConstraint, TypeConstraints

NamedConstraint ::= identifier Constraint | Constraint
Constraint ::= ValueConstraint PresenceConstraint
ValueConstraint ::= SubtypeSpec | empty
PresenceConstraint ::= PRESENT | ABSENT | empty | OPTIONAL

The following additional items are defined in § A.2 for use in the macro notation:

macroreference	"number"
productionreference	"empty"
localtypereference	MACRO
localvaluereference	TYPE
"|"	NOTATION
>	VALUE
astring	value
"string"	type
"identifier"	

The following productions are used in Annex A, with the above items, and items listed at the head of this appendix, as terminal symbols:

MacroDefinition ::=
macroreference
MACRO
"::="
MacroSubstance

MacroSubstance ::=
BEGIN MacroBody END |
macroreference |
Externalmacroreference

MacroBody ::=
TypeProduction
ValueProduction
SupportingProductions

TypeProduction ::=
TYPE NOTATION
"::="
MacroAlternativeList

ValueProduction ::=
VALUE NOTATION
"::="
MacroAlternativeList

SupportingProductions ::= ProductionList | empty

ProductionList ::= Production | ProductionList Production

Production ::=
productionreference

```
        "::="
        MacroAlternativeList

ExternalmacroreferenceReference ::=
        modulereference . macroreference

MacroAlternativeList ::=
        MacroAlternative |
        MacroAlternativeList "|" MacroAlternative

MacroAlternative ::= SymbolList

SymbolList ::= SymbolElement | SymbolList SymbolElement

SymbolElement ::= SymbolDefn | EmbeddedDefinitions

SymbolDefn ::=
        astring |
        productionreference |
        "string" |
        "identifier" |
        "number" |
        "empty" |
        type |
        type(localtypereference) |
        value(MacroType) |
        value(localvaluereference MacroType) |
        value(VALUE MacroType)

MacroType ::= localtypereference | Type

EmbeddedDefinitions ::= <EmbeddedDefinitionList>

EmbeddedDefinitionList ::=
        EmbeddedDefinition |
        EmbeddedDefinitionList EmbeddedDefinition

EmbeddedDefinition ::=
        LocalTypeassignment |
        LocalValueassignment

LocalTypeassignment ::=
        Localtypereference
        "::="
        MacroType

LocalValueassignment ::=
        localvaluereference
        MacroType
        "::="
        MacroValue

MacroValue ::= Value | localvaluereference
```