

Internet Application Protocols

For more info see <http://dsv.su.se/jpalme/abook/>

Copyright © Jacob Palme 2000, 2001, 2002, 2003

Copyright conditions: This document may in the future become part of a book. Copying for non-commercial purposes is allowed on a temporary basis. At some time in the future, the copyright owner may withdraw the right to copy the text. Check for the current copyright conditions at the web site of the author, <http://dsv.su.se/jpalme/abook/>.

This document contains quotes from various IETF standards. These standards are copyright (C) The Internet Society (date). All Rights Reserved. For those quotes, the following copyright conditions apply:

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

Publisher
Not yet published • City

Preliminary Table of Contents

Contents

Introduction

- Overview of the most common Internet protocols and services
- Understanding layering
- Ports and protocols
- Some registered port numbers
- Architectures
- Protocols: Two entities talking to each other using a controlled language
- Ending a connection
- Connection retention
- Chaining, referral, multicasting
- Protocol extension problem
- Intermediaries
- Replication
- IETF standards terminology
- The IETF Golden rules
- Names in the Internet, the Domain Naming System (DNS)
- Basic security techniques
- 1.1 URL, Uniform Resource Locator
- 1.2 URL schemes standardized in RFC 1738
- 1.3 Character set in URLs (not in referenced document)
- Encoding of unsafe characters in URL-s
- 1.4 Top-level URL Syntax:
Common Internet Scheme Syntax
- 1.5 Relative URLs
- 1.6 HTTP URL syntax
- Example of an HTTP Query URL

1.7	Reference to fragments of an HTML document Part of the URL?	
1.8	URL, URI, URN, URC	
Preliminary Table of Contents		iii
1. Introduction to Coding		7
1.1.	Why is coding important?	8
1.2.	Character sets	10
1.1.1.	The UTF-8 encoding of ISO 10646	12
1.1.2.	Limited subsets of character sets	12
1.3.	Textual and binary encoding	13
1.1.3.	Encoding of information structure	14
1.1.4.	Encoding of the start and end of data elements	15
1.1.5.	Encoding of binary data with textual encoding	17
1.1.6.	More About Encoding of Information Structure	17
2. Augmented Backus-Naur Form, ABNF		21
1.1.7.	Linear White Space	22
1.1.8.	Versions of ABNF	23
1.4.	An overview of ABNF syntax constructs	24
1.1.9.	Either-or construct	24
1.1.10.	A series of elements of the same kind	24
1.1.11.	Comments in ABNF	25
1.1.12.	Linear White Space (LWSP)	25
1.1.13.	Comma-separated list	25
1.1.14.	ABNF syntax rules, parentheses	26
1.1.15.	Optional elements	26
1.5.	Examples of use of ABNF	29
1.1.16.	Examples of values matching the syntax in example 4 above:	29
1.1.17.	Example 7 (from RFC822):	30
1.1.18.	Examples of value matching the syntax in example 7 above	30
1.6.	RFC 822 lexical scanner specified in ABNF	30
3. Abstract Syntax Notation, ASN.1		32

1.7.	ASN.1 basic	37
1.1.19.	ASN.1 value notation	37
1.1.20.	ASN.1 terminology	37
1.1.21.	Pre-defined, built-in types in ASN.1	38
1.1.22.	Comments	39
1.1.23.	Format of identifiers	39
1.8.	Simple Types	39
1.1.24.	Integer Type	39
1.1.25.	Subtypes	40
1.1.26.	Boolean Type	41
1.1.27.	Enumerated	42
1.1.28.	Real Type	42
1.1.29.	Bit String	43
1.1.30.	Subtypes	43
1.1.31.	Variants of Bit Strings	44
1.1.32.	Octet String Type	46
1.1.33.	Null Type	46
1.1.34.	Examples of the Use of Size	47
1.1.35.	Character String Types	47
1.9.	Structured types	48
1.1.36.	Inner subtyping	49
1.1.37.	Choice Type	52
1.1.38.	Any Type	53
1.1.39.	Tags	54
1.1.40.	Explicit and Implicit tags	57
1.10.	Special types and Concepts	61
1.1.41.	Time Types	61
1.1.42.	Use of Object Identifiers, Any, External	61
1.1.43.	Object Descriptor and External types	64
1.1.44.	Modules	65
1.11.	Encoding Rules	67
1.1.45.	Basic Encoding Rules (BER)	67
1.1.46.	The Tag or Identifier field	68
1.1.47.	The Length Field in BER	69
1.1.48.	The BER Value Octet	70
1.1.49.	Variants of the encoding of a string with tag	70
1.1.50.	Example of the coding of a SEQUENCE	71
1.1.51.	Different Encoding Rules for ASN.1	73
1.12.	ASN.1 compilers	74

4. HTML and CSS 76

1.13.	(Hypertext Markup Language)	77
1.14.	Cascading Style Sheets (CSS)	79
5.	Extensible Markup Language, XML	82
1.15.	Extensible Markup Language (XML) Introduction	83
1.1.52.	XML versus HTML	84
1.16.	Document Type Definition (DTD)	85
1.17.	XML ELEMENT and its contents	87
1.1.53.	Reserved characters	89
1.1.54.	Empty Elements	90
1.1.55.	Any Specification	90
1.1.56.	Repeated subelements	90
1.1.57.	Choice subelements	92
1.18.	Attributes of XML elements	92
1.1.58.	Use attributes or subelements?	95
1.19.	Formatting XML layout when shown to users (CSS and XLST)	97
1.20.	XML special problems and methods	100
1.1.59.	Putting binary data into XML encodings	100
1.1.60.	Reusing DTD information	100
1.1.61.	Entities	101
1.1.62.	Name Spaces	101
1.1.63.	XLinks and XPointers	102
1.1.64.	Processing instructions	103
1.1.65.	Standalone declarations	103
1.1.66.	XML validation	103
1.1.67.	XHMTL	104
1.21.	A comparison of ABNF, ASN.1-BER/PER and DTD-XML	104
1.1.68.	Comparison RFC822-style headings versus XML and ASN.1	108
1.22.	Other Encoding Languages	109
6.	References	110
7.	Acknowledgements	112
8.	Solutions to exercises	114

1. Introduction to Coding

Objectives

This chapter describes why coding is so important, and introduces the problems which coding attempts to solve

Keywords

coding

records

data structures

characters

1.1. Why is coding important?

The underlying network protocols, like the transport layer of TCP/IP, provide a way of sending a sequence of octets (containers with 8 bits, also often called “bytes”) from the sending port to the receiving port. All information must thus be transformed into a sequence of octets. And the protocol will probably not work, unless the sending and receiving computer agree on how to interpret these octets. The procedure of transforming information into a sequence of octets, is known as “coding”. The procedure of transforming information from this sequence of octets to a data structure easily interpreted by the receiving application, is the reverse process, “uncoding”.

Well, if you have defined your data using a struct in C or a set of records in Pascal, like for example the Pascal code below, cannot you just send these structures as they are from one host to another across the network?

```
flightpointer = ^flight;

flight = RECORD
  airline : String[2];
  flightnumber : Integer;
  nextflight : flightpointer;
END;

passenger = RECORD
  personalname : String [60];
  age : Integer;
  weight : Real;
  gender : Boolean;
  usertexts : ARRAY [1..5] OF flightpointer;
END;
```

In a Pascal program, you can send a record, like a “passenger” record in the code above, to a procedure (= function, method) by just making passenger a parameter in the procedure call. Why can you not do the same when two programs on two different computers communicate through the Internet? Well, there are many reasons why this will not work:

1. The String may not be stored in the same way in the sending and receiving computers. For example, many computers store four 8-bit characters in one 32-bit word. This means that the characters are grouped into groups of four characters and stored in a word. But different computers store characters into words in different order. This means that the sending computer may send

A	B	C	D	E	F	G	H
---	---	---	---	---	---	---	---

, but the receiving computer may re-

ceive DCBAHGFE (this has actually happened to me in a development many years ago, which used a protocol between a Unix server and an MSDOS-based PC).

Table 1: Coding of the character “Ä”

Character set	Representation of “Ä” (hexadecimal)
ISO Latin One	C4
Unicode (ISO 10646), UTF-32	000000C4
Unicode, UTF-8 coding	E2C4
CP850 (old MS-DOS)	8E
ISO 6937/1	C861
old Mac OS	80

2. Different computers might store the same character in different ways, i.e. they may use different bit patterns to represent the same character. As an example, Table 1 shows different ways in which the character “Ä”, which is common in the German and Scandinavian languages, might be represented:
3. Different computers store integers in different ways. Some use 16, some 32, some 64 bits to store an integer. And negative integers are stored in two common different ways, the 1-complement and 2-complement notation.
4. Different computers store floating point numbers in different ways. They assign different number of bits to the mantissa and the exponent, and some use 2, some 10, some 16 as the base.
5. Different computers store Boolean values in different ways. Some computers store Boolean values in an octet, where all non-zero values represent TRUE, other computers use just 1 and 0 for TRUE and FALSE.
6. The receiving computer will have problems with the reference (pointer) “flightpointer”, since it cannot access data in the sending computer.

Thus, if one computer sends data in its internal representation, and another computer receives this, believing it to be in the internal representation of the receiving computer, the data will obviously be misinterpreted. It may work in the special case where both computers have the same architecture, which in some cases might work for some small intranets. But a standard for sending data between any kind of computer must specify exactly how data is to be coded.

1.2. Character sets

The character, as you see it when you read it on paper or on a screen, is called a *glyph*. Thus, for example, the glyph for the letter “O” is an vertical ellipse “o”, and the glyph for the digit “0” is a more narrow vertical ellipse “0”. The same glyph may look somewhat different in different fonts, but it is still the same glyph, for example “A”, “A” and “A”. A font might even render a glyph as quite another graphical form, but it is still the same glyph. The Braggadoco font will for example render the letter “O” as “●”.

A *character set* is a set of glyphs combined with information on how each glyph is to be coded into one or more octets. In Internet standards, several different character sets are used, and a common cause of error in Internet programs is that a character is sent using one character set and one encoding, but received believing it to be another character set and/or another encoding.

Many character sets are variants of the Latin character set, based on the letters A to Z. But there are also completely different character sets, like Cyrillic (ГГДД), Arabic (أَلِفْ حَـ دَـ ذَـ), Hebrew (אָ לֶ מֶ נֶ סֶ), Browallia (ᲀ ᲁ ᲂ ᲃ), Japananese (誕 生), Korean (ᄅᆞᆫ ᄇᆞᆯ) and Chinese (字典網).

The same character set can have more than one encoding specified for that character set. There are also additional encodings which some protocols apply to the sequence of bytes from any character set.

The most common character sets in Internet standards are listed in Table 2.

Table 2: The most common character sets

Name	Included characters	Encoding
US-ASCII	This set has 128 characters. 95 of these are printable characters, the rest are control characters like Carriage Return and Line Feed.	Each character is encoded as one 7-bit byte. This is usually sent as an octet, with the first bit always 0.
ISO 646	This is very similar to US-ASCII, but a few of the characters are called national characters, and can be substituted with other characters in different national variants of ISO 646. The following characters may be replaced with other characters in national sets, and their use can thus cause problems, especially in text files	Same encoding as US-ASCII.

Name	Included characters	Encoding
	transported between computers: £ # \$ € @ [] ^ \ ` { } ~	
ISO 8859-1, also known under the name ISO Latin 1	This set has 256 characters, 190 of them are printable, the rest are control characters. It includes US-ASCII plus a number of additional characters suitable for Western European Languages, like Ä, É and ÿ.	Each character is encoded as exactly one octet. This makes the standard easy to process, but reduces the number of possible characters.
ISO 8859-?	There are a number of different variants of ISO 8859 for different languages or language groups. For example, ISO 8859-2 is suitable for most Eastern European Languages using latin character sets, like Hungarian or Polish. Each set has 256 characters, 190 of which are printable. Many of the sets contain US-ASCII as a subset.	Similar encoding to ISO 8859-1.
ISO 10646, also known as Unicode.	This is the character set meant to replace all other character sets. It has space to hold millions of characters. Every character needed in every language are there, or will be added.	ISO 10646 has more than one encoding. The basic encoding is called UTF-32. It uses two octets for each character. There is also room for more space, if needed, through UTF-32, which uses four octets for each character. The mostly used coding of ISO 10646 in Internet protocols is UTF-8 (see page 12). UTF-8 uses between one and four octets for each character. Special for UTF-8 is that all the US-ASCII characters have exactly the same coding as in US-ASCII. This is important, since many Internet protocols use syntax containing US-ASCII characters and words.
ISO 2022	This is an older solution than ISO 10646 to the problem of including characters from many sets in the same message, for example putting an East European name into a text in a West European language, or showing a dictionary between languages with different sets, such as between Russian and English.	ISO 2022 codes a text as segments. Each segment uses one character set, usually one of the ISO 8859 variants or the ISO 646 variants. Special so-called escape-sequences are put

Name	Included characters	Encoding
	In the Internet, ISO 2022 is mostly used by Asian countries like Japan, China or Korea to switch between English and their native character sets.	into the text to switch between segments.

1.1.1. The UTF-8 encoding of ISO 10646

The UTF-8 [RFC 2279] is an encoding of Unicode with the very important property that all US-ASCII characters have the same coding in UTF-8 as in US-ASCII. This means that protocols, in which special US-ASCII characters have special significance, will work, also with UTF-8. They start with the two or four-octet encodings of ISO 10646 (UTF-32):

UTF-32 range (hex.)	UTF-8 octet sequence (binary)
0000 0000–0000 007F	0xxxxxxx
0000 0080–0000 07FF	110xxxxx 10xxxxxx
0000 0800–0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000–001F FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0020 0000–03FF FFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx
	10xxxxxx
0400 0000–7FFF FFFF	1111110x 10xxxxxx ... 10xxxxxx

The high-order bits are set as specified in the second column above. The rest of the bits, marked with x in the second column above, are filled with those bits from the UTF-32 character whose information is not determined by the high-order bits.

1.1.2. Limited subsets of character sets

In addition to the sets listed in Table 2, many Internet standards use a subset of these standards, for special purposes. Examples of some such subsets are shown in Table 3.

Table 3: Subsets used in some standards

Name	Subset description	Where it is used
specials	"(", ")", "<", ">", "@", ",", ";", ":", "\\", "\"", ".", "[", and "]"	Must be coded when used in e-mail addresses.
non-specials	All printable US-ASCII characters except specials and space	Can be used without special coding in e-mail addresses.
Unsafe	"{", "}", " ", "\\", "^", "~", "[", "]" and "`"	Must be coded when used in URLs
Reserved	";", "/", "?", ":", "@", "=", and "&"	These characters have special meaning in URLs, and must be coded if used without the reserved meaning.
Safe	All printable US-ASCII characters except Unsafe and Reserved characters and space.	Can be used without special coding in URLs.

1.3. Textual and binary encoding

There are two main coding methods, the textual and the binary method.

Textual method: All information is transformed to text format before transmission. Examples: *A floating point number* might be transformed to the textual string of characters: `3.14159`, and this string is then coded using some character set, for example ISO Latin 1, where each character is sent as one octet. *A Boolean value* might be transferred as either the textual string `TRUE` or the textual string `FALSE`, or as the characters `0` or `1`.

Binary method: Information is transformed to a standardized binary format, not dependent on the architecture of a particular computer. For a floating point number, the base, mantissa and exponent are sent as bit strings. Text strings are sent as text strings also with the binary method.

Examples of Internet protocols which use the textual method are:

SMTP	Simple Mail Transfer Protocol
HTTP	Hypertext Transfer Protocol

Examples of Internet protocols which use the binary method are:

LDAP	Lightweight Directory Access Protocol
DNS	Domain Naming System

1.1.3. Encoding of information structure

The information transmitted through networks is not only individual data elements like a number or a text string. There is also structural information. Structural information indicates:

- Where one data element ends and another begins.
- What kind of information is carried by a data element, for example if a number in a meteorological application represents temperature, wind velocity or

Table 4: Encoding of start and end of elements

Method	Description	Example of encoding of the name "John Smith"
Fixed length encoding	A data element has a length specified in the protocol.	J O H N S M I T H
Length encoding	The length of the data element, usually in number of octets, is sent before the element itself.	10-J O H N S M I T H
Delimiter encoding	The end of the data element is marked with some delimiter, some special code which will not appear inside the data element.	J O H N S M I T H ;
Chunked transmission	The information is split into a number of chunks, each chunk is sent using length encoding, but the total length need not be known when sending starts.	4-J O H N 5-S M I T H

humidity.

- Which data elements belong together in structures, for example in a meteorological application, a set of one temperature, one wind velocity and one humidity value may belong together to represent the weather measurements in a certain place at a certain time.

1.1.4. Encoding of the start and end of data elements

Table 4 shows some methods of encoding the start and end of a data element. All of these methods have their particular advantages and disadvantages.

Fixed length encoding has the problem that there is a maximum size of the data (length of the string in the example above). You cannot send data requiring more than the allocated space. An extreme example of the risks with fixed-length encoding is the so-called Y2K or Year 2000 problems, which has caused billions of dollars of cost to companies who used a fixed length of 2 digits instead of 4 for storing the year.

Length encoding has problems for very large objects, where it may be difficult or impossible to compute the size before starting to send. One example is the sending of live sound or video, where you do not know the length of the sound when you begin sending it.

Delimiter encoding has the problem that the delimiter or delimiters cannot be included in the data sent, unless the delimiter is coded in some particular way. Some common methods in Internet protocols of handling this:

7. Have a special escape character preceding a delimiter. For example, if “;” is used as a delimiter, the string “ABC;DEF” might be encoded as “ABC\;DEF”. Any occurrence of the escape character must also be encoded, so that the string “AB\CD;DE” will be encoded as “AB\\CD\;DE”. This method is used in many Internet standards, for example in SMTP.
8. Require duplication of the escape character. For example, if the escape character is “”, the string “AB"BC" "DE” is encoded as “AB" "BC" " " "DE”.
9. Surround the data with double-apostrophe, and duplicate any double-apostrophe in the text. For example, the string “AB"BC" "DE” is encoded as “"AB" "BC" " " "DE"”. This method is used in many Internet standards, for example in SMTP.
10. Encode the data into a limited character set, and then use as delimiters characters outside this set. An example of this is the BASE64 and UUENCODE formats.

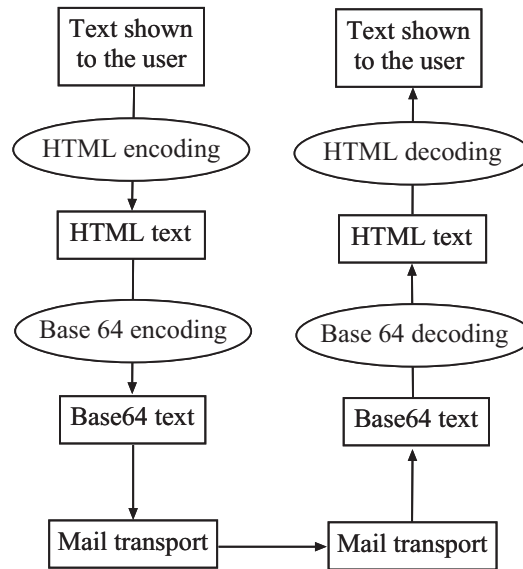


Figure 1: Encoding in several layers

11. Encode the special characters with some sequence of characters which contains the numerical value of the character code. Ex-

amples:

- The *Quoted-Printable encoding method* in MIME will encode the ISO Latin 1 character “Ä” as “=C4”, since “C4” is the hexadecimal byte value of this character.
- The *HTML Character Entity encoding method* of the character “Ä” as “Ä”, where “196” is the decimal byte value of this character.
- The *MIME header encoding method*, where the character “Ä” is encoded as “=?iso-8859-1?q?=C4?=". Here, “iso-8859-1” is the ISO identification of the ISO Latin One character set, “q” indicates that the quoted-printable encoding method is used, and “=C4” is the quoted-printable encoding of “Ä”.
- The *URL encoding method*, where the character “Ä” is encoded as “%C4”, where “C4” is the hexadecimal value of the ISO Latin One character “Ä”.

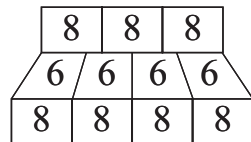
12. Encode the special characters with some sequence of characters which describe the character in words. One example is the encoding of the “Ä” character in HTML (see page 77) as “Ä”, where “Auml” means “A with umlaut”, “umlaut” is the German word for putting two dots on top of a vowel.

In some cases, several different character encoding methods are used on top of each other. They must then be undone in the reverse order to get back the original text. For example, if HTML text is sent in e-mail with the base64 encoding method, then, as shown in Figure 1, the text might first be encoded with the HTML method, and the resulting text might then be encoded once more with the BASE64 method, before it is sent through e-mail.

1.1.5. Encoding of binary data with textual encoding

How do you transport binary data with textual encoding? There are two methods:

- ① If you have an eight-bit transparent transport channels, you can just split the binary data into eight-bit octets and send them as they are. This is usually combined with the length method of delimiting the end of the binary data element, to allow any eight-bit value within the binary data.
- ② Encode the binary data as text. The two most common methods for this are UUENCODING and BASE64.



BASE64 is more reliable and works as follows: Take three octets (24 bits), split them into four 6-bit bytes, and encode each 6-bit byte as one character. Since 6-bit bytes can have 64 different values, 64 different characters are needed. These have been chosen to be those 64 ASCII characters which are known not to be perverted in transport. Since BASE64 requires 4 octets, 32 bits, to encode 24 bits of binary data, the overhead is $8/24$ or 33 %.

1.1.6. More About Encoding of Information Structure

Often you need to transport a complex set of related information elements in a networked protocol. Suppose, for example, that you have the following data structure:

Personal record consists of *age*, *weight* and *name*.

Name consists of two strings, *given name* and *surname*.

Age consists of a positive integer.

Weight consists of a positive decimal value in kilograms.

The two most common methods of encoding this kind of information is the

tag-length-value encoding and the textual encoding.

1.1.1.1. Tag-Length-Value encoding

With the tag-length-value encoding, each element in the data structure is split into three parts, a tag, which specifies whether this is a age, weight, name, given name or surname value, a length, giving the number of octets needed for the value, and then the value. If the value contains several elements, it can consist of a new set of Tag-Length-Value encodings, as shown in Figure 2.

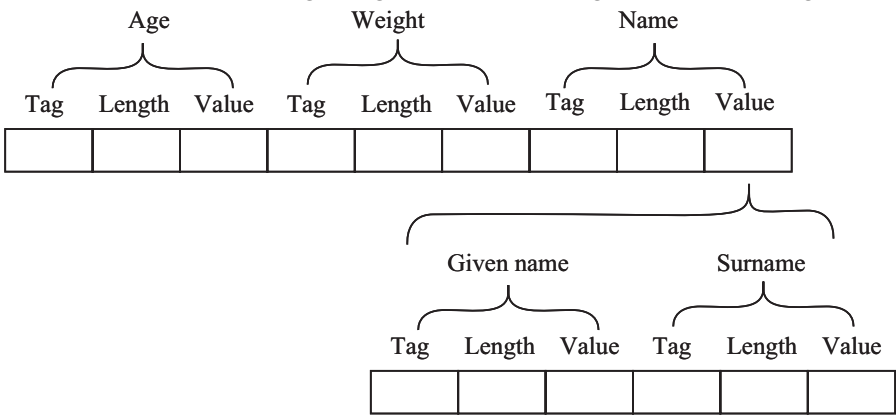


Figure 2: Example of tag-lenght-value encoding

A fuller description of this encoding is shown in Table 1 on page 19.

1.1.1.2. Textual encoding

With textual encoding, the same information might be encoded as the following text string (`CR LF` represents carriage return+line feed = a line break).

Age: 58; Weight: 74.6; Name: John,
Smith `CR LF`

or as the following string:

Table 5: Example of tag-length-value encoding

Information element		Part	Octets	Encoding
Age		Tag	1	The value “0” is chosen to represent “Age” in this protocol.
		Length	1	Always 1
		Value	1	Binary value
Weight		Tag	1	The value “1” is chosen to represent “Weight” in this protocol.
		Length	1	Always 4
		Value	4	First octet exponent with the base 10, then three octets with mantissa, both exponent and mantissa in binary form.
Name		Tag	1	The value “2” is chosen to represent “Name” in this protocol.
		Length	1	The total length of the components.
Com- ponents of Name	Given name	Tag	1	The value “3” is chosen to represent “Given name” in this protocol.
		Length	1	The length of the string
		Value	As many octets as needed for this string	ISO 8859-1
	Sur- name	Tag	1	The value “4” is chosen to represent “Surname” in this protocol.
		Length	1	The length of the string
		Value	As many octets as needed for this string	ISO 8859-1

```
Age: 58 CR LF
Weight: 74.6 CR LF
Name: CR LF
Given Name: John CR LF
Surname: Smith CR LF
```

An example of textual encoding from an actual Internet standard is the e-mail header, an example of which might be:

```
Received: from mail.ietf.net CR LF
by info.dsv.su.se (8.8.8/8.8.8) with ESMTp CR LF
id HAA06480 for <jpalme@dsv.su.se> CR LF
Wed, 22 Jul 1998 07:51:54 +0200 CR LF
Message-ID: <AF4C1AD5F8662ED305D823AF@ietf.net> CR LF
From: Erik Nielsen <erikn@ietf.net> CR LF
To: Jacob Palme <jpalme@dsv.su.se> CR LF
Subject: Example of an e-mail header CR LF
Date: Tue, 24 Jul 1998 21:25:21 -0700 CR LF
```

Textual encoding usually uses the delimiter method. In the example above, “:”, “;”, “<”, “>”, “from”, “by”, “id” and space are used as delimiters. “Received”, “Message-ID”, “From”, “To”, “Subject” and “Date” are used as tags, but in the “Received” field there are subtags “from”, “by”, a n d “id”.

2. *Augmented Backus-Naur Form, ABNF*

Objectives

This chapter describes the most commonly used coding specification method

Keywords

ABNF
coding

When writing syntax specifications for protocols, a special language for syntax specifications is used. There are three common such languages, ABNF (Chapter 2) and XML (Chapter 0) for specifying the syntax of textual protocols, and ASN.1 (Chapter 0) for specifying the syntax of binary tag-length-value-encoded protocols. ABNF was first standardized in [RFC 822] and a revised version was standardized in [RFC 2234]. ABNF and ASN.1 are both based on the Backus-Naur Form, BNF, which became first widely known in the Algol 60 specification in 1958. BNF syntax specifications consists of production rules. Take for example a personal record which might look like this:

```
Age: 58; Weight: 74.6; Name: John,
Smith CR LF
```

Its ABNF specification might be:

```
personal-record = age "; " weight "; " name CR LF
age              = "Age: " integer
weight          = "Weight: " decimal-value
name            = given-name ", " surname
given-name      = 1*LETTER                ; one or more letters
surname         = 1*LETTER
integer         = 1*D                      ; one or more digits
decimal-value   = 1*D "." 0*D              ; zero or more decimals
```

1.1.7. Linear White Space

ABNF has traditionally had problems with indicating where white space is permitted. White space is composed of one or more of the following character codes:

Space	A non-printing break with the same width as a single letter
Horizontal Tab, HT	Moves to the next tab position, sometimes, but not always, there are tab position at every eight column for fixed-width fonts
Line Feed, LF	Moves the cursor to the next line
Carriage Return, CR	Moves the cursor the start of the line
CRLF	CR followed by LF, moves the cursor to the start of the next line

Note: Many computer systems use either only the LF or only the CR as a character to move to the start of the next line. Some Internet standards, for example HTML and HTTP, allows line breaks to be either LF or CR or CRLF. Other Internet standards, for example SMTP, require that all line breaks must

be CRLF.

Here is an example from an old Internet standard, RFC822, the standard for the format of e-mail messages:

```
date = 1*2DIGIT month 2DIGIT           ; day month year
```

Literally, the ABNF below should generate date formats like “25Jul98”. But in reality, the correct date format is “25 Jul 98”, with a space between the words. Some, but not all, later Internet standards specify explicitly where white space is allowed, for example:

```
date = 1*2DIGIT " " month " " 4DIGIT    ; day month year
```

Often (but not in the case of the gap between day, month and year above) where one space is allowed, also a sequence of linear white space characters is allowed. For example, the following three variants are identical according to the e-mail standards:

```
From: "Autumn publishers" <books@autumn.net>
From:  "Autumn publishers"  <books@autumn.net>
From:  "Autumn publishers"
      <books@autumn.net>
```

Some standards even allow comments in parenthesis where white space is allowed. Thus, in e-mail, a fourth equivalent alternative to the “From” field above might be:

```
From: (good books) "Autumn publishers"
      (write to us) <books@autumn.net> (to order our books)
```

1.1.8. Versions of ABNF

There are two commonly used versions of ABNF. The first is the 1982 version, specified in RFC 822 and used, sometimes a little modified, in many Internet standards. Typical of standards using the old ABNF is that they do not specify clearly where comments and linear white space is allowed or required.

The 1997 version, specified in RFC 2234, is when this is written (2000) not yet very much used. It has some new features, which allows the exact specification of things which could only be specified by plain text comments in the old ABNF (see section “RFC 822 lexical scanner specified in ABNF” on page 30).

1.4. An overview of ABNF syntax constructs

1.1.9. Either-or construct

The “/” means either the specification to the left or the specification to the right. Example:

```
answer      = "Answer: " ( "Yes" / "No" )
```

will specify the following two alternative values:

```
Answer: Yes and Answer: No
```

1.1.10. A series of elements of the same kind

There is often a need to specify a series of elements of the same kind. For example, to specify a series of "yes" and "no" we can specify:

```
yes-no-series = *( "yes " / "no " )
```

This specifies that when we send a yes-no-series from one computer to another, we can send for example one of the following strings (double-quote not included):

```
“yes ”      “yes no ”
“yes yes yes ”  “” (an empty string)
```

The “*” symbol in ABNF means “repeat zero, one or more times”, so yes-no-series, as defined above, will also match an empty string. A number can be written before the “*” to indicate a minimum, and a number after the “*” to indicate a maximum. Thus “1*2” means one or two occurrences of the following construct, “1*” means one or more, “*5” means between zero and five occurrences.

If we want to specify a series of exactly five yes or no, we can thus specify:

```
five-yes-or-no = 5*5( "yes " / "no " )
```

and if we want to specify a series of between one and five yes or no, we can specify:


```
one-to-five-yes-or-no = 1*5( "yes " / "no " )
```

1.1.11. Comments in ABNF

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line.

1.1.12. Linear White Space (LWSP)

There is often a need to specify that one or more characters which just show up as white space (blanks) on the screen is allowed. In newer standards, this is done by defining Linear White Space:

```
LWSP char = ( SPACE / HTAB ) ; either one space or one tab
LWSP      = 1*LWSP-char ; one or more space characters
```

LWSP, as defined above, is thus one or more SPACE and HTAB characters. Using LWSP, we can specify for example:

```
yes-no-series = * ( ( "yes" / "no" ) LWSP )
```

examples of a string of this format is:

```
“yes ”      “yes no ”
“no ”      “yes yes yes ”
“”         “yes      yes      no      ”
```

1.1.13. Comma-separated list

Older ABNF specifications often uses a construct "#" which means the same as "*" but with a comma between the elements. Thus, in older ABNF specifications:

```
yes-no-series = *( "yes" / "no" )
```

is meant to match for example the strings

```
“yes ”      “yes no ”
“no ”      “yes yes yes ”
```

while

```
yes-no-series = #( "yes" / "no" )
```

is meant to match the strings

```
“yes ”      “yes, no ”
“no ”      “yes, yes, yes ”
```

The problem with this, however, is that neither of the notations above specify

where LWSP is allowed. Thus, newer ABNF specifications would instead use:

```
yes-or-no      = ( "yes" / "no" )
yes-no-series  = yes-or-no *( LWSP yes-or-no)
```

to indicate a series of “yes” or “no” *separated by LWSP*, or

```
yes-no-series  = yes-or-no *( "," LWSP yes-or-no)
```

to indicate a series of “yes” or “no” separated by “,” and LWSP.

1.1.14. ABNF syntax rules, parentheses

Elements enclosed in parentheses are treated as a single element. Thus, “(elem (foo / bar) elem)” allows the token sequences “elem foo elem” and “elem bar elem”. Example of use of this (from RFC822):

```
authentic      = "From" ":" mailbox ; Single author
                  / ( "Sender" ":" mailbox ; Actual submittor
                     "From" ":" 1#mailbox) ; Multiple authors
                  ; or not sender
```

Example 3, value a:

```
From: Donald Duck <d Duck@disney.com>
```

Example 3, value b:

```
Sender: Walt Disney <walt@disney.com>
From: Donald Duck <d Duck@disney.com>
```

1.1.15. Optional elements

There is often the need to specify that something can occur or can be omitted. This is specified by square brackets. Example:

```
answer = ( "yes" / "no" ) [ ", maybe" ]
```

will match the strings

```
"yes"           "no"
"yes, maybe"    "no, maybe"
```

Square brackets is actually the same as "0*1, the ABNF production above could as well be written as:

```
answer = ( "yes" / "no" ) 0*1( ", maybe" )
```

or

```
answer = ( "yes" / "no" ) *1( ", maybe" )
```

Table 6: Summary of ABNF notation

Notation	Meaning
"/"	either or
n*m(element)	Repetition of between n and m elements
n*n(element)	Repetition exactly n times
n*(element)	Repetition n or more times
*n(element)	Repetition not more than n times
n#m(element)	Same as n*m but comma-separated
[element]	Optional element, same as *1(element)
Example	Meaning
Yes / No	Either Yes or No
1*2(DIGIT)	One or two digits
2*2(DIGIT)	Exactly two digits
1*(DIGIT)	A series of at least one digit
*4(DIGIT)	Zero, one, two, three or four digits
2#3("A")	"A, A" or "A, A, A"
[";" para]	The parameter string can be included or omitted
;	Text from a semicolon (;) to the end of a line is a comment

Exercise 1

Specify, using ABNF, the syntax for a directory path, like "users/smith/file" or "users/smith/WWW/file" with none, one or more directory names, followed by a file name.

(Solutions to the exercises can be found on page 112.)

Exercise 2

Specify, using ABNF, the syntax for Folding Linear White Space, i.e. any sequences of spaces or tabs or newlines, provided there is at least one space or tab after each newline.

Examples:

“**HT HT**”

```
“ HT CR LF  
HT ”  
“ CR LF HT ”
```

Assume SP = Space, HT = Tab, CR = Carriage Return, LF = Line Feed

1.5. Examples of use of ABNF

Example 1, ABNF (from RFC 822):

```
LWSP-char = SPACE / HTAB ; semantics = SPACE
```

Example 2, ABNF (from RFC822):

```
mailbox = addr-spec ; simple address
         / phrase route-addr ; name & addr-spec
addr-spec = local-part "@" domain ; global address
phrase = 1*word ; Sequence of words
word = atom / quoted-string
```

Examples of values matching the syntax in Example 2 above:

```
jpalme@dsv.su.se
Jacob Palme <jpalme@dsv.su.se>
```

Example 3 (from RFC822):

```
optional-field =
/ "Message-ID" ":" msg-id
/ "Resent-Message-ID" ":" msg-id
/ "In-Reply-To" ":" *(phrase / msg-id)
/ "References" ":" *(phrase / msg-id)
/ "Keywords" ":" #phrase
/ "Subject" ":" *text
/ "Comments" ":" *text
/ "Encrypted" ":" 1#2word
/ extension-field ; To be defined
/ user-defined-field ; May be pre-empted
```

Examples of values matching the syntax in Example 3 above:

```
In-Reply-To: <12345*jpalme@dsv.su.se>
In-Reply-To: <12345*jpalme@dsv.su.se> <5678*jpalme@dsv.su.se>
In-Reply-To: Your message of July 26 <12345*jpalme@dsv.su.se>
Keywords: flowers, tropics, evolution
```

Example 4 (from RFC822) demonstrating the use of square brackets ([]) and ():

```
received = "Received" ":"
[ "from" domain] ; sending host
[ "by" domain] ; receiving host
[ "via" atom] ; physical path
*( "with" atom) ; link/mail protocol
[ "id" msg-id] ; receiver msg id
[ "for" addr-spec] ; initial form
```

1.1.16. Examples of values matching the syntax in example 4 above:

```
Received: from mars.su.se (root@mars.su.se Ä130.237.158.10Ä)
by zaphod.sisu.se (8.6.10/8.6.9) with ESMTP
id MAA29032 for <cecilia@sisu.se>
```

1.1.17. Example 7 (from RFC822):

```

authentic = "From" ":" mailbox ; Single author
           / ( "Sender" ":" mailbox ; Actual submittor
             "From" ":" 1#mailbox) ; Multiple authors
           ; or not sender

```

1.1.18. Examples of value matching the syntax in example 7 above

```

From: Sven Svensson <ss@foo.bar>, Per Persson <pp@foo.bar>
Sender: Sven Svensson <ss@foo.bar>

```

Exercise 3

Specify the syntax of a new e-mail header field with the following properties:

Name: “Weather”

Values: “Sunny” or “Cloudy” or “Raining” or “Snowing”

Optional parameters: “;” followed by parameter, “=” and integer value

Parameters: “temperature” and “humidity”

1.1.1.3. Examples of values:

```
Weather: Sunny ; temperature=20; humidity=50
```

```
Weather: Cloudy
```

Exercise 4

An identifier in a programming language is allowed to contain between 1 and 6 letters and digits, the first character must be a letter. Only upper case character are used. Write an ABNF specification for the syntax of such an identifier.

1.6. RFC 822 lexical scanner specified in ABNF

By a lexical scanner is meant the lowest level of the syntax, the rules for scanning characters and combining them into words. Below is part of the lexical scanner from RFC822 as an example of how such a scanner can be specified using ABNF.

```

CHAR      = <any ASCII character>           ; ( 0-177, 0.-127.)
ALPHA     = <any ASCII alphabetic character> ; (101-132, 65.- 90.)
                                                ; (141-172, 97.-122.)
DIGIT     = <any ASCII decimal digit>        ; ( 60- 71, 48.- 57.)
CTL       = <any ASCII control                ; ( 0- 37, 0.- 31.)
            character and DEL>                ; ( 177, 127.)
CR        = <ASCII CR, carriage return>      ; ( 15, 13.)
LF        = <ASCII LF, linefeed>             ; ( 12, 10.)
SPACE     = <ASCII SP, space>                ; ( 40, 32.)
HTAB      = <ASCII HT, horizontal-tab>       ; ( 11, 9.)
<">      = <ASCII quote mark>               ; ( 42, 34.)
CRLF      = CR LF
LWSP-char = SPACE / HTAB                    ; semantics = SPACE

```

Note that much important information above is specified in plain text and not using ABNF constructs. The 1997 version of ABNF includes constructs which mean that much of this can be specified using ABNF constructs. With these new constructs, a code roughly defining the same is specified in the ABNF standard itself as:

```

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 (past newline)
OCTET = %x00-FF ; 8 bits of data
SP = %x20

```

The new constructs allow the specification of character codes using binary (b), decimal (d) or hexadecimal (x) notation.

%d13	is the character with decimal value 13, which is carriage return.
%x0D	is the character with hexadecimal value 0D, which is another way of specifying the carriage return character.
b1101	is the character with binary value 1101, which is a third way of specifying the carriage return character.
%x30-39	means all characters with hexadecimal values from 30 to 39, which is the digits 0-9 in the ASCII character set.
%d13.10	is a short form for %d13 %d10, which is carriage return followed by line feed.

3. *Abstract Syntax Notation, ASN.1*

Objectives

ASN.1 is a strongly typed coding language which gives readable code descriptions and very compact, but difficult to read, binary encoding

Keywords

ASN.1

BER

ASN.1 (Abstract syntax notation 1 [Larmouth 1999, Kaliski 1993]) is an alternative to ABNF for specifying the syntax of complex data structures. While ABNF is mostly used to specify textual encodings, ASN.1 is mostly used to specify binary encodings. The same syntax specification in ASN.1 can be used with different encoding rules, but of course the sending and receiving computer must agree on which encoding rules to use, if they are to understand each other using ASN.1. The mostly used encoding rule for ASN.1 is called BER (Basic Encoding Rules). A short overview of BER can be found on page 67. This book does not give a complete description of all the features of ASN.1.

Most Internet application layer standards use ABNF and textual encodings, but a few use ASN.1, for example SMIME, LDAP and Kerberos.

The main principle of ASN.1 is that new data types can be defined based on simpler types. The example below shows how this is done.

Assume that a meteorological station needs to send a temperature measurement to a meteorological center. The temperature is one single value, it can be encoded in different ways. It can be sent as a **real** value (which in a computer is encoded as a floating-point number, with a mantissa and an exponent) or it can be sent as an **integer** value. It can be given in degrees Celsius, Kelvin or Fahrenheit.

A standard for sending meteorological information must define this. The ASN.1 definition of how temperature information is transferred might look like this:

Temperature ::= REAL -- In degrees Kelvin

This statement just says that the temperature is to be encoded using the ASN.1 rules for encoding floating-point (real) values. **REAL** is a built-in ASN.1 type. ASN.1 has a number of built-in simple data types, like **REAL**, **INTEGER**, **BOOLEAN**, **STRING**, etc. Information which cannot be coded formally in the ASN.1 language can be added as a comment, which is preceded by “--” as “-- In degrees Kelvin” in the example above.

But how does the recipient know that the value sent is a temperature value and not, for example, the floating-point value of the wind velocity or humidity? One way of doing this is to introduce a *tag*. A tag is a label which is sent

before the data value and indicates what kind of information is sent. The ASN.1 definition in that case might be:

```
Temperature ::= [APPLICATION 0] REAL -- In degrees Kelvin
```

This statement says that, in this application (the protocol for sending meteorological data), we let the tag “[APPLICATION 0]” indicate that the data which follows is a temperature reading. Wind velocity and humidity might have different tags:

```
Temperature ::= [APPLICATION 0] REAL
```

```
WindVelocity ::= [APPLICATION 1] REAL
```

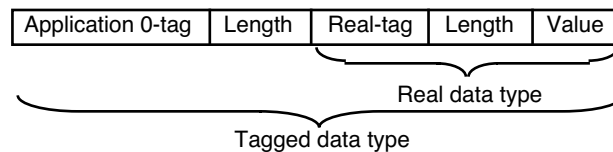
```
Humidity ::= [APPLICATION 2] REAL
```

The three lines above define three new data types, **Temperature**, **WindVelocity**, and **Humidity**, all encoded using the ASN.1 **REAL** type. Note that it is only in this special application that 0, 1 and 2 are tags for **Temperature**, **WindVelocity**, and **Humidity**. In other applications, the tags 0, 1 and 2 may mean something else.

The definition:

```
Temperature ::= [APPLICATION 0] REAL -- In degrees Kelvin
```

will actually define a new tagged data type, based on **REAL**. With explicit tagging, both tags are sent on the line as shown by this figure:



Sometimes, a new data type requires a combination of several values. A complex number, for example, can be coded as two floating-point values, one for the imaginary and one for the real element of the number. In ASN.1 this might be defined as follows:

```
ComplexNumber ::= [APPLICATION 3] SEQUENCE {  

imaginaryPart REAL,  

realPart REAL }
```

More complex data types can thus, as in the example, be defined by a combination of more than one element of simpler types.

One type definition may use separately defined types. For example, the

type for a record containing temperature, wind velocity, and humidity may be defined as:

```
WeatherReading ::= [APPLICATION 4] SEQUENCE {  
  temperatureReading Temperature,  
  velocityReading WindVelocity,  
  humidityReading Humidity }
```

Note that this definition of the new type **WeatherReading** uses the previous definitions of the three types **Temperature**, **WindVelocity**, and **Humidity** as elements. In this way, more and more complex data structures which are needed for some applications can be built using previously defined simpler types. For example, we may want to send a series of weather readings from different altitudes in one transmission as an even more complex object:

```
SeriesOfReadings ::= [APPLICATION 5] SEQUENCE OF AltitudeReading  
  
AltitudeReading ::= [APPLICATION 6] SEQUENCE {  
  altitude Altitude,  
  weatherReading WeatherReading }  
  
Altitude ::= [APPLICATION 7] REAL -- Meters above sea level
```

This contains three ASN.1 productions, where each production refers to types defined in a later production. ASN.1 productions are usually written in this top-down order, but ASN.1 does not require any particular ordering of the productions.

Using the definitions above, the actual bit string (octet string) sent may be partitioned as shown in Figure 3.

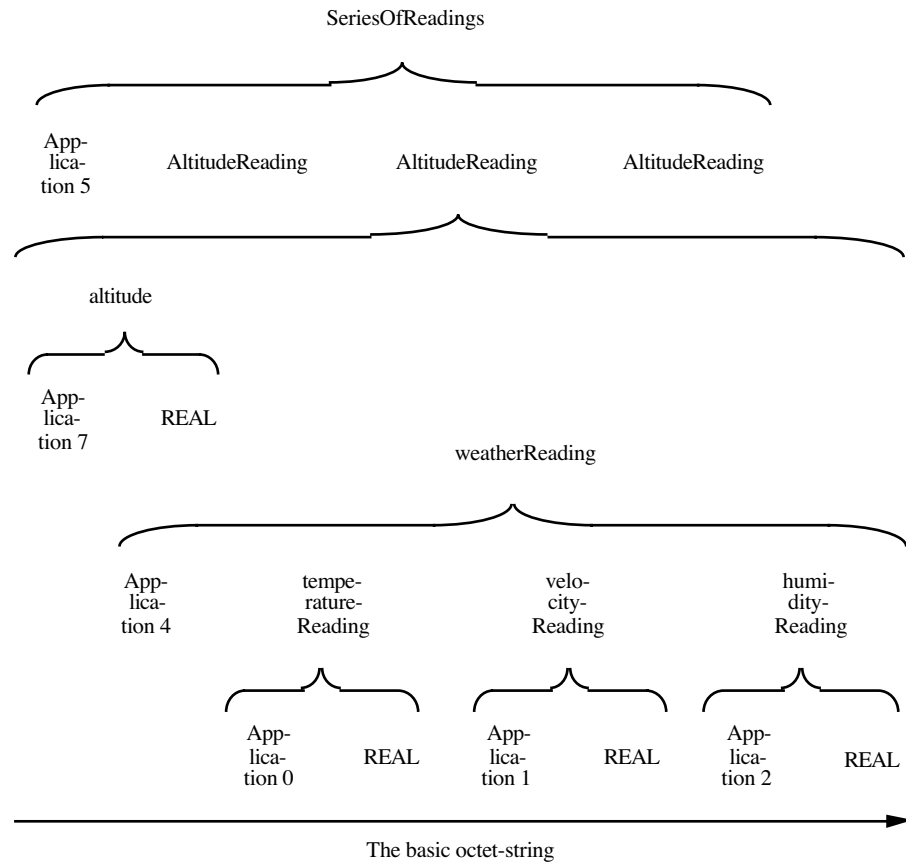


Figure 3: How ASN.1 and BER is used to produce an octet string.

Here is an example of the actual ASN.1 used in an Internet standard. The excerpt below is taken from the LDAP standard (RFC 2251):

```
BindRequest ::= [APPLICATION 0] SEQUENCE {
  version    INTEGER (1 .. 127),
  name       LDAPDN,
  authentication AuthenticationChoice }

AuthenticationChoice ::= CHOICE {
  simple     [0] OCTET STRING,      -- 1 and 2 reserved
  sasl [3] SaslCredentials }

SaslCredentials ::= SEQUENCE {
  mechanism LDAPString,
  credentials OCTET STRING OPTIONAL }
```

1.7. ASN.1 basic

1.1.19. ASN.1 value notation

Information sent via protocols between computers is usually not constant, since there is no need to send constant information. Thus, ASN.1 is mostly used to specify information which is not constant. There is however a notation in ASN.1 for specifying constants, the ASN.1 value notation. It is mostly used to specify constants which are to be used in other ASN.1 declarations. For example, instead of the ASN.1 specification:

```
Windowline ::= GeneralString (SIZE (80))
```

we might use:

```
Windowline ::= GeneralString (SIZE (lineLength))
```

```
lineLength ::= 80
```

The advantage with this is that it is easier to change the `lineLength`, it may be used in many places but defined only once. It is also neat to collect all constants like line lengths in a special area of a standards document.

1.1.20. ASN.1 terminology

A *type* or a data type is a set of permitted values. A type can be defined by enumerating all permitted values, or it can be defined to have an unlimited number of values, like the data types *Integer* and *Real*. A new type, which is defined by a combination of elements of already defined types, is called a *structured* type. Example of a definition of a structured type:

```
ComplexNumber ::= [APPLICATION 3] SEQUENCE  
{   imaginaryPart REAL,  
    realPart    REAL }
```

A specification of a syntax in ASN.1 is called an *abstract syntax*. The syntax used in actual communication between two computers is called a *transfer syntax*. The specification of how an abstract syntax is to be implemented in a transfer syntax is an encoding rule, like the Basic Encoding Rules (BER).

An ASN.1 production is a rule to define one type, based on other already defined types. The syntax for an ASN.1 production is:

1. The name of the new data type (must begin with an upper case letter, A-Z)
2. The operator ::=
3. The definition of the new data type.

Exercise 5

You are to define a protocol for communication between an automatic scale and a packing machine. The scale measures the weight in grams as a floating point number and the code number of the merchandise as an integer. Define a data type **ScaleReading** which the scale can use to report this to the packing machine.

Exercise 6

Some countries use, as an alternative to the metric system, a measurement system based on inches, feet and yards. Define a data type **Measurement** which gives one value in this system, and **Box** which gives the height, length and width of an object in this measurement system. Feet and yards are integers, inches is a decimal value (=floating point value with the base 10).

1.1.21. Pre-defined, built-in types in ASN.1

Table 7 lists the pre-defined, built-in types of ASN.1.

Table 7: Built-in types in ASN.1

Simple types	Character string types	Structured types	"Useful types"
BOOLEAN	NumericString	SET	GeneralizedTime
INTEGER	PrintableString	SET OF	UTCTime
ENUMERATED	TeletexString	SEQUENCE	EXTERNAL
REAL	VideotexString	SEQUENCE OF	ObjectDescriptor
BIT STRING	VisibleString	CHOICE	<i>Warning: Constraints</i>
OCTET STRING	IA5String	ANY	<i>are strongly recom-</i>
NULL	GraphicString	[Tagged]	<i>ended for Graphic,</i>
OBJECT IDENTIFIER	GeneralString	<Different vari-	<i>General, Universal,</i>
	UniversalString	ants	<i>BMP and UTF8</i>
	BMPString	< of ISO 10646,	<i>strings</i>
	UTF8String	not	
	CharacterString	< in the 1998	
		< version	

1.1.22. Comments

Comments in ASN.1 start with two hyphens in direct succession, "--", and end with either two hyphens again, "--" or the end of the row.

1.1.23. Format of identifiers

Field names and constant values in ASN.1 must have names beginning with a lower case letter (a-z). Types must have names beginning with an upper-case letter (A-Z). The case is thus significant in ASN.1 names. Both field names and values can contain all letters (a-z, A-Z, numbers (0-9) and the hyphen character ("-"). Two hyphens in succession are however not allowed, since they are used to indicate the start of a comment.

1.8. Simple Types

1.1.24. Integer Type

The **INTEGER** simple type can have as values all positive and negative integers including 0. Note that there is no maximum value. This is different from integers in computer programming languages, which usually are limited to 32 or 64 bits.

An example of use of an **INTEGER** declaration:

Number-of-years ::= INTEGER

An **INTEGER** declaration may include names of certain values. Example:

Weekday ::= INTEGER { monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(6), sunday(7) }

This does not limit the value of **Weekday** to integers between 1 and 7. **Weekday**, as defined above, can still have as value any positive or negative integer.

1.1.25. Subtypes

It is, however, possible to restrict a new type, based on the **INTEGER** type, to only some values. This is done using the subtyping notation. Example:

Weekday ::= INTEGER { monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(6), sunday(7) } (1 .. 7)

Subtypes are specified with information in parentheses after a type specification, as in the example above. Subtype will limit the set of allowed values to only a subset of the allowed values of the parent type. In the case of the **INTEGER** type, the following commands are allowed in subtype specifications:

Example	Description
1 .. 7	all values between the lower and upper bound
5	a single value
INCLUDES Weekday	all values from another, defined type
2 10	list of values, separated by

Additional constructs are allowed in subtypes to other types than **INTEGER**, this will be described later. Here are some examples of subtype declarations on the **INTEGER** type:

OddSingleDigitPrimes ::= INTEGER (3 | 5 | 7)

SingleDigitPrimes ::= INTEGER (2 | INCLUDES OddSingleDigitPrimes)

PositiveNumber ::= INTEGER (1 .. MAX)

Month ::= (1 .. 12)

Month ::= (1 .. <13)

The two declarations of **Month** above define the same value set. **MAX** and **MIN** means that there is no limit. This is not the same thing as $+\infty$ and $-\infty$), an **INTEGER** cannot have infinity as a value, but it can be of arbitrary size.

Exercise 7

Change the definition of **Measurement** in Exercise 2 so that feet can only have the values 0, 1 or 2 (since 3 feet will be a yard), and so that inches is specified as an integer between 0 and 1199 giving the value in hundreds of an inch (since 1200 or 12 inches will be a foot).

1.1.26. Boolean Type

The Boolean type has only two values, **TRUE** and **FALSE**. Example:

ShopOpen ::= BOOLEAN

It is *not* permitted to write:

Gender ::= BOOLEAN {male (TRUE), female (FALSE) }

but instead, you can write

Gender ::= BOOLEAN

male Gender ::= TRUE

female Gender ::= FALSE

Exercise 8

In an opinion poll, made at the exit door from the election rooms, every voter is asked to indicate which party they voted for. Allowed values are Labour, Liberals, Conservatives or “other”. The age of each voter is also registered as a positive integer above the voting age of 18 years, and the gender is registered. Define a data type to transfer this information from the poll station to a server.

Exercise 9

In the local election in Hometown, there are also two local parties, the Hometown party and the Drivers party. Extend solution 1 to exercise 8 to a new datatype **HometownVoter** where also these two additional parties are allowed.

1.1.27. Enumerated

The **ENUMERATED** type can only have the values which are enumerated in its declaration. The syntax is similar to the **INTEGER** type. Example:

```
DayOfTheWeek ::= ENUMERATED {monday (1), tuesday (2), wednesday (3), thursday (4),
friday (5), saturday (6), sunday (7) }
```

A difference between **ENUMERATED** and **INTEGER** is that the values of the **ENUMERATED** type are not ordered. The following construct:

```
WeekDayNumber ::= INTEGER {monday (1), tuesday (2), wednesday (3), thursday (4),
friday (5), saturday (6), sunday (7) }
```

```
WorkingDayNumber ::= WeekDayNumber ( 1 .. 5 )
```

is thus not permitted, with **ENUMERATED**, you have to define this subtype as:

```
WorkingDay ::= DayOfTheWeek ( monday | tuesday | wednesday | thursday | friday |
saturday | sunday )
```

Compare the following three definitions of DayOfTheWeek:

- ① **DayOfTheWeek** ::= **INTEGER** { monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(6), sunday(7) }
- ② **DayOfTheWeek** ::= **INTEGER** { monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(6), sunday(7) } (1..7)
- ③ **DayOfTheWeek** ::= **ENUMERATED** { monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(6), sunday(7) }

Case ① allows all possible integers as values, case ② and ③ only allows the seven values 1 to 7. Case ② has a defined order, case ③ has no defined order of the values.

1.1.28. Real Type

The **REAL** type includes the following allowed values: $+\infty$, $-\infty$ and values of the form

$M * B^E$, where M and E can be any ASN.1 **INTEGER** and B can only have the value 2 or 10. Examples:

```

Weight ::= [ APPLICATION 0] REAL      -- Measured in grams
pi REAL ::= {314159265358793238462433, 10, 25 }
zero REAL ::= 0
topValue REAL ::= PLUS-INFINITY

```

Exercise 10

In the armed forces, three degrees of secrecy are used: open, secret and top secret. Suggest a suitable datatype to convey the secrecy of a document which is transferred electronically.

Exercise 11

Given the solution to Exercise 10, assume that a new degree extra high secret is wanted. Define an extended version of the protocol defined in Exercise 6 to allow also this value.

1.1.29. Bit String

A **BIT STRING** has as value an ordered string of 0 or more bits. The first bit is numbered 0, the second 1, etc. Examples

```

Gender ::= BIT STRING      -- This BITSTRING indicates the gender of each
                          -- of several individuals

```

```

DotPattern ::= BIT STRING ( SIZE (25)) -- This BITSTRING always contains
                                      -- exactly 25 bits

```

```

Person ::= BIT STRING { gender (0), married (1), adult (2) }

```

Note: BER will encode a **BIT STRING** more compactly than a **SEQUENCE OF BOOLEAN**. With the Packed Encoding Rules (PER) there is no difference.

1.1.30. Subtypes

A subtype specification takes an existing type, and specifies a subtype of its values. The following constructs can be used to specify subtypes of a type:

Table 8 Different kinds of subtypes

Kind of sub-type	Allowed for	Examples
Single value	All types	RetirementAge ::= INTEGER (65)
Range	INTEGER and REAL	AdultAge ::= INTEGER (15 .. MAX) Child ::= INTEGER (1 .. 14)
Contained subtype	All types	Age ::= INTEGER (INCLUDES Child INCLUDES AdultAge)
Size range	SEQUENCE OF , SET OF and all string types	Line ::= General String (SIZE (1..80)) Couple ::= SET SIZE(2) OF Person
Alphabet limitation	Character string types	OctalDigit ::= General String (FROM ("0" "1" "2" "3" "4" "5" "6" "7"))
Inner subtyping	SET , SET OF , SEQUENCE , SEQUENCE OF , CHOICE	Person ::= CHOICE { Male, Female } Males ::= SET WITH Component (Male) OF Person
List of several subtype values	All types	Base ::= INTEGER (2 8 10 16)
Constraint (the actual subtyping restrictions are specified in a comment)	All types	ENCRYPTED { ToBeEnciphered } ::= BIT STRING (CONSTRAINED-BY { -- must be enciphered using the -- DES encipherment standard })

1.1.31. Variants of Bit Strings

- ① **Characteristics ::= BIT STRING {gender(0), adult(1), blueEyed(2), caucasian(3) }**
- ② **Characteristics ::= BIT STRING {gender(0), adult(1), blueEyed(2), caucasian(3) }**
(SIZE (0 .. 4))
- ③ **Characteristics ::= BIT STRING {gender(0), adult(1), blueEyed(2), caucasian(3) }**
(SIZE (4))
- ① Specifies a **BIT STRING** of any length, but with defined names only for its

first four values.

- ② Is similar to ①, but cannot be longer than 4 bits.
- ③ Is similar to ①, but always has exactly 4 bits.

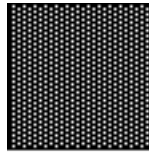
Exercise 12

Assume that you want to define a pattern to cover a monochrome screen. Each pixel on the screen can be either black or white. The pattern is made by repeating a rectangle of N times M pixels over the whole screen. Examples of possible patterns are:

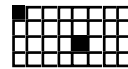
Base



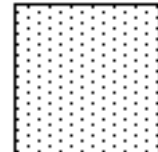
Example of use



Base



Example of use



Specify an ASN.1 data type which you can use to describe different such patterns.

Exercise 13

A store holds paper in the formats A3, A4, A5 and A6. A user wants to know if sheets are available in each of these four formats. Specify a data type to report this to the user.

Exercise 14

What is the difference between these two types, and what does monday mean for each of them?

DayOfTheWeek ::= ENUMERATED { monday(0), tuesday(1), wednesday(2), thursday(3), friday(4), saturday(5), sunday(6) }

DaysOpen ::= BIT STRING { monday(0), tuesday(1), wednesday(2), thursday(3), friday(4), saturday(5), sunday(6) } (SIZE(7))

- 1.1.32. Octet String Type An Octet String specifies a string of zero, one or more octets. This type is often used when you want to transfer data specified according to some other syntax than ASN.1, such as a GIF file. Example:
GifPicture ::= OCTET STRING

1.1.33. Null Type

The Null type has only one allowed value, the value **null**. It can be used to indicate a placeholder for something to be added in the future, or it can be used combined with **OPTIONAL**, where the existence of a value or its absence indicates some information. Example:

**Prisoner ::= SEQUENCE {
 name GeneralString,
 dangerous NULL OPTIONAL }**

Which conveys the same information as

**Prisoner ::= SEQUENCE {
 name GeneralString,
 dangerous BOOLEAN }**

1.1.34. Examples of the Use of Size

```

MonthNumber ::= NumericString (SIZE (1 ..2))
MonthNumber ::= NumericString (SIZE (1 |2))
Base ::= BIT STRING (SIZE ( 0 | 2 .. 7 | 10 ))
Couple ::= SET SIZE(2) OFHuman
BridgeDeal ::= SET SIZE (13) OFPlayingCard
BridgeHand ::= SET SIZE (0..13) OFPlayingCard
lineLength INTEGER 80
Line ::= VisibleString (SIZE (0 .. lineLength))

```

Exercise 15

The X.400 standard specifies that a name can consist of several subfields. One of the subfields is called OrganizationName and can have as value between 1 and 64 characters from the character set PrintableString. Suggest a definition of this in ASN.1.

1.1.35. Character String Types

ASN.1 has several Character String types for different character sets.

NumericString	“0” .. “9” and “ ”
PrintableString	“a” .. “z”, “A” .. “Z”, “0” .. “9” ‘ () + , - . / : = ?
TeletexString	The T.61 or ISO 6937 character set, a set which uses one or two octets to specify more than 255 different characters, for example, the character É is specified by the two characters “ ’ E”.
T61String	
VisibleString	Printable characters, including space, from ISO 646 (“ASCII”), but no format control characters like Carriage Return or Line Feed.
ISO646String	
IA5String	IA5 (ISO 646, “ASCII”).
GraphicString	Can contain characters from several different character sets, using ISO 2022 codes to switch from one character set to another character set within the string. Can only contain printable characters and space, not format control characters.
GeneralString	Same as GraphicString, but can also contain formatting characters.
UniversalString	ISO 10646.
CharacterString	Can contain characters from multiple character sets, using ISO 2022 codes to switch between the sets.

Character Strings have a special kind of subtype only available for Character Strings. It is called Permitted Alphabet, and uses a list of characters allowed in a new type. Example: **PrintableString (FROM("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"))**

1.9. Structured types

Structured types specify new types by combining several components of one or more already defined types. This table lists the basic constructed types in ASN.1.

SET	A list of component fields, like a record in a data base. the components can be included in any order, and the order of the components when transmitted does not convey any information.	Chairmen ::= SET { democratic chairman [0] General String, republican chairman [1] General String }
SEQUENCE	Similar to SET , but the fields must be sent in a certain order.	Ingredients ::= SEQUENCE { peas REAL, eggs INTEGER }
SET OF	Zero, one or more components, all of the same type. The order of the components conveys no information.	Ingredients ::= SET OF Ingredient Couple ::= SET SIZE (2) OF Person
SEQUENCE OF	Like SET OF , but order has significance.	Children ::= SET OF Person
CHOICE	Has as value one of a listed number of alternative types.	Vehicle ::= CHOICE { Bus, Car, Bicycle }

For the **SET OF** and **SEQUENCE OF** types, it is possible to indicate that one or more of the components need not be included. Example:

```
KnownParents ::= SEQUENCE OF {  
    father Male OPTIONAL,  
    mother Female OPTIONAL }
```


Exercise 16

In a protocol for transferring personal data between two computers, a social security number is transferred. This number consists of only digits, blanks and dashes. Name (not split into first name and surname, max 40 characters) can also be transferred if known, and an estimated yearly income can be transferred if known. Both of these values are optional, only the social security number is mandatory. Specify using the **SET** construct of ASN.1 a datatype to transfer this information.

Exercise 17

Assume that a name is to be transferred as two fields, one for given name and one for surname. How can the solution to Exercise 16 be changed to suit this case?

Exercise 18

Define a datatype **FullName** which consists of three elements in given order: *Given name*, *Initials* and *Surname*. *Given name* and *Initials* are optional, but *Surname* is mandatory.

Exercise 19

Define a data type **BasicFamily** consisting of 0 or 1 **husband**, 0 or 1 **wife** and 0, 1 or more **children**. Each of these components are specified as an **IA5String**.

Exercise 20

Define a datatype **ChildLessFamily**, based on **BasicFamily** from Exercise 16. Exercise 21 be changed to suit this case?

1.1.36. Inner subtyping

A special kind of subtypes can be specified for constructed types. This is an inner subtype. By this is meant that you specify a subtype for one or more of the components.

For **SET OF** and **SEQUENCE OF**, the construct **WITH COMPONENT** is used to

specify a subtype of the type of the element. Example:

Age ::= INTEGER

People ::= SET OF Age

Children ::= People (WITH COMPONENT (1 .. 14))

For **SET** and **SEQUENCE**, the construct **WITH COMPONENTS** is used to specify subtypes for one or more of the components. Example 1:

**Person ::= SEQUENCE {
 name GeneralString,
 age INTEGER }**

Adult ::= Person WITH COMPONENTS { ... , age (15 .. MAX) }

Example 2:

**Parents ::= SEQUENCE {
 father Person OPTIONAL,
 mother Person OPTIONAL }**

SingleMother ::= Parents (WITH COMPONENTS { Father ABSENT, ... }

Thus, in a subtype, an element which was **OPTIONAL** in the original type may be specified as **PRESENT**, **ABSENT** or **OPTIONAL** in the subtype.

SingleMother is a subtype of **Person**, specified by specifying a subtype of one of its components, the age component. “...” specifies that all the other components are unchanged.

Example 3:

**NormalName ::= SEQUENCE {
 givenName [0] GraphicString OPTIONAL,
 surName [1] GraphicString OPTIONAL,
 generation [2] GraphicString OPTIONAL,
 age [3] INTEGER
}**

```
RoyalName ::= NormalName
( WITH COMPONENTS {
    givenName PRESENT,
    surName ABSENT,
    generation PRESENT
    age (18.. MAX) }
)
```

Exercise 21

Define a datatype **FullName** which consists of three elements in given order: *Given name*, *Initials* and *Surname*. *Given name* and *Initials* are optional, but *Surname* is mandatory.

Exercise 22

Define a data type **BasicFamily** consisting of 0 or 1 **husband**, 0 or 1 **wife** and 0, 1 or more **children**. Each of these components are specified as an **IA5String**.

Exercise 23

Define a datatype **ChildLessFamily**, based on **BasicFamily** from Exercise 16.

Exercise 24

Given the ASN.1-type:

```
XYCoordinate ::= SEQUENCE {
    x REAL,
    y REAL
}
```

Define a subtype which only allows values in the positive quadrant (where both x and y are ≥ 0).

Exercise 25

Given the ASN.1 type:

```
SET {
  author Name OPTIONAL,
  textbody IA5String }
```

Define a subtype to this, called **AnonymousMessage**, in which no **author** is specified.

1.1.37. Choice Type

The possible values for the Choice type is the total of all the values of all the component types. The choice type indicates that always exactly one of the alternatives will be sent. Example:

```
Identification ::= CHOICE {
  textualname GeneralString,
  identitynumber NumericString }
```

If you want to define a subtype which can only have one of the alternatives in a choice, this can be specified as:

```
TextualIdentification ::= Identification (WITH COMPONENTS {textualname})
```

There is a shortcut notation for this,

```
TextualIdentification ::= textualname < Identification
```

Exercise 26

Given the data types **Aircraft**, **Ship**, **Train** and **MotorCar**, define a datatype **Vessel** whose value can be any of these datatypes.

Exercise 27

What is the difference between the data type:

```
NameListA ::= CHOICE {
  ia5 [0] SEQUENCE OF IA5String,
  gs [1] SEQUENCE OF GeneralString
}
```

and the data type:

```

NameListB ::= SEQUENCE OF CHOICE {
  ia5 [0] IA5String,
  gs [1] GeneralString
}

```

How is it in both alternatives above possible to define a new data type **GeneralNameList** which only can contain a **GeneralString** element?

Exercise 28

The by-laws of a society allows two kinds of votes:

- (a) The voters can select one and only one of 1 .. N alternatives. The alternative which gets the most total votes wins.
- (b) The voters can indicate a score of between 0 and 10 for each of the choices 1 .. N. The choice which gets highest total score wins.

Specify an ASN.1 data type which can be used to report the votings of a person to the vote collection agent, and which can be used for both kinds of votes. The name of the voter shall be included in the report as an **IA5String**.

Exercise 29

Suggest a textual encoding for Exercise 25 using ABNF.

1.1.38. Any Type

The Any type is a way of introducing something, whose format is not defined in the standard, and where you expect future usage to use different format at different times. There are two variants:

- ① **Vehicle ::= ANY**
- ② **SEQUENCE {**
 type-of-vehicle INTEGER,
 Vehicle ::= ANY DEFINED BY type-of-vehicle }

With ①, the receiving computer will have to analyse the value to find out which format it has. With ②, the number (**type-of-vehicle** in the example) will give some kind of information to the receiving computer about the format of

the **ANY**-formatted data.

With the ② syntax, **type-of-vehicle** can either be an **INTEGER** or an **OBJECT-IDENTIFIER**. The difference between an **INTEGER** and an **OBJECT-IDENTIFIER** is that if two different groups, independently define two different extensions, with different format for what they put in the **ANY**, they might choose the same value for **type-of-vehicle**, and then the receiving agent might confuse the two values. **OBJECT-IDENTIFIER** is a special kind of identification tag, which is always globally unique. No two will ever define two **OBJECT IDENTIFIERS** with the same value. The method for defining globally unique **OBJECT IDENTIFIERS** is similar to the method of assigning globally unique domains in the Domain Name System (DNS). The tree structure in Figure 4 is used to distribute **OBJECT IDENTIFIERS**.

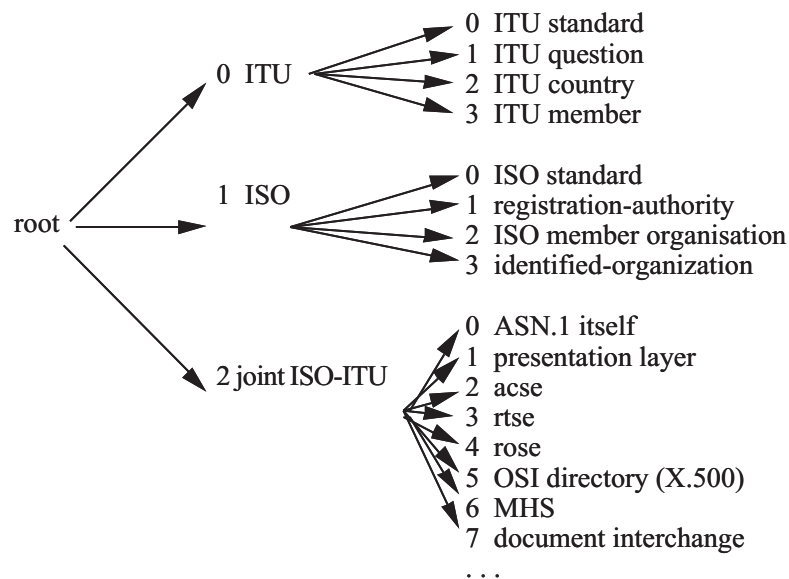


Figure 4: Domain name tree used in selecting **OBJECT IDENTIFIERS**

1.1.39. Tags

Look at the three examples below:

```

Name      ::= SEQUENCE {
    givenName      [0] VisibleString OPTIONAL,
    surName        [1] VisibleString OPTIONAL }
  
```

```

Name      ::= SET {
              givenName      [0] VisibleString,
              surName        [1] VisibleString }

Name      ::= CHOICE {
              numericName     NumericString,
              alphabeticName   VisibleString }

```

In example ①, both elements are optional. The tags [0] and [1] are necessary, because otherwise the receiving computer would not know, when it got only one string, whether this string was givenName or surName.

In example ②, the tags are necessary, because otherwise the receiving computer would not know if the first string was the givenName or the surName, since values of SET types can be sent in arbitrary order.

In example ③, the alternatives have different base type, NumericString and VisibleString, so the receiving computer can look at the UNIVERSAL tag to know which of the alternatives it got.

In summary, the tags for the elements must be different for components in a SET, for components in SEQUENCEs with OPTIONAL elements, and for components in a CHOICE. If the base type is not different, tags must be added to make them different.

Tags are labels used to differentiate between types. Tags are necessary in certain cases, but can be used also when they are not required. It is regarded as good ASN.1 usage to use the tags, also when they are not absolutely necessary. The advantage with using tags, even when they are not needed, is that they will make it easier for an old implementation to handle data in a new format, defined in a newer version of the standard. (This is not true if the Packed Encoding Rules, PER, are used.)

A tag has two components, a *class* component and a *number* component. There are four classes of tags as shown in Table 1.

Table 9: Tag classes

Class	Example	Description
Application	[APPLICATION 3]	Is used in the same way everywhere in an ASN.1 module. Use of this tag has problems, mainly when ASN.1 definitions are exported from one module to another.
Private	[PRIVATE 4]	Allows a company to make its own extensions. Also this tag has problems, because it is not possible to distinguish between two extensions made by different companies.
Context	[7]	This tag is only valid in its immediate context, such as a SET, SEQUENCE or CHOICE. It is the best tag to use if the UNIVERSAL tag is not enough.

The 1994 extension of ASN.1 introduced a fifth tag declaration **AUTOMATIC**. But AUTOMATIC does not define a new tag class, it specifies that the tag is to be computed automatically when compiling the ASN.1 code.

Here is an example of the use of tags:

①

Name

::= SET {

given name

[0] VisibleString,

surname

[1] VisibleString }

②

PersonnelRecord

::= SET {

name

[0] Name,

wage

[1] INTEGER }

Even if these two ASN.1 type declarations occur in the same module, they will not be confused. The tag [0] means something different in the ① and the ② type declaration.

The pre-defined **UNIVERSAL** tags are listed in Table 10.

Table 10: UNIVERSAL tags in ASN.1

<i>Simple types</i>		<i>Structured types</i>	
1	BOOLEAN	16	SEQUENCE
2	INTEGER	16	SEQUENCE OF
3	BIT STRING	17	SET
4	OCTET STRING	17	SET OF
5	NULL	(i)	CHOICE
6	OBJECT IDENTIFIER	(ii)	ANY
9	REAL	(i)	No special tag is needed, the tags of the components are used
10	ENUMERATED	(ii)	The tag is specified inside the ANY value, and can thus be any possible ASN.1 tag
<i>Character String Types</i>		<i>UsefulTypes</i>	
12	UTF8String	7	ObjectDescriptor
18	NumericString	8	EXTERNAL
19	PrintableString	23	UTCTime
20	TeletexString	24	GeneralizedTime
21	VideotexString		
22	IA5String		
25	GraphicString		
26	VisibleString		
27	GeneralString		
28	UniversalString		
29	CharacterString		
30	BMPString		

1.1.40. Explicit and Implicit tags

Suppose you have the following ASN.1 declaration:

```
Name ::= SEQUENCE {
  givenName [0] VisibleString OPTIONAL,
  initials [1] VisibleString OPTIONAL,
  surName [2] VisibleString OPTIONAL }
```

When this is encoded using the Basic Encoding Rules (BER), two tags will be sent for every element. First the Context-Dependent tag [0], [1] or [2], and then the **UNIVERSAL** tag for **VisibleString** (28, see Table 10). This is not really necessary. The declaration can then be changed to:

```
Name ::= SEQUENCE {
  givenName [0] IMPLICIT VisibleString OPTIONAL,
  initials [1] IMPLICIT VisibleString OPTIONAL,
  surName [2] IMPLICIT VisibleString OPTIONAL }
```

The word **IMPLICIT** specifies that only the tag defined in the text ([0], [1] or [2],) need be sent, not the **UNIVERSAL** tag for **VisibleString**.

It is also possible, in the head of an ASN.1 module, to specify that all tags are to be **IMPLICIT** where possible, even if this is not explicitly specified. The head of an ASN.1 module can be

```
DEFINITIONS ::=      - - Implies Explicit tags
DEFINITIONS IMPLICIT TAGS ::=
DEFINITIONS EXPLICIT TAGS ::=
DEFINITIONS AUTOMATIC TAGS ::= (In the 1994 version ASN.1)
```

If the module head specifies **IMPLICIT TAGS**, the ASN.1 code within the module must use **EXPLICIT** where this kind of tag is wanted. If the module head specifies **EXPLICIT TAGS**, the ASN.1 code within the module must use **IMPLICIT** where this is wanted (more about this in the section Modules on page 65).

Exercise 30

Assume an ASN.1-module which looks like shown below; Change this ASN.1 module, so that the same coding is specified, but with tag defaults **IMPLICIT** instead of **EXPLICIT**.

```
WeatherReporting {2 6 6 247 1} DEFINITIONS EXPLICIT TAGS ::=
```

```
BEGIN
```

```
WeatherReport ::= SEQUENCE {
  height [0] IMPLICIT REAL,
  weather [1] IMPLICIT Wrecord
}
```

```

Wrecord ::= [APPLICATION 3] SEQUENCE {
    temp Temperature,
    moist Moisture
    wspeed [0] Windspeed OPTIONAL
}

Temperature ::= [APPLICATION 0] IMPLICIT REAL

Moisture ::= [APPLICATION 1] REAL

Windspeed ::= [APPLICATION 2] REAL

END -- of module WeatherReporting

```

Exercise 31

Which of the tags in the example below can be removed while the receiving computer will still be able to interpret what you send?

```

Record ::= SEQUENCE {
    GivenName [0] PrintableString
    SurName [1] PrintableString }

Record ::= SET {
    GivenName [0] PrintableString
    SurName [1] PrintableString }

Record ::= SEQUENCE {
    GivenName [0] PrintableString OPTIONAL
    SurName [1] PrintableString OPTIONAL }

```

Exercise 32

Which of the tags in the examples below can be removed, while the receiving computer will still be able to deduce what you meant, and assuming that **AUTOMATIC** tagging is not specified.

```

Colour ::= [APPLICATION 0] CHOICE {
    rgb [1] RGB-Colour,
    cmg [2] CMG-Colour,
    freq [3] Frequency
}

```

```

RGB-Colour ::= [APPLICATION 1] SEQUENCE {
  red [0] REAL,
  green [1] REAL OPTIONAL,
  blue [2] REAL
}

CMG-Colour ::= SET {  cyan [1] REAL,
  magenta [2] REAL,
  green [3] REAL
}

Frequency ::= SET {fullness [0] REAL,
  freq [1] REAL
}

```

Exercise 33

The following ASN.1 construct is taken from the 1988 version of the X.500 standard. (**OPTIONALLY-SIGNED** is a macro, macros were replaced with a new construct in the 1994 version of ASN.1.)

```

ListResult ::= OPTIONALLY-SIGNED
CHOICE {
  listInfo SET {
    DistinguishedName OPTIONAL,
    subordinates [1]SET OF SEQUENCE {
      RelativeDistinguishedName,
      aliasEntry [0] BOOLEAN DEFAULT FALSE
      fromEntry [1] BOOLEAN DEFAULT TRUE},
    partialOutcomeQualifier [2]
    PartialOutcomeQualifier OPTIONAL
    COMPONENTS OF CommonResults },
  uncorrelatedListInfo[0] SET OF ListResult }

```

Exercise 34

Is there anything wrong in the ASN.1 code in Exercise 33.

Exercise 35

Why is there no identifier on the element **COMPONENTS OF**? What does it

mean?

Exercise 36

Why are there no context-dependent tags on some of the elements, but not on all of them?

1.10. Special types and Concepts

1.1.41. Time Types

GeneralizedTime is a built-in type for specifying time and date. Its format follows an ISO standard for dates. **UTCTime** is a shorter variant, where year is specied with only two digits (beware!). The same point in time, 9 minutes and 25.2 seconds after 9 p.m in the U.S. Eastern Time Zone can be specified in three ways using **GeneralizedTime**:

time-to-stop-working GeneralizedTime ::= "19880726210925.2" or

time-to-stop-working GeneralizedTime ::= "19880726210925.2Z" or

time-to-stop-working GeneralizedTime ::= "19880726210925.2-0500"

1.1.42. Use of Object Identifiers, Any, External

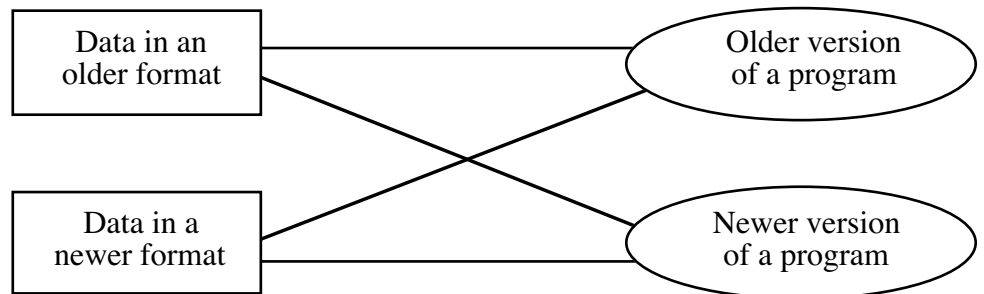


Figure 5: Allow communication between old and new programs

Figure 5 shows a common problem in distributed systems, where many pieces

of software, which have been developed at different times by different people, need to work together. Thus, an older version of a program may receive data from a newer version, in a newer format, which did not even exist when the older version of the program was produced.

ASN.1 contains special constructs to make this possible: constructs for specifying data elements which can be bypassed by older versions of a program and interpreted by newer versions of the same program.

Here is an excerpt from the ASN.1 in the 1988 version of X.420, which shows one way of using these extension facilities:

```
ExtensionsField ::= SET OF HeadingExtension

HeadingExtension ::= SEQUENCE {
  type OBJECT IDENTIFIER,
  value ANY DEFINED BY type DEFAULT NULL NULL }
}

HEADING-EXTENSION MACRO ::=
BEGIN
  TYPE NOTATION ::= "VALUE" type | empty
  VALUE NOTATION ::= VALUE (VALUE OBJECT IDENTIFIER)
END
```

One heading extension, defined in the 1988 version of X.400 using this construct, is:

```
languages HEADING-EXTENSION
VALUE SET OF Language
::= id-hex-languages

Language ::= PrintableString (SIZE (2..2))
```

In the 1992 version of ASN.1, the ANY and MACRO constructs were abolished, and replaced by the new CLASS construct. The above extension facility is with the 1994 X.420 syntax instead defined as:

```
ExtensionsField ::= SET OF IPMSExtension

IPMSExtension ::= SEQUENCE {
  type IPMS-EXTENSION.&id,
  value IPMS-EXTENSION.&Type DEFAULT NULL:NULL }
```

```

IPMS-EXTENSION ::= CLASS {
  &id  OBJECT IDENTIFIER UNIQUE,
  &Type          DEFAULT NULL }
WITH SYNTAX { [VALUE &Type , ] IDENTIFIED BY &id }

```

The heading extension for languages is with the new 1992 syntax defined as:

```

languages IPMS-EXTENSION ::= {VALUE SET OF Language,
IDENTIFIED BY id-hex-languages}

```

```

Language ::= PrintableString (SIZE (2..5) )

```

As is shown in the example above, a typical such extensible element has two subfields, one field with the name **type** and one field with the name **value**. The type field is particular for every kind of extended field. The value field has a structure which is called **ANY DEFINED BY type** with the 1988 notation and **IPMS-EXTENSION.&Type** with the 1992 notation. This means that, for different values of **type**, different ASN.1 specifications will describe the value. A new extension can then be identified by a new **type** value, and a new ASN.1 specification of the value structure, like **SET OF Language** in the example above.

The **type** field in the example above is specified as an **OBJECT IDENTIFIER**. It can also be specified as an **INTEGER**. The difference between **OBJECT IDENTIFIER** and **INTEGER** is that there are rules defined which allows anyone to obtain a new **OBJECT IDENTIFIER**, which will then be different from any other **OBJECT IDENTIFIER** obtained by anyone else. In the case of **integer**, there is no protection against two different developers using the same integer for two different extensions, which would, of course, create a mess if their systems were connected. Thus, in practice, **integer** only allows extensions made by the international standards organizations, while **OBJECT IDENTIFIER** allows anyone to make his own extension, without risk of a conflict with another extension made by some other person or organization.

The value of an extension can (with the 1988 notation) be either **ANY** or **EXTERNAL**. The difference between the two is that **ANY** refers to an extension specified in ASN.1, while **EXTERNAL** allows an extension specified in some language other than ASN.1.

An implementation, which encounters an extended field, can react to the extended field in four different ways:

1. The implementation knows about the extension and utilizes it in the way it

was intended to be used.

2. The implementation receives the unknown fields, removes them and continues handling the message as if they had never been there.
3. The implementation receives the unknown fields, saves them, and transfers them further along with the other data, even though the implementation does not understand and cannot use the information in the extended field.
4. The implementation recognizes that this is an extended field and then gives an error code saying that it cannot handle the data because it contains an extension it does not understand.

Note that (4) is different from the kind of error that was produced when the incoming data were incorrect. Such errors, called *protocol violations*, carry a risk that a program will crash completely or react in unpredictable ways.

For envelope extensions, the X.400 standard for electronic mail specifies for each extension whether reaction (3) (*noncritical* extension) or (4) (*critical* extension) should be used by an implementation which does not understand the extension. For heading extensions, X.400 states that reaction (3) is suitable.

1.1.43. Object Descriptor and External types

Example of use of the **ObjectDescriptor** type:

ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString

This type is used when you use the **ANY** or **EXTERNAL** types, to give a human-readable description of the data type, in addition to the machine-parseable type code.

The **EXTERNAL** type can actually be specified in ASN.1. Its structure is:

EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE

```

{  direct-reference OBJECT-IDENTIFIER OPTIONAL,
   indirect-reference INTEGER OPTIONAL,
   data-value-descriptor ObjectDescriptor OPTIONAL,
   encoding CHOICE {
       single-ASN1-type [0] ANY,
       octet-aligned [1] IMPLICIT OCTET STRING,
       arbitrary [2] IMPLICIT BIT STRING
   }
}
```


This is a more advanced version of **ANY**, where the type of the unspecified data is specified in one or more of three ways: An **OBJECT IDENTIFIER**, An **INTEGER** or a text string. At least one of them must be specified.

1.1.44. Modules

A module is a named collection of ASN.1 type and value definitions. Its structure is as follows:

```
<moduleReference>    <obj-id> DEFINITIONS <tag-defaults> ::=
BEGIN
    EXPORTS <type and value references>;
    IMPORTS <type and value references>
    FROM <moduleReference> <obj-id>;
    ...
    <type and value definitions>
    ...
END
```

EXPORTS and **IMPORTS** are tools for using type definitions from one module in another module. Example of modules with **IMPORTS** and **EXPORTS**:

```
CargoHandling { 1 2 4711 17 } DEFINITIONS EXPLICIT TAGS ::=

BEGIN EXPORTS Box, Container ;

Box ::= SEQUENCE {
    height INTEGER, - - in centimeters
    width INTEGER, - - in centimeters
    length INTEGER } - - in centimeters

Container ::= SEQUENCE
    weight INTEGER, - - in kilograms
    volume Box }

END - - of CargoHandling

TrainCargo { 1 2 4711 18 } DEFINITIONS EXPLICIT TAGS ::=

BEGIN IMPORTS Box, Container FROM CargoHandling { 1 2 4711 17 };
```

```

TrainContainer ::= Container
    ( WITH COMPONENTS
      {   weight ( 0 .. 5000 ), volume }
    )

```

```

Carriage ::= SET SIZE (2..4) OF Container

```

```

END - - of TrainCargo

```

Example of a module specification using dot notation:

```

CargoHandling { 1 2 4711 17 } DEFINITIONS EXPLICIT TAGS ::=

```

```

BEGIN EXPORTS Box, Container ;

```

```

Box ::= SEQUENCE {
    height INTEGER, -- in centimeters
    width  INTEGER, -- in centimeters
    length INTEGER } -- in centimeters
Container ::= SEQUENCE
    weight INTEGER, -- in kilograms
    volume Box }

```

```

END -- of CargoHandling

```

```

TrainCargo { 1 2 4711 18 } DEFINITIONS EXPLICIT TAGS ::=

```

```

BEGIN

```

```

Container ::= CargoHandling{ 1 2 4711 17 }.Container
    ( WITH COMPONENTS
      {   weight ( 0 .. 5000 ), volume }
    )

```

```

Carriage ::= SET SIZE (2..4) OF Container

```

```

END -- of TrainCargo

```

Exercise 37

Given the following ASN.1 module:

```

Driving {1 2 4711 17} DEFINITIONS EXPLICIT TAGS ::=
    BEGIN

```

```
MainOperation ::= SEQUENCE {  
    wheel [0] REAL,  
    brake [1] REAL,  
    gas [2] REAL }
```

END

Define an ASN.1 module **CarDriving**, which imports **MainOperation** from the module above, and defines a new datatype **FullOperation** which in addition to **MainOperation** also includes switching on and of the left and right blinking lights, and setting the lights as unlit, parking lights, dimmed light and full beam.

1.11. Encoding Rules

1.1.45. Basic Encoding Rules (BER)

The Basic Encoding Rules (BER) are the most commonly used encoding rules for interpreting ASN.1 syntax into protocol units to be sent over the net. BER is based on the length-value format (see page 18). Figure 6 shows two examples of BER encodings. Primitive encoding is used for simple types, types which have no components. Constructed encoding is used for constructed types, for example **SET**, **SET OF**, **SEQUENCE**, **SEQUENCE OF**. As is shown by the figure, the value of a constructed type is itself split into a series of Tag-Length-Value objects.

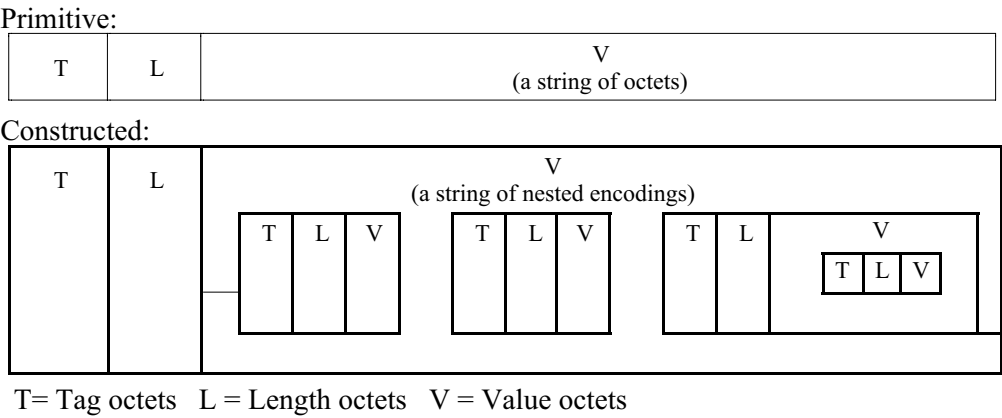


Figure 6 Tag-Length-Value encoding in BER

1.1.46. The Tag or Identifier field

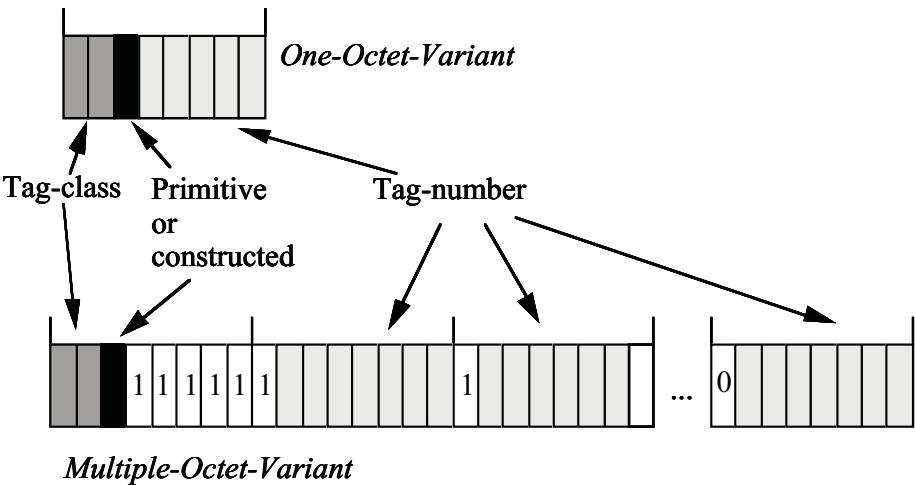


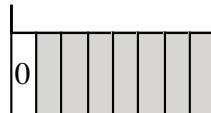
Figure 7: Use of bits in BER encoding

The first two bits contain the tag class, with 00=Universal tag, 01=Application tag, 10=Context tag and 11=Private tag. The third bit is 0 for a primitive type and 1 for a constructed type. If the tag number is between 0 and 30, it is encoded in the remaining give bits (One-Octet-Variant in Figure

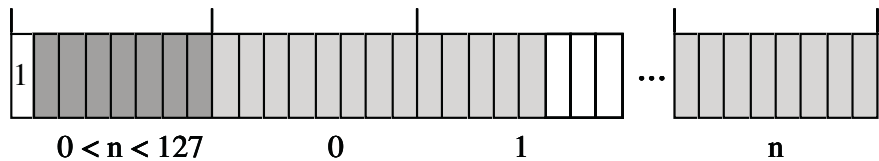
7). If the tag class is higher than 30 (Multiple-Octet-Variant in Figure 7), the remaining five bits are all 1-s, and the tag value is encoded in the last 7 bits of one or more succeeding octets. The first bit of each such succeeding octet is 0 for the last octet, 1 for all but the last octet.

1.1.47. The Length Field in BER

Short form



Long form



Unlimited form, ends with an octet with eight 0-s



Figure 8: The Length field in BER

As is shown in Figure 8, the length field in BER also has a short, one-octet form and a long, multiple-octet form. The short form has the first bit 0, and the remaining 7 bit can contain a length between 0 and 127. In the long form, the first bit is 1, and the remaining 7 bits of the first octet contains the number of additional octets. The length is then encoded as a binary number in the rest of the bits.

There is also an unlimited form. It starts with an octet with 1 in the first 1 and 0 in the rest of the bits, and ends with an octet with eight 0-s. The unlimited form is always constructed, i.e. its value must always be organized into Tag-Length-Value groups. Even though the end is marked with an octet with eight 0-s, it is still possible to have octets with all 0-s in the value, if these octets occur inside the Tag-Length-Value groups. An octet with eight 0-s is

only interpreted as an end of the unlimited form, if it occurs immediately after the end of a Tag-Length-Value group, as is shown below.

I	1	0	0	0	0	0	0	0	I	L	C	...	I	L	C	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	-----	---	---	---	---	---	---	---	---	---	---	---	---	---

1.1.48. The BER Value Octet

Table 11 shows how the BER value octet is defined for different types.

Table 11: The BER value octet

Boolean	One Single Octet. FALSE = 00000000 TRUE = all other values.
Integer	Two-complement notation, coded using the smallest number of necessary bits.
Enumerated	Same coding as Integer.
Null	No value octet at all.
Object Identifier	A packed sequence of integers. The first integer contains the first two labels, after that, one label in each encoded integer.
Set, Sequence, Set-of, Sequence-of	Nested sequences of coding of the components.
Choice, Any	Same code as for the selected element.
Real	Four variants: 0 is represented by no value octets, 01000000 represents PLUS-INFINITY and 01000001 represents MINUS-INFINITY Other values are coded as binary values with the base 2, 8 or 16, or as decimal values according to the ISO 6093 standard. The first octet indicates which coding method is used.
String	Strings have two encoding variants, primitive and constructed. In the primitive form, the values are directly put into the value octets. In the constructed form, the string is split into a series of substring, as if the ASN.1 definition had been: BIT STRING ::= [UNIVERSAL 3] IMPLICIT SEQUENCE OF BIT STRING OCTET STRING ::= [UNIVERSAL 4] IMPLICIT SEQUENCE OF OCTET STRING IA5String ::= [UNIVERSAL 22] IMPLICIT SEQUENCE OF OCTET STRING

1.1.49. Variants of the encoding of a string with tag

Figure 9 shows some examples of the encoding of a string, with and without a

preceding context-sensitive tag.

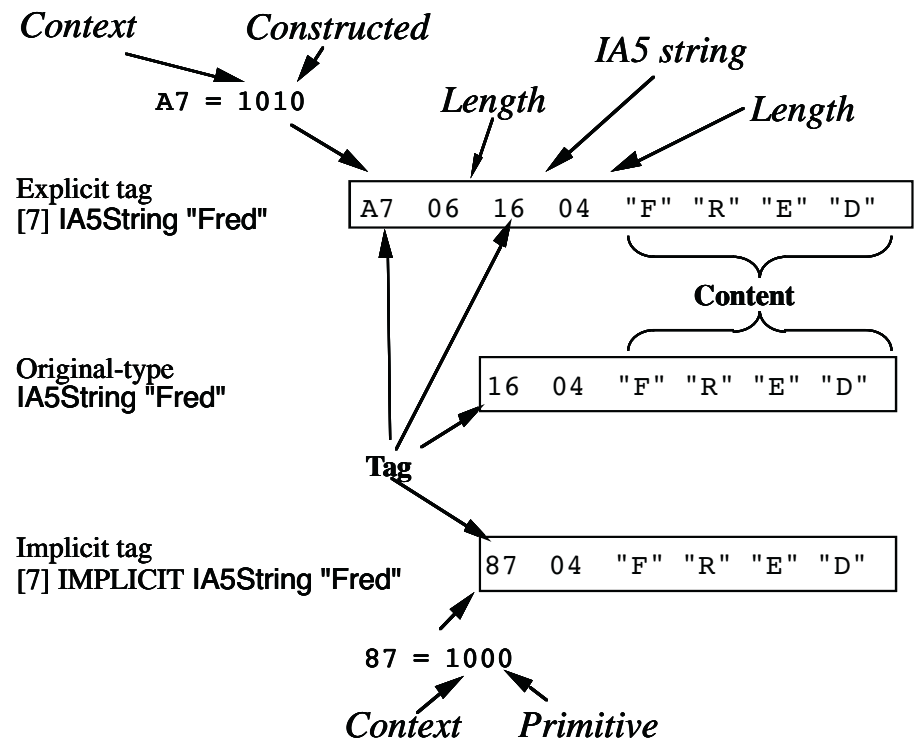


Figure 9: Encoding of a tagged string

1.1.50. Example of the coding of a SEQUENCE

```

HeadOfState ::= [APPLICATION 17] SEQUENCE
{
  name IA5 STRING,
  type ENUMERATED {
    president (0),
    emperor (1),
    king (2) }
  birthyear INTEGER OPTIONAL }

swedishKing ::= {
  name "Carl XVI Gustav",
  type king,
  birthyear 1946 }

```

This might be coded as shown below (hexadecimal numbers):

49	18	<i>Application tag 17 and Length of the whole construct</i>															
16	0F	C	a	r	I		X	V	I		G	u	s	t	a	v	<i>Name</i>
0A	01	02	<i>type = king</i>														
02	02	1E	14	<i>birthyear = 1946</i>													

The hexadecimal value 16 in the first octet of the second line, the tag of the text string, is made up as follows:

$22_{10} = 16_{16} = \text{class universal}(00),$
 form primitive(0), tag number
 IA5String(22)

0	0	0	1	0	1	1	0
---	---	---	---	---	---	---	---

Exercise 38

Given the ASN.1 definition

Surname ::= [APPLICATION 1] IA5String

hername Surname ::= "Mary"

Show its coding in BER

Exercise 39

Given the ASN.1 definitions

Light ::= ENUMERATED {
 dark (0),
 parkingLight (1),
 halfLight (2),
 fullLight (3) }

daylight Light ::= halfLight

give a BER encoding of this value.

Exercise 40

Given the following ASN.1 definitions and explicit tags


```

BreakFast ::= CHOICE {
continental [0] Continental,
english [1] English,
american [2] American }

Continental ::= SEQUENCE {
beverage [1] ENUMERATED {
coffea (0), tea(1), milk(2), chocolade (3) } OPTIONAL,
jam [2] ENUMERATED {
orange(0), strawberry(1), lingonberry(3) } OPTIONAL }

English ::= SEQUENCE {
continentalpart Continental,
eggform ENUMERATED {
soft(0), hard(1), scrambled(2), fried(3) }

Order ::= SEQUENCE {
customername IA5String,
typeofbreakfast Breakfast }

firstorder Order ::= {
customername "Johan",
typeofbreakfast {
    english {
        continentalpart {
            beverage tea,
            jam orange
        }
        eggform fried
    } } }

```

Give an encoding of **firstorder** with BER.

1.1.51. Different Encoding Rules for ASN.1

Most standards based on ASN.1 use the Basic Encoding Rules. They are not very efficient, the redundancy causes about twice as many octets as the Packed Encoding rules. In addition to BER, DER and CER are also used, because they are better suited to security applications. BER allows the same information to be coded in different ways. For example, **TRUE** can in BER be represented by any nonzero octet value, and strings can in BER be encoded

with either definite length or indefinite length encoding. This means that a security checksum may fail for two different BER encodings of exactly the same data. With DER and CER, there are no options for coding the same information in more than one way, and security checksums will thus work better with DER and CER than with BER. See Table 12 for a list of different encoding rules for ASN.1.

Table 12: Different encoding rules

BER = Basic Encoding Rules	Not very efficient, much redundancy, good support for extensions
DER = Distinguished Encoding Rules	No encoding options (for security hashing), always use definite length encoding
CER = Canonical Encoding Rules	No encoding options (for security hashing), always use indefinite length encoding
PER = Packed Encoding Rules	Very compact, less extensible
LWER = Light Weight Encoding Rules	Almost internal structure, fast encoding/decoding

1.12. ASN.1 compilers

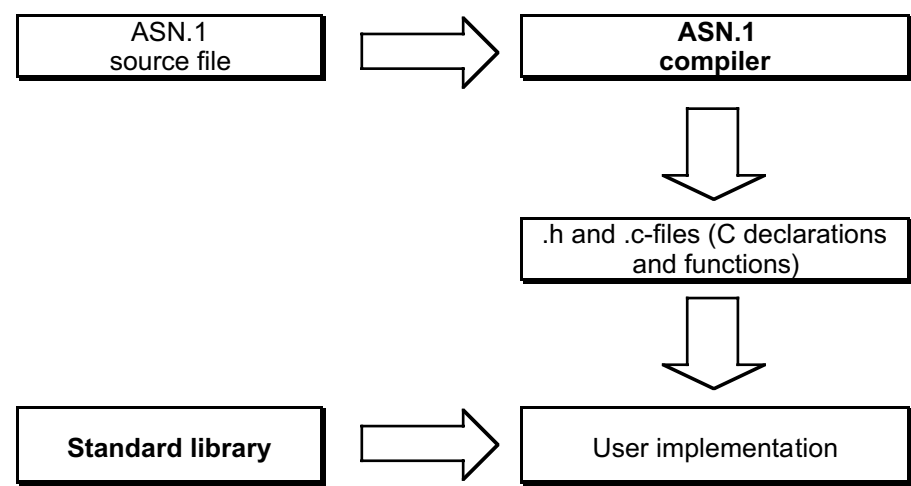


Figure 10: ASN.1 compilers

As shown in Figure 10, the ASN.1 compiler takes ASN.1 declaration files and

compiles this into, usually, source code in the C programming language. This source code is then combined with standard libraries and included as part of the user application source code. Some ASN.1 compilers produce code which directly compiles the ASN.1 into code for exactly this rule. Such compilers need less standard libraries. Other compilers compile to ASN.1 source code into some kind of data structure, which is then interpreted during execution. They need more standard libraries, since these libraries will include the interpreter code.

4. *HTML and CSS*

Objectives

HTML and CSS encode text with markup. The markup controls the layout and gives some structural information about the text.

Keywords

HTML

CSS

W3C

1.13. (Hypertext Markup Language)

This book is not a complete guide to HTML [W3C HTML401]. Here is just a short description of some central concepts of HTML, since these concepts are used later in this book.

A HTML document is a document which contains special codes called *markup*, which control the layout of the document. Example:

HTML document:	What the user sees:
<pre><p>First paragraph containing one boldface word. <p>Second paragraph with a line break
text after the line break.</pre>	First paragraph containing one boldface word. Second paragraph with a line break text after the line break.

As shown in this example, the `<p>` tag indicates the start of a new paragraph, the `` tag indicates bold-face text, the `` tag indicates the end of bold-face text, and the `
` tag indicates a line break.

Since certain characters are used for markup, such as “<”, “>”, “&” and “””, they must be coded if they are to be included as text and not as markup.


Example:

HTML document:	What the user sees:
<pre>Jim&apos;s e-mail address is Jim Sim &gt;;jsim&foo.bar&gt;;.</pre>	Jim's e-mail address is Jim Sim <jsim@foo.bar>.

An HTML document can contain links to other documents. Example:

HTML document:	What the user sees:
<pre>Read the web pageassociated with this book.</pre>	Read the web page associated with this book.

The links to other document contain URIs (see chapter 4.4). To include pictures in an HTML document, you include a link to a separate file, containing the picture in some graphics format, such as for example GIF. Example:

HTML document:	What the user sees:	
<code>This is the logo of the Internet Engineering Task Force.</code>		This is the logo of the Internet Engineering Task Force.

An HTML document is split into main sections as shown in this example:

<pre> <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"> <HTML> <HEAD> <TITLE>Caves and Caverns in Sweden</TITLE> <META name="description" content="This site gives an overview of the most famous Swedish caves."> <META name="keywords" content="Sweden, cave, cavern, speleology, Lummelunda"> </HEAD> <BODY BGCOLOR="#FFFFFF"> <H1>Caves and Caverns in Sweden</H1> <P>The most famous Swedish cave is the Lummelunda Cave on the Island of Gotland in the Baltic Sea. </BODY></HTML> </pre>	Heading line which identifies which dialect of HTML is used
	The head section contains information for the whole document and not directed at some particular part of the document. The head can also contain style sheets and executable code.
	The body section contains the actual text shown to users.

An HTML document can refer to other HTML documents, which are combined to produce the text shown to the user. Example:

HTML documents:		What the user sees:	
frameset.html	<pre> <HTML><HEAD> <TITLE>Framed document</TITLE> </HEAD> <FRAMESET COLS="25%,75%"> <FRAME NAME=left scrolling=no src="left.html"> <FRAMESET ROWS="50%,50%"> <FRAME NAME=top scrolling=no src="top.html"> <FRAME NAME=right scrolling=no src="bottom.html"> </FRAMESET> </FRAMESET> </HTML> </pre>	This is the left frame.	This is the top frame.
left.html	<pre> <HTML><HEAD> <TITLE>Left frame</TITLE> </HEAD><BODY> This is the left frame. </BODY></HTML> </pre>		
top.html	<pre> <HTML><HEAD> <TITLE>Top frame</TITLE> </HEAD><BODY> This is the top frame. </BODY></HTML> </pre>		This is the bottom frame.
bottom.html	<pre> <HTML><HEAD> <TITLE>Bottom frame</TITLE> </HEAD><BODY> This is the bottom frame. </BODY></HTML> </pre>		

1.14. Cascading Style Sheets (CSS)

HTML documents can be combined with style sheets, which specify how different parts of the HTML documents are to be shown to users. The language for these style sheets is called “Cascading Style Sheets” [WR3C CSS1, W3C CSS2]. Example:

HTML document:	What the user sees:
<pre> <html> <head> <title>CSS Example</title> <style type="text/css"> <!-- h1 { font-family: Helvetica; font-size: 16pt} .maintext { font-family: Times; font-size: 12pt} --> </style> </head> <body> <h1>This is the main heading</h1> <div class=maintext> <p>This is the text below the main heading.</p> </div></body></html> </pre>	<p>This is the main heading</p> <p>This is the text below the main heading.</p>

The style sheet in the example above specifies that all text with the tag `<h1>` should be shown with the font Helvetica and the size 16pt, and that all text whose tag has the attribute “class=maintext” should be shown with the font Times and the size 12 pt.

The `<!--` and `-->` commands above will make this text look like comments to old browsers. In the future, when web browsers generally understand the `<style>` element, this will not be necessary any more.

Style sheets can either be put into the `<head>` of the HTML document, or they can be put into separate files, which are referenced by the HTML document. The document above could thus instead have consisted of two files:

HTML document:	What the user sees:
<pre> <html> <head> <title>CSS Example</title> <LINK rel = " s t y l e s h e e t " href="styles.css"></style> </head> <body> <h1>This is the main heading.</h1> <div class=maintext> <p>This is the text below the main heading.</p> </div></body></html> </pre>	<p>This is the main heading</p> <p>This is the text below the main heading.</p>

CSS style sheet file “styles.css”:

```
h1 { font-family: Helvetica; font-size: 16pt}  
.maintext { font-family: Times; font-size: 12pt}
```

One central idea in Cascading Style Sheets is that there can be several different Style Sheets from the same document, which will show it in different ways. Different users may best be supported by style sheets suited to their needs. There is also an option for a user override the style sheet specified by the provider of a web page with his own alternative style sheet.

CSS can also be used with XML, see section 1.19 on page 97.

5. *Extensible Markup Language, XML*

Objectives

XML is a coding format which can combine structural information with layout information to control how XML is shown to users.

Keywords

XML

DTD

CSS

XSLT

1.15. Extensible Markup Language (XML) Introduction

XML (Extensible Markup Language), like ABNF, is a method for specifying nested textual encoding. XML is, however, similar to ASN.1 in that it easily allows complex structures. A particular property of XML is that it can be combined with layout information (using separate standards CSS = Cascading Style Sheets, and XSLT = Extensible Style Language Transformations) to convert the information into human-friendly text.

Like ASN.1, XML consists of two languages, one language for specifying the coding format, corresponding to ASN.1, called DTD (Document Type Definition) and another languages for the actual encoded data, corresponding to BER, called XML. DTD (like ASN.1 and ABNF) is a metalanguage, a language for specifying another language used for the actual encoded data.

XML has many superficial similarities to HTML. It is, however, different from HTML in that HTML has a fixed set of tags and attributes, specified in the HTML specification, while XML allows every application to specify its own tags and their attributes.

When describing ASN.1 and ABNF, it is natural to start by describing the metalanguage, and then go on to describe the actual coding format. With XML, descriptions usually start with the actual coding format, before describing the metalanguage. The reason for this is that the XML coding format is very easy to read and understand, while the metalanguage DTD is rather complex.

The octets sent to describe a person in XML might be:

(Boldface is not part of XML, just used here to make the text more readable.)

```
<PERSON>  
<NAME>John Smith</NAME>  
<BIRTHYEAR>1941</BIRTHYEAR>  
<WAGE>57000</WAGE>  
</PERSON>
```

If you prefer to separate the name into components, the octets sent might

instead be:

```
<PERSON>
  <NAME>
    <FIRST-NAME>John</FIRST-NAME>
    <SURNAME>Smith</SURNAME>
  </NAME>
  <BIRTHYEAR>1941</BIRTHYEAR>
  <WAGE>57000</WAGE>
</PERSON>
```

From these examples, you can see that XML-encoded data consists of a nested structure of tags and data within the tags. In this way, XML is very similar to HTML.

An XML *element* has a start-tag, contents, and an end-tag. Thus, in the example above, `<BIRTHYEAR>1941</BIRTHYEAR>` is an element, and `<BIRTHYEAR>` is the start-tag and `</BIRTHYEAR>` is the end-tag of this element.

The definition of the tags used, in the example `<PERSON>`, `<NAME>`, `<FIRST-NAME>`, `<SURNAME>`, `<BIRTHYEAR>` and `<WAGE>` are not pre-defined in XML, they are chosen by the user or application to suit its needs.

Exercise 41

Here is an example of part of an e-mail heading according to current e-mail standards.

```
From: Nancy Nice <nnice@good.net>
To: Percy Devil <pdevil@hell.net>
Cc: Mary Clever <mclever@intelligence.net>, Rupert Happy
   <rhappy@fun.net>
```

How might the same information be encoded using XML?

1.1.52. XML versus HTML

Here is a comparison of the main similarities and differences between XML and HTML:

Function	HTML	XML
Set of tags	Built-in, predefined set of tags specified in the HTML standard.	Every application or user can define its own element types and select their tags to suite the needs of this particular application.
End-tag	Not always required.	Always required.
Case sensitive	No, for example, <code><TITLE></code> and <code><title></code> are identical.	Yes, <code><TITLE></code> and <code><title></code> are two different tags, specifying two different element types. An element which starts with <code><TITLE></code> must end with <code></TITLE></code> , not with <code></title></code> .

Function	HTML	XML
Acceptance of coding errors	<p>Most web browsers accept many coding errors.</p> <p>Example:</p> <pre><I>Bold-italic text</I></pre> <p>is not correct HTML, but accepted by most web browsers. The example is incorrect, because the elements are incorrectly nested. The element <code><I></code> is neither inside or outside the element <code></code> tag. Correct HTML would be:</p> <pre><I>Bold italic text</I></pre> <p>(Element <code><I></code> inside element <code></code>)</p> <p>or</p> <pre><I>Bold italic text</I></pre> <p>(Element <code></code> inside element <code><I></code>)</p> <p>According to the liberal-conservative rule, it may still be wise to accept certain kinds of inaccurate data. But XML is a reaction to the way this rule has come to be interpreted for HTML, where a web browser is expected to accept and interpret almost any kind of vastly incorrect HTML text.</p> <p>The reason why faults are so common in HTML texts is that they are still often developed manually. Another reason is the multitude of variants of HTML, which make it difficult to test HTML for correctness. Some incorrect constructs (example: <code><CENTER></code>) do in fact work in more browsers than the corresponding correct constructs (<code><DIV ALIGN=CENTER></code> instead of <code><CENTER></code>). In the case of XML, texts will mostly be produced by software, which will reduce the amount of incorrect XML data.</p>	<p>Code must be syntactically correct, and only syntactically correct XML-encoded data should be accepted by an XML processor.</p>
Support in web browsers	Yes.	Yes in some newer versions.
Text layout and style	HTML tags and style sheets.	Style sheets and XSLT transformation code.

1.16. Document Type Definition (DTD)

The Document Type Definition (DTD) is a language for specifying the element types for a particular application of XML. The name of an element type is used in its start and end-tags. To understand this, compare ABNF, ASN.1 and XML:

Table 13: Relation between DTD and XML

Enviroment:	“ABNF”	“ASN.1”	“XML”
Language for specifying the encodings for a particular application.	ABNF	ASN.1	DTD (but not as strong typing as in ASN.1)
Language used to actually encode data.	Text, often as a list of lines beginning with a name, a colon, followed by a value.	BER (or some other ASN.1 encoding rule)	XML

It is not required that XML data has any DTD. You can send XML data without specifying any DTD, but for serious applications you should specify a DTD, since (i) this allows software to be able to check that your XML is syntactically valid (ii) it can be used as an aid in developing software to encode and decode the XML data. An XML document which has correct XML syntax, but no DTD, is said to be *well-formed*. An XML document which also has a DTD, and whose syntax agrees with the DTD, is said to be *valid*.

While a big advantage with XML is that its encoded data is so easy to read, a disadvantage is that the DTD language is not as neat as for example ASN.1.

When an XML text is based on a DTD, this is indicated by a `<!DOCTYPE>` element in the head of the XML text. Thus, an XML text may look like this:

```
<?xml version="1.0"?>
```

Specifies that this is XML-encoded data

```
<!DOCTYPE person SYSTEM "person.dtd">
```

Specifies where to find the DTD. "Person.dtd" can be a complete URL, which gives a globally unique reference to this DTD.

```
<PERSON>
```

Here comes the XML encoded according to this

```
<NAME>John Smith</NAME>
```

DTD.

```
<BIRTHYEAR>1941</BIRTHYEAR>
```

```
<WAGE>57000</WAGE>
```

```
</PERSON>
```

In Table 14 is an example of a DTD and an XML text encoded according to this DTD.

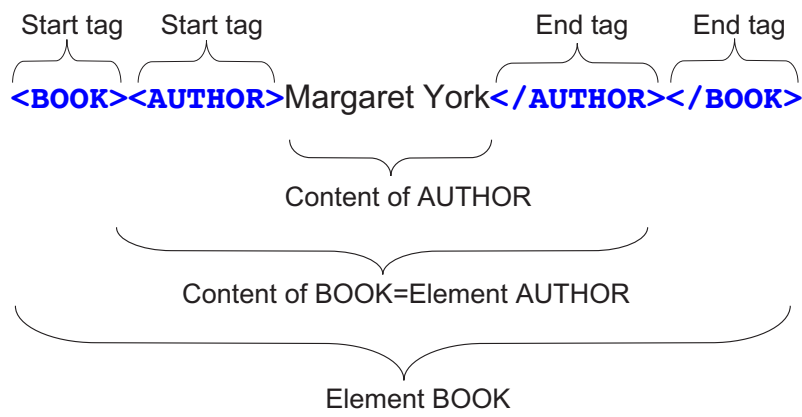
Table 14: An example of an XML text and the corresponding DTD

Explanation:	DTD text:	XML text:
Indicates that this is an XML document.		<code><?xml version="1.0"?></code>
Tells where to find the DTD file ¹ , which specifies the syntax of this XML file. "person.dtd" can be an absolute or a relative URI.		<code><!DOCTYPE person SYSTEM "person.dtd"></code>
Specifies the element type tagged PERSON and that it should always contain, within it, elements tagged NAME , BIRTHYEAR and WAGE .	<code><!ELEMENT PERSON (NAME, BIRTHYEAR, WAGE)></code>	<code><PERSON></code>
Similar to PERSON .	<code><!ELEMENT NAME (FIRST-NAME, SURNAME)></code>	<code><NAME></code>
(#PCDATA) specifies that elements of this element type will contain text outside the tags, in this case "John" and "Smith".	<code><!ELEMENT FIRST-NAME (#PCDATA)></code>	<code><FIRST-NAME>John</code>
	<code><!ELEMENT SURNAME (#PCDATA)></code>	<code></FIRST-NAME></code>
		<code><SURNAME>Smith</code>
		<code></SURNAME></code>
There is no way in DTD to specify that this element type must contain an integer. This is an example where XML/DTD is less strongly typed than ASN.1	<code><!ELEMENT BIRTHYEAR (#PCDATA)></code>	<code></NAME></code>
	<code><!ELEMENT WAGE (#PCDATA)></code>	<code><BIRTHYEAR>1941</code>
		<code></BIRTHYEAR></code>
		<code><WAGE>57000</WAGE></code>
		<code></PERSON></code>

1.17. XML ELEMENT and its contents

¹ The demo files used in this book can be found at <http://dsv.su.se/jpalme/abook/xml/>

ELEMENT and TAG



An XML *element* has a start-tag (example `<PERSON>` in Table 14) and an end-tag (example `</PERSON>`).

The information between the start-tag and the end-tag is the *contents* of the *element*. The contents can either be a piece of text (like “John” in the example in Table 14) or it can be further XML elements (like `<NAME>` inside `<PERSON>` in Table 14) or it can be both text and further XML code.

The DTD declaration of an XML element type (example `<!ELEMENT PERSON (NAME, BIRTHYEAR, WAGE)>`) begins with `<!ELEMENT` followed by the name of the element type, and its contents in parentheses, and ends with `>`.

When the element type allows are further XML elements as contents, their names are listed inside the parentheses, like `(NAME, BIRTHYEAR, WAGE)` in `<!ELEMENT PERSON (NAME, BIRTHYEAR, WAGE)>`. When the element type allows content in plain text, this is specified by the special operator `#PCDATA`.

Many XML applications will regard multiple white space characters as logically identical to a single space character. Thus, many applications will regard the following two XML documents as logically identical:

<code><NAME><FIRST-NAME>John</FIRST-NAME> <SURNAME>Smith</SURNAME> </NAME></PERSON></code>	<code><NAME> <FIRS T-NAME>John </FIRS T-NAME> <SURNAME>Smith </SURNAME> </NAME> </PERSON></code>
--	--

It is, however, up to an XML application to decide whether multiple white space characters are significant or not. And even if they are not logically sig-

nificant, an XML application may let white space influence the layout, in which a document is presented to a reader.

1.1.53. Reserved characters

XML has the same problems as most other textual encodings: Since certain characters are used as delimiters to separate different elements, they cannot occur within plain text. You cannot store:

DTD specification:	Illegal XML data:
<code><!ELEMENT e-mail (#PCDATA)></code>	<pre><?xml version="1.0" ?> <!DOCTYPE e-mail SYSTEM "e-mail.dtd"> <e-mail>"John Smith" <jsmith@foo.bar> </e-mail></pre>

The receiving program will have difficulty interpreting the “<” in “<jsmith@foo.bar>”, it will believe that this is some kind of weird XML tag. To solve this problem, the plain text string must be encoded as “<jsmith@foo.bar>”. The characters which require such special coding are:

Reserved character	Special coding to use instead
<	<
&	&
>	>
'	'
"	"

The inventors av XML apparently have been unhappy with this. Therefore they have invented another, even more convulated way of handling free text data in XML. This alternative method starts the free text with the string “<![CDATA[” and ends it with “]]>”. Example:

DTD specification:	XML data:
<code><!ELEMENT e-mail (#PCDATA)></code>	<pre><?xml version="1.0" ?> <!DOCTYPE e-mail SYSTEM "e-mail.dtd"> <e-mail> <![CDATA["John Smith" <jsmith@foo.bar>]]> </e-mail></pre>

This, of course, means that the string “<![CDATA[” cannot occur in free text in other uses than for this special purpose, and the internal content of the free text cannot use the string “]]>”. In Swedish, we have a proverb about such

things, “No matter how you turn, you will have your back behind you”.

1.1.54. Empty Elements

If an XML element type does not allow any content, this is specified in the DTD with the term `EMPTY`. Example:

DTD specification:	XML data:
<code><!ELEMENT cup EMPTY></code>	<pre><?xml version="1.0" ?> <!DOCTYPE cup SYSTEM "cup.dtd"> <cup></cup></pre>

When there is no content, then a shorter variant of the XML data is to put a “/” at the end of the starting tag, and not specify any end-tag. Thus `<cup></cup>` and `<cup />` are identical. This is allowed even if the element type was not defined as `EMPTY` in the DTD, but happens to have no content in one particular instance. Such a tag, which is both a start-tag and an end-tag at the same time, is called an *empty element tag*.

1.1.55. Any Specification

The ANY specification (example: `<!ELEMENT miscellaneous ANY>`) allows any kind of un-specified XML content. This specification should in most cases be avoided, since it makes it difficult for software to check or interpret the content.

1.1.56. Repeated subelements

Example DTD specification:	XML data:
<pre><!ELEMENT family (husband, wife)> <!ELEMENT husband (#PCDATA)> <!ELEMENT wife (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE family SYSTEM "family.dtd"> <family> <husband>John</husband> <wife>Margaret</wife> </family></pre>

The DTD specification above requires that there is exactly one husband followed by exactly one wife in the XML data. If you want to specify that the family can also, optionally, contain one or more children, you might use the following specification:

Example DTD specification:	XML data:
<pre><!ELEMENT family (husband, wife, child*)> <!ELEMENT husband (#PCDATA)> <!ELEMENT wife (#PCDATA)> <!ELEMENT child (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE family SYSTEM "family.dtd"> <family> <husband>John</husband> <wife>Margaret</wife> <child>Eve</child> <child>Peter</child> </family></pre>

If you want to specify that there must be at least one child, you can specify:

Example DTD specification:	XML data:
<pre><!ELEMENT child-family (husband, wife, child+)> <!ELEMENT husband (#PCDATA)> <!ELEMENT wife (#PCDATA)> <!ELEMENT child (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE child-family SYSTEM "child- family.dtd"> <child-family> <husband>John</husband> <wife>Margaret</wife> <child>Eve</child> <child>Peter</child> </child-family></pre>

Thus, the following operators can be used in a list of subelements:

Code:	Explanation:
a, b	Mandatory a followed by mandatory b.
a b	Either a or b.
a*	0, 1 or more occurrences of a.
a+	1 or more occurrences of a.
a?	0 or one occurrences of a.

Exercise 42

Write a DTD for an XML-variant of the e-mail header in Exercise 41.

From: Nancy Nice <nnice@good.net>

To: Percy Devil <pdevil@hell.net>

Cc: Mary Clever <mclever@intelligence.net>, Rupert Happy <rhappy@fun.net>

1.1.57. Choice subelements

Example DTD specification:	XML data:
<pre><!ELEMENT vehicles (vehicle*)> <!ELEMENT vehicle (bike car)> <!ELEMENT bike (#PCDATA)> <!ELEMENT car (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE vehicles SYSTEM "vehicles.dtd"> <vehicles> <vehicle><bike>Crescent</bike></vehicle> <vehicle><car>Volvo</car></vehicle> </vehicles></pre>

The character “|” specifies either/or as is shown in the example above. It is often combined with additional parenthesis levels, example:

Example DTD specification:	XML data:
<pre><!ELEMENT transport ((bike car)*)> <!ELEMENT bike (#PCDATA)> <!ELEMENT car (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE transport SYSTEM "transport.dtd"> <transport> <bike>Crescent</bike> <car>Volvo</car> </transport></pre>

Exercise 43

Specify DTD and an XML example for a protocol to send either a name (single string), a social-security number (another single string) or both.

1.18. Attributes of XML elements

Like in HTML, an XML element can have attributes on its start-tag. An XML element might for example look like this:

```
<book author ="Margaret Yorke" title="False Pretences"></book>
```

The DTD describing the type for this element might be:

```
<!ELEMENT book EMPTY>
<!ATTLIST book
  author CDATA #REQUIRED
  title CDATA #REQUIRED
>
```

CDATA is the type of the attribute. An XML attribute can have the types listed in Table 16.

An element can have both attributes and content. Example:

DTD specification	XML data
<pre><!ELEMENT book (author, title)> <!ATTLIST book binding (hardback paperback) #REQUIRED color-mode (CMYK RGB GREYS BITMAP) #REQUIRED > <!ELEMENT author (#PCDATA)> <!ELEMENT title (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE book SYSTEM "book.dtd"> <book binding="paperback" colormode="CMYK" > <author>Margaret Yorke</author> <title>False Pretences</title> </book></pre>

For an XML attribute, the DTD can control the use of default values.

Table 15: Default values for XML attributes

DTD term:	Example:	Description:
A single value within quotes at the end of the attribute.	<code><!ATTLIST book binding (hardback paperback) "hardback"></code>	This default value should be assumed if the attribute is not specified in the XML text.
<code>#REQUIRED</code>	<code><!ATTLIST book binding (hardback paperback) #REQUIRED></code>	No default value is allowed, the attribute must always be specified in the XML text.
<code>#IMPLIED</code>	<code><!ATTLIST book binding (hardback paperback) #IMPLIED></code>	No default value, but the attribute is not required. If the attribute is not given, this might mean that it is unknown or not valid.
<code>#FIXED</code>	<code><!ATTLIST book binding (hardback paperback) #FIXED "hardback"></code>	The XML can either contain this attribute or not, but if it is there, it must always have this particular value.

Table 16: Types of XML attributes

Type:	Example:	Description:
CDATA	<code><!ATTLIST book title CDATA #REQUIRED></code>	Any character string.
A list of enumerated values	<code><!ATTLIST book binding (hardback paperback) "hardback"></code>	Restricted to the listed values only.
ID	<code><!ATTLIST book entryno ID #REQUIRED></code>	Gives a name to this particular element. No other element in the XML text can have the same name. Unique names on elements are useful in some cases for programs which manipulate the XML text.
IDREF	<code><!ATTLIST author authorid ID #REQUIRED></code> <code><!ATTLIST book authorid IDREF #REQUIRED></code>	Reference to the unique name, which was given to another element in the XML text. In the example, every element of type author has an ID authorid, and every element of type book has an IDREF referring to the ID of the element for the author of that book.
IDREFS	<code><!ATTLIST author authorid ID #REQUIRED></code> <code><!ATTLIST book authorids IDREFS #REQUIRED></code>	Similar to IDREF, but allows a list of more than one value. Needed in this example, if a book can have more than one author.
ENTITY	DTD text: <code><!ELEMENT LOGO EMPTY></code> <code><!ATTLIST LOGO GIF-FILE ENTITY #REQUIRED></code> <code><!ENTITY DSV-LOGO SYSTEM "dsv-logo.gif"></code> XML text:	This is one way to include binary data in an XML file, by referring to the URI of the binary data. Just like with tags in HTML, the actual binary file is not included, just referenced.

Type:	Example:	Description:
	<code><LOGO GIF-FILE="DSV-LOGO" /></code>	
ENTITIES	DTD text: <pre><!ELEMENT LOGO EMPTY> <![ATTLIST LOGO GIF-FILE ENTITIES #REQUIRED]> <!ENTITY DSV-LOGO SYSTEM "dsv- logo.gif"> <!ENTITY KTH-LOGO SYSTEM "kth- logo.gif"></pre> XML text: <code><LOGO GIF-FILE="DSV-LOGO KTH-LOGO" /></code>	A list of more than one entity.
NMTOKEN	<code><![ATTLIST variable-name #NMTOKEN]></code>	A name, formatted like a variable name in a computer program. Useful when you use XML to generate source program code.
NMTOKENS	<code><![ATTLIST variables #NMTOKENS]></code>	A list of names, similar as for NMTOKEN above.
NOTATION	<code><![ATTLIST SPEECH PLAYER NOTATION (MP3 QUICKTIME) #REQUIRED]></code>	The name of a non-XML encoding.

Exercise 44

Specify DTD and an XML example for a protocol to send a record describing a movie. The record contains a title and a list of people. Each person is identified by the attributes name, and optionally, the attribute role as either actor, photographer, director, author or administrator. As an XML example, use the movie “The Postman Always Rings Twice”, directed by Tay Garnet based on a book by James M. Cain with leading actors Lana Turner and John Garfield.

1.1.58. Use attributes or subelements?

In many cases, you have a choice between use of attributes and subelements.
Example:

DTD specification using attributes:	XML data:
<pre><!ELEMENT book-att EMPTY> <!ATTLIST book-att author #REQUIRED title #REQUIRED ></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE book-att SYSTEM "book- att.dtd"> <book-att author="Margaret Yorke" title="False Pretences"/></pre>
DTD specification using subelements:	XML data:
<pre><!ELEMENT book-sub (author, title)> <!ELEMENT author (#PCDATA)> <!ELEMENT title (#PCDATA)></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE book-sub SYSTEM "book- sub.dtd"> <book-sub> <author>Margaret Yorke</author> <title>False Pretences</title> </book-sub></pre>

There are no fixed rules for when data should be encoded as attributes and as subelements. Both choices above are equally correct. Note however the following differences between attributes and subelements:

Advantage with attributes: There is some rudimentary type control, for example using enumerated attributes, even if the type control is not at all as complete as with ASN.1. Example:

DTD specification:	XML data:
<pre><!ELEMENT book EMPTY> <!ATTLIST book binding (hardback paperback) #REQUIRED color-mode (CMYK RGB GREYS BITMAP) #REQUIRED ></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE book SYSTEM "book.dtd"> <book binding="paperback" colormode="CMYK" /></pre>

Advantage with subelements: Subelements can be repeated multiple times, and can have further inner subelements. Example:

DTD specification:	XML data:
<pre> <!ELEMENT child-family (husband, wife, child+)> <!ELEMENT husband (#PCDATA)> <!ELEMENT wife (#PCDATA)> <!ELEMENT child (#PCDATA)> </pre>	<pre> <?xml version="1.0" ?> <!DOCTYPE child-family SYSTEM "child- family.dtd"> <child-family> <husband>John</husband> <wife>Margaret</wife> <child>Eve</child> <child>Peter</child> </child-family> </pre>

1.19. Formatting XML layout when shown to users (CSS and XLST)

XML can be used as a replacement for HTML. To achieve this, XML is combined with layout information. Special layout languages (CSS and XLST) are available for adding layout information to XML data. CSS or XLST layout specifications are associated with an XML document with a `<?xml-stYLESHEET>` element in the preamble of an XML document. Example:

```

<?xml version="1.0" ?>
<?xml-stYLESHEET type="text/css"
href="mystyles.css"?>

```

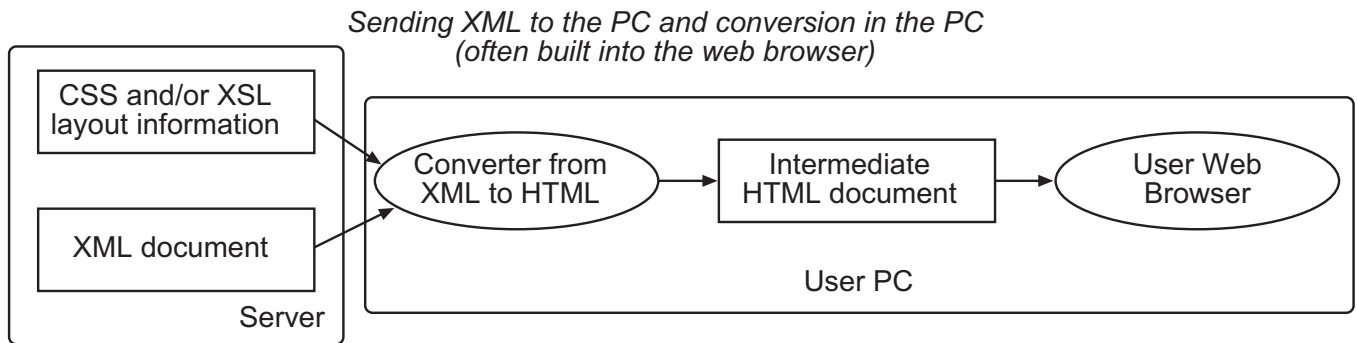
Cascading Style Sheets (see chapter 1.14 on page 79) can be applied to HTML tags or XML elements.

Here is an example of an XML document with a style sheet and how it might be rendered:

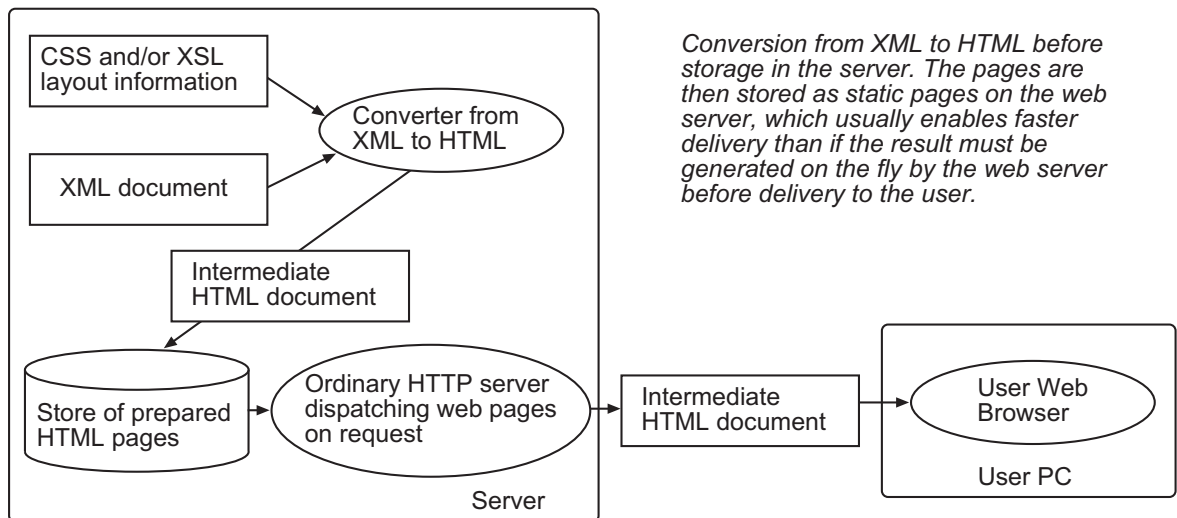
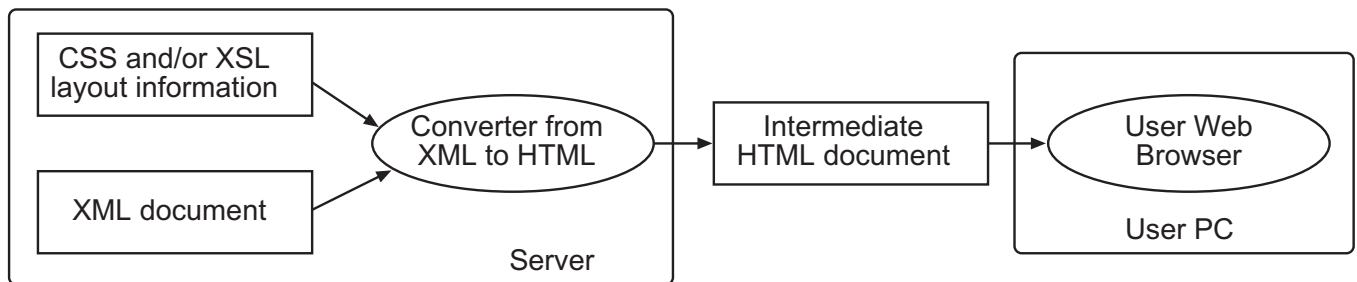
File ticket.css:	
<pre> TITLE { position: absolute; width: 121px; height: 31px; top:25px; left: 86px; font-family: Verdana, sans-serif; font-size: 24pt; font-weight: bold} CLASS { position: absolute; width: 106px; height: 15px; top: 115px; left: 13px; font-family: Verdana, sans-serif; font-size: 12pt; font-weight: bold } FROM { position: absolute; width: 150px; height: 15px; top: 70px; left: 12px; font-family: Verdana, sans-serif; font-size: 14pt; font-weight: bold } TO { position: absolute; width: 150px; height: 15px; top: 70px; left: 166px; font-family: Verdana, sans-serif; font-size: 14pt; font-weight: bold; } DEPART { position: absolute; width: 142px; height: 15px; top: 95px; left: 11px; font-family: Verdana, sans-serif; font-size: 10pt } ARRIVE { position: absolute; width: 128px; height: 15px; top: 95px; left: 167px; font-family: Verdana, sans-serif; font-size: 10pt } CABIN { position: absolute; width: 138px; height: 18px; top: 115px; left: 167px; font-family: Verdana, sans-serif; font-size: 12pt; font-weight: bold } SEAT { position: absolute; width: 138px; height: 18px; top: 115px; left: 247px; font-family: Verdana, sans-serif; font-size: 12pt; font-weight: bold } </pre>	
File ticket.xml:	Visual rendering:
<pre> <?xml version="1.0" ?> <!DOCTYPE TICKET SYSTEM "ticket.dtd"> <?XML:stylesheet type="text/css" href="ticket.css" ?> <TICKET><TITLE>TICKET</TITLE> <CLASS>2 Class</CLASS> <FROM>Oslo</FROM> <TO>Stockholm</TO> <DEPART>Mon 13 Jan 12:13</DEPART> <ARRIVE>Mon 13 Jan 18:45</ARRIVE> <CABIN>Cabin 3</CABIN> <SEAT>Seat 55</SEAT></TICKET> </pre>	<div>TICKET</div> <div> OsloStockholm </div> <div> Mon 13 Jan 12:13Mon 13 Jan 18:45 </div> <div> 2 ClassCabin 3 Seat 5 </div>

Note that with style sheets, you cannot get words like From and To and Class and Cabin and Seat inserted into the visual rendering, if they are not part of the XML values. To solve this problem, you need XSLT. Extensible Style Language Transformations (XSLT) [W3C XSLT 1999] is a more powerful language than CSS. It can be used to describe a series of transformations, which will successively transform an XML document to an HTML document.

Transformation from XML to HTML encoding can be done either in the server or in the client as shown in Figure 11.

Figure 11: Conversion from XML to HTML

Conversion from XML to HTML in the server, before transmission to the PC



HTML does not support alternative versions of the same information for dif-

ferent readers, but with XML, you can use the same XML source data, combined with different CSS and/or XLST layout specifications, in order to produce your data in different format for different readers.

1.20. XML special problems and methods

1.1.59. Putting binary data into XML encodings

All textual encodings have a common problem in that they will not allow binary data, like, for example, a picture in GIF format. There are three ways of handling this problem in XML:

- ① Encode the binary data, using, for example, the BASE64 method (see page 17).
- ② Put the binary data in a separate file, like GIF pictures in HTML:
``
- ③ Use method ②, but combine it with the MHTML method (see page 666) to concatenate all the files into a single compound file.

1.1.60. Reusing DTD information

You may have a need to define some general DTD element types, and then use them in several other DTD element types. This can be done by an include functionality. The name of the include functionality in XML is ENTITY. Example of use of ENTITIES in DTD files::

General DTD specifications: e name person.dtd)	XML data:
ELEMENT person (name, birthyear)> ELEMENT name (#PCDATA)> ELEMENT birthyear (#PCDATA)> ATTLIST person gender (male female) #REQUIRED status (unmarried married divorced widow widower) #REQUIRED	<?xml version="1.0" ?> <!DOCTYPE family SYSTEM "family.dtd"> <family> <person gender="male" status="married"> <name>John Smith</name> <birthyear>1958 </birthyear> </person> <person gender="female" status="married"> <name>Eliza Tennyson</name> <birthyear>1959 </birthyear> </person> </family>
TD using this specification: ile name family.dtd)	
ELEMENT family (person+)> ENTITY % person SYSTEM "person.dtd"> erson;	

After defining **person** in the file **person.dtd** above, this element type can

then be used in a number of different new DTDs by just referencing them as shown in the file family.dtd above.

1.1.61. Entities

Entities are ways of referencing data defined elsewhere. They can be external, as in the example in section 1.1.60, or they can be internal references within a file. Example:

```
<!ENTITY KTH "Kungliga Tekniska Högskolan">
<DESCRIPTION>&KTH; is a technical university.</DESCRIPTION>
```

is identical to

```
<DESCRIPTION>Kungliga Tekniska Högskolan is a technical
university.</DESCRIPTION>
```

In fact, the special codes for certain characters defined in section 1.1.53, like " are built-in entities.

1.1.62. Name Spaces

When you want to combine different DTD sets, perhaps developed by different people at different times, there is a risk that several of the sets will use the same element type name for different purposes.

Example: Suppose you have two DTDs, one about war, one about geography. Both contain elements with the same tag <desert>. In the war DTD, this element describes the act of deserting from an army. In the geography DTD, this element describes a kind of arid region. Suppose now that for a particular application, you want to combine element types from both these DTDs.

Part of the war DTD: (file name war.dtd)	XML data:
<!ELEMENT war:desert (deserter*)> <!ELEMENT war:deserter (#PCDATA)>	<?xml version="1.0" ?> <!DOCTYPE desertations-in-deserts SYSTEM "desertations-in-deserts.dtd"> <desertations-in-deserts xmlns:war="http://dsv.su.se/jpalme/a- book/xml/war.dtd" xmlns:geography="http://dsv.su.se/jpalme/a- book/xml/geography.dtd"> <war:desert> <deserter>John Smith</deserter> </war:desert> <geography:desert> Sahara</geography:desert> </desertations-in-deserts>
Part of the geography DTD: (file name geography.dtd)	
<!ELEMENT geography:desert (#PCDATA)>	
Use of these two DTDs in a new DTD: (file name desertaions-in-deserts.dtd)	
<!ENTITY % war:desert SYSTEM "war.dtd"> %war; <!ENTITY % geography:desert SYSTEM "geography.dtd"> %geography; <!ELEMENT desertations-in-deserts (war:desert, geography:desert)> <!ATTLIST desertaions-in-deserts xmlns:war CDATA #IMPLIED xmlns:geography CDATA #IMPLIED>	

The `xmlns:war="http://dsv.su.se/jpalme/a-book/xml/war.dtd"` and `xmlns:geography="http://dsv.su.se/jpalme/a-book/xml/geography.dtd"` attributes need not refer to any real file, but should contain a unique URL for this name space.

The character “.” is not permitted in XML identifiers except to separate the name space name and the following identifier from that name space.

1.1.63. XLinks and XPointers

It is possible to put links into an XML document in the same way as in an HTML document, for example:

```
<a href="http://dsv.su.se/jpalme/a-book/">Web pages for this
book</a>
```

If you do this in XML, you should define the `<a>` element type and its attribute `href` in the DTD, just like you define other XML element types. Additionally, XML has special constructs XLinks and XPointers. They are more powerful than the `<a>` tag in HTML: An element defined for other purposes can at the same time become a link, you have better ways of linking to parts of a target document than in HTML, and with Xlinks (specified in the Extensible Linking Language, XLL) you can create bi-directional links, links which are

fully specified in both linked documents.

1.1.64. Processing instructions

Elements like

```
<?xml version="1.0" ?>  
<?xml-stylesheet type="text/css" href="mystyles.css"?>
```

are called *processing instructions*, because they instruct the recipient how to process the XML document.

The default character set in XML is UTF-8. If you are using some other character set, such as ISO 8859-1, you have to indicate this in the first processing instruction in the XML file. For example, you can specify

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

to indicate that the character set used in the XML document is ISO 8859-1.

1.1.65. Standalone declarations

When you look at XML files, you may find that the first line is not `<?xml version="1.0" ?>` but instead `<?xml version="1.0" standalone="yes" ?>` or `<?xml version="1.0" standalone="no" ?>`. This is supposed to indicate whether some information in some other file (like a DTD declaration) is needed to understand the XML content. You need not specify `standalone="no"` in every XML file which is based on a DTD. `standalone="no"` is required only if information in the DTD (or some other external file, such as one reference in an ENTITY declaration) is required in order to correctly interpret the XML. For example, if the DTD specifies defaults or fixed values for attributes, then this information is necessary to correctly interpret the XML code, and then this declaration should be `standalone="no"`. The whole `standalone` declaration is optional, and many XML applications do not use it at all.

1.1.66. XML validation

When you are developing specifications using DTD and XML, it is essential to be able to check your specifications for correctness. There is software available to do this. I have been using the validator on the net at <http://www.stg.brown.edu/service/xmlvalid/> to validate the examples given in

this book.

1.1.67. XHTML

XHTML is a variant of HTML which is at the same time also correct XML.

The main differences from ordinary HTML are:

- All tags must be lower case, e.g. `<a href>` and not ``
- All tags must be ended, e.g. `<p>First paragraph
second line.</p>`
- No syntax errors allowed, e.g. not `<p>Strong text</p>`

1.21. A comparison of ABNF, ASN.1-BER/PER and DTD-XML

Table 17 shows an example of the same information as encoded with ABNF, ASN.1-BER and DTD-XML.

Table 18 compares some properties of the three encoding methods.

Table 17: The same information with ABNF, ASN.1 and XML

BNF specification:	ASN.1 specification:	DTD specification:
<pre> family = "Family" CRLF *(Person) "End of Family" person = "Person" CRLF " Name: " 1*A CRLF " Birthyear: " 4D CRLF " Gender: " ("Male"/"Female") CRLF " Status: " ("unmarried"/ "married"/ "divorced"/ "widow"/ "widower") </pre>	<pre> = SEQUENCE OF Person := SEQUENCE { name VisibleString, birthyear INTEGER, gender Gender, status Status } := ENUMERATED { male(0), female(1) } = ENUMERATED { unmarried(0), married(1), divorced(2), widow(3), widower(4) } </pre>	<pre> <!ELEMENT family (person+)> <!ELEMENT person (name, birthyear)> <!ELEMENT name (#PCDATA)> <!ELEMENT birthyear (#PCDATA)> <!ATTLIST person gender (male female) #REQUIRED status (unmarried married divorced widow widower) #REQUIRED > </pre>
Example of textual encoding:	Example of BER encoding:	Example of XML encoding:
<pre> family person Name: John Smith Birthyear: 1958 Gender: Male Status: Married person Name: Eliza Tennyson Birthyear: 1959 Gender: Female Status: Married End of Family </pre>	<p>(Each box represents one octet. Two-character codes are hexadecimal numbers, one character codes are characters)</p> <pre> 30 34 30 16 1A 0A J o h n S m i t h 02 02 07 A6 0A 01 00 0A 01 01 30 1A 1A 0E E l i z a T e n n y s o n 02 02 07 A7 0A 01 01 0A 01 01 </pre>	<pre> <?xml version="1.0" ?> <!DOCTYPE family SYSTEM "family.dtd"> <family> <person gender="male" status="married"> <name>John Smith</name> <birthyear>1958 </birthyear> </person> <person gender="female" status="married"> <name>Eliza Tennyson</name> <birthyear>1959 </birthyear> </person> </family> </pre>
3 octets (excluding newlines)	54 octets	258 octets (excluding newlines and leading spaces)
% efficiency ²	57 % efficiency ¹	12 % efficiency ¹

² As compared to PER.

The PER (unaligned variant) encoding of the same ASN.1 and the same data would be the following 31 octets:

00000010	(number of persons in family)	000011 10	(14 characters)
00001010	(10 characters)	100010 1	E
1001010	J	1101100	l
1 101111	o	1101001	i
11 01000	h	1 111010	z
110 1110	n	11 00001	a
0100 000		010 0000	
10100 11	S	1010 100	T
110110 1	m	11001 01	e
1101001	i	110111 0	n
1110100	t	1101110	n
1 101000	h	1111001	y
00 000010	(2 octets)	1 110011	s
00 00011110	100110 (1958)	11 01111	o
0	(male)	110 1110	n
0 01	(married)	0000 0010	(2 bytes)
		0000 01111010 0111	(1959)
		1	(female)
		001	(married)

Note 1: Many thanks to Jean-Paul Lemaire, who helped me with the BER and PER encodings.

Note 2: The success of many Internet application layer protocols with very inefficient textual encodings apparently indicates that the efficiency is not a very important factor in determining the success of an application layer protocol.

Note 3: Compression programs (like zip, gz, etc.) can compress almost any textual encoding to near-maximal efficiency. This, however, only works for large files. Small files are not compressed very efficiently with compression programs. To test this, I tried to compress the XML encoding above using the Zip encoding. It actually became 14 % larger after compression. I also tested a file where I repeated the XML encoding above 11 times, with the same XML elements and tags, but different content. This larger file, after compression with Zip encoding, became 53 % as efficient as the PER encoding, or about as high efficiency as with the BER encoding.

Table 18: Comparison of ABNF, ASN.1-BER and DTD-XML

	ABNF	ASN.1	DTD+XML
Level	Low level, can specify almost any textual encoding.	High level, strongly typed, you define the exact data types to use .	High level, but not as good type facilities as ASN.1.
Encoded format	Text.	With for example Basic Encoding Rules (BER), a binary format, or Packed Encoding Rules (PER), a very efficient binary format, or other encoding rules.	Text.
Readability of meta-language	OK.	Good.	Acceptable.
Readability of encoded data	Very good.	Very bad unless special reader program is used.	Very good.
Efficiency of data packing, as compared to maximum efficiency.	Usually not so good.	About 50 % with BER, almost 100 % with PER.	Not so good.
Binary data	Must be encoded, for example using BASE64, which however adds 33 % redundancy.	Can easily be included as is.	Must be encoded, for example using BASE64, or sent as separate files.
Layout facilities	None, but the high freedom allows specification of rather readable formats.	None.	Can be combined with layout languages to produce highly readable output (comparable to HTML-based web documents).

Below are quoted two messages from an e-mail discussion about the pros and cons of ASN.1:

```

From: Marshall T. Rose <mrose@dbc.mtview.ca.us>
Date: 12 jul 1995 05:12
... ..

Combining ASN.1 and high-performance is oxymoronic.

ASN.1 is probably the greatest failure of the OSI effort, it led
hundreds of engineers, including myself, to devise data structures that
were far too complicated for their own good.

```

(Oxymoron = Self-contradiction)

(Marshall T. Rose is a well-known previous OSI expert who has turned into one of the most vocal OSI enemies. OSI is a set of standards which in the 1980s were competing with the Internet standards. Today, most OSI standards

have failed, a few of them have been accepted in the Internet, for example X.500 as used in the LDAP standard.)

```
From: Colin Robbins <c.robbsins@nexor.co.uk>
Date 13 Jul 1995 16:58

Let me see if I have understood this debate.
X.400 is a brontosarus, because it uses ASN.1.
SMTP is a monkey because it does not.

Where does that leave the SNMPv2 Protocol, desgined by the Internet
community, co-author one Marshall T. Rose. It uses ASN.1. I thought
leopards didn't change their spots!

There are plenty or reasons to knock X.400, but the use of ASN.1 is not
one of them. Sure it has its faults, but BOTH the Internet and OSI
communities are using it.
```

1.1.68. Comparion RFC822-style headings versus XML and ASN.1

Many standards have used the so-called RFC822-style header format, which is usually specified using ABNF. Below is an example of how the same information can be encoded in this format as compared to XML:

RFC822 example:

```
From: Father Christmas <fchristmas@northpole.arctic>
```

XML encoding of the same information:

```
<from>
  <user-friendly-name>Father Christmas</user-friendly-name>
  <e-mail-address>
    <localpart>fchristmas</localpart>
    <domainpart>
      <domainelement>northpole</domainelement>
      <domainelement>arctic</domainelement>
    </domainpart>
  </from>
```

Besides noting that XML in this example requires about five times as many characters, another difference is that XML uses the same characters for framing in all levels, while the RFC822 example uses three different notations in five levels:

Level 1: Newline between headers.

Level 2: “:” between header name and header value.

Level 3: “<” and “>” to separate localpart from e-mail address.

Level 4: “@” to separate localpart from domainlist.

Level 5: “.” to separate the domain component in the list of domain elements.

It is of course an advantage with XML that you do not have to invent new

framing characters at each level, and also maybe new rules about forbidden characters or characters that need to be quoted at each level.

1.22. Other Encoding Languages

ABNF, ASN.1 and XML are not the only encoding languages. Some other existing languages are Corba and XDR (External Data Representation, [RFC 1832]). Both XDR and Corba represent data in a format which is more similar to the way it is stored internally in data handled by common programming languages like C and Pascal. XDR is somewhat similar to ASN.1, but tags and length encoding are used more sparsely. An application using XDR may then have to include type and length information into the defined data structures, while with ASN.1 tag and length are included in the encoding rules. On the other hand, XDR avoids some unnecessary tags, and will thus probably give somewhat more efficient encodings than BER. XDR is used in the ONC RPC (Remote Procedure Call) and the NFS* (Network File System).

Corba is integrated with a programming API for transmission of data between applications running on different hosts. And some protocols, for example the Domain Naming System (DNS) do not use any encoding language at all, their encodings are specified in the form of English-language text and tables.

6. *References*

Objectives

Books and websites for further reading

Keywords

Book

Web site

Reference	Source	Comment
Larmouth 1999:	ASN.1 Complete, by John Larmouth, Morgan Kaufmann Publishers 1999.	An ASN.1 tutorial.
Kaliski 1993:	A Layman's Guide to a Subset of ASN.1, BER, and DER, by Burton S. Kaliski Jr. 1993, http://www.rsa.com/rsalabs/pkcs/ .	A 36-page introduction to the of BER.
RFC 822:	RFC822 Standard for the format of ARPA Internet text messages. D. Crocker. Aug-13-1982. (Status: STANDARD)	This early e-mail standard specifies the commonly used version of ABNF.
RFC 2234:	RFC2234 Augmented BNF for Syntax Specifications: ABNF. D. Crocker, Ed., P. Overell. November 1997.	New version of ABNF used in standards.
RFC 2279:	RFC2279 UTF-8, a transformation format of ISO 10646. F. Yergeau. January 1998. (Obsoletes RFC2044)	Specification of the UTF-8 encoding for the ISO 10646=Unicode character set.
RFC 1345:	RFC1345 Character Mnemonics and Character Sets. K. Simonsen. June 1992.	A comprehensive listing of character sets and the characters within them.
RFC 1832:	RFC 1832 XDR: External Data Representation Standard.	Specification of the XDR encoding.
RFC 2045:	2045 Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. N. Freed & N. Borenstein. November 1996.	Contains specification of the Quoted-Printable and BASE64 encoding.
Harold 1999:	XML Bible, by Elliott Rusty Harold, IDG Books, Foster City, CA, U.S.A., 1999.	A very thorough and readable guide to all aspects of XML. Some chapters updated after publication, and corrections loaded from the web.
W3C XSLT 1999:	XSL Transformations (XSLT), W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt	A language for transforming XML documents to HTML documents for presentation when shown to users.
W3C CSS1 1996:	Cascading Style Sheets, level 1, W3C Recommendation 17 Dec 1996, http://www.w3.org/TR/REC-CSS1	The standard for level 1 of cascading style sheets.
W3C CSS2 1998:	Cascading Style Sheets, level 2, CSS2 Specification, W3C Recommendation 12-May-1998, http://www.w3.org/TR/REC-CSS2/	The standard for level 2 of cascading style sheets.
W3C HTML401 1999:	HTML 4.01 Specification, W3C Recommendation 24 December 1999, http://www.w3.org/TR/html401/	The standard describing the HTML markup language.
Bourett 2000:	XML Namespaces FAQ, by Ronald Bourett, February 2000, http://www.informatik.tu-darmstadt.de/DSV1/staff/bourett/xml/NamespacesFAQ.htm	Tries to explain the complex issues with namespaces in XML.

7. Acknowledgements

Objectives

People who helped getting this book better.

Keywords

Expert

Mailing list

Many people have helped me in the writing of this book. I have sent draft chapters of various chapters to mailing lists with experts on the varioups protocols and methods and got very useful feedback. Here are some of the people who have helped me: Andrew Waugh, Olivier Dubuisson, Jean-Paul Lemaire, Richard Lander, Lars Marius Garshol.

8. Solutions to exercises

Objectives

Solving the exercises.

Keywords

Solution

Facit

Exercise 1 solution

```
path = [ "/" ] *( directory-name "/" ) file-name
```

Exercise 2 solution

```
LWSP = 1*( SP / HT / ( CR LF ( SP / HT ) )
```

Exercise 3 solution

```
weather-header = "Weather:" LWSP weathertype 0*2( parameter )
weathertype    = "Sunny" / "Cloudy" / "Raining" / "Snowing"
parameter      = ( ";" ( LWSP "temperature" / "humidity" ) ) "=" 1*DIGIT
```

Exercise 4 solution

```
ALPHA = "A" / "B" / "C" / "D" / "E" / "F" / "G" / "H" / "I" / "J" / "K"
/ "L" / "M" / "N" / "O" / "P" / "Q" / "R" / "S" / "T" / "U" / "V" / "X"
/ "Y" / "Z"
DIGIT = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
Identifier = ALPHA *5( ALPHA / DIGIT )
```

Exercise 5 solution**1.1.1.4. Solution alternative 1 to Exercise 1**

```
ScaleReading ::= [APPLICATION 0] SEQUENCE { weight Weight,
itemno Itemno
}
```

```
Weight ::= [APPLICATION 1] REAL -- in grams
```

```
Itemno ::= [APPLICATION 2] INTEGER
```

1.1.1.5. Solution alternative 2 to Exercise 1

```
ScaleReading ::= [APPLICATION 0] SEQUENCE {
weight REAL, -- in grams
itemno INTEGER
}
```

Warning: The use of the **APPLICATION** tag is not recommended in the 1994 version of ASN.1. So with the 1994 style of ASN.1, use:

1.1.1.6. Solution alternative 3 to Exercise 1

```
ScaleReading ::= SEQUENCE { weight Weight,
                             itemno Itemno
                           }
```

```
Weight ::= REAL -- in grams
```

```
Itemno ::= INTEGER
```

Exercise 6 solution

```
Box ::= SEQUENCE{
    height Measurement,
    width Measurement,
    length Measurement
}
```

```
Measurement ::= SEQUENCE {
    yards INTEGER,
    feet INTEGER,
    inches REAL
}
```

Exercise 7 solution

```
Measurement ::= SEQUENCE {
    yards INTEGER,
    feet INTEGER (0 .. 2),
    inches INTEGER (0 .. 1199)
}
```

Exercise 8 solution

```
Voter ::= SEQUENCE {
    vote Vote,
    age Age,
    gender Gender
}
```

```
Age ::= INTEGER ( 18 .. MAX )
```

```
Vote ::= INTEGER {
    labour(0),
    liberals (1),
    conservatives (2),
    other (3)
} (0 .. 3)
```

Gender ::= BOOLEAN

Alternative definiton of “Vote”:

```
Vote ::= ENUMERATED {
    labour(0),
    liberals (1),
    conservatives (2),
    other (3)
}
```

Exercise 9 solution

```
HomeTownVoter ::= SEQUENCE {
    hometownvote Sthvote,
    age Age,
    gender Gender
}
```

```
HomeTownVoter ::= SEQUENCE {
    hometownvote Sthvote,
    age Age,
    gender Gender
}
```

Note, some people claim that it would be allowed to write:

```
} ( INCLUDES Vote | 4 | 5 )
```

as the last line above, but other people claim this is not allowed.

Exercise 10 solution

1.1.1.7. Alternative 1

```
Secrecy ::= INTEGER { open(1), secret(2), topsecret(3) } (1..3)
```

1.1.1.8. Alternative 2 (better)

```
Secrecy ::= ENUMERATED { open(1), secret(2), topsecret(3) }
```

Exercise 11 solution

1.1.1.9. Alternative 1

```
StabSecrecy ::= INTEGER { open(1), secret(2), topsecret(3), extratopsecret(4) }
(INCLUDES Secrecy | 4 )
```

1.1.1.10. Alternative 2 (better)

(better according to ASN.1 experts)

```
StabSecrecy ::= ENUMERATED { open(1), secret(2), topsecret(3), extratopsecret(4) }
```

Exercise 12 Solution**Alternative 1**

```
Pattern ::= SEQUENCE {
    height INTEGER,
    width INTEGER,
    pattern BIT STRING -- row by row
}
```

Alternative 2

```
Row ::= BIT STRING

Pattern ::= SEQUENCE {
    height INTEGER,
    width INTEGER,
    pattern SEQUENCE OF Row
}
```

Exercise 13 Solution

```
InStore ::= BIT STRING {
    a3 (0),
    a4 (1),
    a5 (2),
    a6 (3)
} (SIZE(4))
```

Exercise 14

What is the difference between these two types, and what does monday mean for each of them?

```
DayOfTheWeek ::= ENUMERATED { monday(0), tuesday(1), wednesday(2),
    thursday(3), friday(4), saturday(5), sunday(6) }
```

```
DaysOpen ::= BIT STRING { monday(0), tuesday(1), wednesday(2),
    thursday(3), friday(4), saturday(5), sunday(6) } (SIZE(7))
```

Solution

DayOfTheWeek can have as value one of the seven days, and the value **monday**

designates that single day.

DaysOpen can have as value a bit string, which specifies for each day, whether a shop is open or not on that day. **monday** is the name of the first bit, which is true if the shop is open on Mondays, and false if it is closed on Mondays.

Exercise 15 Solution

1.1.1.13. Solution taken from X.411, 1998 version

ub-organization-name-length INTEGER ::= 64

OrganizationName ::= PrintableString
(SIZE (1 .. ub-organization-name-length))

1.1.1.14. Solution, using new constructs from the 1994 version of ASN.1:

Name {INTEGER : name-length} ::= PrintableString (Size(1..name-length))

OrganizationDirectorName ::= Name {64}

Exercise 16 solution

1.1.1.15. Solution 1

PersonRecord ::= SET {

pnumber Pnumber,
name Nametype OPTIONAL,
income Incometype OPTIONAL
}

Pnumber1 ::= [APPLICATION 1] PrintableString

(FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | " "))

Pnumber ::= Pnumber1 (SIZE (13))

Nametype ::= GeneralString (SIZE (1 .. 40))

Incometype ::= INTEGER (0 .. MAX)

1.1.1.16. Solution 2

```

PersonRecord ::= SET {
    pnumber Pnumber,
    name Nametype OPTIONAL,
    income Incometype OPTIONAL
}

Pnumber1 ::= PrintableString (FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | " "))

Pnumber ::= Pnumber1 (SIZE (13))

Nametype ::= GeneralString (SIZE (1 .. 40))

Incometype ::= INTEGER (0 .. MAX)

```

1.1.1.17. Solution 3

```

Pnumber1 ::= PrintableString (FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | " "))

PersonRecord ::= [APPLICATION 0] SET {
    pnumber Pnumber1 (SIZE (13))
    name GeneralString (SIZE (1 .. 40)) OPTIONAL,
    income INTEGER (0 .. MAX) OPTIONAL
}

```

Note: With the 1994 version of ASN.1, you might also write:

```

Pnumber1 ::= PrintableString (FROM ("0" .. "9" | "-" | " "))

```

Exercise 17 Solution**1.1.1.18. Solution 1**

```

PersonRecord ::= SET {
    pnumber Pnumber,
    gname GNametype OPTIONAL,
    sname SNametype OPTIONAL,
    Income Incometype OPTIONAL
}

Pnumber1 ::= PrintableString
(FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | " "))

Pnumber ::= Pnumber1 (SIZE (13))

```


GNametype ::= [APPLICATION 0] GeneralString (SIZE (1 .. 40))

SNametype ::= GeneralString (SIZE (1 .. 40))

Incometype ::= INTEGER (0 .. MAX)

1.1.1.19. Solution 2

**PersonRecord ::= SET {
 pnumber Pnumber,
 name Nаметype OPTIONAL,
 income Incometype OPTIONAL
 }**

**Pnumber1 ::= PrintableString
 (FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | " "))**

Pnumber ::= Pnumber1 (SIZE (13))

**Nametype ::= SEQUENCE {
 sName GeneralString (SIZE (1 .. 40)),
 gName GeneralString (SIZE(1 .. 40))
 }**

Incometype ::= [APPLICATION 3] INTEGER (0 .. MAX)

Question: Why is the solution below not correct?

**PersonRecord ::= [APPLICATION 0] SET {
 pnumber Pnumber,
 gname Nаметype OPTIONAL,
 sname Nаметype OPTIONAL,
 income Incometype OPTIONAL
 }**

**Pnumber1 ::= PrintableString
 (FROM ("0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | " "))**

Pnumber ::= Pnumber1 (SIZE (13))

Nametype ::= GeneralString (SIZE (1 .. 40))

Incometype ::= INTEGER (0 .. MAX)

Answer: The receiving computers cannot know if a name with only one component is only a **gname** or only a **sname**.

Exercise 21 solution

```

FullName ::= SEQUENCE {
  givenName [0] IA5String OPTIONAL,
  initials [1] IA5String OPTIONAL,
  surname [2] IA5String
}

```

Question: Can the tags in the solution above be removed?

Yes, you can always remove one of the tags, since it will then get the **UNIVERSAL** tag of **IA5String**, which is different than the other user-defined tags.

If you have **AUTOMATIC** tagging set, you can remove all the tags. Otherwise, two of them must be kept, since the elements must have different tags to separate them. If the first two elements had not been **OPTIONAL**, then the tags would not have been required, since then the elements could be separated by their order in the **SEQUENCE**.

Exercise 22 solution

```

BasicFamily ::= SEQUENCE {
  husband [0] IA5String OPTIONAL,
  wife [1] IA5String OPTIONAL,
  children [2] SEQUENCE OF IA5String OPTIONAL
}

```

With automatic tagging, the tags above can be removed.

Question: Is **SEQUENCE OF** or **SET OF** best in this exercise? Answer: If you want to indicate the order of birth the children, **SEQUENCE OF** is better.

Exercise 23 solution

```

ChildLessFamily ::= BasicFamily
( WITH COMPONENTS {
  ... , children ABSENT
}
)

```

Exercise 24

Given the ASN.1-type:

```
XYCoordinate ::= SEQUENCE {
  x REAL,
  y REAL
}
```

Define a subtype which only allows values in the positive quadrant (where both x and y are ≥ 0).

solution

```
PositiveCoordinate ::= XYCoordinate
  ( WITH COMPONENTS {
    x (0 .. MAX)
    y (0 .. MAX)
  }
)
```

Exercise 25

Given the ASN.1 type:

```
ET {
  author Name OPTIONAL,
  textbody IA5String }
```

Define a subtype to this, called **AnonymousMessage**, in which no **author** is specified.

solution**1.1.1.20. Solution 1**

```
AnonymousMessage ::= Message
  ( WITH COMPONENTS {... , author ABSENT }
)
```

1.1.1.21. Solution 2

```

AnonymousMessage ::= Message
( WITH COMPONENTS {
  author ABSENT,
  textbody }

```

Exercise 26 solution

```

Vessel ::= CHOICE {
  aircraft Aircraft,
  ship Ship,
  train Train,
  motorcar MotorCar

```

Exercise 27 solution**1.1.1.22. Solution 1**

```

GeneralNameListA ::= gs < NameListA

```

```

GeneralNameListB ::= NamelistB
( WITH COMPONENT
(WITH COMPONENTS {gs} )
)

```

1.1.1.23. Solution 2

```

GeneralNameListA ::= NameListA ( WITH COMPONENTS {gs} )

```

```

GeneralNameListB ::= NamelistB
( WITH COMPONENT
(WITH COMPONENTS {gs} ) )

```

Exercise 28 solution

```

Vote ::= SEQUENCE {
    voterName IA5String,
    votevalue CHOICE {
        chosenAlternative AlternativeNumber,
        setvalue SET OF SEQUENCE {
            alternative AlternativeNumber,
            score INTEGER ( 0 .. 10 )
        }
    }
}

```

Exercise 29 solution

```

vote          = voter-name "/" (One-choice / Choice-list )
voter-name    = "" name ""
name          = 1*namechar
namechar      = <any printable ASCII character except ">
One-choice    = "Single:" 1*DIGIT
Choice-list   = "Multiple:" 1#(alternative LWSP score)
alternative   = 1*DIGIT
Score         = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" / "10"

```

Exercise 30 solution

WeatherReporting {2 6 6 247 1} DEFINITIONS IMPLICIT TAGS ::=

BEGIN

WeatherReport ::= SEQUENCE {

height [0] REAL,

weather [1] Wrecord

}

Wrecord ::= [APPLICATION 3] EXPLICIT SEQUENCE {

temp Temperature,

moist Moisture

wspeed [0] EXPLICIT Windspeed OPTIONAL

}

Temperature ::= [APPLICATION 0] REAL

Moisture ::= [APPLICATION 1] EXPLICIT REAL

Windspeed ::= [APPLICATION 2] EXPLICIT REAL

END - - of module

WeaterhReporting

Exercise 31 solution

Record ::= SEQUENCE {	}	Both tags can be removed
GivenName [0] PrintableString		
SurName [1] PrintableString }		
Record ::= SET {	}	One of the tags can be removed, since if you remove one of them, that element will have the UNIVERSAL tag for PrintableString, which is different from the context-dependent tag [1].
GivenName [0] PrintableString		
SurName [1] PrintableString }		
Record ::= SEQUENCE {	}	
GivenName [0] PrintableString OPTIONAL		
SurName [1] PrintableString OPTIONAL }		

Exercise 32 solution

The tags which can be removed are those shown in italics below.

```
Colour ::= [APPLICATION 0] CHOICE {
  rgb [1] RGB-Colour,
  cmg [2] CMG-Colour,
  freq [3] Frequency
}
```

```
RGB-Colour ::= [APPLICATION 1] SEQUENCE {
  red [0] REAL,
  green [1] REAL OPTIONAL,
  blue [2] REAL
}
```

```
CMG-Colour ::= SET {
  cyan [1] REAL,
  magenta [2] REAL,
  green [3] REAL
}
```

```

Frequency ::= SET {
  fullness [0] REAL,
  freq [1] REAL
}

```

Exercise 33 solution

```

ListResult ::= OPTIONALLY-SIGNED
CHOICE {
  listInfo SET {
    DistinguishedName OPTIONAL,
    subordinates [1] SET OF SEQUENCE {
      RelativeDistinguishedName,
      aliasEntry [0] BOOLEAN DEFAULT FALSE
      fromEntry [1] BOOLEAN DEFAULT TRUE},
    partialOutcomeQualifier [2]
    PartialOutcomeQualifier OPTIONAL
    COMPONENTS OF CommonResults },
  uncorrelatedListInfo [0] SET OF Listresult }

```

Exercise 34 solution

Yes, two comma characters are missing:

```

ListResult ::= OPTIONALLY-SIGNED
CHOICE {
  listInfo SET {
    DistinguishedName OPTIONAL,
    subordinates [1] SET OF SEQUENCE {
      RelativeDistinguishedName,
      aliasEntry [0] BOOLEAN DEFAULT FALSE, -- ⇐ This comma is missing
      fromEntry [1] BOOLEAN DEFAULT TRUE},
    partialOutcomeQualifier [2]
    PartialOutcomeQualifier OPTIONAL, -- ⇐ This comma is missing
    COMPONENTS OF CommonResults },
  uncorrelatedListInfo [0] SET OF Listresult }

```

Exercise 35 solution

COMPONENTS OF is not a data type, and can thus not have any identifier. It copies a series of separately defined type elements, and is useful if you have a series of standard elements, like **CommonResults**, which is to be used in many places.

Exercise 36 solution

In a **SET** all the elements must have different type. It is then necessary to give a context tag only on all but one of the elements.

Exercise 37 solution

```
CarDriving { 1 2 4711 18 } DEFINITIONS EXPLICIT TAGS ::=
```

```
BEGIN
```

```
IMPORTS MainOperation FROM Driving {1 2 4711 17};
```

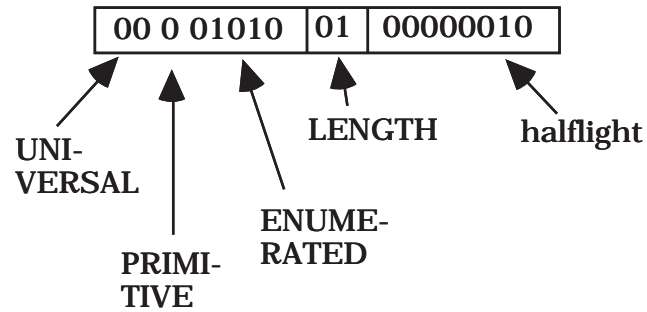
```
FullOperation ::= SEQUENCE {
    COMPONENTS OF MainOperation,
    blink SEQUENCE {
        on BOOLEAN,
        left BOOLEAN },
    light ENUMERATED {
        dark(0),
        parkingLight (1),
        dimmedLight (2),
        fullBeam (3)
    } }
} }
```

```
END -- of module CarDriving
```

Note: Since there was no **EXPORTS** statement in Driving, all objects in it are exported.

Exercise 38 solution

APPLI- CATION	CON- STRUC- TED	Tag nr.	Length	UNI- VER- SAL	PRIMI- TIVE	IA5STRING	Length	Character codes			
01	1	00001	6	00	0	10110	4	M	a	r	y
61			06	16			04	M	a	r	y

Exercise 39 solution**Exercise 40 solution**

element	encoding	Octet
beverage	(context explicit tag) 101 00001 (ENUMERATED) 000 01010	2
tea	(length) 1 (value) 00000001	2
jam	(context explicit tag) 101 00010 (ENUMERATED) 000 01010	2
orange	(length) 1 (value) 00000000	2
continentalpart	(SEQUENCE) 001 10000 (length) 8 beverage tea jam orange	10
eggform fried	(ENUMERATED) 000 01010 (length) 1 (value) 00000101	3
english	(SEQUENCE) 001 10000 (length) 10 continentalpart	12
typeofbreakfast	(context explicit tag) 100 00001 (length) 12 english	14
customername	(IA5string) 00010110 (length) 5 ("Johan") "J" "o" "h" "a" "n"	7
firstorder	(SEQUENCE) 001 10000 (length) 21 customername typeofbreakfast	23

Exercise 41 solution

```

<?xml version="1.0" ?>
<!DOCTYPE header SYSTEM "header.dtd">
<header>
  <from>
    <person>
      <user-friendly-name>Nancy Nice</user-friendly-name>
      <local-id>nnice</local-id>
      <domain>good.net</domain>
    </person>
  </from>
  <to>
    <person>
      <user-friendly-name>Percy Devil</user-friendly-name>
      <local-id>pdevil</local-id>
      <domain>hell.net</domain>
    </person>
  </to>
  <cc>
    <person>
      <user-friendly-name>Mary Clever</user-friendly-name>
      <local-id>mclever</local-id>
      <domain>intelligence.net</domain>
    </person>
    <person>
      <user-friendly-name>rupert happy</user-friendly-name>
      <local-id>rhappy</local-id>
      <domain>fun.net</domain>
    </person>
  </cc>
</header>

```

Exercise 42 solution

```

<!ELEMENT header (from, to?, cc?)>
<!ELEMENT from (person)>
<!ELEMENT to (person+)>
<!ELEMENT cc (person+)>
<!ELEMENT person (user-friendly-name, local-id, domain)>
<!ELEMENT user-friendly-name (#PCDATA)>
<!ELEMENT local-id (#PCDATA)>
<!ELEMENT domain (#PCDATA)>

```

Exercise 43 solution

DTD specification:	XML examples:
<pre> <!ELEMENT id (name social- security-no both)> <!ELEMENT both (name, social- security-no)> <!ELEMENT name (#PCDATA)> <!ELEMENT social-security-no (#PCDATA)> </pre>	<pre> <?xml version="1.0" ?> <!DOCTYPE id SYSTEM "id.dtd"> <id><social-security-no>410201- 1410 </social-security-no></id> <?xml version="1.0" ?> <!DOCTYPE id SYSTEM "id.dtd"> <id><both><name>Eliza Doolittle</name> <social-security-no>410201-1410 </social-security- no></both></id> </pre>

	<pre><?xml version="1.0" ?> <!DOCTYPE id SYSTEM "id.dtd"> <id><name>Eliza Doolittle</name> </id></pre>
--	--

Note: The following will not work:

<pre><!ELEMENT id (name social-security-no (name, social-security- no))> <!ELEMENT name (#PCDATA)> <!ELEMENT social-security-no (#PCDATA)></pre>
--

This will not work, because the receiving program will not be able to know, when it starts to scan <name> whether this is the first or the third branch of the choice.

Exercise 44 solution

DTD specification:	XML data:
<pre><!ELEMENT movie (title, person+)> <!ELEMENT title (#PCDATA)> <!ELEMENT person EMPTY> <!ATTLIST person name CDATA #REQUIRED role (actor photographer director author administrator) #IMPLIED ></pre>	<pre><?xml version="1.0" ?> <!DOCTYPE movie SYSTEM "movie.dtd"> <movie> <title> The Postman Always Rings Twice</title> <person name="Lana Turner" role="actor" /> <person name="John Garfield" role="actor" /> <person name="Tay Garnet" role="director" /> <person name="James M. Cain" role="author" /> </movie></pre>

