Introduction to ASN.1

Simple and structured types

Basic concepts

Version 1.1.2

http://www.powerasn.com

Presentation

- Problem:
  - Heterogeneous systems
  - Multiple programming languages
- How to exchange information?
- Abstract Syntax Notation 1

ASN.1 is everywhere

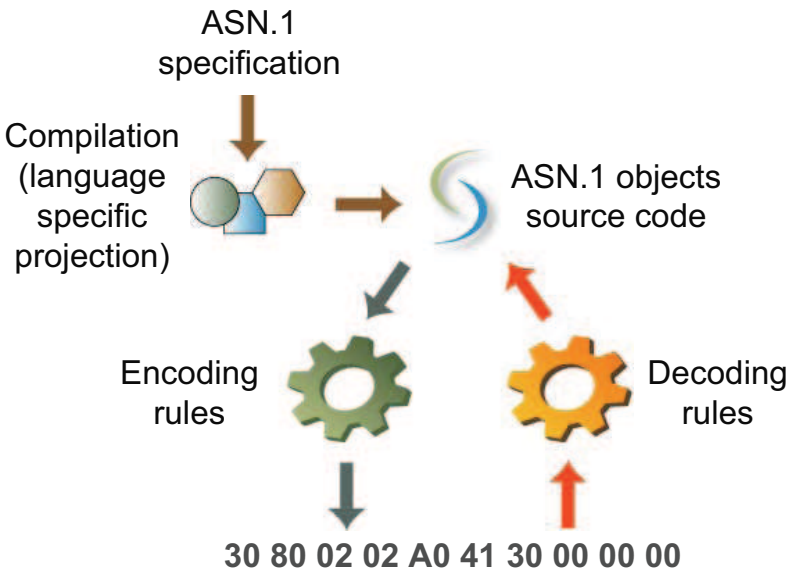
ASN.1 is:

- A formal notation to describe data types
- A specification of encoding / decoding rules

Is it used in many fields, such as:

- Aeronautics: ATN
- Telecommunications: VoIP, MAP
- Network protocols: SNMP
- Security: cryptography, digital signature

ASN.1 processes



ASN.1 components

- A notation syntax used to represent constrained data types: this syntax is abstract as it is not linked with a particular transfer syntax (encoding / decoding processes)

- Standard transfer syntaxes:
  - BER: Basic Encoding Rules
  - CER: Canonical Encoding Rules
  - DER: Distinguished Encoding Rules

  } Octet-based transfer syntaxes

  - PER: Packed Encoding Rules

  } Bit-based

  - XER (B-XER,C-XER, E-XER): XML Encoding rules

Simple types

- INTEGER
- REAL
- ENUMERATED
- BOOLEAN
- OCTET STRING, BIT STRING
- String types (NumericString, VisibleString, …)
- UTCTime
- GeneralizedTime
- RELATIVE IDENTIFIER
- OBJECT IDENTIFIER
- NULL (no associated value)

Identifiers

- RELATIVE IDENTIFIER
- OBJECT IDENTIFIER

Identifier ::= NumericValue
Identifier ::= NumericValue Id

Id ::= "." Identifier

- Example: 2.5.29.15 (certificate key usage)

Structured types

Placeholders for inner ASN.1 elements

- SEQUENCE → Ordered
- SET → Unordered

- SEQUENCE OF
- SET OF

} Collections

- EMBEDDED PDV, …

Structured types elements

Inner ASN.1 elements can be declared:

- Mandatory: must be initialized
- OPTIONAL: may not be initialized
- With a DEFAULT value: this value is used if not initialized

OPTIONAL and DEFAULT are mutually exclusive

↳ These declarations along with a transfer syntax define the encoding (or decoding) process success or failure

Transparent types

- CHOICE → Alternative

↳ Defined set of possible (tagged) elements
Each element must be uniquely identified
This identification is linked with encodings

- OpenType → Formerly ANY

↳ "Blob" type

↳ Do not appear in encodings

Types constraints

- Basic types (examples):
  - INTEGER: minimum / maximum values
  - REAL: minimum / maximum values
  - String types: characters restrictions, length, regular expressions match
  - ENUMERATED: list of accepted (significant) values
  - Time (UTC, Generalized): time validity
  - IDENTIFIER: positive integer values with restrictions
- Collections:
  - minimum / maximum size
  - Elements types (inner elements must belong to the same type)

Simple types derived types examples

```
Version     ::= INTEGER

Name        ::= UTF8String SIZE(1..50)
-- Not empty and cannot exceed 50 characters

Gender      ::= ENUMERATED {
   male,
   female
}
```

Values assignments

- General format:

```
instance    ::=    Type        Value
                   |
                   <tag>Value</tag>     -- XML format
```

- Examples:

```
Int          ::=  INTEGER

my-Value     ::=  Int       3
```

```
bool-Instance ::=
        BOOLEAN
        TRUE
```


SEQUENCE derived type example

```
Individual ::= SEQUENCE {
   first       UTF8String,
   last        UTF8String,
   age         INTEGER (0..MAX)     OPTIONAL,
   gender      Gender               DEFAULT  female
}

myself Individual ::= {  -- 'age' is not mandatory
   first "foo", last "bar", gender male
}
```


SET OF derived type example

```
Individuals ::= SET OF Individual

myIndividuals Individuals ::= {
   {first "Paul", last "Smith", age 24, gender male},
   {first "John", last "Smith", age 30, gender male},
   {first "Pamela", last "Smith"}
   -- 'gender' DEFAULT value applies if not mentioned
}
```


CHOICE derived type example

```
Time ::= CHOICE {
   utc              UTCTime,
   general          GeneralizedTime
}

myTime Time ::= {utc "0612242359Z"}
```

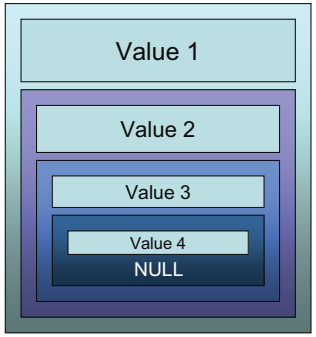
Only one inner ASN.1 element encoded

Generic linked list example 1

```
LinkedList1 ::= SEQUENCE {
   value          OpenType,
   next           NextElement
}

NextElement ::= CHOICE {
   other          LinkedList1,
   noElement      NULL
}
```

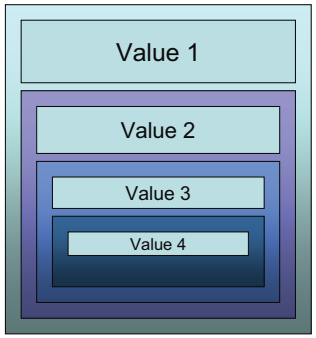
Example of 4 values



Generic linked list example 2

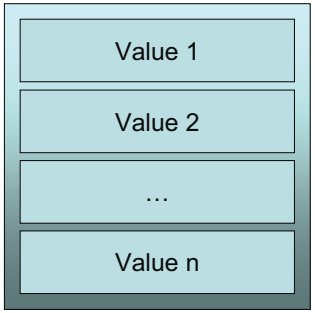
```
LinkedList2 ::= SEQUENCE {
   value          OpenType,
   next           LinkedList2      OPTIONAL
}
```

Example of 4 values



Generic linked list example 3

```
LinkedList3 ::= SEQUENCE OF OpenType
```



Specs evolution: types substitution

Replace any non OpenType element with CHOICE:

```
MyValue ::= INTEGER
```

```
MyValue ::= CHOICE {
        val1        INTEGER,
        val2        REAL
}
```

Specs evolution: extensibility

Use of ellipsis "…" within SEQUENCE (OF) and ENUMERATED types and types constraints (ASN.1 97 specifications)

Example:

```
Individual ::= SEQUENCE {
   first      UTF8String,
   last       UTF8String,
   age        INTEGER (0..MAX)     OPTIONAL,
   gender     Gender               DEFAULT  female,
   …
}
```


First specification extension

```
Individual ::= SEQUENCE {
   first      UTF8String,
   last       UTF8String,
   age        INTEGER (0..MAX)     OPTIONAL,
   gender     Gender               DEFAULT  female,
   …,
   [[2: address    UTF8String,
        town       UTF8String]]
}
```

Can be omitted: refers to version 2 extra components

Next specification extension

```
Individual ::= SEQUENCE {
   first      UTF8String,
   last       UTF8String,
   age        INTEGER (0..MAX)     OPTIONAL,
   gender     Gender               DEFAULT  female,
   …,
   [[2: address    UTF8String,
        town       UTF8String]],
   [[3: email      GeneralString]]
}
```


Comments on extensibility

- Opening and closing double square brackets used to group extensions together
- An optional version number can be mentioned

- Double squares are not mandatory:

```
   [[2: address    UTF8String,
        town       UTF8String]],
   email           GeneralString
```

Module description

```
ModuleName      DEFINITIONS
   IMPLICIT TAGS
   -- Module header section
        ::=
   BEGIN
        -- IMPORTS section;
        -- EXPORTS section;

        -- Derived types declarations section
   END
```


CHOICE invalid definition 1

```
Invalid1 ::= CHOICE {
   str1        PrintableString,
   str2        PrintableString
}
```

Why is this definition incorrect?

CHOICE invalid definition 1 (ctn'd)

```
Invalid1 ::= CHOICE {
   str1        PrintableString,
   str2        PrintableString
}
```

2 elements of the same type: decoding ambiguity

CHOICE invalid definition 2

```
Invalid2 ::= CHOICE {
   blob        OpenType,
   octets      OCTET STRING
}
```

Why is this definition incorrect?

CHOICE invalid definition 2 (ctn'd)

```
Invalid2 ::= CHOICE {
   blob      OpenType,
   octets    OCTET STRING
}
```

- OpenType can hold an OCTET STRING: similar to previous case

Basic ASN.1 notation limits

```
TwoInt ::= SEQUENCE {
   val1      INTEGER      OPTIONAL,
   val2      INTEGER      OPTIONAL
}
```

- Why is this description incorrect?

Basic ASN.1 notation limits (ctn'd)

```
TwoInt ::= SEQUENCE {
   val1      INTEGER      OPTIONAL,
   val2      INTEGER      OPTIONAL
}
```

What happens if only one INTEGER is initialized?

- Decoding ambiguity: val1 or val2 received?

Basic ASN.1 notation limits (end)

```
OtherAmbiguous ::= SEQUENCE {
   val1      INTEGER      DEFAULT v1(1),
   val2      INTEGER      OPTIONAL
}
```

Same problem if the applied transfer syntax does not encode non-initialized DEFAULT-valued elements

- Decoding ambiguity if a single INTEGER is encoded

Non-ambiguous declaration

Use tags!

```
Unambiguous ::= SEQUENCE {
   val1      [0]   INTEGER        OPTIONAL,
   val2            INTEGER        OPTIONAL
}
```

↳ Refer to ASN.1 tagging presentation…

Conclusion

- ASN.1 is used in various domains including security
- It is composed of:
  - A notation syntax
  - A set of standard transfer syntaxes, including XML-like
- ASN.1 is an improved working environment
- Allows backward compatible evolutions
- Transfer syntaxes ensure systems interoperability
- Many implementations available in many languages (Perl, PHP, C, C++, Java, .NET, …)
- Efficient octet-based and bit-based implementations for high performances and real-time communications