ASN.1: abstract syntax

György RÉTHY, János Zoltán SZABÓ
Test Competence Center, Ericsson Hungary
Gusztáv ADAMIS
BME TMIT
adamis@tmit.bme.hu
Friday, March 13, 2015

ASN.1 abstract syntax

INTRODUCTION

Classical way of protocol specification & implementation

"ASN.1 way" of protocol specification & implementation

textual/table specification
(rarely SDL/MSC)

Procedures

Structure & information content of protocol elements (messages, IEs etc.)

Protocol specification

Procedures

Structure & information content of protocol elements (messages, IEs etc.)

ASN.1 abstact syntax

Manual encoding of protocol elements & procedures

Protocol imple-mentation

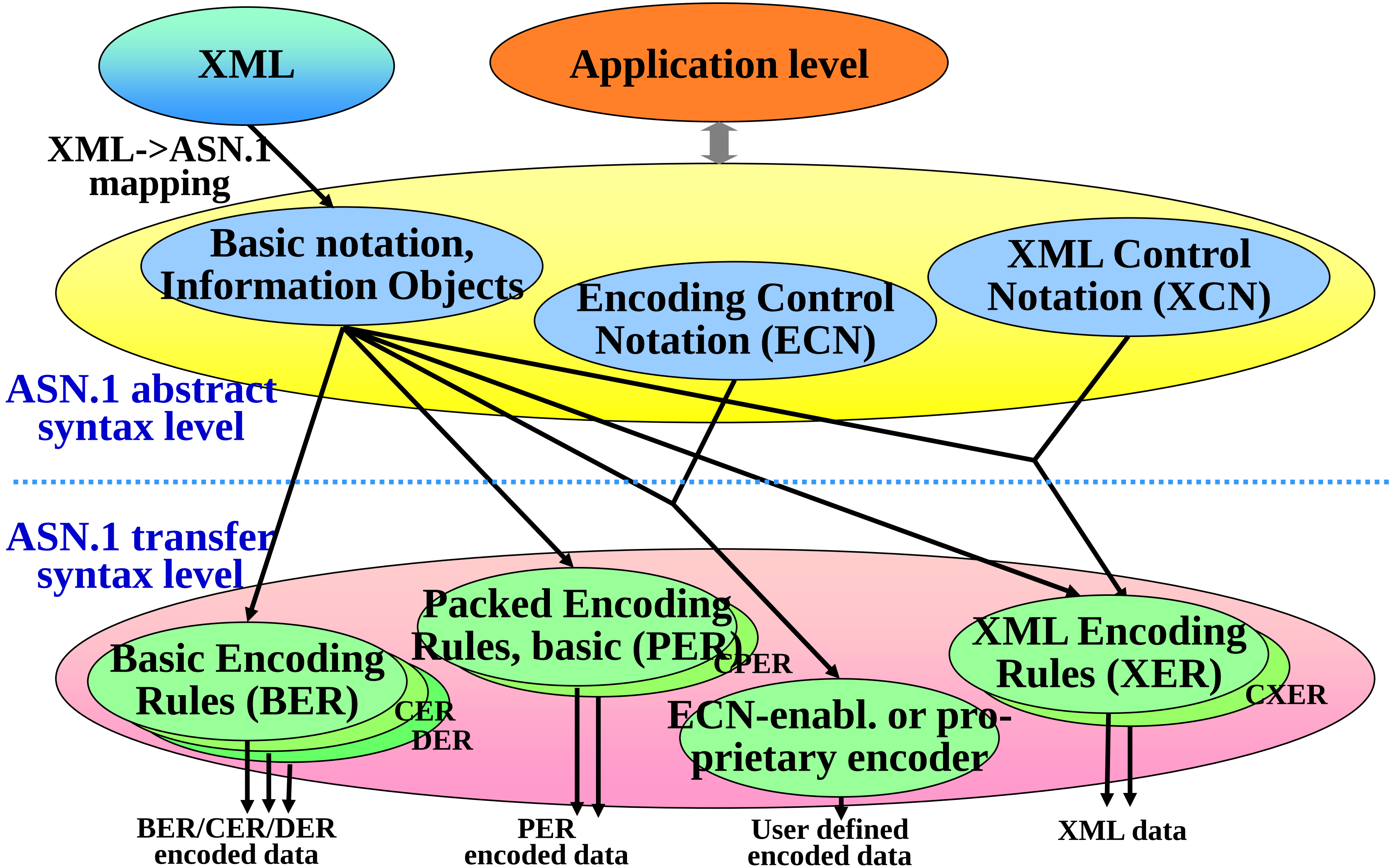
Manual encoding of procedures

Automatic encoding of ASN.1 elements by a generic purpose ASN.1 encoder

May be 3rd party implementation

ASN.1 transfer syntax

Structure of ASN.1



ASN.1 - 1997 (multipart) version - 1

X.680

WILL BE SUPERSEEDED SOON!

Recommendation X.680 (12/97) - Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation
Corrigendum 1 (06/99) to Recommendation X.680
Corrigendum 2 (03/00) to Recommendation X.680
Amendment 1 (06/99) to Recommendation X.680 - Relative object identifiers
Amendment 2 (06/99) to Recommendation X.680 - ASN.1 semantic model
Corrigendum 3 (02/01) to Recommendation X.680
Erratum to Corrigendum 3 (05/01) to Recommendation X.680
Corrigendum 4 (03/01) to Recommendation X.680

X.681

Recommendation X.681 (12/97) - Information technology - Abstract Syntax Notation One (ASN.1): Information object specification
Corrigendum 1 (06/99) to Recommendation X.681
Amendment 1 (06/99) to Recommendation X.681 - ASN.1 semantic model

ASN.1 - 1997 (multipart) version - 2

WILL BE SUPERSEEDED SOON!

X.682

Recommendation X.682 (12/97) - Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification

Corrigendum 1 (03/00) to Recommendation X.682

Corrigendum 2 (02/01) to Recommendation X.682

Corrigendum 3 (03/01) to Recommendation X.682

X.683

Recommendation X.683 (12/97) - Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications

Amendment 1 (06/99) to Recommendation X.683 - ASN.1 semantic model

Recommended reading

- **John Larmouth, *ASN.1 Complete*,**
  Elsevier-Morgan Kaufmann, 1999
  - **Downloadable PDF version: http://www.oss.com/asn1/larmouth.html**
- **Olivier Dubuisson, *ASN.1 - Communication between heterogeneous systems*,**
  Elsevier-Morgan Kaufmann, 2000
  - **Downloadable PDF version: http://www.oss.com/asn1/dubuisson.html**

ASN.1 abstract syntax

CONTENTS

ASN.1 abstract syntax — Naming conventions -1

- **Characters used in names**

  A to Z *(latin capital)*
  a to z *(latin small)*
  0 to 9 *(digits)*
  "-" *(hyphen)*

- ***Naming rules***
  - **General rules:**
    first character is never a digit or hyphen
    last character is never a hyphen
    hyphen shall not be followed by another hyphen
  - **First character is a capital letter for:**
    Type name *(typereference)*
    Module name
    Information object set name *(objectsetreference = typereference)*

ASN.1 abstract syntax — Naming conventions -2

- *Naming rules (cont.)*
  - First character is a small letter for:
    value name *(valuereference)*
    identifier (component, named value, enumeration etc.)
    information object name *(objectreference = valuereference)*
  - ALL CAPITAL letters used in:
    Information object class name *(objectclassreference)*
  - First character is an ampersand (&):
    Information object class fields names
    than *typereference* for type field, value set field, object set field
    valuereference for value field, object field
  - Word (in user defined syntax for an information object class):
    ALL capital letters & no digits

ASN.1 abstract syntax

White-space & newline

- **White-space may contain:**

  HORIZONTAL TABULATION (9)
  SPACE (32)
  LINE FEED (10)
  VERTICAL TAB (11)
  FORM FEED (12)
  CARRIAGE RETURN (13)

- **Newline may contain:**

  LINE FEED (10)
  VERTICAL TAB (11)
  FORM FEED (12)
  CARRIAGE RETURN (13)

ASN.1 abstract syntax Reserved words

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

ASN.1 abstract syntax

Module definition

Example (fields are separated by <u>white-space</u>)

Module-Name { <optional object identifier> } — *-- name is mandatory*

DEFINITIONS — -- mandatory keyword

<u>EXPLICIT TAGS</u> -- or
IMPLICIT TAGS -- or
AUTOMATIC TAGS
— *optional (see later)*

EXTENSIBILITY IMPLIED — *-- optional*

::= — -- mandatory keyword

BEGIN — -- mandatory keyword

EXPORTS *<comma separated list>* ; — -- optional

IMPORTS *<comma separated list>* FROM *<module identifier>*
<comma separated list> FROM *<module identifier>*; — -- optional

<assignments> — *-- definitions in the module body*

END

CONFORMANCE Lab

ASN.1 abstract syntax

Assignments

- **Type**

  UpperIdentifier ::= <type>

- **Value**

  lowerIdentifier <type> ::= <value>

- **Value set**

  UpperIdentifier <type> ::= <valueset>

- **Information object class**

  FULLUPPERIDENTIFIER ::= CLASS { … } | CLASSNAME

- **Information object**

  lowerIdentifier CLASSNAME ::= <object>

- **Information object set**

  UpperIdentifier CLASSNAME ::= <objectset>

ASN.1 abstract syntax

CONTENTS

1. Basic conventions, ASN.1 module definition

2. Built-in ASN.1 types

NULL, BOOLEAN, INTEGER, BIT STRING, OCTET STRING, ENUMERATED, SEQUENCE (OF), SET (OF), CHOICE, selection type, Character string types, REAL, Hole types: ANY, EMBEDDED PDV, EXTERNAL, Unrestricted character string

ASN.1 abstract syntax — NULL, BOOLEAN

- **NULL**
  - The only information it carries: was it SENT or NOT

```
Type-1              ::=     NULL
value-1        NULL ::=     NULL
```

- **BOOLEAN**
  - Logical variable or constant

```
Type-2              ::=     BOOLEAN
value-2     BOOLEAN ::=     TRUE      --  FALSE
```

ASN.1 abstract syntax — INTEGER

- INTEGER
  - positive, negative integer number or (+) 0
  - named numbers (does NOT restrict assignment of other values)

```
Sample1 ::=      INTEGER { zero (0), one (1), other (8) }
value-3     Sample1 ::= zero                  --  value is zero
value-4     Sample1 ::= 5                     -- "- 0" is illegal
```

ASN.1 abstract syntax — BIT STRING

- Sequence of individual bits

  b1 BIT STRING ::= '0001011100110101'B

  b2 BIT STRING ::= 'F0F0'H --> '1111000011110000'B

- Named bit list
  - Just a name of the position: does not restricts the length and value of the string

    B1 ::= BIT STRING { first (0), second (1), trailing (6) }

    b3 B1 ::= '001001100011'B

  - Does not specify trailing zeros (can be added or removed at coding)

    b4 B1 ::= { first, trailing } --> '1000001'B
    --> '100000100'B
    --> '10000010000'B

    b5 B1 ::= { } --> any number of '0's

    b6 B1 ::= 'AA'H --> '10101010'B

ASN.1 abstract syntax — OCTET STRING

- Sequence of individual octets
  - value notation using hstring

missing hex character is interpreted as zero

o1 OCTET STRING ::= ‘278EF’H --> ‘278EF0’ H

most less
significant

- value notation using bstring

missing bits are interpreted as zero

o2 OCTET STRING ::= ‘0001011011101’B --> ‘16E8’H

ASN.1 abstract syntax — *ENUMERATED*

- **ENUMERATED: List of named items**
  - **==>> from ASN.1 abstract syntax point of view has no numeric value !**

    E1 ::= ENUMERATED { first, second, third, fourth, fifth }

first	second	third	fourth	fifth
0	1	2	3	4

  - **At coding an integer value is assigned: starting at "0", step 1**
  - **The value to be coded can be specified: 'hardcoded' values jumped over in automatic numbering**

    E2 ::= ENUMERATED { first, second, third(5), fourth, fifth(1) }

first	second	third(5)	fourth	fifth(1)
0	2	5	3	1

  - **Numeric value is NOT ALLOWED to be referenced within ASN.1!**

    e1 E1 ::= first **-> valid**

    e2 E1 ::= ~~0~~ **-> invalid**

ASN.1 abstract syntax — ENUMERATED, extension

- **The problem:**

Transmitter (version 2)

```
ENUMERATED {
yourIncome,
yourDebit,
accountClosed,
accountReopened
}
```

added in version 2

sent:
ENUMERATED:3

Receiver(version 1)

```
ENUMERATED {
yourIncome, yourDebit,
accountClosed
}
```

Received: ???
(decoding error)

- **Solution: indicate possible additions in the type definition**

```
ENUMERATED {
yourIncome,
yourDebit,
accountClosed, ...,
accountReopened
}
```

called ellipsis

insertion point:
identified by the ellipsis
(additional enumerations can be inserted here, following the comma)

sent:
ENUMERATED:3

```
ENUMERATED {
yourIncome, yourDebit,
accountClosed, ...
}
```

Received: added value
(no error, call local exception handling proc.)

ASN.1 abstract syntax — Rules to extend ENUMERATED

- All values - root and additional - shall be **unique**

  A ::= ENUMERATED { a, b, … , c(0) } --> *invalid as a = c*

  B ::= ENUMERATED { a, b, … , c(2) } --> *valid*

- Values of additional enumerations shall monotonously **increase**

  C ::= ENUMERATED { a, b, … , c, d(2) } --> *invalid as c = d*

  D ::= ENUMERATED { a, b, … , c(3), d(2) } --> *invalid, it should be c < d*

  E ::= ENUMERATED { a, b, … , c(2), d(3) } --> *valid*

- First additional "automatic" value will be the **smallest not used** value in the root

  F ::= ENUMERATED { a, b, c(0), … , d, e } --> *d=3, e=4 (c=0, a=1, b=2)*

  G ::= ENUMERATED { a, b, c(10), … , d, e } --> *d=2, e=3 (a=0, b=1, c=10)*

  H ::= ENUMERATED { a, b, c(0), … , d, e, f(4) } --> *invalid as e = f*

The numeric value assigned to an enumeration has NO significance within the abstract syntax !

ASN.1 abstract syntax — SEQUENCE

- SEQUENCE: ordered group of types
  - Sending order of “contained” types (components) is determined by the textual order in the ASN.1 spec.
  - Each component consist of a unique (within the type) identifier and a Type => called a “named type”
  - Components may be mandatory, optional or have a default value (default: a mandatory information not necessarily sent on the line)

```
S1  ::=  SEQUENCE {  first       INTEGER,
                     second      BIT STRING,
                     third       OCTET STRING    OPTIONAL,
                     fourth      BOOLEAN         DEFAULT  TRUE   }
```

  - It can be empty, i.e. without components:

```
S2  ::=  SEQUENCE { }
```

ASN.1 abstract syntax — Tagging example -1

- **The problem:**

Transmitter

```
SEQUENCE {
yourIncome      INTEGER OPTIONAL,
yourDebit       INTEGER OPTIONAL,
accountClosed   BOOLEAN
}
```

sent:
INTEGER:5000,
BOOLEAN:FALSE

Receiver

Is yourIncome OR yourDebit received?

- **Solution: add a distinguishing label**

```
SEQUENCE {
yourIncome      [0] INTEGER OPTIONAL,
yourDebit           INTEGER OPTIONAL,
accountClosed       BOOLEAN
}
```

sent:
0+INTEGER:5000,
BOOLEAN FALSE

Be happy: it is yourIncome !

ASN.1 abstract syntax — When tags shall be distinct - SEQUENCE

- In a SEQUENCE
  - Components following an OPTIONAL or DEFAULT component shall have distinct tags up to the first mandatory component (including alternatives of all referenced CHOICEs!)

```
F ::= SEQUENCE { first          INTEGER,
                 second         INTEGER,
                 third          INTEGER    OPTIONAL,
                 fourth         INTEGER              }    --> invalid
```

```
G ::= SEQUENCE { first          INTEGER,
                 second         INTEGER,
                 third    [0]   INTEGER    OPTIONAL,
                 fourth         OCTET STRING         }    --> valid
```

```
H ::= SEQUENCE { first          INTEGER,
                 second         INTEGER,
                 third    [0]   INTEGER    OPTIONAL,
                 fourth   [1]   INTEGER              }    --> valid
```

ASN.1 abstract syntax — SEQUENCE - additions

- COMPONENTS OF
  - transports components of another SEQUENCE: for BER coded signals saves the header of the referenced SEQUENCE (!)

```
ParentSeq  ::= SEQUENCE  {  first      INTEGER        OPTIONAL,
                            second     ENUMERATED { one, two }
                                                  DEFAULT one    }
NewSeq     ::= SEQUENCE {  COMPONENTS OF ParentSeq,
                            new        INTEGER                    }
```

  - Identifiers in the parent type and in the new type shall be different
  - Before coding the new type is created by a COMPONENTS OF transformation: components of the referenced type are inserted into the new type
  - any constraint to the referenced (outer SEQUENCE) type is ignored

```
NewSeq   =>   SEQUENCE {   first      INTEGER        OPTIONAL,
                           second     ENUMERATED { one, two }
                                                 DEFAULT one ,
                           new        INTEGER                    }
```

ASN.1 abstract syntax — *Extension of a SEQUENCE*

- **The general case: type extensible at the middle of root components**

Example:

```
I  ::=   SEQUENCE {   a          A,
                      b          B,
                      ... !1,
                      ... ,
                      c          C   }
```

root components

insertion point: identified by the 1st ellipsis

end of insertions: identified by the 2nd ellipsis

(following the comma there are root elements again); note, that no exception handling can be specified for this point

--> type extensible in later versions

```
I'  ::=   SEQUENCE {   a          A,
                       b          B,
                       ... !1,
                       d          D,
                       e          E,
                       f          F,
                       ... ,
                       c          C   }
```

additional components

--> type added in version 2 (d D)

--> type added in version 2 (e E)

--> type added in version 3 (f F)

--> type extended in versions 2 and 3

ASN.1 abstract syntax — Rules for extended SEQUENCE

– All additional types up to ad including the first mandatory root component shall have distinct tags

Example (non-automatic tagging is supposed!)

```
J ::= SEQUENCE {  a          INTEGER,
                  b          INTEGER,
                  ... !1,
                  d    [0]   INTEGER OPTIONAL,
                  e    [1]   OCTET STRING,
                  f    [2]   INTEGER,
                  ... ,
                  c1         INTEGER OPTIONAL,
                  c2         OCTET STRING
                }
```

tags needed due to addition INTEGER and optional INTEGER in root

tag needed due to OCTET STRING in root

additional components

root components

ASN.1 abstract syntax — Rules for extended SEQUENCE

– In automatic tagging environment, if none of the root components is a tagged type, the additional components **shall not** be tagged types

Example (automatic tagging is supposed!)

K ::= SEQUENCE {
a INTEGER,
b INTEGER,
... !1,
d ~~[0]~~ INTEGER,
e ~~[1]~~ OCTET STRING,
f ~~[2]~~ INTEGER,
... ,
c1 INTEGER OPTIONAL,
c2 OCTET STRING
}

additional components

root components

ASN.1 abstract syntax — Extension of a SEQUENCE

- **Type extensible at the end of root components**

J ::= SEQUENCE { a A,
b B,
... !1~~,~~
~~...~~
}

root components

insertion point:
identified by the 1st ellipsis

"end of extensions" marker
is optional

- **Type extensible at the beginning**

K ::= SEQUENCE { ... !1,
... ,
a A,
b B
}

root components

insertion point:
identified by the 1st ellipsis

ASN.1 abstract syntax SET

- **SET: similar to SEQUENCE but unordered**
  - **Syntax is the same (including extension) but additional rules apply:**

    1. Sending order of components is undetermined

```
S3 ::= SET      { first     INTEGER,
                  second    OCTET STRING   OPTIONAL,
                  fourth    BOOLEAN        DEFAULT TRUE }
```

ASN.1 abstract syntax SET

– 2. all components, root and additional, shall have distinct tags (including alternatives of all referenced CHOICEs!)

```
I  ::=  SET    { first              INTEGER,
                 second             INTEGER,
                 third      [0]     INTEGER   OPTIONAL  }   --> invalid
```

```
J  ::=  SET    { first              INTEGER,
                 second     [1]     INTEGER,
                 third      [0]     INTEGER   OPTIONAL  }   --> valid
```

```
K  ::=  SET    { first              INTEGER,
                 second     [0]     INTEGER   OPTIONAL,
                 third              OCTET STRING        }   --> valid
```

```
I'  ::=  SET   {first       [0]     INTEGER,
                second              INTEGER,
                ...,
                third               INTEGER             }   --> invalid
```

ASN.1 abstract syntax — CHOICE

- **Set of alternative types**
  - Each alternative shall be marked by a unique (within this type) identifier

```
SampleChoice1 ::=   CHOICE {
                        first        INTEGER,
                        second       BIT STRING,
                        third        OCTET STRING
                    }
```

- **Additional alternatives can be inserted following the root alternatives only:**

```
SampleChoice2 ::=   CHOICE {
                first        INTEGER,
                second       BIT STRING,
                third        OCTET STRING,
                ...!1,
                fourth       ENUMERATED,
                fifth        NULL,
                ...
                    }
```

root alternatives

additional alternatives

"end of extensions" marker is optional

ASN.1 abstract syntax — Tagging example -2

- **The problem**:

Transmitter

```
CHOICE{
yourIncome       INTEGER,
accountClosed    BOOLEAN,
yourDebit        INTEGER
}
```

sent:
INTEGER:5000

Receiver

Is yourIncome OR yourDebit received?

- **Solution: add a distinguishing label**

```
CHOICE{
yourIncome [0] INTEGER,
accountClosed  BOOLEAN,
yourDebit      INTEGER
}
```

sent:
0+INTEGER:5000

Be happy: it is yourIncome !

ASN.1 abstract syntax — When taggs shall be distinct - CHOICE

- In a CHOICE
  - all alternatives - including all alternatives of all referenced CHOICE(s)! - shall have distinct tags

```
I ::= CHOICE { first              INTEGER,
               second             INTEGER }      --> invalid
```

```
J ::= CHOICE { first              INTEGER,
               second   [0]       INTEGER }      --> valid
```

```
K ::= CHOICE { first    [0]       INTEGER,
               second             FirstChoice }  --> invalid (see tags first & a)
```

```
L ::= CHOICE { first              FirstChoice,
               second             SecondChoice } --> invalid (see tags a & c
                                                 -->     and also tags b & d )
```

```
FirstChoice  ::= CHOICE { a [0] INTEGER , b [1] INTEGER }   --> valid
SecondChoice ::= CHOICE { c [0] INTEGER , d [1] INTEGER }   --> valid
```

ASN.1 abstract syntax — CHOICE

- **All alternatives (root and additional) shall have distinct tags:**

```
SampleChoice2 ::=    CHOICE  {
        first              INTEGER,
        second    [0]      INTEGER,
        third              OCTET STRING,
        ...!1,
        fourth    [1]      OCTET STRING,
        fifth              BIT STRING,
        ...
    }
```

root alternatives

additional alternatives

all alternatives shall have distinct types

"end of extensions" marker is optional

ASN.1 abstract syntax

Version brackets

- Grouping of elements added in a given version
  - can be used in SEQUENCE, SET and CHOICE, NOT in ~~ENUMERATED~~

```
Sample ::= SEQUENCE {     a    A,
                          b    B,
                          ...,
                        [[ c   C,
                          d    D ]], -- stuff added in version 2
                          ...        }             -- version 2 specification

Sample ::= SEQUENCE {     a    A,
                          b    B,
                          ...,
                        [[ c   C,
                          d    D ]], -- stuff added in version 2
                        [[ e   E,
                          f    F ]], -- stuff added in version 3
                          ...        }             -- version 3 specification
```

ASN.1 abstract syntax — *Selection type*

- Permanently selects one alternative of a CHOICE
  - Applicable to CHOICE parent types only
  - Both type and value notations always includes the identifier of the selected alterative

```
MySeq  ::=  SEQUENCE{
                      a        INTEGER,
                      b        chosen < MyChoice
                     }
```

this whole denotes the type OCTET STRING

```
where
MyChoice ::=  CHOICE  {
                      chosen       OCTET STRING,
                      other        BIT STRING,
                      another      INTEGER
                     }
```

identifier of the selected alternative is not present!

```
myValueForMySeq MySeq ::= {
                      a          5,
                      b          'AA'H
                   }
```

ASN.1 abstract syntax — *SEQUENCE OF and SET OF*

- **SEQUENCE OF, SET OF: repeated values of a single type**
  - **SEQUENCE OF: sending order of values has semantic significance**
  - **SET OF: sending order of values does NOT have semantic significance**
  - **Each component is marked by an identifier (mandatory)**
  - **Components may be mandatory, optional or have a default value (default: a mandatory information not necessarily sent on the line)**

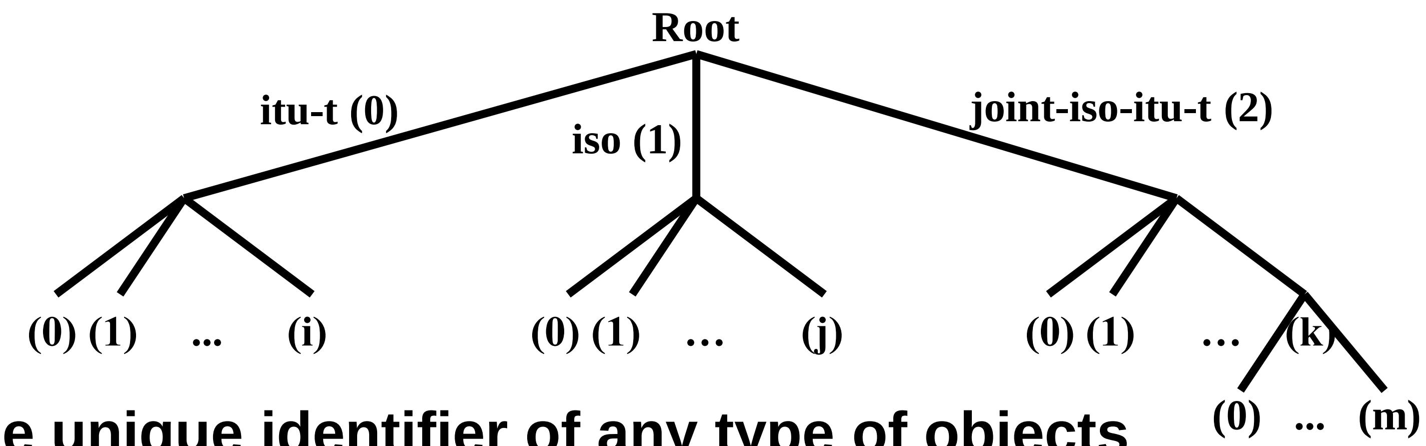
```
S1  ::=   SEQUENCE   OF  AnotherType

where    AnotherType  ::=   INTEGER        -- or another definition

         AnotherType  ::=  CHOICE {  first      INTEGER,
                                     second     BIT STRING,
                                     third      OCTET STRING,
                                     fourth     BOOLEAN   }

valueRef-1  S1  ::=  {  0, 1, 15,  7  }                -- when AnotherType is integer
valueRef-2  S1  ::=  { first : 1, third : '0F'H }      -- when AnotherType is choice
```

ASN.1 abstract syntax — OBJECT IDENTIFIER



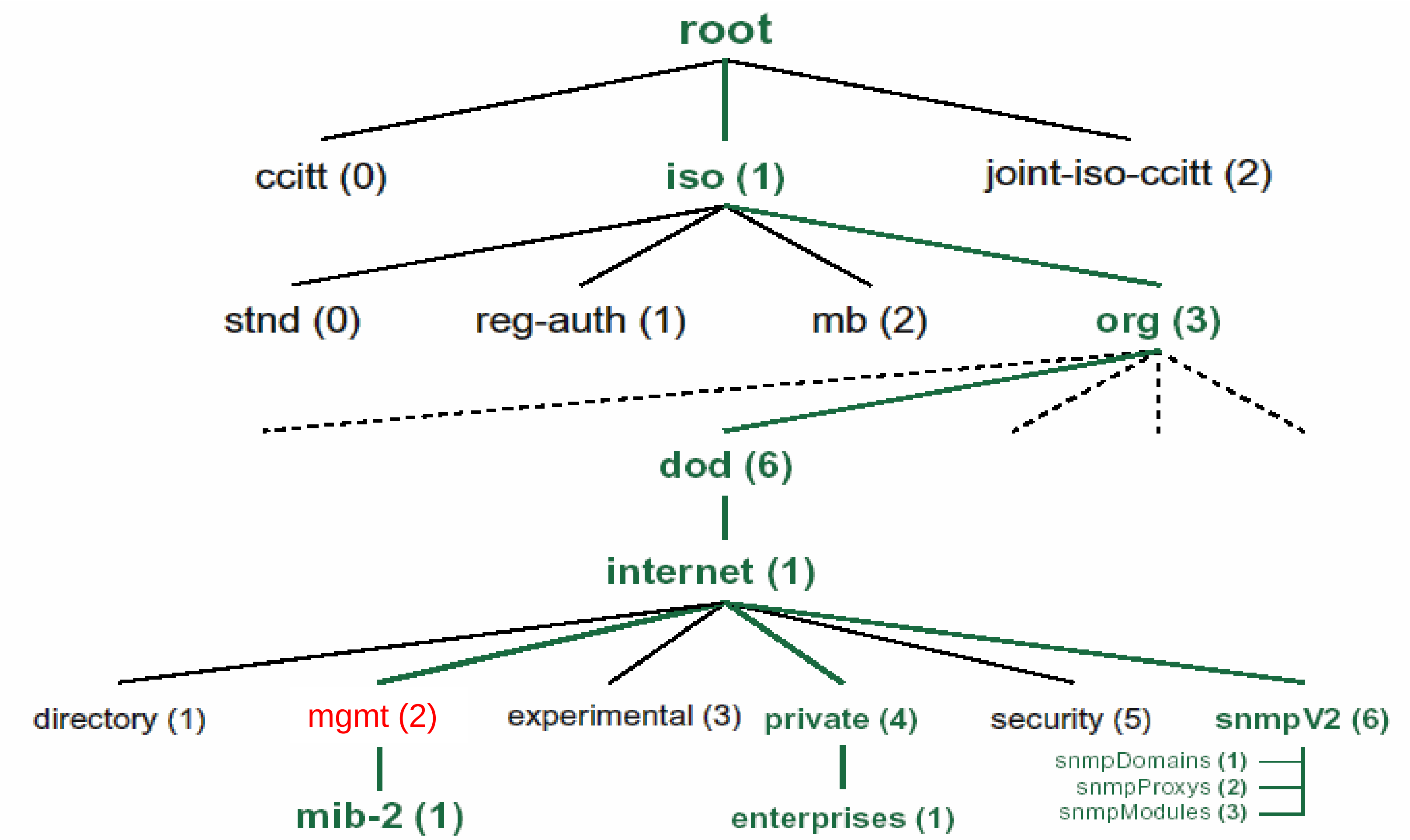
- Worldwide unique identifier of any type of objects
  - OID contains of a series of Object ID components in curly brackets
  - A component may be:

    a name (only for names defined in ITU-T Rec. X.660)

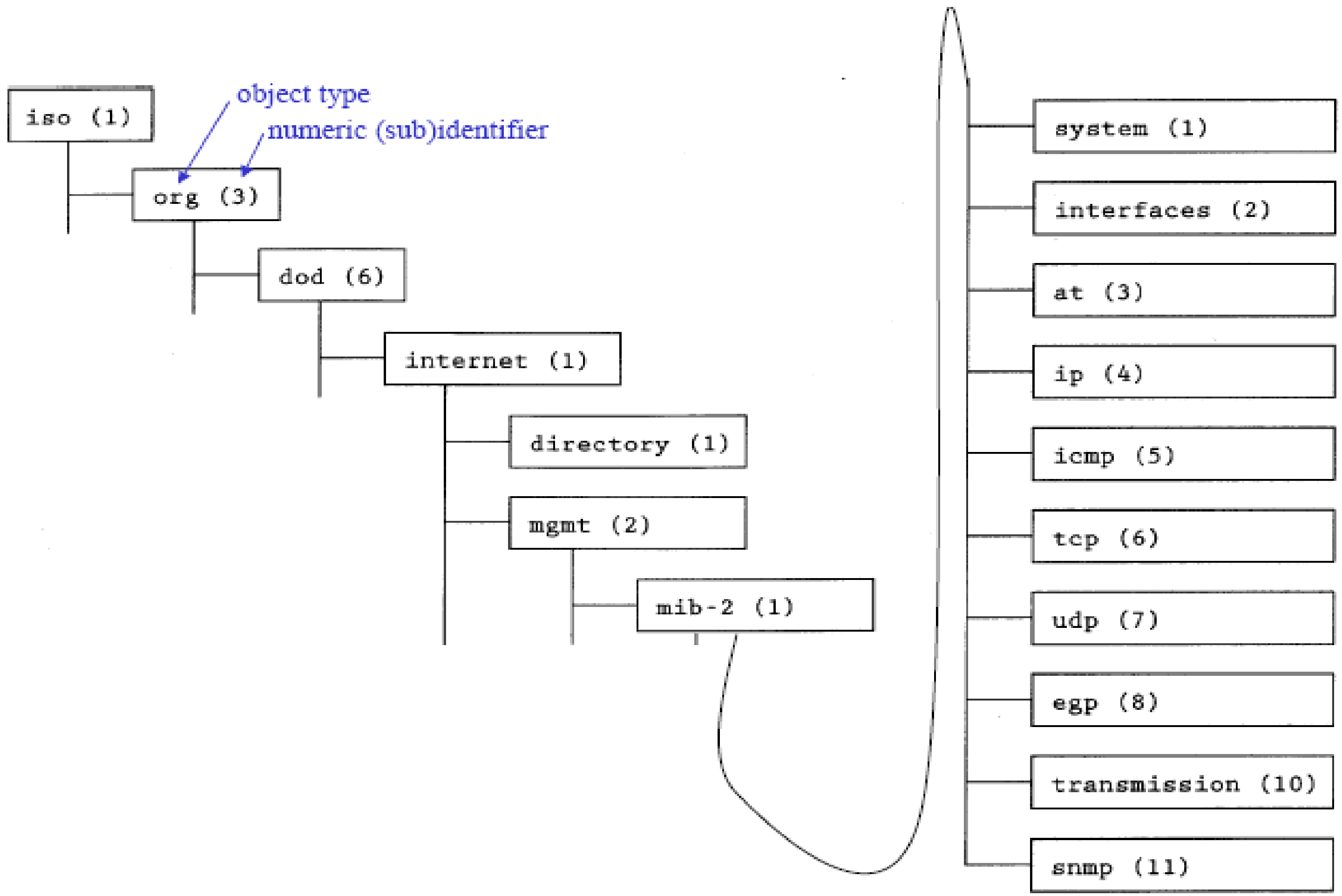
    an integer number

    a name and a number

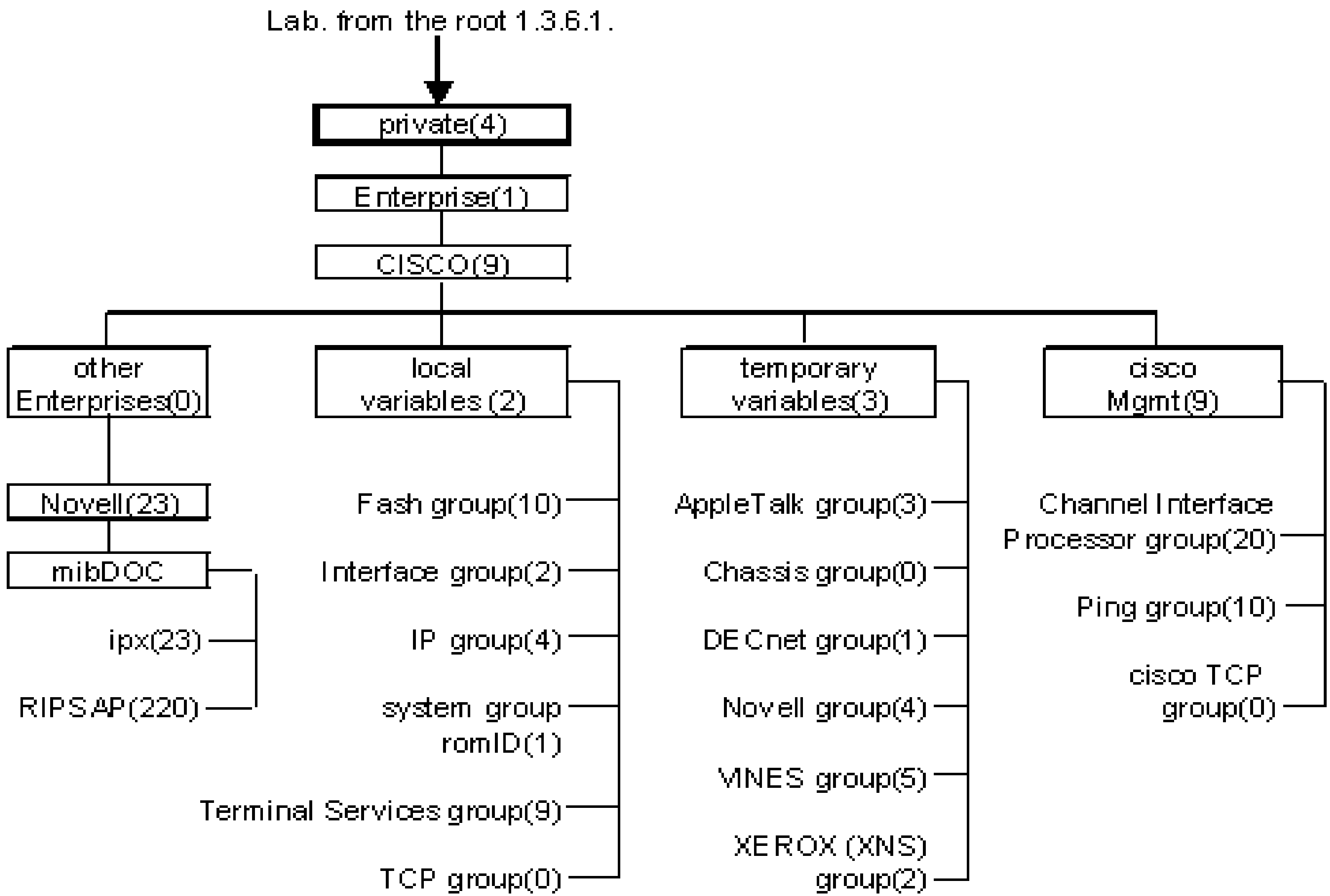
MIB


root
ccitt (0)
iso (1)
joint-iso-ccitt (2)
stnd (0)
reg-auth (1)
mb (2)
org (3)
dod (6)
internet (1)
directory (1)
mgmt (2)
experimental (3)
private (4)
security (5)
snmpV2 (6)
mib-2 (1)
enterprises (1)
snmpDomains (1)
snmpProxys (2)
snmpModules (3)

MIB-2

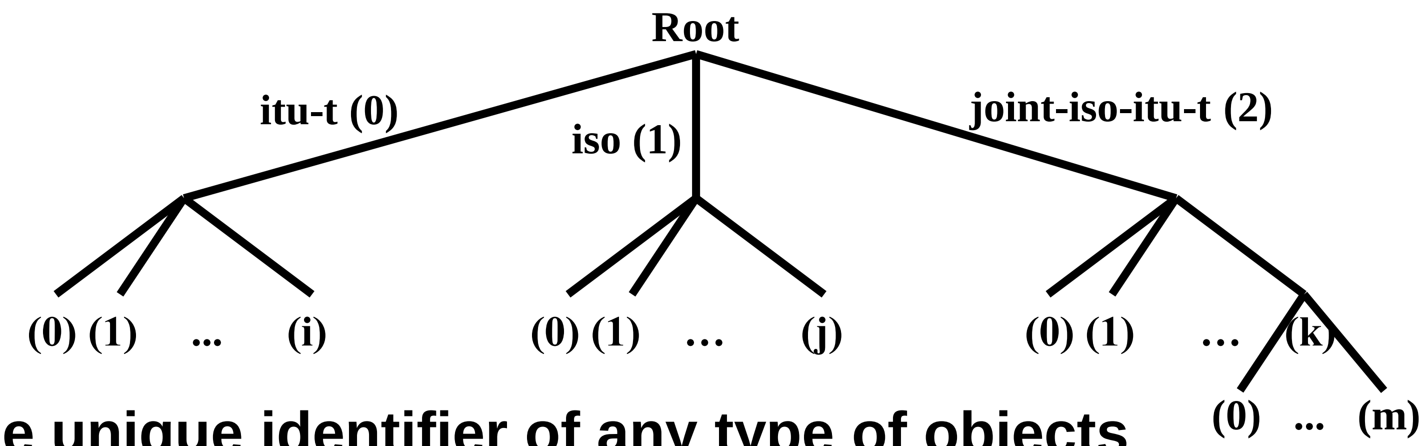


iso (1)
object type
numeric (sub)identifier
org (3)
dod (6)
internet (1)
directory (1)
mgmt (2)
mib-2 (1)
system (1)
interfaces (2)
at (3)
ip (4)
icmp (5)
tcp (6)
udp (7)
egp (8)
transmission (10)
snmp (11)



Cisco MIB rész egy eleme: 1.3.6.1.4.1.9.2.1.46 bufferFail

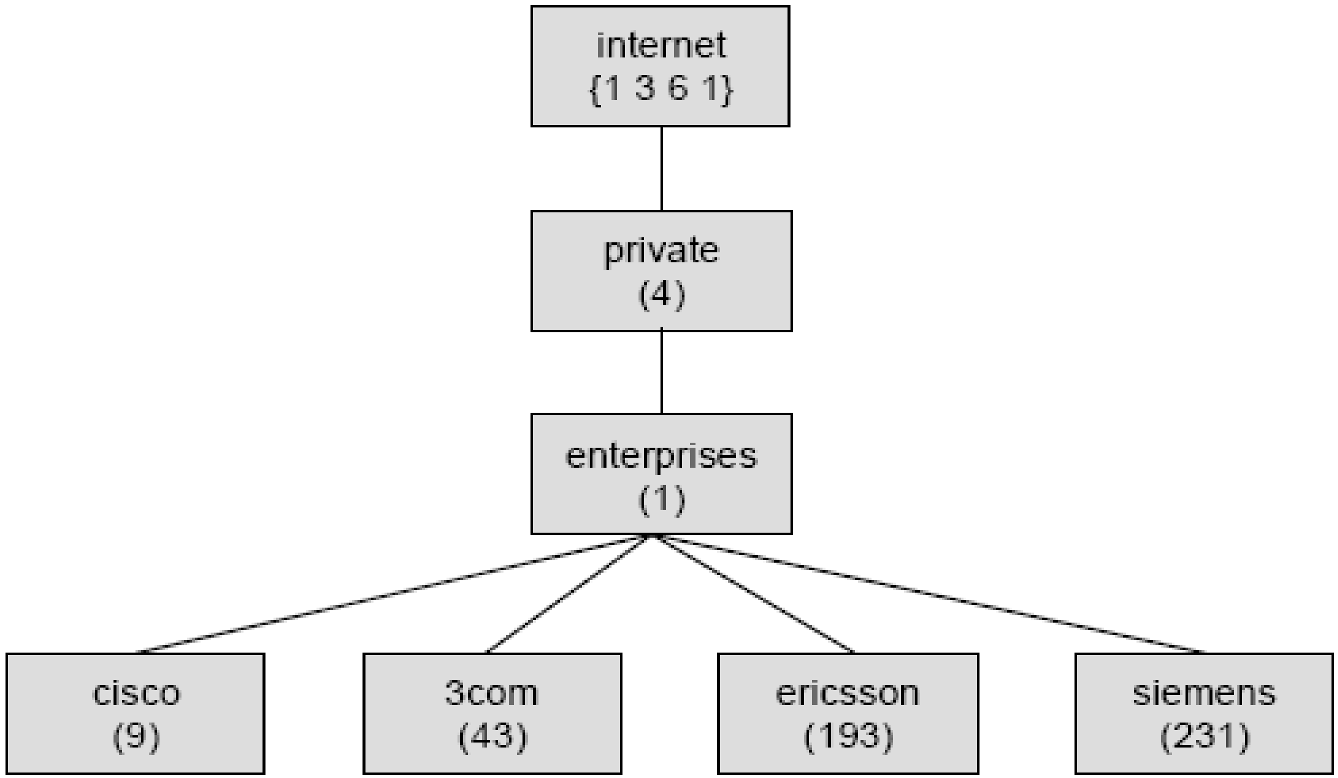
ASN.1 abstract syntax — OBJECT IDENTIFIER



- **Worldwide unique identifier of any type of objects**
  - OID contains of a series of Object ID components in curly brackets
  - A component may be:
    - a name (only for names defined in ITU-T Rec. X.660)
    - an integer number
    - a name and a number

```
ftam        OBJECT IDENTIFIER ::= { iso  standard  8571 }
ericsson    OBJECT IDENTIFIER ::= { itu-t(0) identified-organization(4)
            etsi(0) reserved(127) etsi-identified-organization(0) ericsson(5) }
```

Some examples for manufacturers



Useful object identifier values

- Names defined in ITU-T Rec. X.660:

itu-t(0)	question (1) administration (2) network-operator (3) identified-organization(4)
iso(1)	standard (0) member-body(2) identified-organization (3)
joint-iso-itu-t(2)	

- Known Ericsson identifiers:

{iso member-body bsi(826) disc(0) ericsson(1249)}

{iso identified-organization dod(6) internet(1) private (4) enterprise (1)
ericsson-Business-Communication(193) }

{itu-t identified-organization etsi(0) reserved(127)
etsi-identified-organization(0) ericsson(5) }

ASN.1 abstract syntax — Object descriptor

- **Textually identifies an object**
  - **Not a free text but a textual identifier allocated by an Authority**
  - **Not unambiguous (unlike the object identifier)**
  - **Its type definition (a graphical string with changed type ID):**

    ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString

ASN.1 abstract syntax — Restricted character string types

- Restricted character string types
  - BMPString | GeneralString | GraphicString | IA5String | ISO646String | NumericString | PrintableString | TeletexString | T61String | UniversalString | UTF8String | VideotexString | VisibleString
  - NumericString has the object identifier and object descriptor:
    { joint-iso-itu-t asn1(1) specification(0) characterStrings(1) numericString(0) }
    and
    "NumericString character abstract syntax”
  - PrintableString has the object identifier and object descriptor:
    { joint-iso-itu-t asn1(1) specification(0) characterStrings(1) printableString(1) }
    and
    ”PrintableString character abstract syntax"
  - See more details at coding!

ASN.1 abstract syntax — *Character string types*

UNIVERSAL

- **12 UTF8String = UniversalString**
- **18 NumericString** -> 0..9 (space)
- **19 PrintableString** -> A .. Z a .. z 0 .. 9 ‘ ( ) + , - . / : = ? (space)
- **20 TeletexString (T61String)** -> ITU-T Rec. T.61
- **21 VideotexString** -> ITU-T Rec. T.100 & T.101
- **22 IA5String = UniversalString (BasicLatin)** (including control characters!)
- **25 GraphicString** -> All G sets + SPACE
- **26 VisibleString (ISO646String) ->** ISO/IEC 646:1991characters
- **27 GeneralString** -> All G and C sets + SPACE + DELETE
- **28 UniversalString** -> ISO/IEC 10646-1 (all characters in the Universe)
  (character space: 128 groups/256 plane/ 256 row/256 cell = 2 147 483 648)
- **30 BMPString =** UniversalString (Bmp) (Basic Multilingual Plane)
  (all characters in all living languages; can be encoded on 16 bits!)

Note1: further details see in Table 3/X.680

Note2: A "CharacterStringList" notation can also be used for UniversalString, UTF8String, BMPString or IA5String -> {0,0,7,15}; for IA5String only also -> {0,7}

ASN.1 abstract syntax — Time types

- **GeneralizedTime**
  - Its type definition (a visible string with changed type ID):

    GeneralizedTime ::= [UNIVERSAL 24] IMPLICIT VisibleString
  - Contains calendar date, time of day (any precision) and local time differential factor according to ISO 8601
  - Formats : YYYYMMDDhhmm[ss](”Z”) | (”+|-”hhmm) |empty
    local time: “20010921214525.3” (2001. Sept. 21, 21h 45m 25.3 s)
    coordinated universal time (GMT): “20010921214525.3Z”
    local time, 5 hours retarded from GMT: “20010921214525.3-0500”
- **Universal time**
  - Its type definition (a visible string with changed type ID):

    UTCTime ::= [UNIVERSAL 23] IMPLICIT VisibleString
  - Contains calendar date, time (precision of 1 min. or 1 sec.) and local time differential factor from coordinated universal time
  - Formats: YYMMDDhhmm[ss](”Z”) | (”+|-”hhmm)
    local time, 5 hours retarded from UTC: “010921214525.3-0500”
    coordinated universal time (UTC): “010921214525.3Z”

ASN.1 abstract syntax

CONTENTS

1. Basic conventions, ASN.1 module definition
2. Built-in ASN.1 types
   NULL, BOOLEAN, INTEGER, OCTET STRING, BIT STRING, SEQUENCE (OF), SET (OF), CHOICE, Character string types, REAL, ANY, EMBEDDED PDV
3. Tagging
   Why we need?, types of tags, IMPLICIT, EXPLICIT, AUTOMATIC tagging, tagging rules

ASN.1 abstract syntax — Tagging example -3

- **The problem (again):**

Transmitter — **Receiver**

```
SET {
  toBePaid      INTEGER,
  haveMoney     BOOLEAN,
  toBeGet       INTEGER
}
```

sent:
INTEGER, BOOLEAN,
INTEGER

was the order toBePaid, haveMoney and toBeGet OR toBeGet, haveMoney and toBePaid?

- **Solution: additional label**

```
SET {
  toBePaid   [0] INTEGER,
  haveMoney      BOOLEAN,
  toBeGet        INTEGER
}
```

sent:
INTEGER, BOOLEAN,
0+INTEGER

the order was toBeGet, haveMoney and toBePaid !

ASN.1 abstract syntax — Tagging - 1

- Tagging: a transformation creating a new type from the old one
  - Commonly used: to identify a new (application or implementation specific) type, to make type components of a constructed type distinct

```
A ::= SEQUENCE { first          INTEGER  OPTIONAL,
                 second         INTEGER  OPTIONAL }
B ::= SEQUENCE { first      [0] INTEGER  OPTIONAL,
                 second         INTEGER  OPTIONAL }
```

Types not distinct

Distinct types

- Tag classes:
  - UNIVERSAL: used by the ASN.1 standard only
    --> built-in ASN.1 types
  - APPLICATION: user defined types in application standards
  - PRIVATE: vendor-specific user defined type
  - context-specific: default if no tag class defined
    --> most commonly used to make type components distinct
  - only the use of UNIVERSAL is strictly limited

ASN.1 abstract syntax — Tagging - 1

- Tagging mechanisms: implicit, explicit, automatic
  - Default for the module defined in the module header
  - In IMPLICIT tagging environment:

implicitly tagged   explicitly tagged

```
C ::= SEQUENCE { first   [0]            INTEGER  OPTIONAL,
                 second  [1] EXPLICIT   INTEGER  OPTIONAL }
```

  - In EXPLICIT tagging environment :

explicitly tagged   implicitly tagged

```
D ::= SEQUENCE { first   [0]            INTEGER  OPTIONAL,
                 second  [1] IMPLICIT   INTEGER  OPTIONAL }
```

  - If default is AUTOMATIC: components of all SEQUENCEs, SETs and CHOICEs are tagged in implicit environment with context-specific tags before encoding by the tool, except types containing even a single hand-written tag in the ASN.1 spec.

in ASN.1

```
E ::= SEQUENCE { first   INTEGER       OPTIONAL,
                 second  OCTET STRING  OPTIONAL }
```

tagged before encoding

```
E ::= SEQUENCE { first   [0]  INTEGER       OPTIONAL,
                 second  [1]  OCTET STRING  OPTIONAL }
```

ASN.1 abstract syntax | *UNIVERSAL class tags*

- **UNIVERSAL 0** *Reserved for use by the encoding rules*
- **UNIVERSAL 1** BOOLEAN
- **UNIVERSAL 2** INTEGER
- **UNIVERSAL 3** BIT STRING
- **UNIVERSAL 4** OCTET STRING
- **UNIVERSAL 5** NULL
- **UNIVERSAL 6** OBJECT IDENTIFIER
- **UNIVERSAL 7** ObjectDescriptor
- **UNIVERSAL 8** EXTERNAL | INSTANCE OF
- **UNIVERSAL 9** REAL
- **UNIVERSAL 10** ENUMERATED
- **UNIVERSAL 11** EMBEDDED PDV
- **UNIVERSAL 12** UTF8String
- **UNIVERSAL 13** Relative OID
- **UNIVERSAL 14-15** *Reserved*
- **UNIVERSAL 16** SEQUENCE | SEQUENCE OF
- **UNIVERSAL 17** SET | SET OF
- **UNIVERSAL 18-22** Character string types
- **UNIVERSAL 23** UTCTime
- **UNIVERSAL 24** GeneralizedTime
- **UNIVERSAL 25-28** Character string types
- **UNIVERSAL 29** CHARACTER STRING
- **UNIVERSAL 30** BMPString
- **UNIVERSAL 31-...** *Reserved*

ASN.1 abstract syntax — Character string types

UNIVERSAL

- **12 UTF8String = UniversalString**
- **18 NumericString** -> 0..9 (space)
- **19 PrintableString** -> A .. Z a .. z 0 .. 9 ‘ ( ) + , - . / : = ? (space)
- **20 TeletexString (T61String)** -> ITU-T Rec. T.61
- **21 VideotexString** -> ITU-T Rec. T.100 & T.101
- **22 IA5String = UniversalString (BasicLatin)** (including control characters!)
- **25 GraphicString** -> All G sets + SPACE
- **26 VisibleString (ISO646String) ->** ISO/IEC 646:1991characters
- **27 GeneralString** -> All G and C sets + SPACE + DELETE
- **28 UniversalString** -> ISO/IEC 10646-1 (all characters in the Universe)
  (character space: 128 groups/256 plane/ 256 row/256 cell = 2 147 483 648)
- **30 BMPString =** UniversalString (Bmp) (Basic Multilingual Plane)
  (all characters in all living languages; can be encoded on 16 bits!)

Note1: further details see in Table 3/X.680

Note2: A "CharacterStringList" notation can also be used for UniversalString, UTF8String, BMPString or IA5String -> {0,0,7,15}; for IA5String only also -> {0,7}

ASN.1 abstract syntax — *Tagging rules*

- The tag applied to a CHOICE or an open type always shall be explicit

```
F ::=  SEQUENCE {  first    [0]            INTEGER                          OPTIONAL,
                   second   [1] EXPLICIT   CHOICE { alt1  OCTET STRING }  }
```

- SEQUENCE: tagging rules for COMPONENTS OF:
  - Automatic tagging applied only if NONE of the elements in the referencing type (containing the COMPONENTS OF notation) is tagged
  - Decision on tagging is made BEFORE the COMPONENTS OF transformation
  - Automatic tagging is carried out AFTER the transformation
  - During automatic tagging tags within the referenced SEQUENCE are suppressed (i.e. once automatic tagging is decided, it is done independently of tags within the parent type)

ASN.1 abstract syntax — *Module definition*

Example **(fields are separeted by white-space)**

Module-Name { <optional object identifier> } — *-- name is mandatory*

DEFINITIONS — -- mandatory keyword

EXPLICIT TAGS -- or
IMPLICIT TAGS -- or
AUTOMATIC TAGS

optional (see later)

EXTENSIBILITY IMPLIED — *-- optional*

::= — -- mandatory keyword

BEGIN — -- mandatory keyword

EXPORTS *<comma separated list>* **;** — -- optional

IMPORTS *<comma separated list>* **FROM** *<module identifier>*
<comma separated list> **FROM** *<module identifier>***;** — -- optional

<assignments> — *-- definitions in the module body*

END

ASN.1 abstract syntax

CONTENTS

ASN.1 abstract syntax — Subtyping: features

- Creates a new type from the parent type!

  Parent ::=INTEGER (0..31)   NewOne ::= Parent (0..15)

- May be an expression using set aritmetics
  - INTERSECTION (^), UNION (|) & EXCEPT

  Expr1 ::= INTEGER (ALL EXCEPT (0..15))   Expr2 ::= INTEGER (0..1|4..6)

- May contain extension markers (...) and exceptions(!)

  Values ::= INTEGER (0..15, … !1)   Values ::= INTEGER (0..15, …, 0..31 !1)

- Very often parameterized

  ThisIsIt {INTEGER:maxIterations, INTEGER:maxNumber} ::=
      SEQUENCE (SIZE (0.. maxIterations)) OF INTEGER (0.. maxNumber)

ASN.1 abstract syntax — Subtyping - 1

- **Single value constraint** (applicability)

  Zero ::= INTEGER (0)   Yes-No ::= IA5String ("Yes" |"No")

- **Value range constraint** (applicability)

  Seven-bit-range ::= INTEGER (0..127)

  PI ::= REAL ({mantissa 3141, base 10, exponent -3} ..
  {mantissa 3142, base 10, exponent -3})

- **Permitted alphabet** (applicability)
  - applies to restricted character string types only (except CHARACTER STRING!)

  CapitalA ::= IA5String (FROM ("A"))
  String1 ::= IA5String (FROM ("A".. "Z") UNION FROM ("a".. ”z”))
  String2 ::= IA5String (FROM ("A".. "Z" UNION "a".. "z"))

  String1 = String2 ???   String1 $\neq$ String2 !!

  String1: strings containing all capital or all small letters only,
  String 2: capital and small letters can be mixed within a string

ASN.1 abstract syntax — Subtyping - 2

- Size constraint (applicability)
  - applies to BIT STRING, OCTET STRING, SET OF, SEQUENCE OF and character string types (including CHARACTER STRING)

    String-sequence ::= SEQUENCE (SIZE(1..5)) OF IA5String (SIZE(10))
- Contained sub-type constraint (applicability)
  - uses another ASN.1 type(s) to create the subtype

    Subtype1 ::= INTEGER (0..63) Subtype2 ::= INTEGER (16..31)
    NewSubtype ::= INTEGER (Subtype1 EXCEPT Subtype2)
- Selection type
  - Applicable to CHOICE only
  - Restricts type to a single choice (in the standard -> type notation)
  - This single choice shall be referenced in value notations!

    ParentChoice ::= CHOICE { otherCase INTEGER, thisCase BIT STRING }
    NewChoice ::= thisCase < ParentChoice
    myValue NewChoice ::= ‘01010101’B

ASN.1 abstract syntax — Subtyping - 3

- Inner subtyping (applicability)
  - The exception: does NOT create a new type but resctricts values allowed to be used => tagged and coded exactly as the parent type !
  - Both allowed value range and presence may be constrained

```
Parent-type  ::=  SEQUENCE {  flag       BOOLEAN              OPTIONAL,
                              complex    ParentNew            OPTIONAL,
                              value      INTEGER              OPTIONAL,
                              values     SEQUENCE OF INTEGER           }
ParentNew    ::=  SEQUENCE {  number     INTEGER (0..32)      OPTIONAL,
                              flag2      BOOLEAN              OPTIONAL  }

Born-type    ::=  Parent-type ( WITH COMPONENTS {
                              flag                        ABSENT,
                              complex    (BornNew)        PRESENT,
                              value                       OPTIONAL,
                              values     (SIZE(0..5))
                                         (WITH COMPONENT ( 0..63 ) ) } )
BornNew      ::=  ParentNew  ( WITH COMPONENTS { number ( 0 ..7 ) } )
```

ASN.1 abstract syntax — Subtyping - 4

- **Inner subtyping - specification for SET and SEQUENCE (cont)**

```
Parent-type ::= SEQUENCE { flag     BOOLEAN              OPTIONAL,
                           complex  Parent2              OPTIONAL,
                           value    INTEGER              OPTIONAL,
                           values   SEQUENCE OF INTEGER           }
```

– **Full specification**

```
Born-1 ::= Parent-type ( WITH COMPONENTS {
                      mandatory  -->  flag                                    ,
 "complex" shall be ABSENT       -->
 constrained & remains optional  -->  value   ( 0 .. 65535 )  OPTIONAL ,
                      mandatory  -->  values                  PRESENT     } )
```

– **Partial specification**

```
Born-2 ::= Parent-type ( WITH COMPONENTS {
        keyword for partial spec.           -->  ... ,
 'flag" & "complex" remain OPTIONAL          -->
 constrained & remains optional             -->  value   ( 0 .. 65535 )           ,
                      mandatory             -->  values                  PRESENT  } )
```

ASN.1 abstract syntax — Subtyping - 5

- Inner subtyping - specification for CHOICE (cont)

```
Parent-type ::= CHOICE { flag      BOOLEAN,
                         complex   Parent2,
                         value     INTEGER,
                         values    SEQUENCE OF INTEGER }
```

– Full specification

```
Born-3 ::= Parent-type ( WITH COMPONENTS {
        flag ,

        value    ( 0 .. 65535 )  } )
```

may be chosen → flag

"complex" & "values" shall never be chosen →

constrained & may be chosen → value

```
Born-4 ::= Parent-type ( WITH COMPONENTS {
        values    PRESENT  } )
```

the only choice → values

– Partial specification

"value" is the only legal choice

```
Born-5 ::= Parent-type ( WITH COMPONENTS { ... , value PRESENT } )
```

"value" shall never be chosen

```
Born-6 ::= Parent-type ( WITH COMPONENTS { ... , value ABSENT } )
```

- **Type Constraint** [(applicability)]
  - Constraints an open type to an ASN.1 type

open type:
a set of
all ASN.1
types

type constraint → **ENUMERATED**

Example

notation for an open type | type constraint

MY-CLASS-2.&Argument (INTEGER)

ASN.1 abstract syntax — Applicability of subtypes

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

ASN.1 abstract syntax — Content constraint

- Restricts the content of the parent type
  - May apply to BIT STRING & OCTET STRING only
  - The octet- or bit string contains an encoded abstract value;
    the encoding rule always defined;
    specifying the abstract value is not mandatory
  - When used this may be the only type constraint applied

```
ConstrainedType3  ::=  OCTET STRING
                       ( CONTAINING  EmbeddedType
                         ENCODED BY { joint-iso-itu-t asn (1) basic-encoding (1) } )

ConstrainedType4 ::= OCTET STRING
                       ( CONTAINING  EmbeddedType
                         -- encoded by the same rule as the whole  module-- )

ConstrainedType5 ::= OCTET STRING ~~( SIZE ( 0 .. maxLength )~~
                       ( ENCODED BY { joint-iso-itu-t asn (1) basic-encoding (1) }
                         -- practically a comment describes the abstract value
                         -- to be encoded into the parent type --              )
```

--> invalid as no another constraint may be applied to a content constrained type

ASN.1 abstract syntax

CONTENTS

1. Basic conventions
2. Built-in ASN.1 types
   NULL, BOOLEAN, INTEGER, OCTET STRING, BIT STRING, SEQUENCE (OF), SET (OF), CHOICE, Character string types, REAL, ANY, EMBEDDED PDV
3. Tagging
   Why we need?, types of tags, IMPLICIT, EXPLICIT, AUTOMATIC tagging, tagging rules
4. Subtyping
   General, single value, value range, size constraint, type constraint, permitted alphabet, contained subtype, inner subtyping
5. Extensibility
   ASN.1 model of extensions, extension marker, exception identifier, extensibility rules, version brackets

ASN.1 abstract syntax — Extensibility - general

- The only legal way to add elements to defined types LATER
- Any unknown element at an insertion point shall NOT be treated as an error by the ASN.1 decoder!!!
- Extensibility implied
  - implicitly for all extensible types by the module header
  - explicitly including an extension marker at the insertion point
- What can be extended
  - TYPES: CHOICE, SEQUENCE, SET, ENUMERATED
  - Type CONSTRAINTS (see later) : single value, value range, size constraint and permitted alphabet

ASN.1 abstract syntax — Exception handling

- Instructs the APPLICATION!, how to handle
  - type constraint violation (see later)
  - unknown elements at an extension insertion point
- Consist of:

  an exception identifier(!) and

  an exception handling rule optional Type:value

Examples

A ::= SEQUENCE (SIZE (0 .. 15 !1)) OF INTEGER (0 .. 65535 !2)

INTEGER is assumed

......

-- Error handling rules: 1 - ignore additional elements ; 2 - relay value ....

B ::= SEQUENCE (a A,

b B,

... ! IA5String:"not comprehended element received")

C ::= SEQUENCE (SIZE (0 .. 15 !)) OF INTEGER

--> comment in ASN.1 or specified in the procedures description or

--> take implementation-dependent action

Extensibility rules -1

- **ASN.1 extensibility rules apply**
  - **Single value, value range, size constraint:**

Original X.680(12/97): **ALL (root and additional) elements are used in set aritmetics:**

INTEGER ((0..15, ..., 32..63) | (16..31, ...))

--> range 0..63, type extensibility is not defined clearly!

Corrigendum 1 (06/99): **in set aritmetic ONLY the root values of components are used => extensible ONLY if there is an ellipsis at the outmost level**

INTEGER ((0..15, ..., 32..63) | (16..31, ...)) --> inextensible, range 0..31
INTEGER ((0..31, ..., 32..63) ^ (0..63, ...)) --> inextensible, range 0..31
INTEGER ((0..15) | (16..31), ...) --> extensible type, range 0..31

This is the only LEGAL understanding for ALL ASN.1'97 compliant specs.!

BUT the X.680 (12/97) rule is followed by the OSS syntax checker v5.0.4, so correct behaviour always to be checked when a 3rd party tool is used!

ASN.1 abstract syntax — Extensibility rules -2

ASN.1 extensibility rules (cont.)

- If an extensible type is further constrained, the new type does NOT inherit extensibility (for extensibility ellipsis shall explicitly be included)

```
A  ::=  INTEGER ( 0..63, ... )       --> extensible (parent) type
B  ::=  A ( 0..31)                   --> inextensible, range 0..31
C  ::=  A ( 0..31, ... )             --> extensible, range 0..31
D  ::=  A                            --> extensible, range 0..63
```

- Constrained subtype: ONLY root values are used => the new type does NOT inherit extensibility

```
E  ::=  INTEGER ( A )                --> inextensible, range 0..63
F  ::=  INTEGER ( A, ... )           --> extensible, range 0..63
```

- Inner subtyping: has no effect (the new type is extensible if and only if the parent type is extensible)

```
Parent-type  ::= SEQUENCE {  value    INTEGER       OPTIONAL,
                             values   SEQUENCE OF INTEGER, ...  }
Born-type    ::= Parent-type  ( WITH COMPONENTS { value   PRESENT } )
```

=> Born-type inherits extensibility of Parent-type

ASN.1 abstract syntax — When can a type be extended ?

- Checking extensibility with conceptual elements
  - Add a conceptual element at each insertion point
  - Its tag is distinct from any tag in the root (after the COMPONENTS OF transformation and applying automatic tags)
  - But matches the tag of other conceptual elements
  - Requirements for distinct tags may not be violated

Example

```
Ex-1 ::=  SEQUENCE  {  field1      CHOICE {
                                         alt1     INTEGER,
                                         ... ! 1
                                         A
                                                          } OPTIONAL,
                       ... ! 2
                       B                                          }
```

conceptual elements

--> illegal: a version 1 system can not decide if the new element inserted at point 1 (and chosen) or at point 2 (and field1 is missing)

ASN.1 abstract syntax — *Parameterization*

- **Parameterization - general**

```
FirstType { INTEGER: param-value } ::=
        SEQUENCE {
            first     INTEGER       DEFAULT  param-value,
            second    SecondType             { TRUE, 5 },
            third     ThirdType { param-value }
        }
```

"IN"
=> formal parameters

"OUT"
=> actual parameters

```
SecondType { BOOLEAN: paramLOG, INTEGER: paramINT } ::=
        CHOICE {
            altern1          BOOLEAN ( paramLOG ),
            altern2          INTEGER  ( paramINT )
        }
```

Governor   dummy reference

ASN.1 abstract syntax — Parameterization

- **Parameterization of type notations**
  - Formal parameter may be:

```
type:        { DummyType },
value:       { Type: dummyvalue }                or
             { DummyType: dummyvalue }
valueset:    { Type: Dummyvalueset }             or
             { DummyType: Dummyvalueset }
```

```
Sample1 { Par-Type } ::= CHOICE {    altern1        BOOLEAN,
                                     altern2        Par-Type        }

Sample2 { INTEGER: par-value } ::= SEQUENCE {
                                     first    INTEGER  DEFAULT  par-value }

Sample3 { ParType, ParType: parValue } ::= SEQUENCE {
                                     seethis        ParType ( parValue )   }

Names { IA5String: ExtraNames } IA5String ::= { "Johny" | "Freddy" | ExtraNames }
```

for the actual parameter

```
=>    ToBeInvited  IA5String  ::=  { Names { { "Jill" | "Mary" | "Sue"} } }
```

for value set notation

Rules of parameterization - 1

- Each dummy reference shall be used

  Sample1 { Par-Type } ::= CHOICE { altern1 BOOLEAN,
  altern2 INTEGER } --> invalid

- Defined type/value shall not be just the dummy reference

  Sample2 { Par-Type } ::= ~~Par-Type~~ --> invalid

- If the formal parameter is a type or value set, it shall not be passed as a tagged type to a circular reference

  Sample3 { Par-Type } ::= SEQUENCE { a INTEGER,
  b Sample3 { ~~[0]~~ Par-Type } } --> invalid

- A dummy type reference shall not refer to a dummy reference, which has a governor

  Sample4 { INTEGER: Par-valueSet, ~~Par-valueSet:~~ parvalue } ::= --> invalid
  -- whatever type assignment follows --

  Sample5 { Par-Type , Par-Type: parvalue } ::= --> valid
  -- whatever type assignment follows --

Rules of parameterization - 2

- Circular reference (direct or indirect !) is not allowed except for dummy types if the reference is OPTIONAL or a CHOICE, where a non-circular alternative exists

Sample6 { INTEGER: parValue } ::=
SEQUENCE { a INTEGER,
b ~~Sample6 { parValue }~~ OPTIONAL } --> invalid

Sample7 { ParType } ::=
SEQUENCE { a INTEGER,
b Sample7 { ParType } } --> invalid

Sample8 { ParType } ::=
SEQUENCE { a INTEGER,
b Sample8 { ParType } OPTIONAL} --> valid

Sample9 { ParType } ::=
SEQUENCE { a INTEGER,
b MyChoice { ParType } }

MyChoice {ParMyType} ::=
CHOICE { first INTEGER,
my Sample7 { ParType } } --> valid