Internetanwendungstechnik

Externe Datenrepräsentation

Gero Mühl

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Kommunikations- und Betriebssysteme (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

Motivation

- Sockets unterstützen per se nur den Transport von Byteströmen bzw. Bytearrays

- Bei der Nutzung von Sockets muss der Programmierer daher von Hand
  - Die Struktur der Datenströme bzw. Nachrichtentypen festlegen
  - Die zu versendenden Applikationsdaten in Bytes umwandeln (aka. *Serialisierung, Marshalling*)
  - Die empfangenen Bytes wieder zurück in Applikationsdaten umwandeln (aka. *Deserialisierung, Unmarshalling*)
  - Bei verketteten, im Speicher nicht zusammenhängenden, Datenstrukturen ist dies nicht trivial („*Deep Copy*“ vs. „*Shallow Copy*“)

- Heterogenität macht alles noch deutlich komplexer!

Motivation

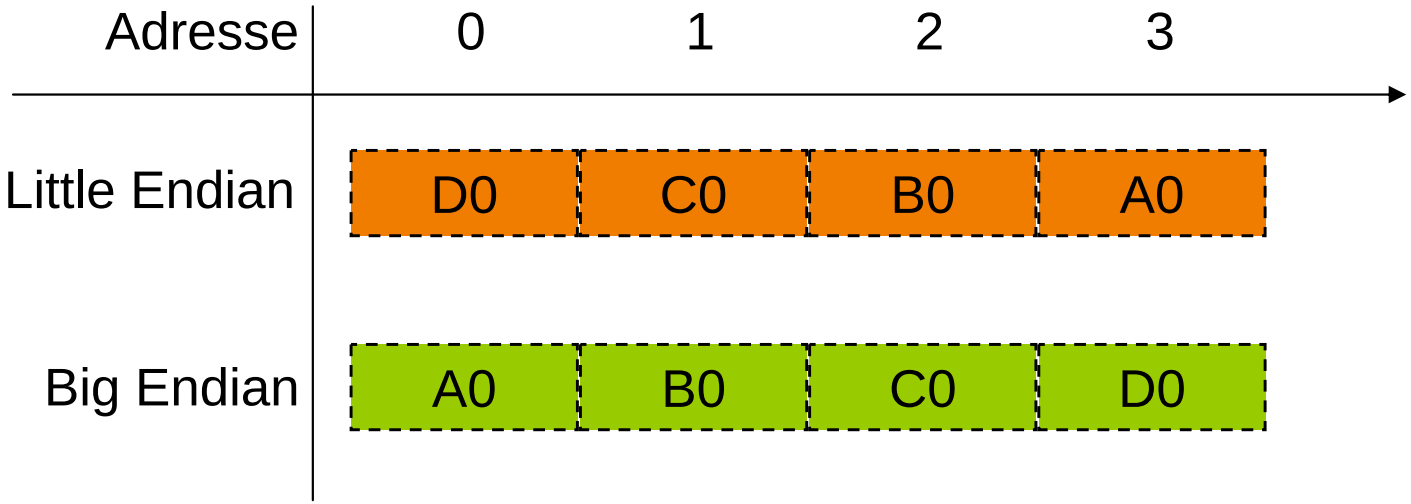
- Sinnvoll wäre eine Bibliothek mit Routinen
  - zur Serialisierung und Deserialisierung von
    - primitiven Datentypen,
    - zusammengesetzte Datentypen (z.B. Arrays) und
    - verketteten Datenstrukturen (z.B. Listen und Graphen)
  - zum Senden und Empfangen ganzer Nachrichten (inklusive der enthaltenen Datenstrukturen) mit automatischer Fragmentierung beim Sender und Defragmentierung beim Empfänger

- Die Bibliothek müsste für verschiedenste Hardware, Betriebssysteme und Programmiersprachen verfügbar sein, um auch in heterogenen Systemen anwendbar zu sein

⇒ Externe Datenrepräsentationen

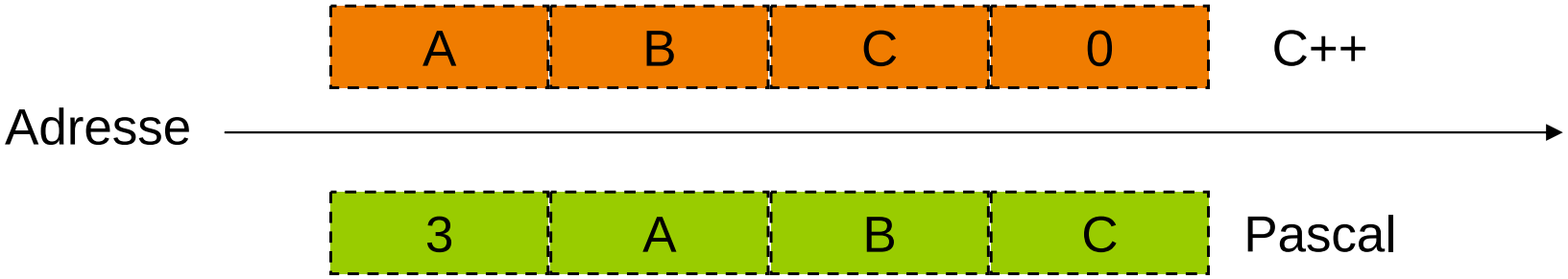
Heterogenität – Hardware-Architekturen

- Verschiedene Hardware-Architekturen speichern die Bytes eines Wortes (hier 32 Bit) in unterschiedlicher Reihenfolge
- ***Little Endian*** (z.B. Intel x86) vs. ***Big Endian*** (z.B. Motorola G4) am Beispiel des Wertes 0xA0B0C0D0



Heterogenität – Programmiersprachen

- Verschiedene Programmiersprachen speichern Datentypen unterschiedlich
- Z.B. String „ABC“

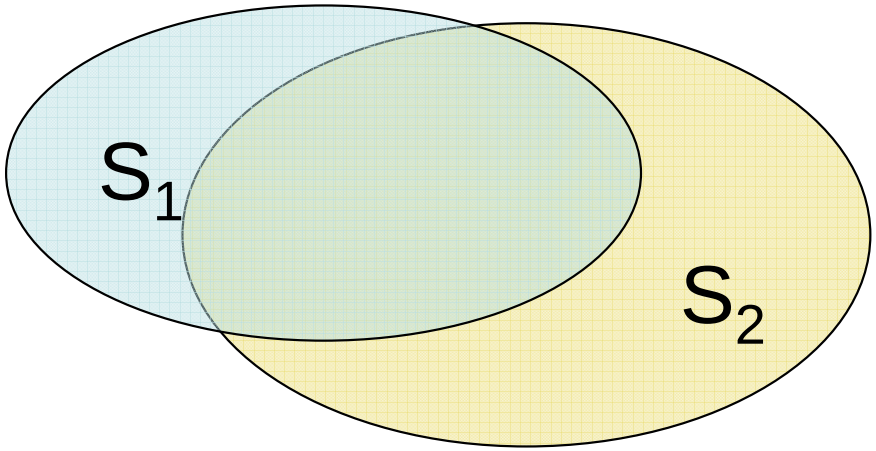


→ Beim Datenaustausch muss passend konvertiert werden

- Meist definiert man ein kanonisches Transferformat

Transformation zwischen Darstellungen

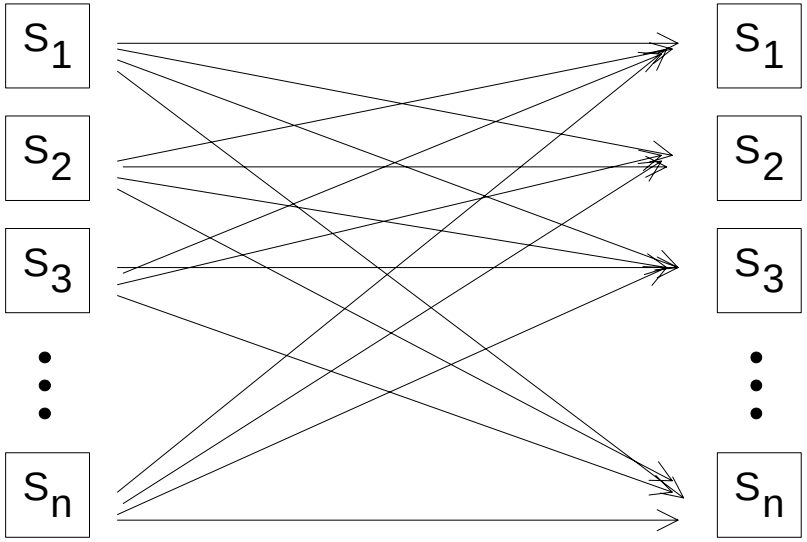
- Vorhandene Heterogenität der Repräsentationen
- ⇒ Transformation zwischen verschiedenen Darstellungen notwendig
- Hierbei gehen evtl. Informationen verloren
- 2 mögliche Realisierungen



Paarweise Transformation

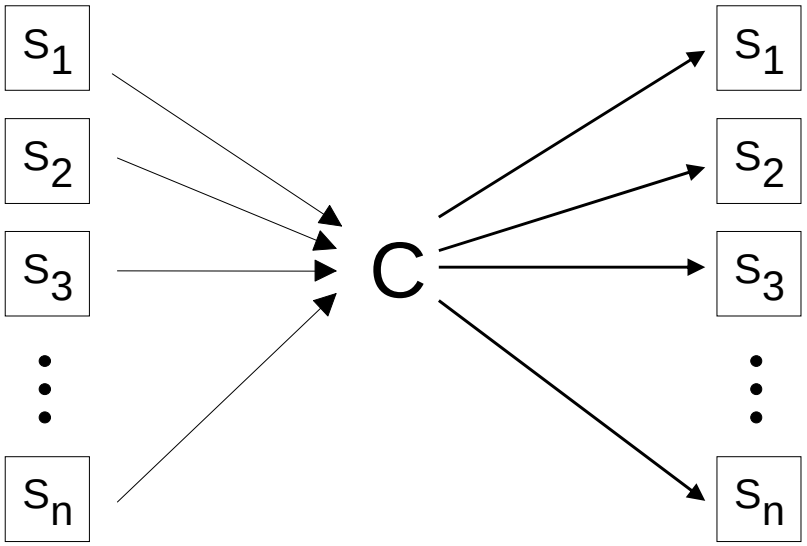
- Paarweise Transformation zwischen *n* verschiedenen lokalen Repräsentationen
- Entweder Sender oder Empfänger muss transformieren
  - „Sender makes it right“ vs. „Receiver makes it right“
  - → eine Transformation pro Kommunikation

→ n^2-n verschiedene Transformationen



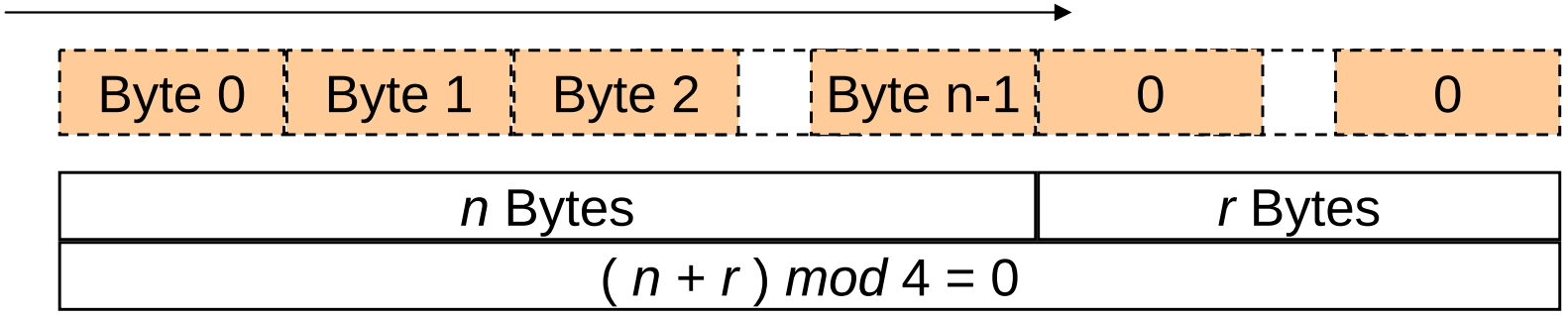
Kanonische Datenrepräsentation

- Verwendung einer kanonischen Datenrepräsentation *C*
- Vorteil: Keine Informationen über die Repräsentation des Gegenübers notwendig
- 2 Transformationen pro Kommunikation notwendig (jeweils von S_i nach *C* und von *C* nach S_j) → Mehr Aufwand
- Nur *n* verschiedene Transformationen für jede der beiden Richtungen (nach *C*, von *C*) notwendig



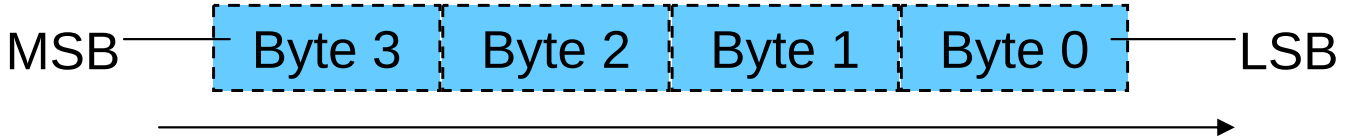
XDR (SUN RPC)

- eXternalDataRepresentation (XDR) definiert in [RFC 1014]
- Es wird stets Byte i vor Byte i+1 gesendet
- Jeweils 1 Byte muss korrekt interpretiert werden
- Es werden keinerlei Meta-Informationen (wie z.B. Typinformationen) gesendet!
- Daten werden in Blöcken zu Vielfachen von 4 Bytes kodiert

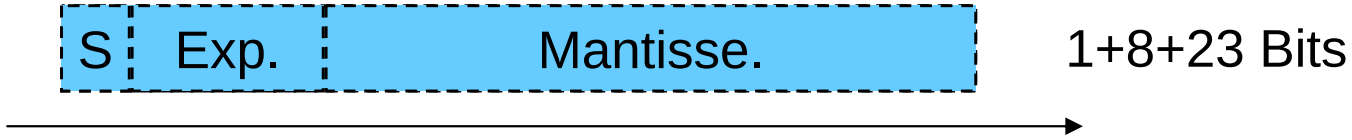


XDR (SUN RPC)

- Beispiele für unterstützte Datentypen und deren Repräsentation
  - 32 Bit Signed Integer (Big Endian-Format, 2er Komplement)

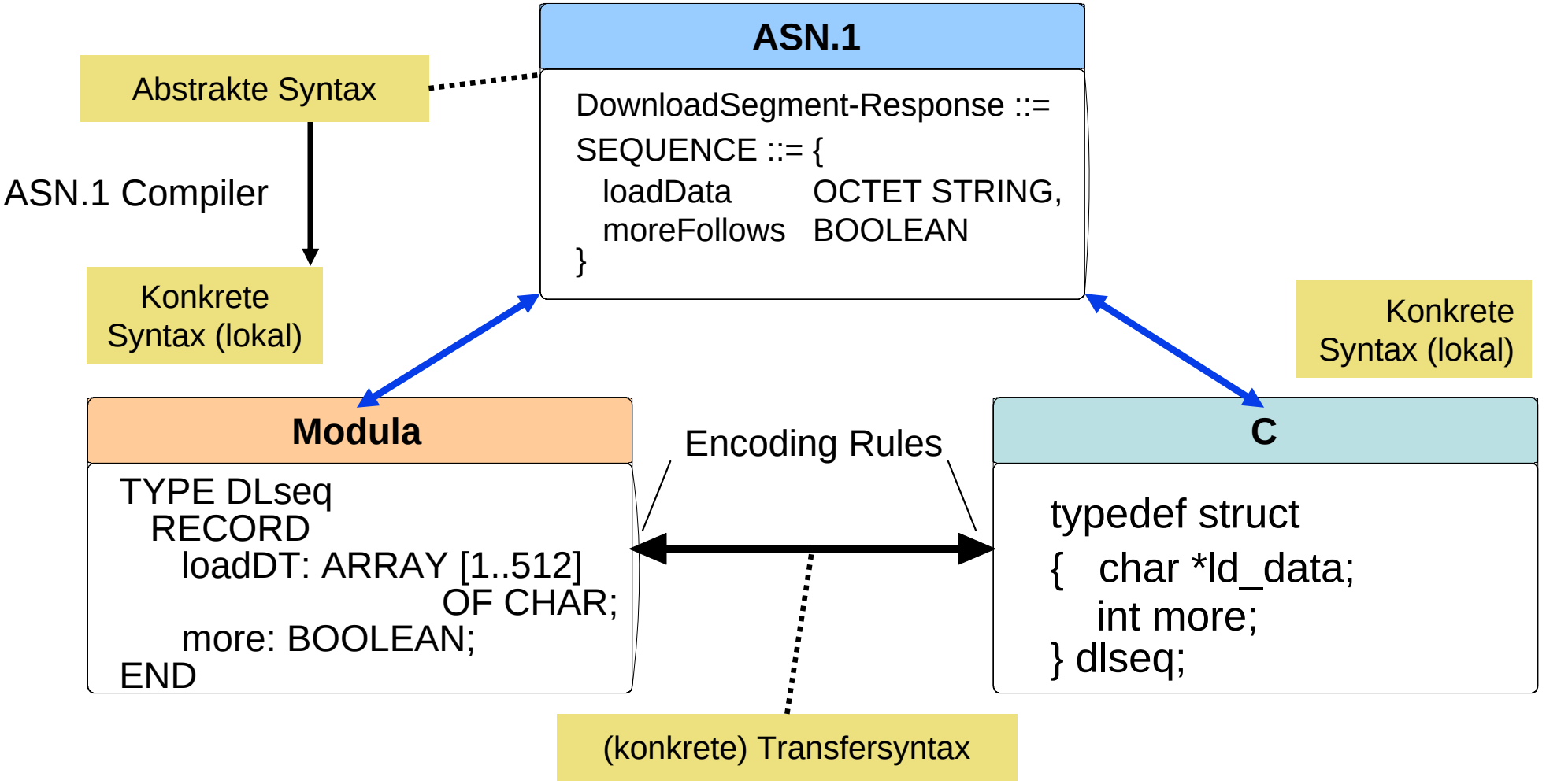


  - 32 Bit Single Precision Float (IEEE Standard-Format)



- Viele weitere Datentypen werden unterstützt:
  - Boolean, Opaque, Enumeration, Structure, Array, Union, ...

ISO Standard Abstract Syntax Notation.1 (ASN.1)



ASN.1

- Abstract Syntax
  - Beschreibt die allgemeine Struktur von Daten (unabhängig von Programmiersprachen etc.)
- Concrete Syntax
  - Beschreibt die lokale Darstellung der Daten (abhängig von Programmiersprache etc.)
- Transfer Syntax
  - Beschreibt die Darstellung der Daten bei der Übertragung (z.B. als Tripel <Datentyp, Byteanzahl, Bytes>)
- Encoding Rules (z.B. Basic Encoding Rules (BER))
  - Beschreiben die Umwandlung von der konkreten in die Transfer Syntax und vice versa

ASN.1 – Compiler

- Überprüft die Syntax einer Abstract Syntax Specification
- Erzeugt aus der Abstract Syntax die Concrete Syntax (z.B. für C)
- Erzeugt die Routinen für Codierung/Decodierung (z.B. in C)

ASN.1 Beispiel – Order Processing [DUB00]

```
Order ::= SEQUENCE {
   header Order-header,
   items SEQUENCE OF Order-line}
```

Typdefinition durch ::=
Typen beginnen mit Großbuchstaben
Komponentennamen mit Kleinbuchstaben

```
Order-header ::= SEQUENCE {
   number Order-number,
   date Date,
   client Client,
   payment Payment-method}
```

Sequenz von Komponenten

```
Order-number ::= NumericString (SIZE (12))
```

String aus Zahlen mit fester Länge

```
Date ::= NumericString (SIZE (8)) -- DDMMYYYY
```

Kommentar

ASN.1 Beispiel – Order Processing

```
Client ::= SEQUENCE {
   name PrintableString (SIZE (1..40)),
   street PrintableString (SIZE (1..50)) OPTIONAL,
   postcode NumericString (SIZE (10)),
   town PrintableString (SIZE (1..30)),
   country PrintableString (SIZE (1..20))
   DEFAULT default-country }

default-country PrintableString ::= "France“

Payment-method ::= CHOICE {
   check NumericString (SIZE (15)),
   credit-card Credit-card,
   cash NULL}

Credit-card ::= SEQUENCE {
   type Card-type,
   number NumericString (SIZE (20)),
   expiry-date NumericString (SIZE (6)) -- MMYYYY -- }

Card-type ::= ENUMERATED { cb(0), visa(1),
   eurocard(2), diners(3) }
```

variable beschränkte Länge

optionale Komponenten

default Komponenten

alternative Komponenten

NULL Wert

Aufzählungstyp

ASN.1 Beispiel – Order Processing

```
Order-line ::= SEQUENCE {
   item-code Item-code,
   quantity Quantity,
   price Cents }

Item-code ::= NumericString (SIZE (7))

Cents ::= INTEGER

Quantity ::= CHOICE {
   units [0] INTEGER,
   millimeters [1] INTEGER,
   milligrams [2] INTEGER }

Module-order DEFINITIONS
   AUTOMATIC TAGS ::=
   BEGIN
   ...
   END
```

Mehr Informationen durch zusätzliche Typdefinition

Tags notwendig bei CHOICE, SET und optionalen Parametern in SEQUENCE, falls Komponenten sonst wegen gleichem Typ nicht unterschieden werden könnten. Alternative: AUTOMATIC TAGS

Moduldefinition

ASN.1 Beispiel – Order Processing

```
Protocol DEFINITIONS AUTOMATIC TAGS ::=
   BEGIN
   IMPORTS Order, Delivery-report, Item-code,
      Quantity, Order-number FROM Module-order;
   PDU ::= CHOICE {
      question CHOICE {
         question1 Order,
         question2 Item-code,
         ... },
      answer CHOICE {
         answer1 Delivery-report,
         answer2 Quantity,
         ... }
   }
   END
```

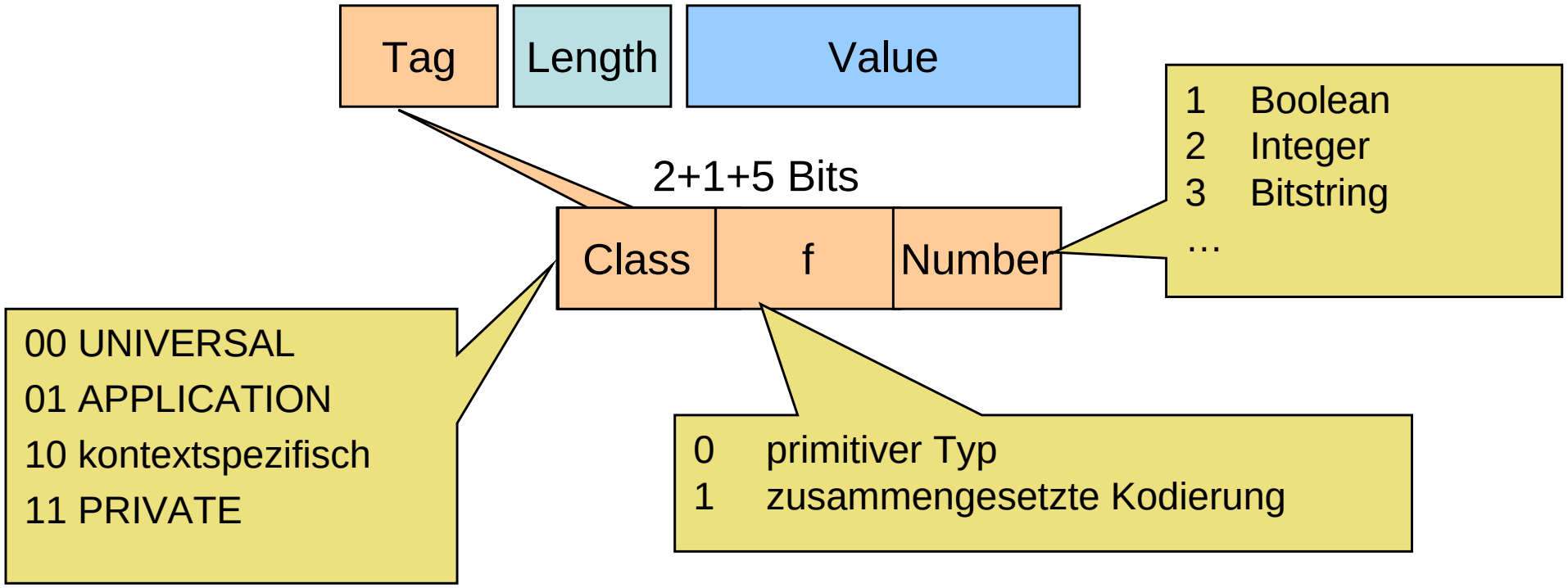
Importieren von Typdefinitionen

PDU = Protocol Data Unit

... = Erweiterungen in zukünftigen Versionen möglich

Basic Encoding Rules (BER) for ASN.1

- Daten werden als Folge von Oktetts (=Bytes) kodiert
  - Nummerierung der Bits: 8 7 6 5 4 3 2 1
  - höchste Wertigkeit: Bit 8
- TLV-Codierung

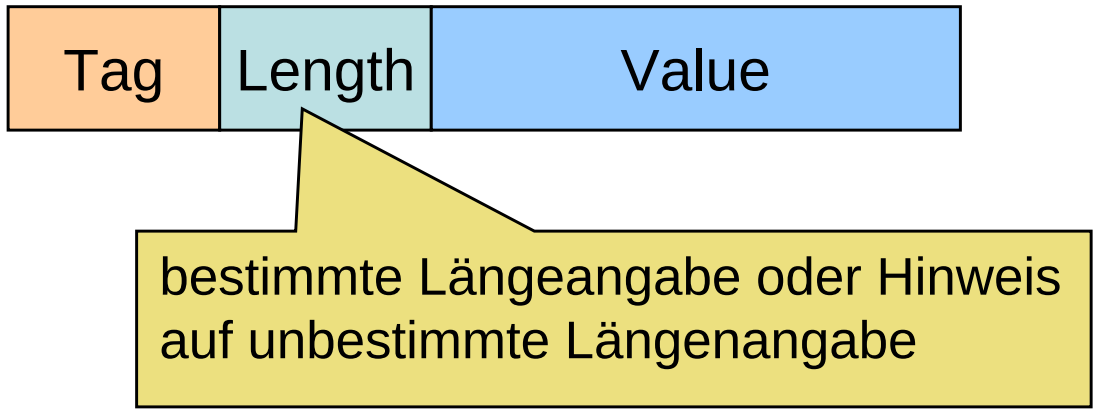


BER – Typnummer im Tag-Feld

> n≤30	Kodierung in einem Oktett	ccfx xxxx
> n>30	1. Oktett	ccf1 1111
	2. Oktett	1xxx xxxx
	3. Oktett	1xxx xxxx
	…	
	n. Oktett	0xxx xxxx

BER – Längenangabe

- Länge-Feld: gibt die Länge des Wert-Feldes an
- Zwei Arten der Längenangabe
  - Bestimmt
  - Unbestimmt



BER – Längenangabe

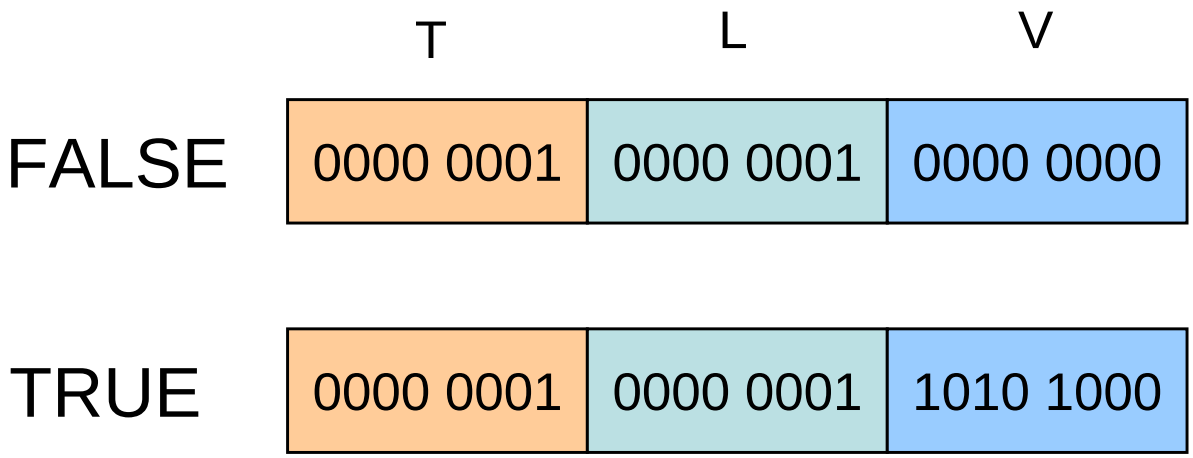
- Bestimmte Form der Längenangabe
  - Länge ≤ 127 ein Oktett 0**xxx xxxx**
  - Länge > 127 1. Oktett 1**nnn nnnn** (**n** ≤ 126 Oktetts für die Länge)
    2. Oktett **xxxx xxxx**
    …
    n+1. Oktett **xxxx xxxx**

- Unbestimmte Form der Längenangabe

  1. Oktett 1000 0000
  2. Oktett (Nutzdaten)
  .....
  n. Oktett (Nutzdaten)
  n+1. Oktett 0000 0000 (end of contents –
  n+2. Oktett 0000 0000 EOC)

BER – Beispiele

- Zwei Boolean Werte
  - FALSE = 0000 0000
  - TRUE ≠ 0000 0000
- Kodierung mit TLV



BER – Beispiele

- INTEGER (im 2er-Komplement)

78	0000 0010	0000 0001	0100 1110	
-296	0000 0010	0000 0010	1111 1110	1101 1000

- SEQUENCE S ::= SEQUENCE { p1 Typ1, p2 Typ2 }

00 1 10000	x..Länge..x	Tag Typ 1	Länge p1	Wert p1
Tag Typ 2	Länge p2	Wert p2		

oder

00 1 10000	1000 0000	Tag Typ 1	Länge p1	Wert p1
Tag Typ 2	Länge p2	Wert p2	0000 0000	0000 0000

Kritik der TLV-Kodierung

- ☺ ASN.1 sehr mächtig (z.B. SET, CHOICE, OPTIONAL)
- ☺ BER bietet entsprechend vielfältige Möglichkeiten

- 👎 Kodierung sehr aufwendig (Ressourcenverbrauch ☹)
  - > viele Anwendungen kennen die Daten, so dass Tag und Länge unnötig sind
  - > variable Länge, einzelne Bits, variabel lange Integer, etc. orientieren sich nicht an den Anforderungen der Rechnerhardware (ineffizient, aufwendige Verarbeitung)
- 👎 Mächtigkeit geht vor Schnelligkeit („OSI vs. TCP/IP“)

Java Object Serialization

- Erlaubt die streambasierte Übertragung serialisierter Objekte
  - Z.B. über TCP- oder UDP-Sockets oder Parameter bei Java RMI
- Klassen die serialisiert werden sollen, müssen das Markerinterface java.io.Serializable implementieren (gilt auch für alle Klassenfelder)
- Zugriff auf Implementierungen der Klassen notwendig
- Benötigt keinen Klassen-spezifischen Code
  - Benutzt die Reflektions-Mechanismen von Java
- Binäres Format mit Meta-Informationen
- Kommt auch mit zyklischen Datenstrukturen zurecht
  - Jedes Objekt wird einmal geschrieben und ihm wird eine Referenz zugewiesen
  - Rückwärtsverweise auf Referenzen für bereits geschriebene Objekte

Java Object Serialization

- Spezielle Darstellungen für null Objekte, neue Objekte, Klassen, Arrays, Strings und Rückwertsverweise
- Es ist wichtig zu wissen, von welcher Klasse ein Objekt ist!
  - Stream enthält auch Informationen über alle geschriebenen Klassen, deren Version und deren Superklassen (falls serialisierbar) → Instanzen der Klasse ObjectStreamClass
  - Ermöglicht dem Empfänger auch, die benötigten Klassen zu laden
- Ein Reset des Streams löscht die Informationen über bereits geschriebene Objekte (und Klassen)
  - Ermöglicht die neuerliche Übertragung von (z.B. geänderten) Objekten
  - Ineffizient, da Klasseninformationen auch gelöscht werden
- Objekt-Implementierungen können Serialisierung durch Implementierung spezieller Methoden beeinflussen

Klonen mittels Serialisierung

```
public class ObjectCloner {

   public static byte[] toByteArray(Object obj) throws IOException{

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ObjectOutputStream oos = new ObjectOutputStream(baos);
      oos.writeObject(obj);
      oos.flush(); baos.flush();
      return baos.toByteArray();
   }

   public static Object fromByteArray(byte[] ba)
                        throws IOException, ClassNotFoundException{

      ByteArrayInputStream bais = new ByteArrayInputStream(ba);
      return new ObjectInputStream(bais).readObject();
   }
```

Klonen mittels Serialisierung

```
public static Object clone(Object obj) {
    Object clone = null;
    try {
        clone = fromByteArray(toByteArray(obj));
    } catch (Exception exception) {
        exception.printStackTrace();
    }
    return clone;
}

public static void main(String[] args) {
    Date d = new Date();
    Date d2 = (Date)ObjectCloner.clone(d);
    System.out.println(d==d2); // false
}
}
```

Gleichheitstest mittels Serialisierung

```
public class ObjectComparer {

   public static boolean equals(Object obj1, Object obj2) {

      boolean b = false;
      try {
         byte[] ba1 = ObjectCloner.toByteArray(obj1);
         byte[] ba2 = ObjectCloner.toByteArray(obj2);
         return Arrays.equals(ba1,ba2);
      } catch (Exception exeption) { // ignored }
      return b;
   }

   public static void main(String[] args) {

      Date d = new Date();
      Date d2 = (Date)ObjectCloner.clone(d);
      System.out.println(ObjectComparer.equals(d,d2)); // true
   }

}
```

XML-basierte Serialisierung

- eXtensible Markup Language (XML)
- XML-Dokumente
  - sind hierarchisch strukturiert
  - enthalten Informationen über die eigene Struktur
  - basieren auf Klartext
    - „Menschen-lesbar“
    - Wiederherstellung archivierter Dokumente leichter
    - Größere Länge als binäre Formate
- XML-Parser und andere XML-Tools können genutzt werden

XML-basierte Serialisierung

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE koml SYSTEM "http://koala.ilog.fr/XML/koml12.dtd">
<koml version='1.2'>
  <classes>
    <class name='java.net.URL'>
      <field name='port' type='int'/>
      <field name='file' type='java.lang.String'/>
      <field name='host' type='java.lang.String'/>
      <field name='protocol' type='java.lang.String'/>
    </class>
  </classes>
  <object class='java.net.URL' id='i2'>
    <value type='int' name='port'>80</value>
    <value type='java.lang.String‘ name='file'>/koala/XML/</value>
    <value type='java.lang.String‘ name='host'>www.inria.fr</value>
    <value type='java.lang.String' name='protocol'>http</value>
  </object>
</koml>
```

Literatur

1. Sun Microsystems, Inc. *Java Object Serialization Specification 1.5.0*, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serial-title.html.

2. O. Dubuisson. *ASN.1 Communication between Heterogeneous Systems*. Morgan Kaufmann Publishers, 2000. translated by Philippe Fouquart. http://asn1.elibel.tm.fr/en/book/, http://www.oss.com/asn1/booksintro.html

3. M. Campione, K. Walrath, and A. Huml. *The Java Tutorial. The Java Series*. Addison-Wesley, 3rd edition, 2001. http://java.sun.com/docs/books/tutorial/

Fragen?