ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

04 - Protocol Specification, Part III

Stefan Rührup

University of Freiburg
Computer Networks and Telematics

Summer 2009

CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG

Overview

- **In previous parts of this chapter:**
  - Modeling behaviour with state machines, state charts

- **Part III:**
  - Specifying data/message formats

Data Format Specification

- Structuring data is part of the design process
  ... however, it is usually not of primary importance.
- Design decisions should not be driven by data format or encoding issues
- A defined data/message format and the corresponding encoding is important for the interoperability

- Some notation needed during the specification process
  (...and also for documentation)

Tabular and Box Notation

- Tabular Notation
  - Listing of message fields with types, lengths and descriptions
- Box Notation
  - Table of message fields, where the size of the boxes indicate the field length
  - used by the IETF in RFCs

Tabular Notation, Example

Message type: INFORMATION Significance: Local (Note 1) Direction: Both				
Information element	**Reference (subclause)**	**Direction**	**Type**	**Length**
Protocol discriminator	4.2	Both	M	1
Call reference	4.3	Both	M (Note 2)	2-*
Message type	4.4	Both	M	1
Sending complete	4.5	Both	O (Note 3)	1
Display	4.5	$n \to u$	O (Note 4)	(Note 5)
Keypad facility	4.5	$u \to n$	O (Note 6)	2-34
Signal	4.5	$n \to u$	O (Note 7)	2-3
Called party number	4.5	both	O (Note 8)	2-*

Contents of an ISDN Information Message

ISDN Basic Call Control Specification [ITU Q.931]

Tabular and Box Notation, Example

8	7	6	5	4	3	2	1	Octet
Protocol discriminator								1
0	0	0	0	Length of call reference value (in octets)				2
Call reference value								3
0	Message type							etc.
Other information elements as required								

Bits 8	7	6	5	4	3	2	1	
0	0	0	0	0	0	0	0	Escape to nationally specific message type (Note)
0	0	0	-	-	-	-	-	*Call establishment message:*
			0	0	0	0	1	– ALERTING
			0	0	0	1	0	– CALL PROCEEDING
			0	0	1	1	1	– CONNECT
			0	1	1	1	1	– CONNECT ACKNOWLEDGE
			0	0	0	1	1	– PROGRESS
			0	0	1	0	1	– SETUP
			0	1	1	0	1	– SETUP ACKNOWLEDGE
0	0	1	-	-	-	-	-	*Call information phase message:*
			0	0	1	1	0	– RESUME
			0	1	1	1	0	– RESUME ACKNOWLEDGE

General Message Organization and Message Types

ISDN Basic Call Control Specification [ITU Q.931]

Box Notation, Example

- Data format specification of the internet protocols
- TCP and IP packets have a header and a data part
- Header information:
  - destination address (IP) and port (TCP)
  - source address and port
  - TTL (IP), sequence number (TCP), and checksums (both)
  - flags and options
  - etc.

Box Notation, Example (1)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

ASCII Box Notation for the IPv4 Header Format [RFC 791]

Box Notation, Example (2)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

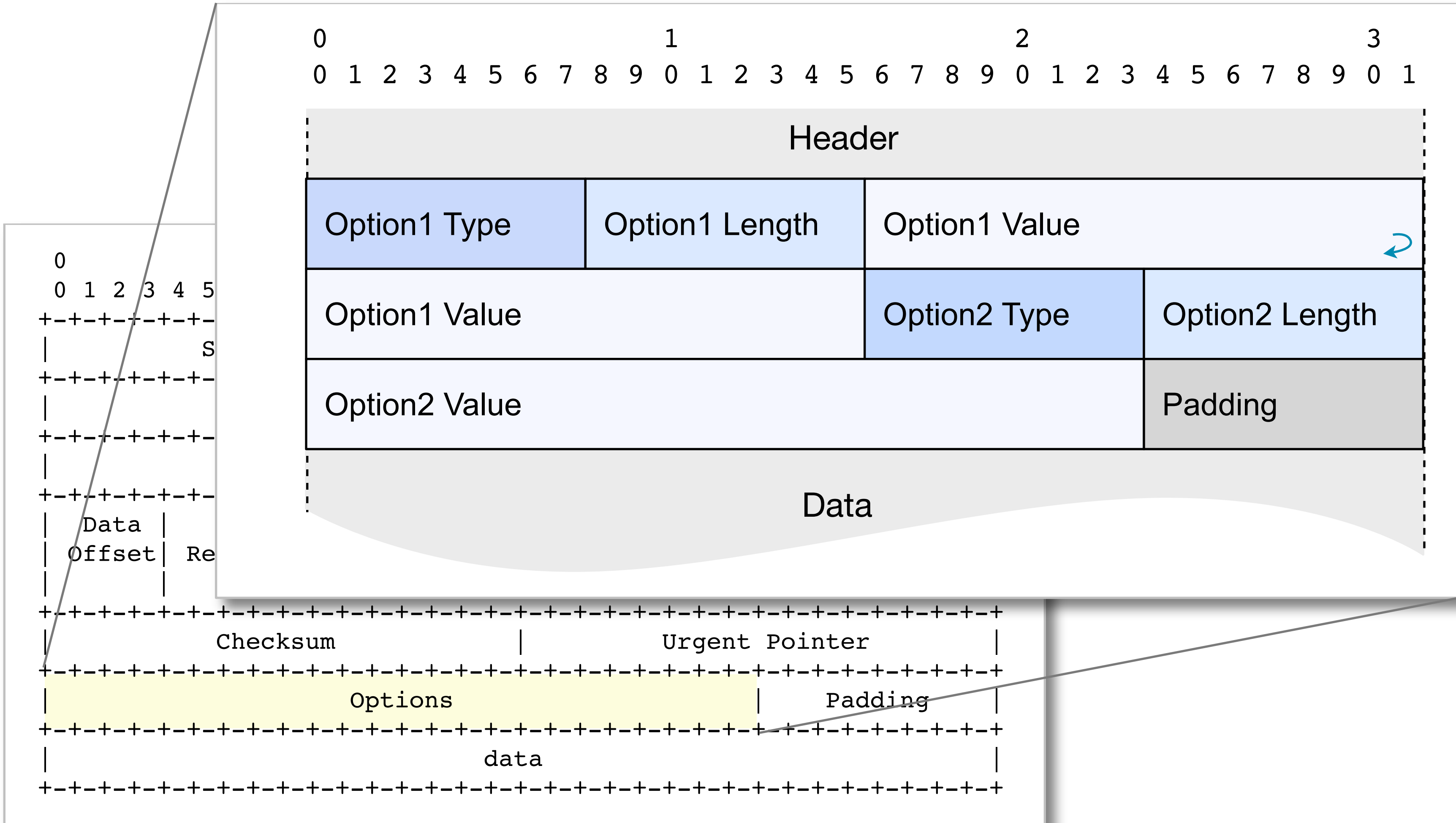
ASCII Box Notation for the TCP Header Format [RFC 793]

Box Notation and TLVs

- **Box Notation**
  - Intuitive way of describing a data format
  - Field types/encoding have to be specified separately
  - Limitations: no variable length fields

- **Type-Length-Value (TLV)**
  - Representation of variable size or optional message fields
  - Can be parsed without understanding the meaning of the field

TLV Example



Option Field in the TCP Header [RFC 793]

Getting more formal: ABNF

- **Augmented Backus-Naur Form**
- Defined in RFC 5234 (older def. in RFC 822)
- Definition language for some IETF protocols

- BNF describes context-free grammars (Chomsky 2) by derivation rules of the form `<symbol> ::= expression`
- Left side symbols: *non-terminals*, right side symbols: *terminals* and *non-terminals*
- Right side expression: sequence of symbols (or choice of sequences)

ABNF Example

```
date-time       =   [ day-of-week "," ] date time [CFWS]
day-of-week     =   ([FWS] day-name)
day-name        =   "Mon" / "Tue" / "Wed" / "Thu" /
                    "Fri" / "Sat" / "Sun"
date            =   day month year
day             =   ([FWS] 1*2DIGIT FWS)
month           =   "Jan" / "Feb" / "Mar" / "Apr" /
                    "May" / "Jun" / "Jul" / "Aug" /
                    "Sep" / "Oct" / "Nov" / "Dec"
year            =   (FWS 4*DIGIT FWS)
time            =   time-of-day zone
time-of-day     =   hour ":" minute [ ":" second ]
hour            =   2DIGIT
minute          =   2DIGIT
second          =   2DIGIT
zone            =   (FWS ( "+" / "-" ) 4DIGIT)
```

FWS = folding white space

[The Internet Message Format, RFC 5322]

ABNF Notation conventions

- Rule names are not case sensitive
- Certain rules (*core rules*) are in upper case.
- Rules *can* be put in brackets < > as in BNF,
  but this is not mandatory
- A colon ; starts a comment
- Binary and hexadecimal values: `%b1101`, `%h98C3`

Operators (1)

- **Concatenation**

```
rule := r1 r2 r3
```

- **Alternation**

```
rule := r1 / r2 / r3
```

- **Group**

```
rule := r1 / (r2 / r3) / r4
```

Operators (2)

- **Repetition**

```
nRule    ; n repetitions
n*mRule  ; min*max repetitions, default: 0*infinity
```

- **Optional occurence**

```
[Rule]
*1Rule
```

- **Value range**

```
OCTAL = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"
OCTAL = %x30-37
```

Core Rules

`ALPHA  =  %x41-5A / %x61-7A; A-Z / a-z`	
`BIT    =  "0" / "1"`	
`CHAR   =  %x01-7F`	any 7-bit US-ASCII character, excluding NUL
`CR     =  %x0D`	carriage return
`CRLF   =  CR LF`	Internet standard newline
`CTL    =  %x00-1F / %x7F`	controls
`DIGIT  =  %x30-39`	0-9
`DQUOTE =  %x22`	" (Double Quote)
`HEXDIG =  DIGIT / "A" / "B" / "C" / "D" / "E" / "F"`	
`HTAB   =  %x09`	horizontal tab
`LF     =  %x0A`	linefeed
`LWSP   =  *(WSP / CRLF WSP)`	linear white space rule
`OCTET  =  %x00-FF`	8 bits of data
`SP     =  %x20`	
`VCHAR  =  %x21-7E`	visible printing character
`WSP    =  SP / HTAB`	white space

[RFC 5234]

Example

```
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line      = Request-Line | Status-Line

message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, separators, and quoted-string>

Request-Line   = Method SP Request-URI SP HTTP-Version CRLF

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

message-body = entity-body
             | <entity-body encoded as per Transfer-Encoding>

...
```

[HTTP/1.1, RFC 2616]

Example

HTTP 204 Message:

```
0000  00 23 6c 89 55 cf 00 1d  7e ac d6 d2 08 00 45 00   .#l.U... ~.....E.
0010  00 b1 76 73 00 00 38 06  d7 bf 4a 7d 27 8b c0 a8   ..vs..8. ..J}'...
0020  01 64 00 50 cb a6 2f 3c  9e e7 7a 57 85 68 80 18   .d.P../< ..zW.h..
0030  00 6a da 91 00 00 01 01  08 0a e8 57 1d 93 2e 74   .j...... ...W...t
0040  41 60 48 54 54 50 2f 31  2e 31 20 32 30 34 20 4e   A`HTTP/1 .1 204 N
0050  6f 20 43 6f 6e 74 65 6e  74 0d 0a 43 6f 6e 74 65   o Conten t..Conte
0060  6e 74 2d 4c 65 6e 67 74  68 3a 20 30 0d 0a 43 6f   nt-Lengt h: 0..Co
0070  6e 74 65 6e 74 2d 54 79  70 65 3a 20 74 65 78 74   ntent-Ty pe: text
0080  2f 68 74 6d 6c 0d 0a 44  61 74 65 3a 20 46 72 69   /html..D ate: Fri
0090  2c 20 32 32 20 4d 61 79  20 32 30 30 39 20 30 37   , 22 May  2009 07
00a0  3a 34 34 3a 30 33 20 47  4d 54 0d 0a 53 65 72 76   :44:03 G MT..Serv
00b0  65 72 3a 20 47 46 45 2f  32 2e 30 0d 0a 0d 0a      er: GFE/ 2.0....
```

TCP

HTTP

Text based encoding in type : value format with *CRLF* as separator

HTTP/1.1 204 No Content *CRLF*
Content-Length: 0 *CRLF*
Content-Type: text/html *CRLF*
Date: ...

How to use ABNF?

- **Just for specification**
- **...or to generate a parser:**

1. Format description in ABNF
2. Automatic parser generator
   Generates program code that parses the input according to the ABNF grammar. Empty callback functions for ABNF rules are inserted into the code.
3. Implementation of callback functions
4. Integration in application

CSN.1

- **Concrete Syntax Notation One**
- Data encoding description developed by the GSM community
- Based on BNF
- Defines valid encoded data streams on a bit level
- used in ETSI and 3GPP standard documents

- see Annex B of [3GPP TS 24.007]
  (www.3gpp.org/ftp/Specs/archive/24_series/24.007)
- ...or www.csn1.info

CSN.1 Example (1)

```
<bit> ::= {0 | 1}

<octet> ::= {0 | 1} {0 | 1} {0 | 1} {0 | 1}
            {0 | 1} {0 | 1} {0 | 1} {0 | 1} ;

<octet> ::= <bit>*8 ;

<octet string> ::= <octet>**;

<octet string(40)> ::= <octet>*(8*(4+1)) ;

<all bit strings> ::= null |  {<all bit strings> {0 | 1}} ;
```

CSN.1 definitions example

CSN.1 Example (2)

```
< PSI6 message content > ::=
  < PAGE_MODE : bit (2) >
  < PSI6_CHANGE_MARK : bit (2) >
  < PSI6_INDEX : bit (3) >
  < PSI6_COUNT : bit (3) >
  { { < NonGSM Message : < Non-GSM Message struct > > **
                                    -- The Non-GSM Message struct is repeated until:
      { < spare bit > * 3 00000 }   -- A) val(NR_OF_CONTAINER_OCTETS) = 0, or
      < padding bits > } //         -- B) the PSI message is fully used
  ! < Distribution part error : bit (*) = < no string > > ;
< NonGSM Message struct > ::=
  < NonGSM Protocol Discriminator : bit(3) >
  < NR_OF_CONTAINER_OCTETS : bit(5) exclude 00000 } >
  { < CONTAINER : bit(8) > } * (val(NR_OF_CONTAINER_OCTETS)) ;
```

Packet System Information Type 6
GSM/EDGE Radio Access Network Specification
[3GPP TS 44.060 v8.4.0]

CSN.1 Core Rules (1)

- B1: A *bit string* is an ordered sequence of symbols of {0,1}
- B2: *null* denotes the empty string
- B3: *Concatenation* is described by succession of strings
- B4: *Choices* are indicated by “|”
  Delimiters for a string set description are “{” and “}”
- B5: Delimiters for a *reference* to a string are “<” and “>”
- B6: *Definitions* have the form

  ```
  <reference> ::= <string set>
  ```

CSN.1 Core Rules (2)

- B7: A *spare bit* is 0 when sent and can be 0 or 1 when received. It is denoted by `<spare bit>`

- B8: *Padding bits* are filling bits. They are usually 0, sometimes a padding sequence is defined. Matching and non-matching a bit of a padding sequence is defined by L and H.

```
<spare padding> ::= L {null | <spare padding>};
```

Some more examples...

```
<bit> ::= {0 | 1}

<octet> ::= {0 | 1} {0 | 1} {0 | 1} {0 | 1}
            {0 | 1} {0 | 1} {0 | 1} {0 | 1} ;

<octet> ::= <bit>*8 ;

<octet string> ::= <octet>**;

<octet string(40)> ::= <octet>*(8*(4+1)) ;

<all bit strings> ::= null |  {<all bit strings> {0 | 1}} ;
```

CSN.1 Advanced Rules

- A1: *Labels* can be added to references or string sets:
  `<label : reference>` or `<label : string set>`

- A2: *Exponents* define repetitions of elements:
  `<string>(expression)` or `<string> * expression`
  where *expression* is a constant mathematical expression.
  The *infinite exponent* is denoted by **.

- G1: *Comments* start with “--” and terminate at the end of line

Further rules (1)

- Send construction: *<str_decoded>* = *<str_encoded>*
  e.g. `<spare bits> ::= null | <spare bits> { <bit> = 0 }`

- Functions: *function(label)*
  e.g. `data ::= <bit> * val(Length)`

- Intersection: *<string1>* **and** *<string2>* (or "&")
  both string1 and string2 have to match on the same data

- Exclusion: *<string1>* **exclude** *<string2>* (or "-")
  string1 has to match and string2 must not match

(cf. www.csn1.info)

Further rules (2)

- Error indications: *<valid_string>* `!` *<invalid_string>*
  like a choice, but a decoding error should be thrown
- Subclassing: *<reference == string>*
  limits a definition, e.g. `data ::= <bit> * val(Length)`
- Integer subclassing: *<reference := 0xValue>*
- Option: [*string*]
  e.g. `[<bit>]` is equivalent to `<bit> | null`
- Truncations: {*string set*} //
  i.e. items of the string set are optional

(cf. www.csn1.info)

CSN.1 Review

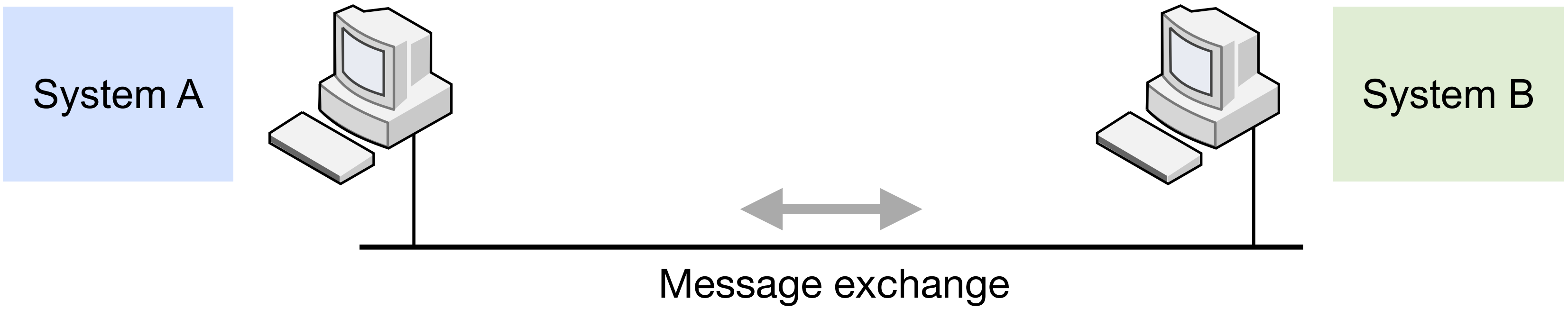
- Formal description method with simple rules
- Compilable
- Evolving standard, extensions and non-standard notation are often used.
- More complex encoding descriptions are difficult to write and understand

ASN.1

- **Abstract Syntax Notation One**
- Abstract data structure description language
- Data type definition + separate encoding rules

- **The Standard:** ITU-T X.680 series
  see www.itu.int/ITU-T/studygroups/com10/languages/
- Literature (see www.asn1.org/books)
  - John Larmouth: “ASN.1 Complete”, 1999
  - Olivier Dubuisson “ASN.1 - Communication between Heterogeneous Systems”, 2000

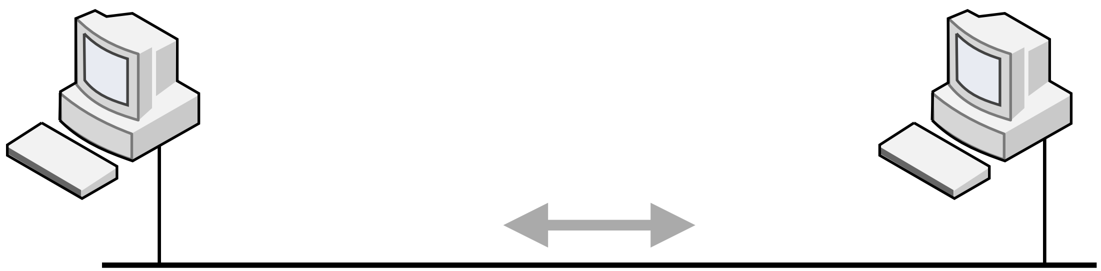
Goal of ASN.1



- Provide abstract syntax definition for communication among heterogeneous systems
- Methods to represent, encode/decode data
- Independent of platform-specific encoding

Example

C++
Linux



Pascal
Windows

Message exchange

concrete syntax

```
typedef struct {
  char   name[40];
  int    age;
  enum { win=0, linux=1, mac=2 } belief;
} Record;
```

concrete syntax

```
type BTYPE = { win, linux, mac };
type Record = RECORD
  name : STRING[40];
  age : INTEGER;
  belief : array [BTYPE] of integer;
END;
```

Abstract syntax

```
Record ::= SEQUENCE {
  name      PrintableString (SIZE (0..40)),
  age       INTEGER,
  belief    ENUMERATED { win(0), linux(1), mac(2) }
}
```

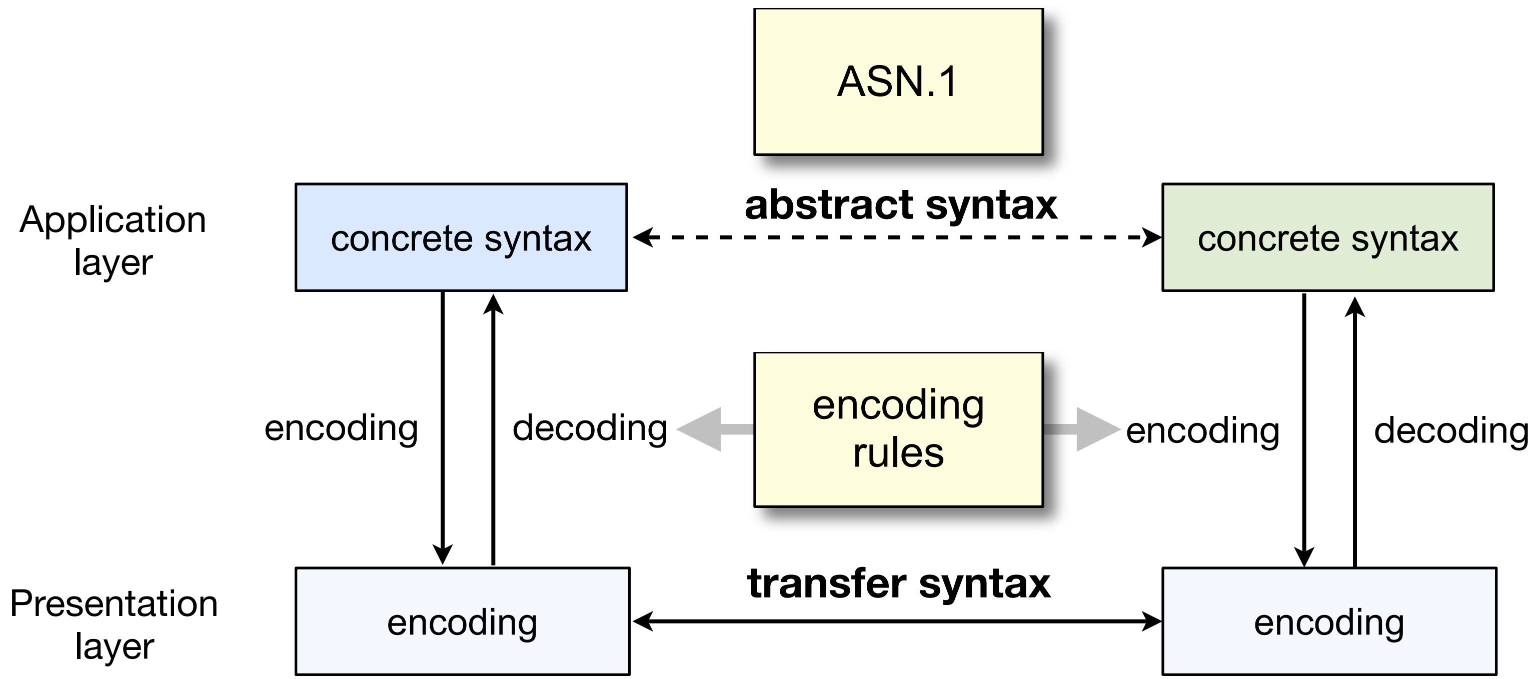
abstract syntax

concrete syntax ←----→ concrete syntax

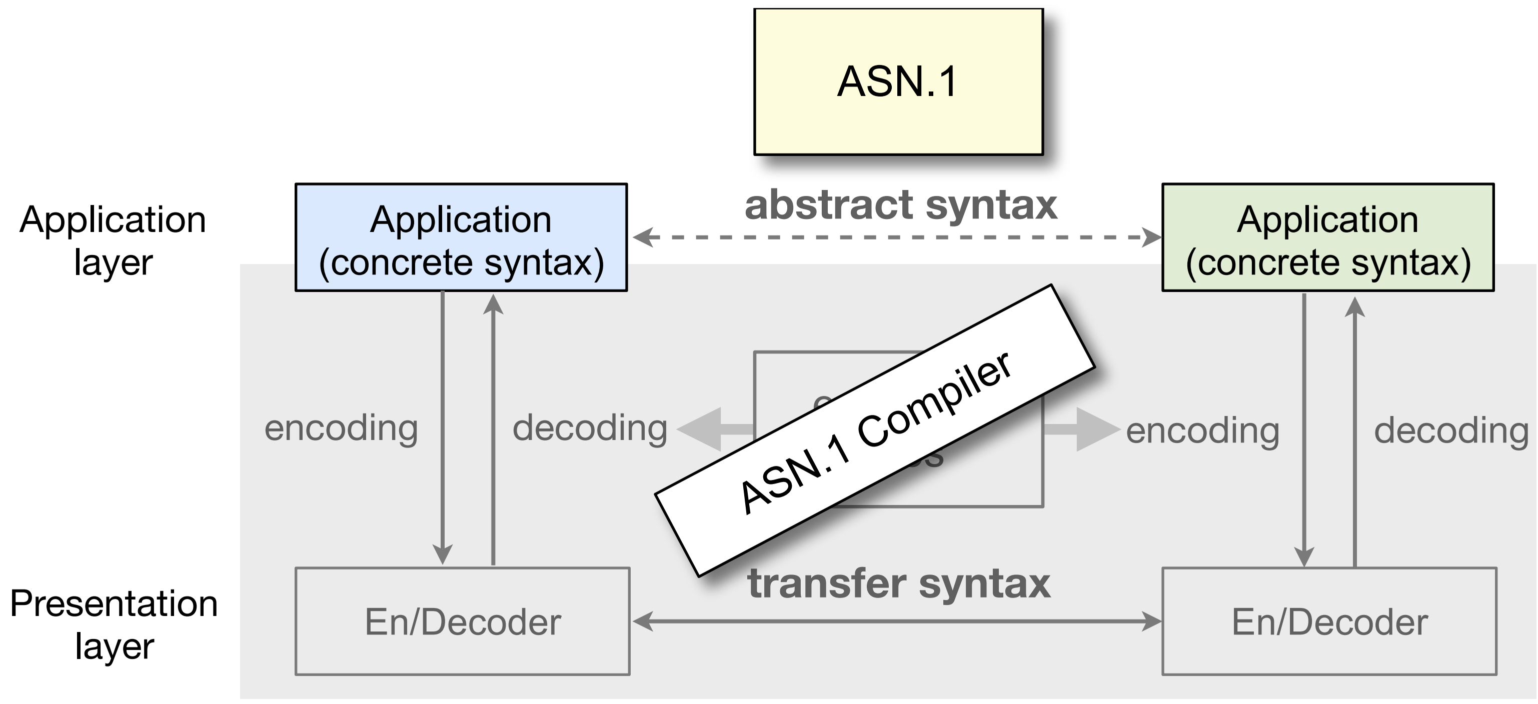
```
typedef struct {
  char   name[40];
  int    age;
  enum { win=0, linux=1, mac=2 } belief;
} Record;
```

```
type BTYPE = { win, linux, mac };
type Record = RECORD
  name : STRING[40];
  age : INTEGER;
  belief : array [BTYPE] of integer;
END;
```

Abstract syntax and transfer syntax



Abstract syntax and transfer syntax



ASN.1 Example

```
Protocol DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  ProtocolMessage ::= SEQUENCE {
                       Header  ::= ProtocolHeader,
                       Data    ::= ProtocolPayload,
                       Trailer ::= Checksum }
  ProtocolHeader ::= SEQUENCE {
                      SourceAddr ::= BIT STRING(SIZE(16)),
                      DestAddr ::= BIT STRING(SIZE(16)),
                      Flags ::= BIT STRING(SIZE(4)) }
  ProtocolData ::= OCTET STRING(SIZE(512))
  Checksum ::= CHOICE {
                crc16 BIT STRING(SIZE(17)),
                crc32 BIT STRING(SIZE(33)) }
END
```

Notation conventions

- All identifiers, references, keywords begin with a letter and may contain digits or single dashes
- ASN.1 keywords (such as built-in data types) consist of upper-case letters, except for some string types
- Module and type reference names start upper case, value reference names start lower case.
- Comments start with a double dash: `--comment`
- String notation: `“string”`
- Binary and hex value notation: `‘101011’B, ‘89AFBB09’H`

The Module

- Basic element of an ASN.1 specification
- Contains the type and value assignments

```
ModuleName DEFINITIONS ::=
BEGIN
  -- assignments
END
```

more general:

```
ModuleName {<object identifiers>}
DEFINITIONS <tagging clause> ::=
BEGIN
EXPORTS <export clause>;
IMPORTS <import clause>;
<assignments>
END
```

The Module

- Modules can import and export type definitions from/to other modules

```
ModuleName DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
EXPORTS Type1,Type2;
IMPORTS Type1 FROM Module1;

Type2 ::= TypeDefinition
  -- assignments
END
```

Type definitions

- General notation:

```
TypeReferenceName ::= TypeDefinition
```

- Examples:

```
Gender ::= BOOLEAN

Data ::= OCTET STRING

Identifyer ::= INTEGER

CountryID ::=  UTF8String
```

Value assignments

- General notation:

```
ValueReferenceName Type ::= Value
```

- Examples:

```
checked BOOLEAN ::= TRUE

data BIT STRING ::= ‘01001011’B

year INTEGER ::= 1985

name PrintableString ::= “Brown”
```

Predefined Types

	Primitive Types		Constructed Types	
basic types	BOOLEAN		CHOICE	choice between types
	NULL	used for recursive types	SEQUENCE	ordered structure of different types
	INTEGER		SET	unordered structure of different types
	REAL		SEQUENCE OF	ordered structure of the same type
	BIT STRING		SET OF	unordered structure of the same type
	OCTET STRING			
	ENUMERATED			
	OBJECT IDENTIFIER			
	RELATIVE-OID			
	EXTERNAL			
string types	NumericString			
	PrintableString			
	...String			
	UTCTime			
	GeneralizedTime			

Subtype constraints

- Subtypes can be derived with constrained size, constrained value range

```
Day ::= ENUMERATED{
          mon(0),tue(1),wed(2),thu(3),fri(4),sat(5),sun(6)}
Crc32 ::= BIT STRING (SIZE (33))
FibNr ::= INTEGER (0|1|2|3|5|8|13)
NonEmptyString ::= OCTET STRING(SIZE (1..MAX))

PositiveInt ::= INTEGER (0<..MAX)

ConstrainedString ::= VisibleString(PATTERN “regularExpression”)
```

Constructed Types

ASN.1 Constructed Types	Description	C++
`CHOICE`	choice between types	`union`
`SEQUENCE`	ordered structure of different types	`struct`
`SET`	unordered structure of different types	`struct`
`SEQUENCE OF`	ordered structure of the same type	`list or array`
`SET OF`	unordered structure of the same type	`list or array`

Constructed Types: Sequence

```
Record ::= SEQUENCE {
    length    INTEGER,
    options   OCTET STRING,
    flag      BOOLEAN DEFAULT FALSE,
    number    INTEGER OPTIONAL,
    ...
}
```

default value

optional field

extension marker: other types can be expected here

```
List ::= SEQUENCE OF INTEGER
```

- SET follows the same syntax. SET can be used instead if ordering does not matter (probably smaller encoding)

Constructed Types: Choice

```
Identifier ::= CHOICE {
    sin   [0] PrintableString,
    matNr [1] INTEGER,
    pan   [2] OCTET STRING
}
```

with explicit tagging (for unambiguous encoding)

```
Module DEFINITIONS AUTOMATIC TAGS := BEGIN

  Identifier ::= CHOICE {
      sin   PrintableString,
      matNr INTEGER,
      pan   OCTET STRING
  }

END
```

with automatic tagging

Extensibility

- Extension markers allow future extensions

root type

```
Packet ::= SEQUENCE {
    address BIT STRING (SIZE(3)),
    ...
}
```

the extension marker can be placed at the end of sequences, sets and choices

1st extension

```
Packet ::= SEQUENCE {
    address BIT STRING (SIZE(3)),
    ...,
    flag BOOLEAN,
    ttl INTEGER
}
```

Extension groups

- Extension addition groups

root type
(version 1)

```
Packet ::= SEQUENCE {
    address BIT STRING (SIZE(3)),
    ...
}
```

1st extension
(version 2)

```
Packet ::= SEQUENCE {
    address BIT STRING (SIZE(3)),
    ...,
    [[
      flag BOOLEAN,
      ttl INTEGER
    ]]
}
```

version brackets indicate an extension group; all included fields must be present in version 2

Tagging

- Optional fields and choices can lead to ambiguous encodings.

```
List ::= SEQUENCE {
    number   INTEGER OPTIONAL,
    code     INTEGER,
    value    INTEGER OPTIONAL
}
```

ambiguity!

- Tagging helps to identify data fields

```
List ::= SEQUENCE {
    number [0] EXPLICIT INTEGER OPTIONAL,
    code   [1] IMPLICIT INTEGER,
    value  [2] EXPLICIT INTEGER OPTIONAL
}
```

- *This should actually not be an issue on the abstract level ...*

Global Tagging

- Global tagging options (used in module definitions):
  - EXPLICIT TAGS (tags always inserted)
  - IMPLICIT TAGS (tags inserted, if necessary)
  - **AUTOMATIC TAGS** (automatically done by the compiler)

method of choice

```
Module DEFINITIONS AUTOMATIC TAGS := BEGIN

  Identifier ::= CHOICE {
      sin   PrintableString,
      matNr INTEGER,
      pan   OCTET STRING
  }

END
```

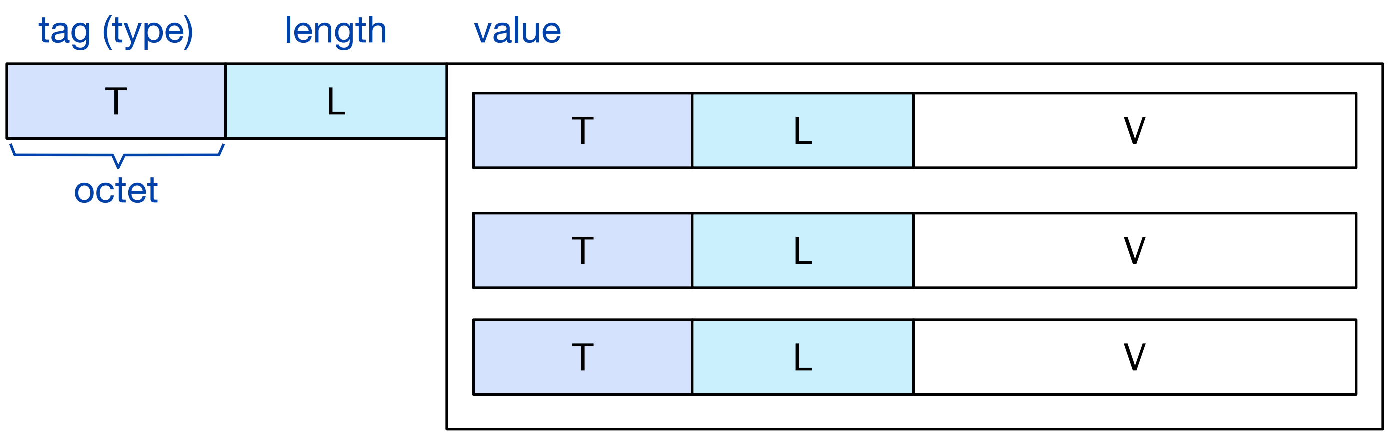
Encoding

- ASN.1 does not determine the transfer syntax
- Standardized encoding rules:
  - Basic encoding rules (BER) [X.690]
  - Canonical and Distinguished Encoding Rules (CER,DER)
  - Packed Encoding Rules (PER) [X.691]
  - XML Encoding Rules (XER) [X.693]

- Specification of specialized encoding rules:
  Encoding Control Notation (ECN) [X.692]

Basic Encoding Rules (BER)

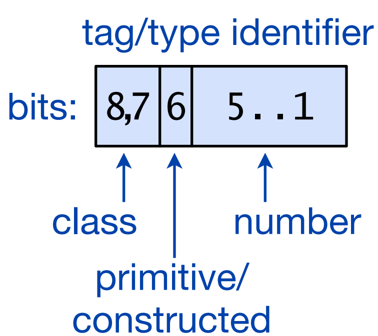
- Rules for encoding abstract data into concrete data
- Encoding in TLV Style (tag-length-value)
- TLVs can be cascaded, i.e. the value can contain a sequence of TLVs



BER Types

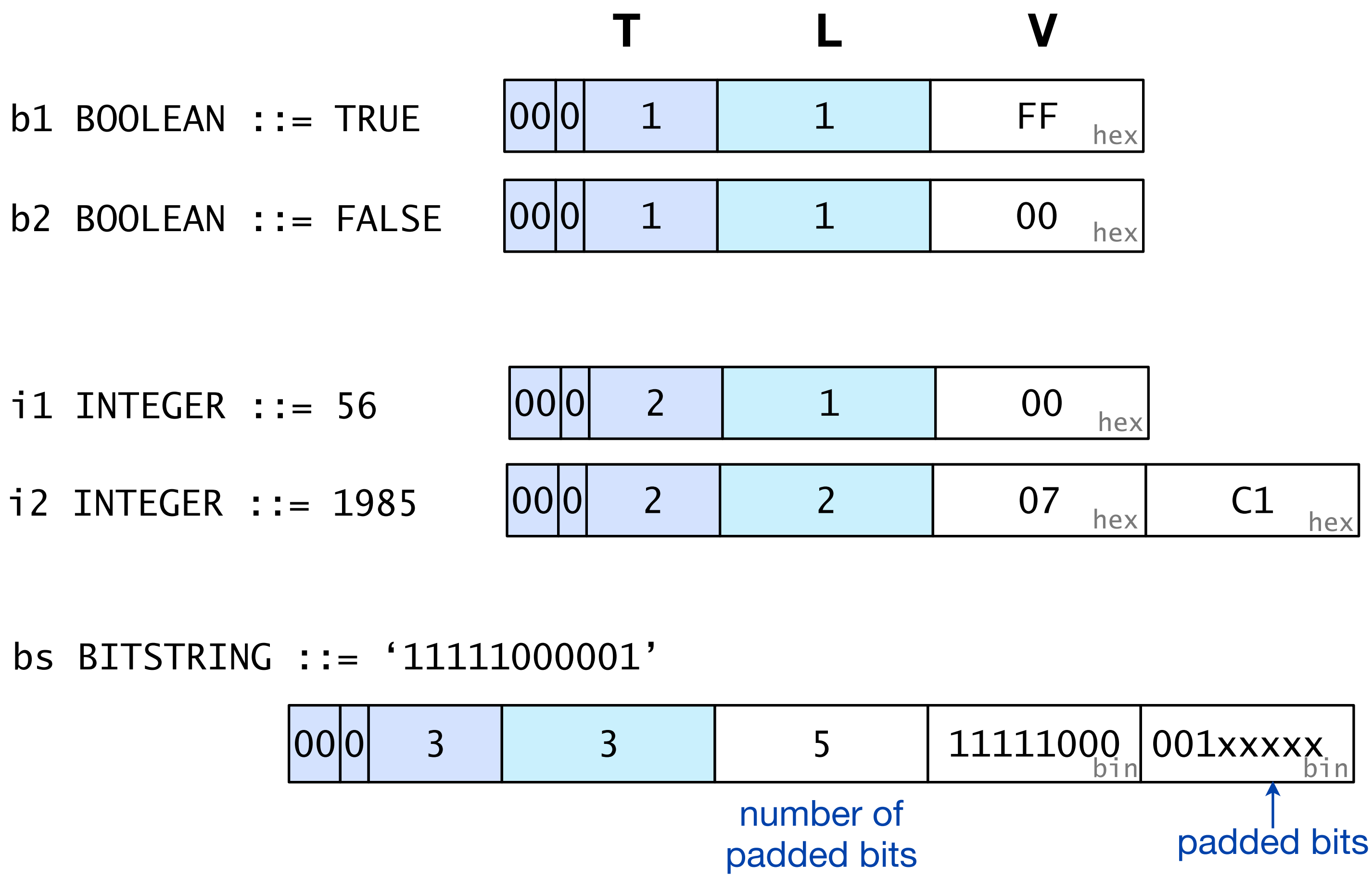
- Types are primitives or compounds and belong to different classes, indicating universal ASN.1 types or application-specific definitions

Class	bit 8	bit 7
Universal	0	0
Application	0	1
Context-specific	1	0
Private	1	1



Name	P/C	Number
BOOLEAN	P	1
INTEGER	P	2
OCTET STRING	P/C	4
SEQUENCE	C	16
...		

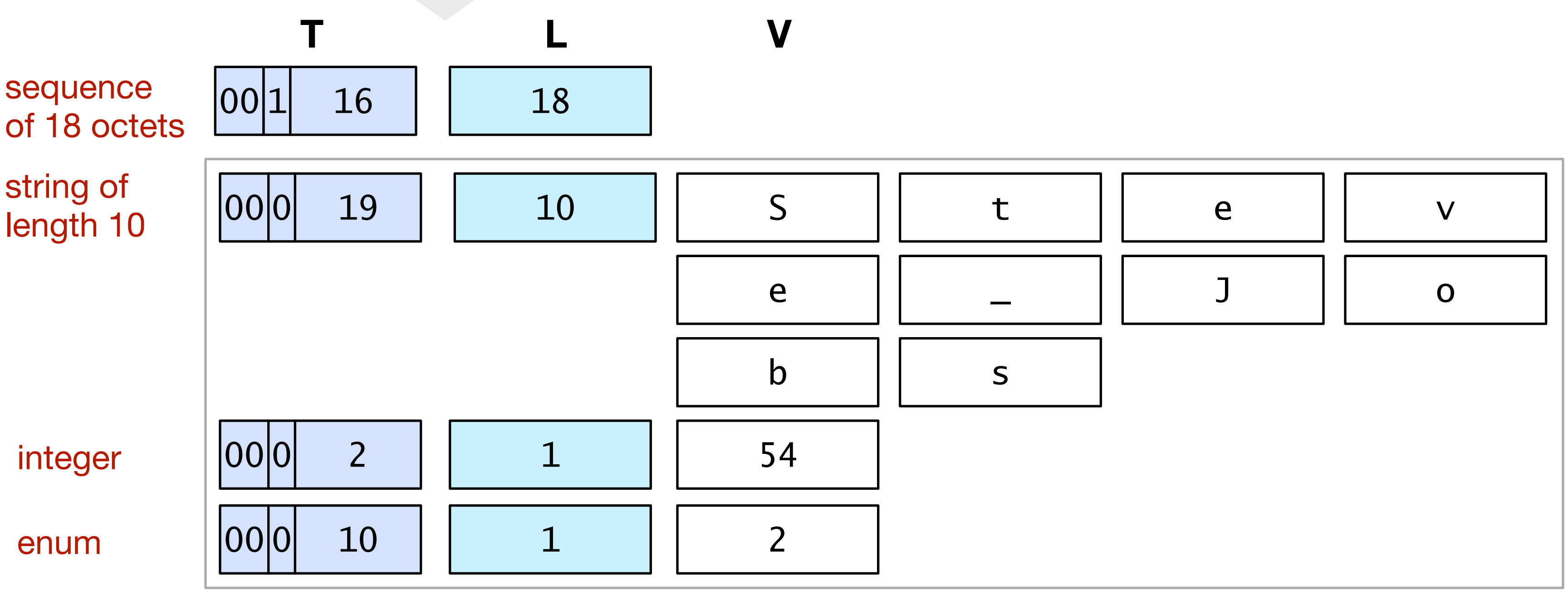
BER Examples (1)



BER Example (2)

```
Record ::= SEQUENCE {
  name      PrintableString (SIZE (0..40)),
  age       INTEGER,
  belief    ENUMERATED { win(0), linux(1),
                         mac(2) }
}
```

{“Steve Jobs”,54,2}



BER and other encodings

- **Shortcomings of the Basic Encoding Rules:**
  - Lengthy encoding. Type and length fields are always octets, e.g. a boolean value requires 3 octets = 24 bits
  - Encoding rules with some degrees of freedom (e.g. different ways of cascading TLVs)
- **CER/DER** (Canonical and Distinguished Encoding Rules)
  - Restrictions on BER such that there is only one possible encoding (injectivity; used in security protocols, e.g. exchange of certificates)
- **PER** (Packed Encoding Rules)
  - size reduction by giving up the TLV format

Packed Encoding Rules (PER)

- PER format: [P][L][V] instead of TLV:
  optinal preamble, optional length, optional value
- Series of bits instead of octets
- Tags are **not** encoded
- Preambles are used for choices
- Length is only encoded when necessary
  (e.g. no type size limitation by the ASN.1 specification)
- Optional components of a sequence or set is indicated in a bitmap preceding the value encoding

[Dubuisson 2000]

Packed Encoding Rules (PER)

- Two variants: aligned and unaligned
  - Aligned encoding inserts padding bits to reach the octet length
  - Unaligned encoding does not and is therefore more compact (however, it requires more processing)
- More compact encoding than BER, and also smaller processing overhead.

[Dubuisson 2000]

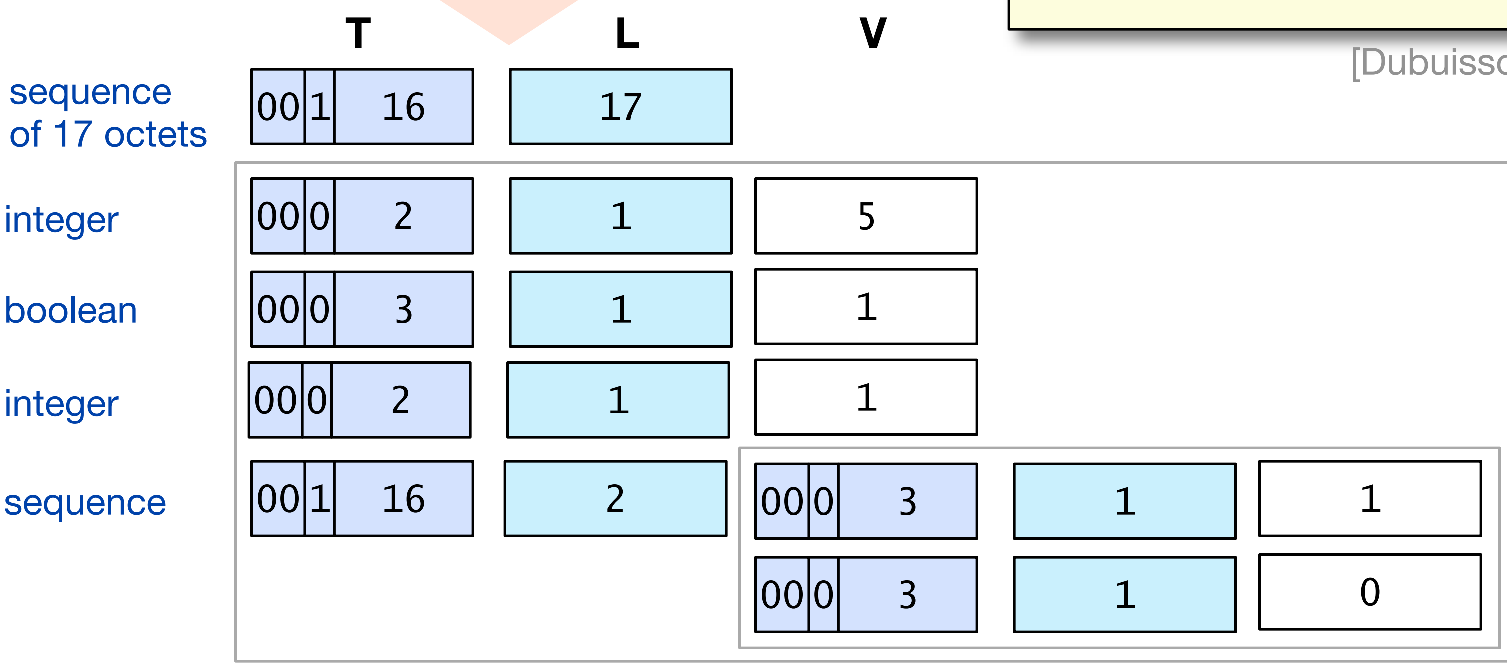
BER vs. PER, Example (1)

```
SEQUENCE {
  a INTEGER (0..7),
  b BOOLEAN,
  c INTEGER (0..3),
  d SEQUENCE { d1 BOOLEAN,
               d2 BOOLEAN }
}
```

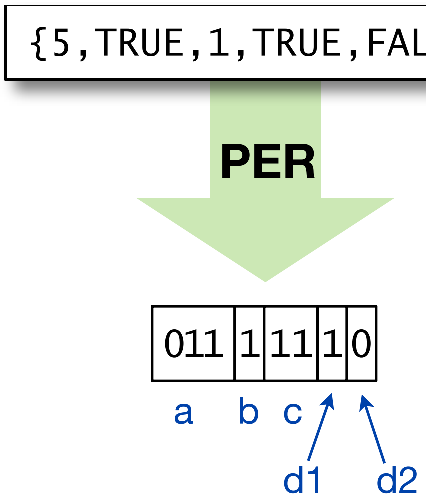
[Dubuisson 2000]

{5,TRUE,1,TRUE,FALSE}

BER



BER vs. PER, Example (2)




```
SEQUENCE {
  a INTEGER (0..7),
  b BOOLEAN,
  c INTEGER (0..3),
  d SEQUENCE { d1 BOOLEAN,
               d2 BOOLEAN }
}
```

[Dubuisson 2000]

- PER can reduce the encoding length significantly in this example
- Note, that the data format is not extensible

PER Encoding Example

- Sequence with extension additions and optional fields
- Extensions have to be considered in the encoding by additional indicators

```
Ax ::= SEQUENCE {
 a INTEGER (250..253),
 b BOOLEAN,
 c CHOICE {
  d INTEGER,
  ...,
  [[
     e BOOLEAN,
     f IA5String
  ]],
  ...
 },
 ...,
 [[
    g NumericString (SIZE(3)),
    h BOOLEAN OPTIONAL
 ]],
 ...,
 i BMPString OPTIONAL,
 j PrintableString OPTIONAL
}
```

[ITU X.961]

PER Encoding Example

```
Ax ::= SEQUENCE {
 a INTEGER (250..253),
 b BOOLEAN,
 c CHOICE {
  d INTEGER,
  ...,
  [[
     e BOOLEAN,
     f IA5String
  ]],
  ...
 },
 ...,
 [[
     g NumericString
(SIZE(3)),
     h BOOLEAN OPTIONAL
 ]],
 ...,
 i BMPString OPTIONAL,
 j PrintableString
OPTIONAL
}
```

```
{ a  253,  b  TRUE,  c  e : TRUE,  g  "123",  h  TRUE }
```

PER (aligned encoding)

1	00	11	1	1	0000000xx	00000001	1xxxxxxx	0000000	1	00000010	1	0010 0011 0100	1xx

[ITU X.961]

PER Encoding Example

```
Ax ::= SEQUENCE {
 a INTEGER (250..253),
 b BOOLEAN,
 c CHOICE {
  d INTEGER,
  ...,
  [[
     e BOOLEAN,
     f IA5String
  ]],
  ...
 },
 ...,
 [[
    g NumericString
(SIZE(3)),
    h BOOLEAN OPTIONAL
 ]],
 ...,
 i BMPString OPTIONAL,
 j PrintableString
OPTIONAL
}
```

[ITU X.961]

1	extension additions present
00	optional fields i,j present
11	a = 253
1	b = TRUE
1	extension addition in c's choice
0000000xx	selection of choice c.e
00000001	length of c.e
1xxxxxxx	c.e = TRUE
0000000	number of extension additions
1	1st extension addion is present
00000010	length of extension addition
1	optional value h is present
0010 0011 0100	g=123
1xx	h=TRUE

```
{ a  253,  b  TRUE,  c  e : TRUE,
  g  "123",   h  TRUE }
```

XML Encoding Rules (XER)

- Basic Idea: Delimit ASN.1 values with XML markups

ASN.1

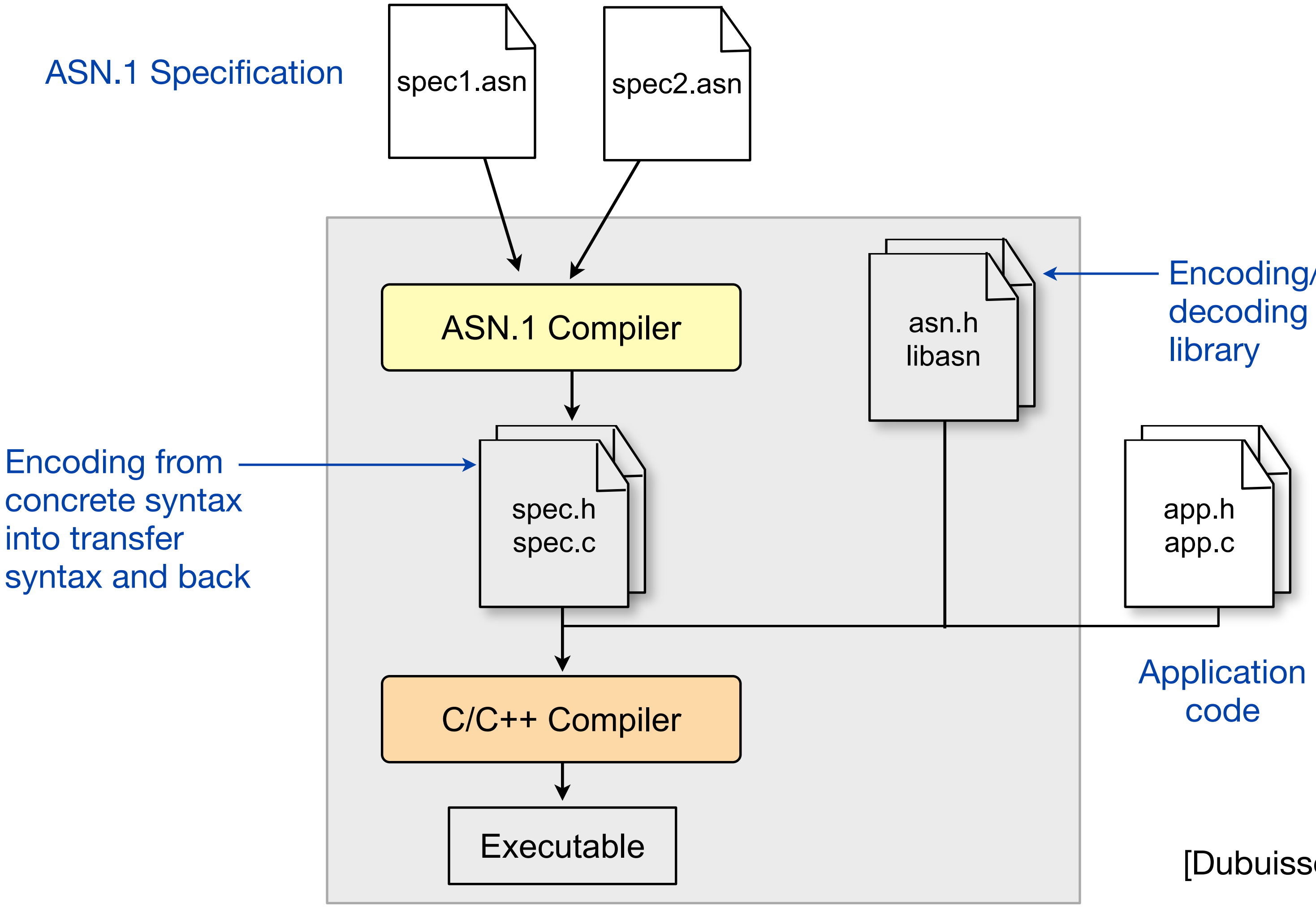
```
Record ::= SEQUENCE {
  name     PrintableString (SIZE (0..40)),
  age      INTEGER,
  belief   ENUMERATED { win(0), linux(1),
                        mac(2) }
}
```

XML

```
<Record>
  <name></name>
  <age></age>
  <belief></belief>
</Record>
```

[Dubuisson 2000]

ASN.1 Compiler



[Dubuisson 2000]

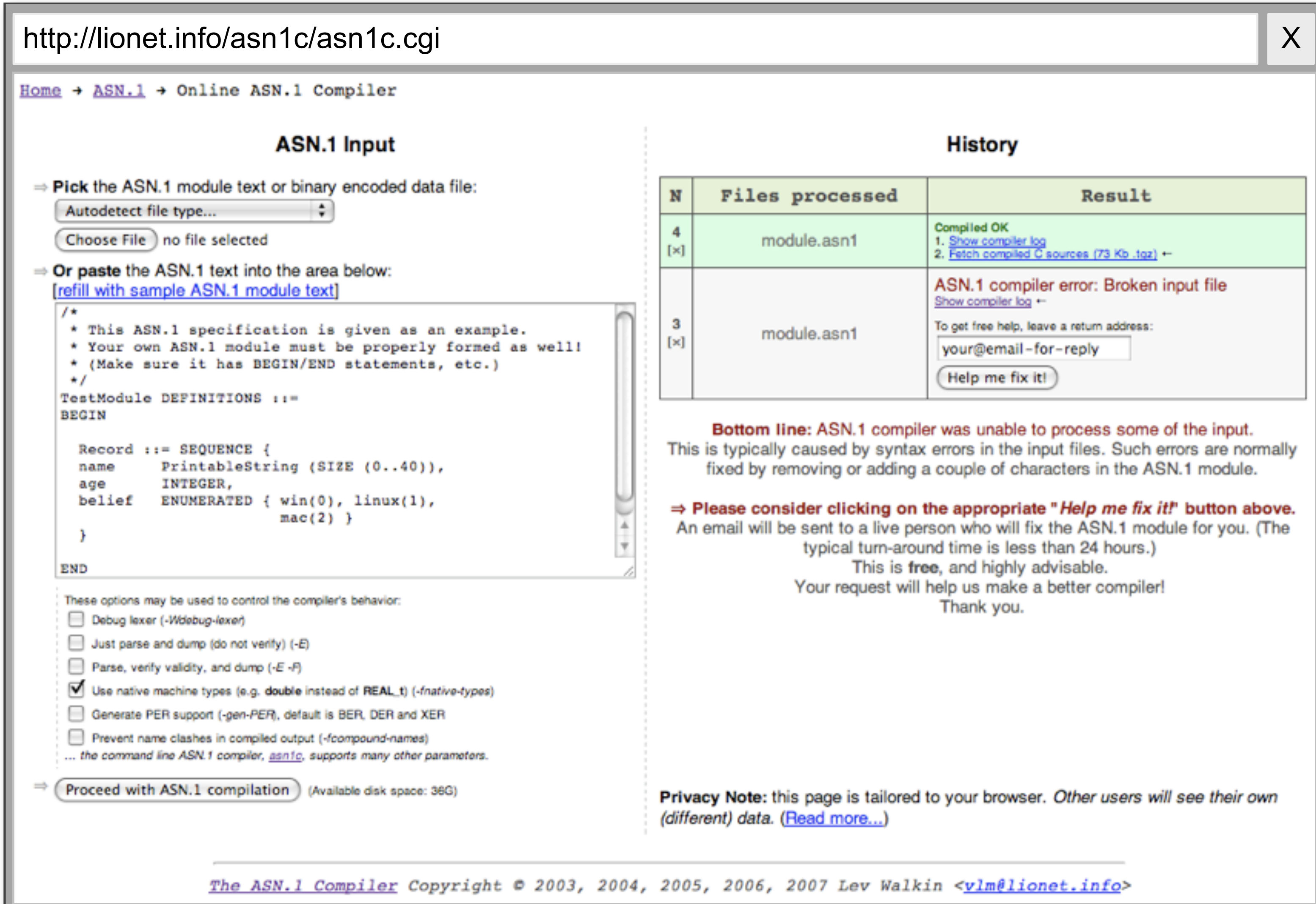
ASN.1 Compilers

- Many commercial tools

- **asn1c** - an open source ASN.1 compiler by Lev Walkin (http://lionet.info/asn1c/)
- converts ASN.1 specifications into C/C++
- creates data structures and functions for encoding and (de)serialization

- **Online version**: http://lionet.info/asn1c/asn1c.cgi

asn1c online compiler



asn1c Example

ASN.1 specification (record.asn)

```
Record ::= SEQUENCE {
  name      PrintableString (SIZE (0..40)),
  age       INTEGER,
  belief    ENUMERATED { win(0), linux(1),
                         mac(2) }
}
```

asn1c output (record.h)

```
/* Dependencies */
typedef enum belief {
    belief_win   = 0,
    belief_linux = 1,
    belief_mac   = 2
} e_belief;

/* Record */
typedef struct Record {
    PrintableString_t  name;
    long     age;
    long     belief;
} Record_t;
```

ASN.1 Review

- Flexible and powerful notation
- Compilable
- ASN.1 has become an important standard in telecommunications, e.g. used by ETSI, ITU, 3GPP
- ASN.1+BER: extensible, but lengthy encoding
- ASN.1+PER yields a quite compact encoding
- However, there are examples, where a manual encoding is more compact

Data Formats Review (1)

- **Box Notation**
  - Intuitive, less flexible
- **ABNF (RFC 2234)**
  - simple, readable, extensible, designed for text encoding
  - less suitable for complex data structures
- **CSN.1**
  - compact notation
- **ASN.1**
  - important standard in telecommunications
  - can be validated and compiled
  - comprehensive tool support

Data Formats Review (2)

- **ASN.1 versus CSN.1**
  - ASN.1 defines data structures similar to definitions in C
  - Encoding rules define valid encodings
  - En-/decoder convert messages into local data structures

  - CSN.1 defines valid encoded bit streams
  - Bit stream is decoded by a parser that invokes a defined function whenever it recognizes a valid element

Data Formats Review

- Comparison of ASN.1, CSN.1 and Tabular notation (presented by Telecom Modus within a 3GPP WG meeting)

	CSN.1	ASN.1 + unaligned PER	ASN.1 + BER	Tabular
Compactness (ranking)	1st	2nd	4th	3rd
Extensibility (ranking)	4th	3rd	1st	2nd
Optional values	yes	yes	yes	yes
Default values	no	yes	yes	yes
Partial decoding	yes	yes	yes	yes

[TSG-RAN Working Group 2 Meeting March 1999]

Lessons learned

- There are standardized methods for data type definition with tool support
- The corresponding en-/decoder can be automatically generated
- The choice of the method depends on the requirements (compact encoding versus extensibility)