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

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

Manual encoding of protocol elements & procedures

Protocol specification

textual/table specification (rarely SDL/MSC)

Manual encoding of procedures

Automatic encoding of ASN.1 elements by a generic purpose ASN.1 encoder
May be 3rd party implementation

Protocol implementation

ASN.1 abstract syntax

ASN.1 transfer syntax
Structure of ASN.1

Application level

XML

XML -> ASN.1 mapping

Basic notation, Information Objects

Encoding Control Notation (ECN)

XML Control Notation (XCN)

ASN.1 abstract syntax level

ASN.1 transfer syntax level

Basic Encoding Rules (BER)

Packed Encoding Rules, basic (PER)

ECN-enabl. or proprietary encoder

XML Encoding Rules (XER)

XML data

User defined encoded data

PER encoded data

BER/CER/DER encoded data

ECN-encoding rules
ASN.1 - 1997 (multipart) version - 1

X.680

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

WILL BE SUPERSEDED SOON!
**ASN.1 - 1997 (multipart) version - 2**

**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

WILL BE SUPERSEEDED SOON!
Recommended reading

  - Downloadable PDF version: [http://www.oss.com/asn1/larmouth.html](http://www.oss.com/asn1/larmouth.html)

- Olivier Dubuisson, *ASN.1 - Communication between heterogeneous systems*, Elsevier-Morgan Kaufmann, 2000
ASN.1 abstract syntax

CONTENTS

1. Basic conventions

2. Built-in ASN.1 types (basics of tagging and extensibility)
   NULL, BOOLEAN, INTEGER, BIT STRING, OCTET STRING,
   ENUMERATED, SEQUENCE (OF), SET (OF), CHOICE, Character string
   types, time types, REAL, ANY & other hole types: EMBEDDED PDV,
   EXTERNAL, Unrestricted character string

3. Tagging - additions
   Types of tags, IMPLICIT, EXPLICIT, AUTOMATIC tagging, tagging rules

4. Subtyping & constraints
   Single value, value range, size constraint, type constraint, permitted
   alphabet, contained subtype, inner subtyping

5. Extensibility - further rules
   ASN.1 model of extensions, extension marker, exception identifier,
   extensibility rules, version brackets
**ASN.1 abstract syntax**

**Naming conventions**

- **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
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

- **ABSENT**
- **ABSTRACT-SYNTAX**
- **ALL**
- **APPLICATION**
- **AUTOMATIC**
- **BEGIN**
- **BIT**
- **BMPString**
- **BOOLEAN**
- **BY**
- **CHARACTER**
- **CHOICE**
- **CLASS**
- **COMPONENT**
- **COMPONENTS**
- **CONSTRAINED**
- **CONTAINING**
- **DEFAULT**
- **DEFINITIONS**
- **EMBEDDED**
- **EMBEDDED**
- **ENCODING**
- **END**
- **ENUMERATED**
- **EXCEPT**
- **EXPLICIT**
- **EXPORTS**
- **EXTERNAL**
- **FALSE**
- **FROM**
- **GeneralizedTime**
- **GeneralString**
- **GraphicString**
- **IA5String**
- **IDENTIFIER**
- **IMPLIED**
- **IMPLICIT**
- **IMPORTS**
- **INCLUDES**
- **IMPLIED**
- **IN\-**
- **INSTANCE**
- **INTEGER**
- **INTERSECTION**
- **ISO646String**
- **MAX**
- **MIN**
- **MINUS-INFINITY**
- **NULL**
- **NumericString**
- **ObjectDescriptor**
- **OCTET**
- **OF**
- **OPTIONAL**
- **PATTERN**
- **PDV**
- **PLUS-INFINITY**
- **PRESENT**
- **PRINTABLESTRING**
- **PRIVATE**
- **REAL**
- **RELATIVE-OID**
- **SEQUENCE**
- **SET**
- **SIZE**
- **STRING**
- **SYNTAX**
- **T61String**
- **TAGS**
- **TeletexString**
- **TRUE**
- **TYPE-IDENTIFIER**
- **UNION**
- **UNIQUE**
- **UNIVERSAL**
- **UniversalString**
- **UTCTime**
- **UTF8String**
- **VideotexString**
- **VISIBLESTRING**
- **WITH**
### ASN.1 abstract syntax

**Module definition**

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

```
Module-Name { <optional object identifier> }  -- name is mandatory
DEFINITIONS

{  EXPPLICIT TAGS  -- or
  IMPLICIT TAGS  -- or
  AUTOMATIC TAGS

EXTENSIBILITY IMPLIED
  ::=  -- mandatory keyword
BEGIN
  EXPORTS  <comma separated list> ;  -- optional
  IMPORTS  <comma separated list> FROM <module identifier>  -- optional
    <comma separated list> FROM <module identifier>;
  <assignments>  -- definitions in the module body

END
```
**ASN.1 abstract syntax**

- **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**
  - 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**
  - 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**

- **Sequence of individual bits**
  
  \[
  b1 \quad \text{BIT STRING} ::= \text{‘000101110011010’B} \\
  b2 \quad \text{BIT STRING} ::= \text{‘F0F0’H} \quad \rightarrow \quad \text{‘111000011110000’B}
  \]

- **Named bit list**

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

  \[
  B1 ::= \text{BIT STRING }\{ \text{first (0), second (1), trailing (6) }\} \\
  b3 \quad B1 ::= \quad \text{‘001001100011’B}
  \]

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

  \[
  b4 \quad B1 ::= \quad \{ \text{first, trailing} \} \quad \rightarrow \quad \text{‘1000001’B} \\
  \rightarrow \quad \text{‘10000100’B} \\
  \rightarrow \quad \text{‘1000001000’B} \\
  b5 \quad B1 ::= \quad \{ \} \quad \rightarrow \quad \text{any number of ‘0’s} \\
  b6 \quad B1 ::= \quad \text{‘AA’H} \quad \rightarrow \quad \text{‘10101010’B}
  \]
**ASN.1 abstract syntax**

**OCTET STRING**

- Sequence of individual octets
  - value notation using hstring
    - missing hex character is interpreted as zero
    
    \[ o1 \text{ OCTET STRING } ::= \text{ ‘278EF’H } \quad \text{--> ‘278EF0’ H } \]

- value notation using bstring
  - missing bits are interpreted as zero
  
  \[ o2 \text{ OCTET STRING } ::= \text{ ‘0001011011101’B } \quad \text{--> ‘16E8’H } \]
**ASN.1 abstract syntax**

- **ENUMERATED**: List of named items
  - `E1 ::= ENUMERATED { first, second, third, fourth, fifth }
  - 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) }
    - Numeric value is NOT ALLOWED to be referenced within ASN.1!
  - `e1 E1 ::= first` -> valid
  - `e2 E1 ::= 0` -> invalid

> **==>>** from ASN.1 abstract syntax point of view has no numeric value!
The problem:

Transmitter (version 2)

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

Receiver (version 1)

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

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)

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**
  
  \[ \text{A} ::= \text{ENUMERATED} \{ a, b, \ldots, c(0) \} \quad \text{--> invalid as } a = c \]
  
  \[ \text{B} ::= \text{ENUMERATED} \{ a, b, \ldots, c(2) \} \quad \text{--> valid} \]

- Values of additional enumerations shall monotonously **increase**
  
  \[ \text{C} ::= \text{ENUMERATED} \{ a, b, \ldots, c, d(2) \} \quad \text{--> invalid as } c = d \]
  
  \[ \text{D} ::= \text{ENUMERATED} \{ a, b, \ldots, c(3), d(2) \} \quad \text{--> invalid, it should be } c < d \]
  
  \[ \text{E} ::= \text{ENUMERATED} \{ a, b, \ldots, c(2), d(3) \} \quad \text{--> valid} \]

- First additional “automatic” value will be the **smallest not used** value in the root
  
  \[ \text{F} ::= \text{ENUMERATED} \{ a, b, c(0), \ldots, d, e \} \quad \text{--> } d=3, e=4 (c=0, a=1, b=2) \]
  
  \[ \text{G} ::= \text{ENUMERATED} \{ a, b, c(10), \ldots, d, e \} \quad \text{--> } d=2, e=3 (a=0, b=1, c=10) \]
  
  \[ \text{H} ::= \text{ENUMERATED} \{ a, b, c(0), \ldots, d, e, f(4) \} \quad \text{--> invalid as } e = f \]

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

\[
S1 ::= \text{SEQUENCE} \{ \text{first INTEGER, second BIT STRING, third OCTET STRING OPTIONAL, fourth BOOLEAN DEFAULT TRUE} \}
\]

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

\[
S2 ::= \text{SEQUENCE} \{ \}
\]
**The problem:**

Transmitter

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

Receiver

```
Is yourIncome OR yourDebit received?
```

```
sent: INTEGER:5000, BOOLEAN:FALSE
```

**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!
```
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 INTEGER OPTIONAL [0],
  fourth OCTET STRING
}  --> valid

H ::= SEQUENCE {
  first INTEGER,
  second INTEGER,
  third INTEGER OPTIONAL [0],
  fourth INTEGER [1]
}  --> 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   }
The general case: type extensible at the middle of root components

Example:

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

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

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

--> type added in version 2

--> type added in version 2

--> type added in version 3

--> type extended in versions 2 and 3
- All additional types up to ad including the first mandatory root component shall have distinct tags

Example (non-automatic tagging is supposed!)

```asn1
definition
J := SEQUENCE {
    a  INTEGER,
    b  INTEGER,
    ... !1,
    d  INTEGER OPTIONAL,
    e  OCTET STRING,
    f  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!)

\[
\begin{align*}
K & ::= \text{SEQUENCE} \{ \\
   & \quad a \quad \text{INTEGER}, \\
   & \quad b \quad \text{INTEGER}, \\
   & \quad \ldots \ !1, \\
   & \quad d \quad [0] \quad \text{INTEGER}, \\
   & \quad e \quad [1] \quad \text{OCTET STRING}, \\
   & \quad f \quad [2] \quad \text{INTEGER}, \\
   & \quad \ldots \ , \\
   & \quad c1 \quad \text{INTEGER OPTIONAL}, \\
   & \quad c2 \quad \text{OCTET STRING} \\
\}
\end{align*}
\]
**ASN.1 abstract syntax**

**Extension of a SEQUENCE**

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

  \[ J ::= \text{SEQUENCE} \{ \text{a A, b B, ... !1, ... } \} \]

  - insertion point: identified by the 1st ellipsis
  - "end of extensions" marker is optional

- **Type extensible at the beginning**

  \[ K ::= \text{SEQUENCE} \{ \text{... !1, ... , a A, b B} \} \]

  - insertion point: identified by the 1st ellipsis
  - root components
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 }
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**

- 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
  }
  ```

- “end of extensions” marker is optional
### The problem:

Transmitter

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

Receiver

Is yourIncome OR yourDebit received?

---

### Solution: add a distinguishing label

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

Sent:

```
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

<table>
<thead>
<tr>
<th>CHOICE</th>
<th>first</th>
<th>second</th>
<th>INTEGER, INTEGER</th>
<th>--&gt; invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>L ::=</td>
<td>CHOICE</td>
<td>{</td>
<td>first</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>second</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER, INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>--&gt; invalid</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHOICE</th>
<th>first</th>
<th>second</th>
<th>INTEGER, INTEGER</th>
<th>--&gt; valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>J ::=</td>
<td>CHOICE</td>
<td>{</td>
<td>first</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>second [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTEGER, INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>--&gt; valid</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHOICE</th>
<th>first</th>
<th>second</th>
<th>INTEGER, FirstChoice</th>
<th>--&gt; invalid (see tags first &amp; a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K ::=</td>
<td>CHOICE</td>
<td>{</td>
<td>first</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>second [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FirstChoice</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>--&gt; invalid</td>
<td>(see tags first &amp; a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHOICE</th>
<th>first</th>
<th>second</th>
<th>FirstChoice, SecondChoice</th>
<th>--&gt; invalid (see tags a &amp; c and also tags b &amp; d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L ::=</td>
<td>CHOICE</td>
<td>{</td>
<td>first</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>second</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FirstChoice</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SecondChoice</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>--&gt; invalid</td>
<td>(see tags a &amp; c and also tags b &amp; d)</td>
</tr>
</tbody>
</table>

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

SecondChoice ::= CHOICE { c [0] INTEGER, d [1] INTEGER } --> valid
All alternatives (root and additional) shall have distinct tags:

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

- 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 alternative

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

where

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

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

This whole denotes the type OCTET STRING

Identifier of the selected alternative is not present!
**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)

\[ S1 ::= \text{SEQUENCE OF AnotherType} \]

where \( \text{AnotherType} ::= \text{INTEGER} \quad \text{-- or another definition} \)

\[ \text{AnotherType} ::= \text{CHOICE} \{ \text{first} : \text{INTEGER}, \quad \text{second} : \text{BIT STRING}, \quad \text{third} : \text{OCTET STRING}, \quad \text{fourth} : \text{BOOLEAN} \} \]

valueRef-1 \( S1 ::= \{ 0, 1, 15, 7 \} \quad \text{-- when AnotherType is integer} \)

valueRef-2 \( S1 ::= \{ \text{first} : 1, \text{third} : \text{'0F'H} \} \quad \text{-- when AnotherType is choice} \)
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
Cisco MIB rész egy eleme: 1.3.6.1.4.1.9.2.1.46 bufferFail
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 \quad \text{OBJECT IDENTIFIER} \ ::= \ \{ \text{iso} \ \text{standard} \ 8571 \ \}

ericsson \quad \text{OBJECT IDENTIFIER} \ ::= \ \{ \text{itu-t}(0) \ \text{identified-organization}(4) \ 
etsi(0) \ \text{reserved}(127) \ \text{etsi-identified-organization}(0) \ \text{ericsson}(5) \ \}$
Some examples for manufacturers

internet
{1 3 6 1}

private
(4)

enterprises
(1)

cisco
(9)

3com
(43)

ericsson
(193)

siemens
(231)
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) }
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
- 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: YYYYMMDDhhmm[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

- **The problem (again):**
  - Transmitter

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

- **Solution: additional label**
  - Transmitter

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

- **Receiver**
  - was the order `toBePaid, haveMoney and toBeGet` OR `toBeGet, haveMoney and toBePaid`?

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

  - the order was `toBeGet, haveMoney and toBePaid`!
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 }

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
Tagging mechanisms: implicit, explicit, automatic

- Default for the module defined in the module header
- In IMPLICIT tagging environment:
  \[ C ::= \text{SEQUENCE} \{ \text{first} \ [0] \text{INTEGER} \text{ OPTIONAL,} \]
  \[ \text{second} \ [1] \text{EXPLICIT INTEGER} \text{ OPTIONAL} \} \]

- In EXPLICIT tagging environment:
  \[ D ::= \text{SEQUENCE} \{ \text{first} \ [0] \text{INTEGER} \text{ OPTIONAL,} \]
  \[ \text{second} \ [1] \text{IMPLICIT INTEGER} \text{ OPTIONAL} \} \]

- 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
\[ E ::= \text{SEQUENCE} \{ \text{first} \text{ INTEGER} \text{ OPTIONAL,} \]
\[ \text{second} \text{ OCTET STRING} \text{ OPTIONAL} \} \]
tagged before encoding
\[ E ::= \text{SEQUENCE} \{ \text{first} \ [0] \text{INTEGER} \text{ OPTIONAL,} \]
\[ \text{second} \ [1] \text{OCTET STRING} \text{ OPTIONAL} \} \]
### ASN.1 abstract syntax

| 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 class tags

| 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 646:1991 characters
- **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}
The tag applied to a CHOICE or an open type always shall be **explicit**

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

**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 separated by **white-space**)

```plaintext
Module-Name { <optional object identifier> }  -- name is mandatory
DEFINITIONS

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

EXTENSIBILITY IMPLIED
 ::=
BEGIN
EXPORTS  <comma separated list> ;  -- optional
IMPORTS  <comma separated list>  FROM <module identifier>;  -- optional
<assignments>  -- definitions in the module body
END
```

---

---
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
4. Subtyping
   - General, single value, value range, size constraint, type constraint, permitted alphabet, contained subtype, inner subtyping
**ASN.1 abstract syntax**

**Subtyping: features**

- Creates a new type from the parent type!
  
  \[
  \text{Parent ::= INTEGER (0..31)} \quad \text{NewOne ::= Parent (0..15)}
  \]

- May be an expression using set aritmetics
  
  - \text{INTERSECTION (\(\wedge\))}, \text{UNION (|)} \& \text{EXCEPT}
    
    \[
    \text{Expr1 ::= INTEGER (ALL EXCEPT (0..15))} \quad \text{Expr2 ::= INTEGER (0..1|4..6)}
    \]

- May contain extension markers (...) and exceptions(!)
  
  \[
  \text{Values ::= INTEGER (0..15, ... !1)} \quad \text{Values ::= INTEGER (0..15, ..., 0..31 !1)}
  \]

- Very often parameterized
  
  \[
  \text{ThisIsIt \{INTEGER:maxlTerations, INTEGER:maxNumber\} ::=}
  \]
  
  \[
  \text{SEQUENCE (SIZE ( 0.. maxlTerations)) OF INTEGER (0.. maxNumber)}
  \]
**ASN.1 abstract syntax**

**Subtyping - 1**

- **Single value constraint** *(applicability)*
  
  \[
  \text{Zero} ::= \text{INTEGER (0)} \quad \text{Yes-No} ::= \text{IA5String ("Yes" |"No")}
  \]

- **Value range constraint** *(applicability)*
  
  \[
  \text{Seven-bit-range} ::= \text{INTEGER (0..127)}
  \]
  
  \[
  \text{PI} ::= \text{REAL ( } \{\text{mantissa 3141, base 10, exponent -3}\} \ldots \{\text{mantissa 3142, base 10, exponent -3}\} \text{ )}
  \]

- **Permitted alphabet** *(applicability)*
  
  - applies to restricted character string types only ( except CHARACTER STRING!)
  
  \[
  \text{CapitalA} ::= \text{IA5String ( FROM ("A"))}
  \]
  
  \[
  \text{String1} ::= \text{IA5String ( FROM ("A".. "Z") UNION FROM ("a".. "z"))}
  \]
  
  \[
  \text{String2} ::= \text{IA5String ( FROM ("A".. "Z" UNION "a".. "z"))}
  \]

\[
\text{String1 } = \text{ String2 } ???
\]

\[
\text{String1} \neq \text{ String2 } !!
\]

\[
\text{String1: strings containing all capital or all small letters only, String 2: capital and small letters can be mixed within a string}
\]
ASN.1 abstract syntax

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

```plaintext
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

```plaintext
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!

```plaintext
ParentChoice ::= CHOICE { otherCase INTEGER, thisCase  BIT STRING }
NewChoice ::= thisCase < ParentChoice
myValue NewChoice ::= '01010101'B
```
Inner subtyping (applicability)
- The exception: does NOT create a new type but restricts 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)
})
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  
  mandatory values  
} )

– Partial specification

Born-2 ::= Parent-type ( WITH COMPONENTS {  
  keyword for partial spec. ... ,  
  ‘flag” & “complex” remain OPTIONAL  
  constrained & remains optional  
  mandatory values  
} )
**ASN.1 abstract syntax**

- **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 {
      may be chosen flag ,
      “complex” & “values” shall never be chosen value ( 0 .. 65535 )
    })

    Born-4 ::= Parent-type ( WITH COMPONENTS {
      the only choice values PRESENT }
    )

  - **Partial specification**
    
    Born-5 ::= Parent-type ( WITH COMPONENTS { ... , value PRESENT }
    “value” is the only legal choice

    Born-6 ::= Parent-type ( WITH COMPONENTS { ... , value ABSENT }
    “value” shall never be chosen
ASN.1 abstract syntax

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

Example

notation for an open type: MY-CLASS-2.&Argument
type constraint: ( INTEGER )

open type: a set of all ASN.1 types

type constraint: ENUMERATED
## ASN.1 abstract syntax

<table>
<thead>
<tr>
<th>Bit String</th>
<th>Boolean</th>
<th>Choice</th>
<th>Embedded-pdv</th>
<th>Enumerated</th>
<th>External</th>
<th>Instance-of</th>
<th>Integer</th>
<th>Null</th>
<th>Object class field type</th>
<th>Object Identifier</th>
<th>Octet String</th>
<th>open type</th>
<th>Real</th>
<th>Relative Object Identifier</th>
<th>Restricted Character</th>
<th>String Types</th>
<th>Sequence</th>
<th>Sequence-of</th>
<th>Set</th>
<th>Set-of</th>
<th>Unrestricted Character</th>
<th>String Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Value</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Contained Subtype</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Value Range</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Size Constraint</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Permitted Alphabet</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Type constraint</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Inner Subtyping</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pattern constraint</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
**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
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
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

\[ A ::= \text{SEQUENCE} \left( \text{SIZE} \left( 0 .. 15 !1 \right) \right) \text{OF INTEGER} \left( 0 .. 65535 !2 \right) \]

\[ B ::= \text{SEQUENCE} \left( a \ A, \ b \ B, \ ... ! \text{IA5String:”not comprehended element received”} \right) \]

\[ C ::= \text{SEQUENCE} \left( \text{SIZE} \left( 0 .. 15 ! \right) \right) \text{OF INTEGER} \]

\[ \rightarrow \text{comment in ASN.1 or specified in the procedures description or} \]
\[ \rightarrow \text{take implementation-dependent action} \]
**ASN.1 abstract syntax**

**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 arithmetics:

  \[
  \text{INTEGER ( (0..15, ..., 32..63) | (16..31, ...) )}
  \]

  \[\rightarrow \text{range 0..63, type extensibility is not defined clearly!}\]

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

  \[
  \begin{align*}
  \text{INTEGER ( (0..15, ..., 32..63) | (16..31, ...) )} & \rightarrow \text{inextensible, range 0..31} \\
  \text{INTEGER ( (0..31, ..., 32..63) \^ (0..63, ...) )} & \rightarrow \text{inextensible, range 0..31} \\
  \text{INTEGER ( (0..15) | (16..31), ... )} & \rightarrow \text{extensible type, range 0..31}
  \end{align*}
  \]

  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)
    
    | Symbol | Syntax                  | Explanation                       |
    |--------|-------------------------|-----------------------------------|
    | 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
    
    | Symbol | Syntax                  | Explanation                       |
    |--------|-------------------------|-----------------------------------|
    | 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)
    
    | Symbol | Syntax                  | Explanation                       |
    |--------|-------------------------|-----------------------------------|
    | 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 |
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
} OPTIONAL,

A conceptual elements

... ! 2
B

--> 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 - general**

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

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

  Governor dummy reference

  "IN" => formal parameters

  "OUT" => actual parameters
**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

```asn1
Sample1 { Par-Type } ::= CHOICE {
                       altern1 BOOLEAN,
                       altern2 Par-Type }  
```

#### Sample2

```asn1
Sample2 { INTEGER: par-value } ::= SEQUENCE {
                       first INTEGER DEFAULT par-value }
```

#### Sample3

```asn1
Sample3 { ParType, ParType: parValue } ::= SEQUENCE {
                       seethis ParType ( parValue ) }
```

#### Names

```asn1
Names { IA5String: ExtraNames } IA5String ::= { “Johny” | “Freddy” | ExtraNames }
```

**=>**

```asn1
ToBeInvited IA5String ::= { Names { { “Jill” | “Mary” | “Sue”} } }
```

**for the actual parameter** for value set notation
**ASN.1 abstract syntax**

**Parameterization**

- **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 } ::=  
    -- whatever type assignment follows --
    ```

    --> invalid

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

    --> valid
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 }  

Sample7 { ParType } ::= 
  SEQUENCE { 
    a INTEGER, 
    b Sample7 { ParType } }  

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

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

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

---

Sample6 is invalid because of circular referencing.

Sample7 is invalid because of circular referencing.

Sample8 is valid because of non-circular alternative.

Sample9 is valid because of non-circular alternative.