*:96 Internet application layer protocols and standards

Compendium 6[v1] - *Coding*

Not allowed during the exam

Last revision: 7 Sep 2003

ABNF

ASN.1

HTML

XML

Internet Application Protocols

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

Publisher
Not yet published • City

Preliminary Table of Contents

Compendium 6 page 3

Contents

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 underlaying 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 tje 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 happended 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 recieves 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 (ГДД), Arabic (وَحَدَ), Hebrew (אלף), Browallia (ไทย), Japananse (誕生), Korean (처음 어) 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 ¿.	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 UF-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, 1 4 1 5 9, 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 metheorological 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.	JOHN□□□□□□SMITH□□□
Length encoding	The length of the data element, usually in number of octets, is sent before the element itself.	10-JOHN□SMITH
Delimeter encoding	The end of the data element is marked with some delimiter, some special code which will not appear inside the data element.	JOHN□SMITH;
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-JOHN5-SMITH

humidity.

- Which data elements belong together in structures, for example in a metheorological 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.

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

7. Have a special escape character preceding a delimeter. For example, if "`;`" is used as a delimeter, the string "`ABC;DEF`" might be encoded as "`ABC\;DEF`". Any occurence 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 delimeters characters outside this set. An example of this is the BASE64 and UUENCODE formats.

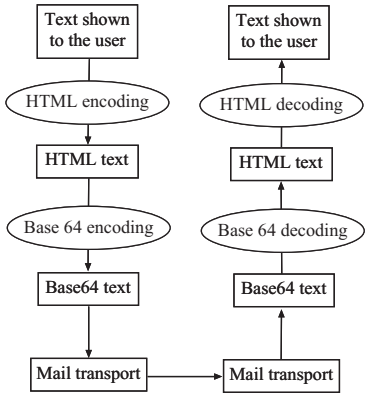


Figure 1: Encoding in several layers

11. Encode the special characters with some sequence of characters which contains the numerical value of the character code. Examples:
    - *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 "`&#196;`", where ""196"" is the decimal byte value of this character.
    - *The MIME header encoding method*, where the character "Ä" is encodes 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 "`&Auml;`" where "`Auml`" means "A with umlaut", "umlaut" is the German word for putting two dots on top of a wovel.

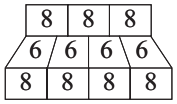
In some cases, several different character encoding methods are used on top of each other. They must then be undone in the reverse order to get back the original text. For example, if HTML text is sent in e-mail with the base64 encoding method, then, as shown in Figure 1, the text might first be encoded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

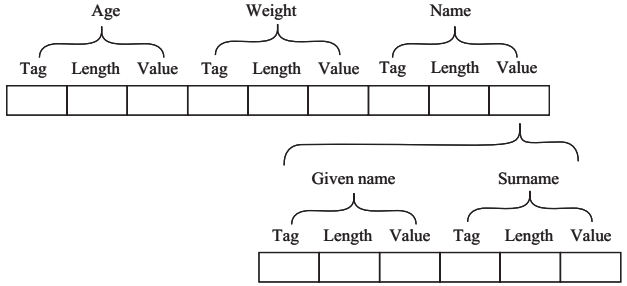


Figure 2: Example of tag-lenght-value encoding

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.
Components 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
	Surname	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 delimeter method. In the example above, "`:`", "`;`", "`<`", "`>`", "from", "by", "id" and space are used as delimeters. "`Received`", "`Message-ID`", "`From`", "`To`", "`Subject`" and "`Date`" are used as tags, but in the "`Received`" field there are subtags "`from`", "`by`", and "`id`".

ugmented Backus-Naur Form,

2. A

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 linenar 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 ocurrences of the following construct, "1*" means one or more, "*5" means between zero and five occurences.

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 <dduck@disney.com>
```

Example 3, value b:

```
Sender: Walt Disney <walt@disney.com>
From:   Donald Duck <dduck@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 emenent, 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"    ":"         ; one per relay
               ["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 Ä130.237.158.10Å)
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 langauge 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:

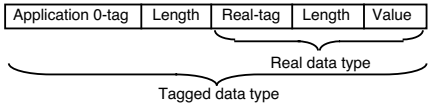
```
Temperature     ::= [APPLICATION 0] REAL

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 AltituteReading

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.

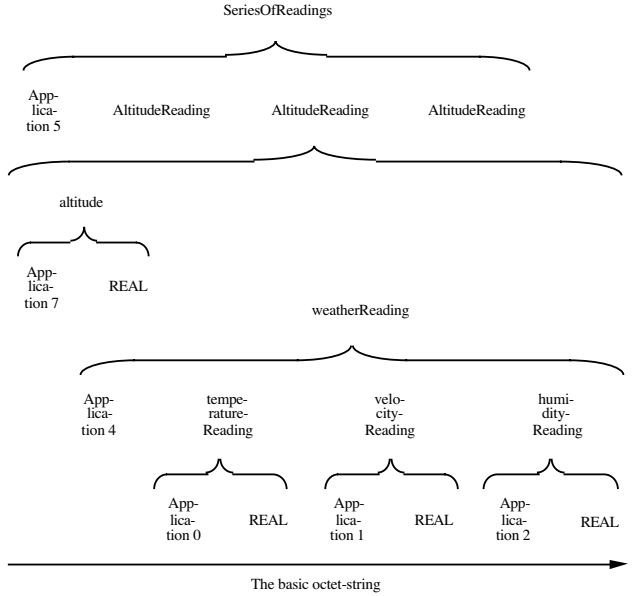


Figure 3: **How ASN.1 and BER is used to produce an octet string.**

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 lengthes 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 isn 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 yeards. 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	*Warning: Constraints are strongly recommended for Graphic, General, Universal, BMP and UTF8 strings*
OCTET STRING	IA5String	ANY	
NULL	GraphicString	[Tagged]	
OBJECT IDENTIFIER	GeneralString	*<Different variants*	
	UniversalString	*< of ISO 10646, not*	
	BMPString	*< in the 1998*	
	UTF8String	*< version*	
	CharacterString		

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 parenthesises 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 +∞ and -∞), 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 yeard), 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 inces 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: +∞, --∞ 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 ::= {31415926535879323846243 3, 10, 25 }

zero REAL   ::= 0                                    dividuals

topValue REAL  ::= PLUS-INFINITY                     n
```

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 10 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 i

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 subtype	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 enciphermed 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

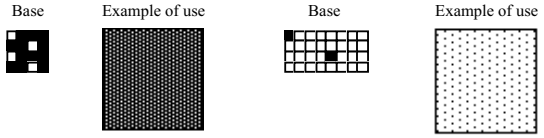
first four values.

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:



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 I2))
 Base ::= BIT STRING (SIZE ( 0 I 2 .. 7 I 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 se PrintableString. Suggest a definition of this in ASN.1.

1.1.35. Character String Types

ASN.1 has several Character String types for different charactersets.

NumericString“	“0” .. “9” and “ ”
PrintableString	“a”..“z”, “A”..“Z”, “0”..“9” ' ( ) + , - . / : = ?
TeletexString T61String	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”.
VisibleString ISO646String	Printable characters, including space, from ISO 646 (”ASCII”), but no format control characters like Carriage Return or Line Feed.
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" I "1" I "2" I "3" I "4" I "5" I "6" I "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 19. 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
Childen ::= 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 specifed 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:

```
ET {
 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 alterantives 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 IDENTIFIER**s 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 IDENTIFIER**s.

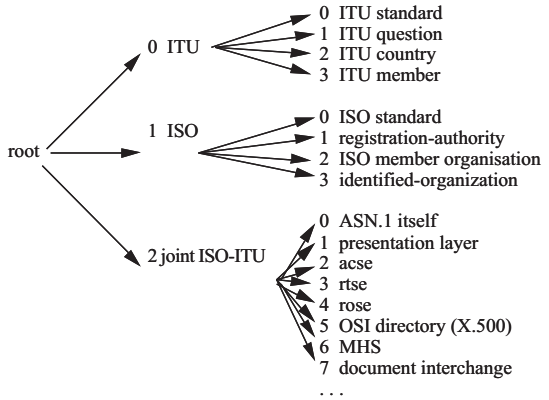


Figure 4: Domain name tree used in selecting OBJECT IDENTIFIERs

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

Simple types		*Structured types*	
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

Character String Types		*UsefulTypes*	
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 specificing 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

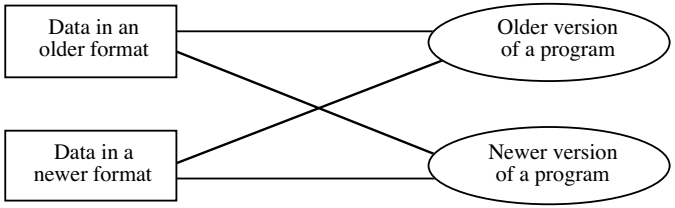


Figure 5:Allow communication between old and new programs

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 types 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-IDENTFIER 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.

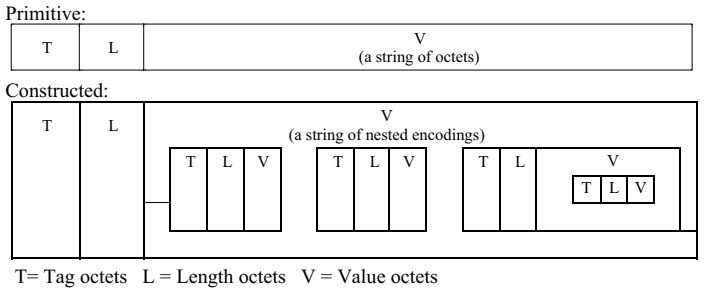


Figure 6 Tag-Length-Value encoding in BER

1.1.46. The Tag or Identifier field

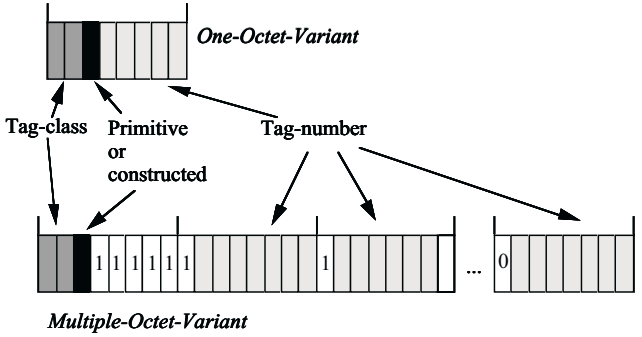


Figure 7: Use of bits in BER encoding

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 suceeding octet is 0 for the last octet, 1 for all but the last octet.

1.1.47. The Length Field in BER

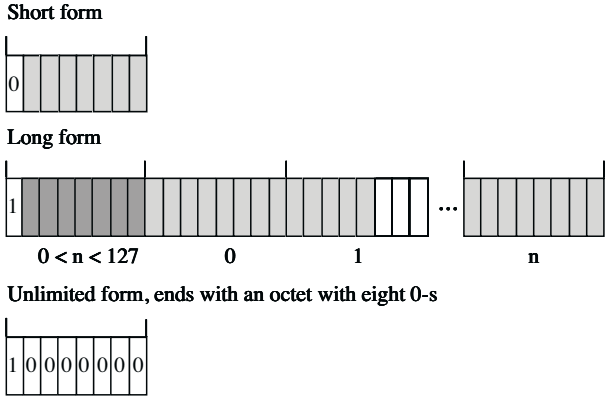
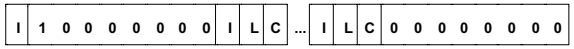


Figure 8: The Length field in BER

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 cotet 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 sitll 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.



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.

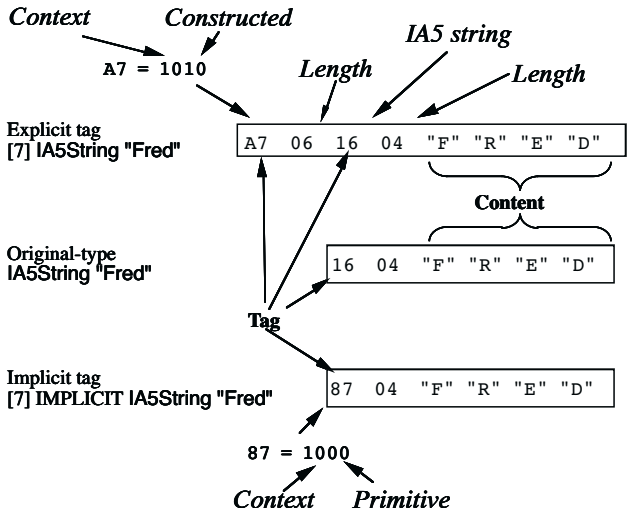


Figure 9: Encoding of a tagged string

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 OPTIIONAL }

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


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

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

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

$22_{10} = 16_{16}$= 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 defintions 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

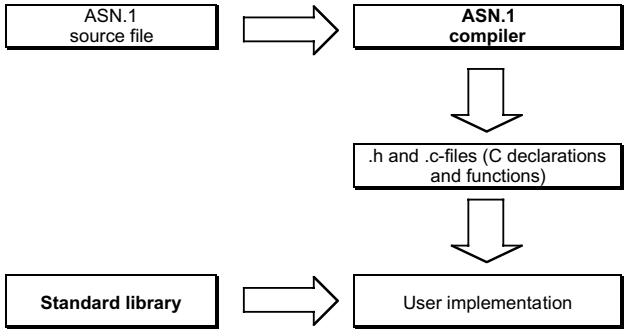


Figure 10: ASN.1 compilers

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.

6

7

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:
`<p>First paragraph containing one <b>boldface</b> word.` `<p>Second paragraph with a line break text after the line break.`	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 `<b>` tag indicates bold-face text, the `</b>` tag indicates the end of bold-face text, and the `<br>` 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:
`Jim&apos;s e-mail address is Jim Sim &gt;jsim&foo.bar&gt;.`	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:

`Read the`
`<a href="http://dsv.su.se/jpalme/abook/">`
`web page</a>associated with this book.`
	Read the web page associated with this book.

The links to other document contain URIs (see chapter ¿¿¿). 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:
`<IMG SRC="ietflogo.gif" BORDER="0">This is the logo of the Internet Engineering Task Force.`	 This is the logo of the Internet Engineering Task Force.

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

```
<!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>
```

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	`<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>`
		This is the left frame.	This is the top frame.

left.html	`<HTML><HEAD>`
	`<TITLE>Left frame</TITLE>`
	`</HEAD><BODY>`
	`This is the left frame.`
	`</BODY></HTML>`		

top.html	`<HTML><HEAD>`
	`<TITLE>Top frame</TITLE>`
	`</HEAD><BODY>`
	`This is the top frame.`
	`</BODY></HTML>`
			This is the bottom frame.

bottom.html	`<HTML><HEAD>`
	`<TITLE>Bottom frame</TITLE>`
	`</HEAD><BODY>`
	`This is the bottom frame.`
	`</BODY></HTML>`		

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:

`<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>`
	This is the main heading
	This is the text below the main heading.

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:

`<html>`
`<head>`
`<title>CSS Example</title>`
`<LINK rel="stylesheet"`
`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>`
	This is the main heading
	This is the text below the main heading.

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.

2

8

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, **<TITLE>** and **<title>** are identical.	Yes, **<TITLE>** and **<title>** are two different tags, specifying two different element types. An element which starts with **<TITLE>** must end with **</TITLE>**, not with **</title>**.

Function	HTML	XML
Acceptance of coding errors	Most web browsers accept many coding errors.	Code must be syntactically correct, and only syntactically correct XML-encoded data should be accepted by an XML processor.
	Example: `<B><I>Bold-italic text</B></I>` is not correct HTML, but accepted by most web browsers. The example is incorrect, because the elements are incorrectly nested. The element **<I>** is neither inside or outside the element **<B>** tag. Correct HTML would be: `<B><I>Bold italic text</I></B>` *(Element **<I>** inside element **<B>**)* or `<I><B>Bold italic text</B></I>` *(Element **<B>** inside element **<I>**)* 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 interpet almost any kind of vastly inccorrect HTML text. 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: **<CENTER>**) do in fact work in more browsers than the corresponding correct constructs (**<DIV ALIGN=CENTER>** instead of **<CENTER>**). In the case of XML, texts will mostly be produced by software, which will reduce the amount of incorrect XML data.	
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>
```

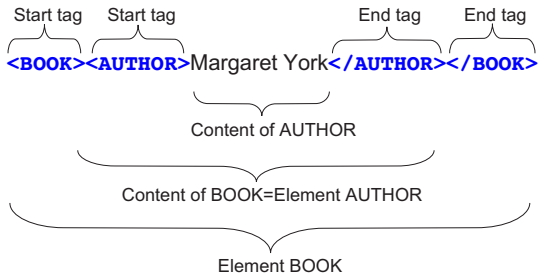
In Table 14 is an example of a DTD and an XML text encoded according to this DTD.

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

Explanation:	DTD text:	XML text:
Indicates that this is an XML document.		`<?xml version="1.0"?>`
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.		`<!DOCTYPE person SYSTEM "person.dtd">`
Specifies the element type tagged PERSON and that it should always contain, within it, elements tagged NAME, BIRTHYEAR and WAGE.	`<!ELEMENT PERSON (NAME, BIRTHYEAR, WAGE)>`	`<PERSON>`
Similar to PERSON.	`<!ELEMENT NAME (FIRST-NAME, SURNAME)>`	`<NAME>`
(#PCDATA) specifies that elements of this element type will contain text outside the tags, in this case "John" and "Smith".	`<!ELEMENT FIRST-NAME (#PCDATA)>`	`<FIRST-NAME>John</FIRST-NAME>`
	`<!ELEMENT SURNAME (#PCDATA)>`	`<SURNAME>Smith</SURNAME>`
		`</NAME>`
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	`<!ELEMENT BIRTHYEAR (#PCDATA)>`	`<BIRTHYEAR>1941</BIRTHYEAR>`
	`<!ELEMENT WAGE (#PCDATA)>`	`<WAGE>57000</WAGE>`
		`</PERSON>`

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/



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 parenthesises, 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:

```
<NAME><FIRST-NAME>John</FIRST-NAME>
<SURNAME>Smith</SURNAME>
</NAME></PERSON>
```

```
<NAME>
<FIRST-NAME>John
</FIRST-NAME>
<SURNAME>Smith
</SURNAME>
</NAME>
</PERSON>
```

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:

`<!ELEMENT e-mail (#PCDATA)>`	`<?xml version="1.0" ?>`
	`<!DOCTYPE e-mail SYSTEM "e-mail.dtd">`
	`<e-mail>"John Smith" <jsmith@foo.bar>`
	`</e-mail>`

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:

`<!ELEMENT e-mail (#PCDATA)>`	`<?xml version="1.0" ?>`
	`<!DOCTYPE e-mail SYSTEM "e-mail.dtd">`
	`<e-mail>`
	`<![CDATA["John Smith" <jsmith@foo.bar>]]>`
	`</e-mail>`

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:

`<!ELEMENT cup EMPTY>`	`<?xml version="1.0" ?>`
	`<!DOCTYPE cup SYSTEM "cup.dtd">`
	`<cup></cup>`

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:

`<!ELEMENT family (husband, wife)>`
`<!ELEMENT husband (#PCDATA)>`
`<!ELEMENT wife (#PCDATA)>`
	`<?xml version="1.0" ?>`
	`<!DOCTYPE family SYSTEM "family.dtd">`
	`<family>`
	`  <husband>John</husband>`
	`  <wife>Margaret</wife>`
	`</family>`

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:

`<!ELEMENT family (husband, wife, child*)>`
`<!ELEMENT husband (#PCDATA)>`
`<!ELEMENT wife (#PCDATA)>`
`<!ELEMENT child (#PCDATA)>`
	`<?xml version="1.0" ?>`
	`<!DOCTYPE family SYSTEM "family.dtd">`
	`<family>`
	`  <husband>John</husband>`
	`  <wife>Margaret</wife>`
	`  <child>Eve</child>`
	`  <child>Peter</child>`
	`</family>`

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

Example DTD specification:	XML data:

`<!ELEMENT child-family (husband, wife, child+)>`
`<!ELEMENT husband (#PCDATA)>`
`<!ELEMENT wife (#PCDATA)>`
`<!ELEMENT child (#PCDATA)>`
	`<?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>`

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 occurences of a.
`a+`	1 or more occurences of a.
`a?`	0 or one occurences 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:

`<!ELEMENT vehicles (vehicle*)>`
`<!ELEMENT vehicle (bike | car)>`
`<!ELEMENT bike (#PCDATA)>`
`<!ELEMENT car (#PCDATA)>`
	`<?xml version="1.0" ?>`
	`<!DOCTYPE vehicles SYSTEM "vehicles.dtd">`
	`<vehicles>`
	`  <vehicle><bike>Crescent</bike></vehicle>`
	`  <vehicle><car>Volvo</car></vehicle>`
	`</vehicles>`

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:

`<!ELEMENT transport ((bike | car)*)>`
`<!ELEMENT bike (#PCDATA)>`
`<!ELEMENT car (#PCDATA)>`
	`<?xml version="1.0" ?>`
	`<!DOCTYPE transport SYSTEM "transport.dtd">`
	`<transport>`
	`  <bike>Crescent</bike>`
	`  <car>Volvo</car>`
	`</transport>`

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

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

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.	`<!ATTLIST book`
	`binding (hardback | paperback)`
	`"hardback">`
		This default value should be assumed if the attribute is not specified in the XML text.

`#REQUIRED`	`<!ATTLIST book`
	`binding (hardback | paperback)`
	`#REQUIRED>`
		No default value is allowed, the attribute must always be specified in the XML text.

`#IMPLIED`	`<!ATTLIST book` `binding (hardback | paperback) #IMPLIED>`	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.

`#FIXED`	`<!ATTLIST book`
	`binding (hardback | paperback) #FIXED`
	`"hardback">`
		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`	`<!ATTLIST book` `title CDATA #REQUIRED>`	Any character string.

A list of enumerated values	`<!ATTLIST book`
	`binding (hardback | paperback)`
	`"hardback">`
		Restricted to the listed values only.

`ID`	`<!ATTLIST book entryno ID #REQUIRED>`	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`	`<!ATTLIST author authorid ID`
	`#REQUIRED>`
	`<!ATTLIST book authorid IDREF`
	`#REQUIRED>`
		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`	`<!ATTLIST author authorid ID`
	`#REQUIRED>`
	`<!ATTLIST book authorids IDREFS`
	`#REQUIRED>`
		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:
	`<!ELEMENT LOGO EMPTY>`
	`<!ATTLIST LOGO GIF-FILE ENTITY`
	`#REQUIRED>`
	`<!ENTITY DSV-LOGO SYSTEM "dsv-logo.gif">`
	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 **`<IMG>`** tags in HTML, the actual binary file is not included, just referenced.

Type:	Example:	Description:
	`<LOGO GIF-FILE="DSV-LOGO"/>`	

ENTITIES	DTD text:
	`<!ELEMENT LOGO EMPTY>`
	`<!ATTLIST LOGO GIF-FILE ENTITIES #REQUIRED>`
	`<!ENTITY DSV-LOGO SYSTEM "dsv-logo.gif">`
	`<!ENTITY KTH-LOGO SYSTEM "kth-logo.gif">`
	XML text:
	`<LOGO GIF-FILE="DSV-LOGO KTH-LOGO"/>`
		A list of more than one entity.

NMTOKEN	`<!ATTLIST variable-name #NMTOKEN>`	A name, formatted like a variable name in a computer program. Useful when you use XML to generate source program code.
NMTOKENS	`<!ATTLIST variables #NMTOKENS>`	A list of names, similar as for NMTOKEN above.
NOTATION	`<!ATTLIST SPEECH PLAYER NOTATION ( MP3 | QUICKTIME ) #REQUIRED>`	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:

`<!ELEMENT book-att EMPTY>`
`<!ATTLIST book-att`
`  author #REQUIRED`
`  title #REQUIRED`
`>`
	`<?xml version="1.0" ?>`
	`<!DOCTYPE book-att SYSTEM "book-att.dtd">`
	`<book-att`
	`author="Margaret Yorke"`
	`title="False Pretences"/>`

DTD specification using subelements:	XML data:

`<!ELEMENT book-sub (author, title)>`
`<!ELEMENT author (#PCDATA)>`
`<!ELEMENT title (#PCDATA)>`
	`<?xml version="1.0" ?>`
	`<!DOCTYPE book-sub SYSTEM "book-sub.dtd">`
	`<book-sub>`
	`<author>Margaret Yorke</author>`
	`<title>False Pretences</title>`
	`</book-sub>`

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:

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

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

DTD specification:	XML data:

<!ELEMENT child-family (husband, wife, child+)>
<!ELEMENT husband (#PCDATA)>
<!ELEMENT wife (#PCDATA)>
<!ELEMENT child (#PCDATA)>
	<?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>

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:

```
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 }
```

File ticket.xml:

```
<?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>
```

Visual rendering:

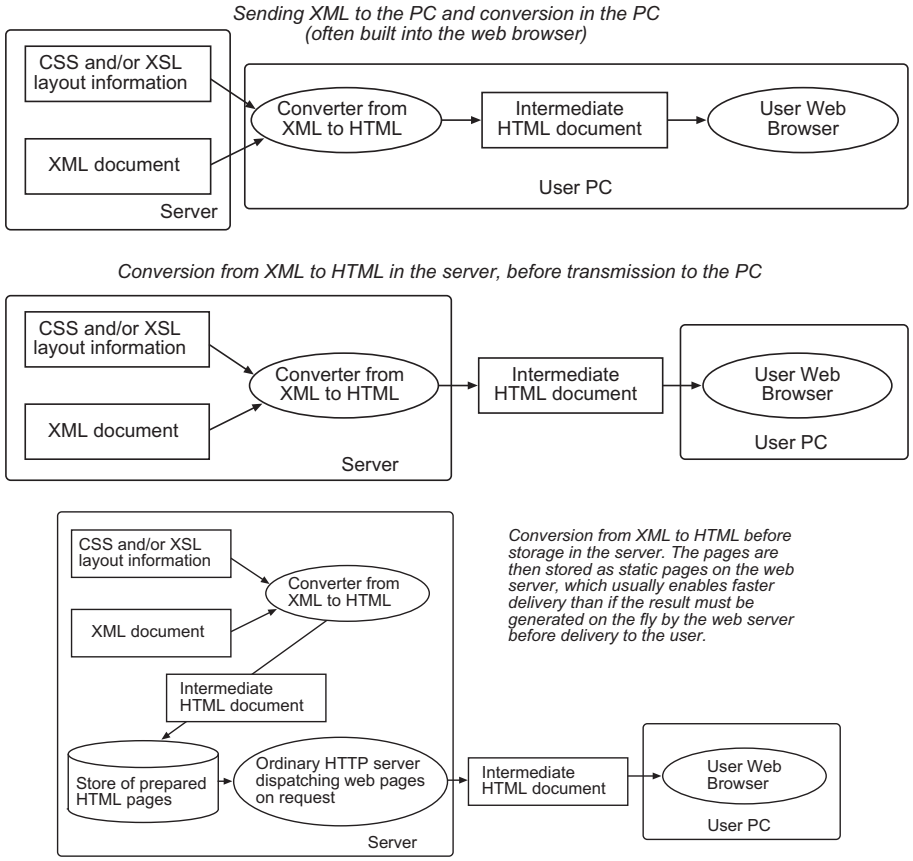
TICKET

Oslo	Stockholm
Mon 13 Jan 12:13	**Mon 13 Jan 18:45**
2 Class	Cabin 3 Seat 5

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



HTML does not support alternative versions of the same information for different 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: `<IMG SRC="image.gif">`

③ Use method ②, but combine it with the MHTML method (see page ¿¿¿) 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::

neral DTD specifications:
e name person.dtd)

```
ELEMENT person (name, birthyear)>
ELEMENT name (#PCDATA)>
ELEMENT birthyear (#PCDATA)>
ATTLIST person
gender ( male | female )
        #REQUIRED
status ( unmarried | married |
         divorced | widow |
         widower ) #REQUIRED
```

TD using this specification:
ile name family.dtd)

```
ELEMENT family (person+)>
ENTITY % person SYSTEM "person.dtd">
erson;
```

XML data:

```
<?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>
```

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 entitites.

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. Additionaly, 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. XHMTL

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 `<A HREF=>`
- All tags must be ended, e.g. `<p>First paragraph<br/>second line.</p>`
- No syntax errors allowed, e.g. not `<p><strong>Strong text</p></strong>`

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:

`mily = "Family"`
`CRLF *(Person)`
`"End of Family"`
 	
`rson = "Person" CRLF`
`" Name: " 1*A CRLF`
`" Birthyear: " 4D CRLF`
`" Gender: "`
`("Male"/"Female") CRLF`
`" Status: "`
`("unmarried"/ "married"/`
`"divorced"/ "widow"/`
`"widower" )`
	= 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) }
		`<!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`
		`>`

:ample of textual encoding:	Example of BER encoding:	Example of XML encoding:

`mily`
`rson`
`Name: John Smith`
`Birthyear: 1958`
`Gender: Male`
`Status: Married`
`rson`
`Name: Eliza Tennyson`
`Birthyear: 1959`
`Gender: Female`
`Status: Married`
`d of Family`
	(Each box represents one octet. Two-character codes are hexadecimal numbers, one character codes are characters)
	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
	02 02 07 A7
	0A 01 01
	0A 01 01
		<?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>

) 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 becaome 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 oxymornonic.

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.robbins@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-auther 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 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

6. References

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 ASN.1 and 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 a 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 some newer standards.
RFC 2279:	RFC2279 UTF-8, a transformation format of ISO 10646. F. Yergeau. January 1998. (Obsoletes RFC2044)	Specification of the UTF-8 encoding format 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 standard.
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 methods.
Harold 1999:	XML Bible, by Eliott Rusty Harold, IDG Books, Foster City, CA, U.S.A., 1999.	A very thorough and readable guilde to all aspects of XML. Some chapters have been updated after publication, and can be downloaded 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 neat layout 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 text 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 issue of name spaces 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
directory-name = 1* (ALPHA / DIGIT )
directory-name = 1* (ALPHA / DIGIT )
```

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 Nametype 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 Nametype OPTIONAL,
sname 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)
```

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 {
GivenName [0] PrintableString
SurName [1] PrintableString  }
```

Both tags can be removed

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

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].

```
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

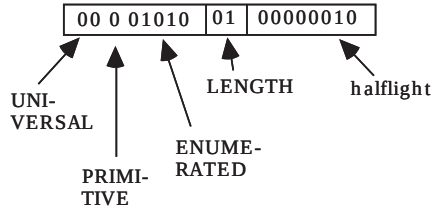
```
CarDriving { 1 2 4711 18 } DEFINITIONS EXPLICIT TAGS ::=
BEGIN
IMPORTS MainOperation FROM Driving {1 2 4711 17};

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	(contextexplicit 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:

```
<!ELEMENT id ( name | social-
security-no | both)>
<!ELEMENT both (name, social-
security-no)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT social-security-no
(#PCDATA)>
```

XML examples:

```
<?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>
```

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

Note: The following will not work:

```
<!ELEMENT id ( name | social-security-no | (name, social-security-
no))>
<!ELEMENT name (#PCDATA)>
<!ELEMENT social-security-no (#PCDATA)>
```

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:
<!ELEMENT movie (title, person+)> <!ELEMENT title (#PCDATA)> <!ELEMENT person EMPTY> <!ATTLIST person name CDATA #REQUIRED role (actor | photographer | director | author | administrator) #IMPLIED >	<?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>

A Summary of ASN.1 Types and their Usage

Abstract: The basics of how to design ASN.1 code is shown through a practical example.

By Jacob Palme (**http://www.palme.nu/jacob/**)

This document is also available in HTML format at URL

http://dsv.su.se/jpalme/internet-course/solving-asn-1-exercise.html

ASN.1 type	Usage
ANY	Data, whose format is to be specified in the future or by someone else. Example: **FutureData ::= SEQUENCE { type VisibleString, value ANY }**
BitString	A string of Boolean values. Example: **DaysOpen ::= BitString { monday (0), tuesday (2), wednesday (3), thursday (4), friday (5), saturday (6), sunday (7) }**
Boolean	A single Boolean (true/false, or 1/0) value. Example: **Gender ::= BOOLEAN -- Male=true, Female=false**
CharacterString: **NumericString** **PrintableString** **TeletexString** **VideotexString** **VisibleString** **IA5String** **GraphicString** **GeneralString** **UniversalString**	Character strings using different sets of allowed characters. Example: **Surname ::= VisibleString**
CHOICE	One of a list of different types, combined with a value of the chosen type. Example 1: **CHOICE { car Motorcar, bike Bicycle, boat Boat }** Example 2: Tags needed since all elements must be of different type: **CHOICE { registrationnumber [1] VisibleString, name [2] VisibleString }**
ENUMERATED	Can have any of a limited set of enumerated values. Example: **Weekday ::= ENUMERATED { monday (0), tuesday (2), wednesday (3), thursday (4), friday (5), saturday (6), sunday (7) }**
INTEGER	An integer value, example: **Age ::= INTEGER (0 .. MAX)**
OCTET STRING	A string of octets, whose data is not specified in ASN.1. Example: **GIF-Picture ::= OCTET STRING**
REAL	A real value, example: **Windvelocity ::= REAL**
SEQUENCE	Several different types of data in sequence. Example: **Name ::= SEQUENCE { Givenname VisibleString, Surname VisibleString }**
SEQUENCE OF	A list of one or more items of the same type. Example: **Family ::= SEQUENCE OF Name**
SET	Same as **SEQUENCE**, but no assumed order.
SET OF	Same as **SEQUENCE OF**, but no assumed order.

Example of how you can Think when Solving an ASN.1 Exam Question

All page references are to pages in the book ASN.1 The Tutorial & Reference by Douglas Steedman.

Question 2 in the exam 1999-11-09

Below is a specification of a proposed addition to Internet e-mail. The specification is based on ABNF.

Write a specificaction which will convey the same information using ASN.1. You need only translate the syntax (down to "**`Note:`**") not the explanatory text which comes after "**`Note:`**".

Note: Your solution need only transfer the information, not the syntactical form.

Supersedes

Syntax

```
    Supersedes-field          = "Supersedes:" " " identifier
                                *(identifier)
                                optional-parameter-list
                                CRLF

    optional-parameter-list   = *( ";" " " parameter )

    parameter                 = parameter-name [ "="
                                parameter-value ]

    parameter-name            = "noshow" / "show" / "repost"
                                private-parameter /
                                future-parameter
```

`Note:` There is no comma between multiple values, and that each Message-ID value is to be surrounded by angle brackets.

`Warning:` Some software may not work correctly with comments in header fields, especially comments in other places than at the beginning and end of the field value.

Warning: This header MUST be spelled "Supersedes" and not "Supercedes".

Semantics

The Supersedes header identifies previous correspondence, which this message supersedes. Different messaging agents such as user agents, mailing list expanders and mailing list archives. A user agent is expected to handle this field in much the same way as the In-Reply-To and References header.

Note: The Message-ID of a superseding message MUST be different from the Message-ID of the superseded message. The Message-ID of the superseded message is used as value in the "Supersedes:" header, not in the Message-ID of the superseding message.

Parameters:

noshow — In the opinion of the sender, this message makes such a minor change to the superseded version, that a recipient, who has already seen the previous verson, will probably not want to see the new version, unless the user explicitly asks for it.

show — In the opinion of the sender, this message makes such a large change to the superseded version, that a recipient, who has already seen the previous version, will probably want to see the new version, too.

repost — This document is a document which is repeatedly, at regular or irregular intervals, reposted, such as FAQs or mailing list monthly information.

None of these parameters have values. The "noshow" and the "show" parameters are mutually exclusive, but both of them can occur together with the "repost" parameter.

How to solve this exam question

First analyse what information is sent from with this protocol element. The information sent is:

1. That this is a Supersedes header field.
2. A list of one or more identifiers of superseded messages.
3. A list of one or more optional parameters.
4. Each optional parameter can have a name, and an optional value.
5. The parameters noshow, show and repost are specified, additional private or future parameters can be added.

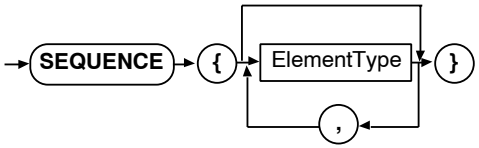
Note that the ":", ";", "=" are part of the ABNF syntactical structure, and should not be sent, since ASN.1 has its own, alternative methods of syntactically structuring the information sent.

How is this information formatted with ASN.1?

1. That this is a Supersedes header field

Presumably, there is a list of header fields, of which this is one. To create a list of elements of the same type, you use **SEQUENCE OF**.

Look at the syntax for **SEQUENCE OF** on page 141:



Using this syntax, you can construct it as follows:

HeaderFields ::= SEQUENCE OF Header

Each header contains information on which header it is, and the data for this header. Thus, we need two main parts of each header, name and value, so a **SEQUENCE** might be used. See the syntax for **SEQUENCE** on page 141:

```
Header ::= SEQUENCE {     headername,          -- Not complete yet
                          headerdata }
```

"**headername**" and "**headerdata**" have the syntax for ElementType, which you can find on page 795 of Compendium 1. As you can see there, it is a NamedType, possibly followed by **OPTIONAL** or **DEFAULT**.

And the syntax of NamedType can be found on page 139 as an identifer and a type:

```
Header ::= SEQUENCE {     headername Headernametype,
                          headerdata Headerdatatype }
```

Headernametype should tell which of a number of known headers this is. Since there are, presumably, a limited number of known headers, **ENUMERATED** is suitable. Thus, we could specify **Headernametype** as:

```
Headernametype ::= ENUMERATED {     From (0), To (1), Cc (2), Date (3),
                                    Supersedes (4) }
```

There are probably more values, but that is not part of this question.

The syntax for **ENUMERATED** can be found on page 135 of Compendium 1. It uses Namednumber, whose syntax can be found on page 139.

Another alternative would be to use a text string:

Headernametype ::= VisibleString

2. A list of one or more identifiers of superseded messages,

3. A list of one or more optional parameters

Headerdata consists of two main groups of data in sequence, so we use the **SEQUENCE** type:

```
Headerdatatype ::= SEQUENCE {     identifierlist Identifierlisttype,
                                  parameterlist Parameterlisttype OPTIONAL }
```

The identifierlist consists of one or more identifiers, which are text strings:

Identifierlisttype ::= SEQUENCE OF VisibleString

4. A list of one or more optional parameters

5. The parameters noshow, show and repost are specified, additional private or future parameters can be added

Since this is a list of one or more parameters, we use **SEQUENCE OF**:

```
Parameterlisttype ::= SEQUENCE OF Parameter
```

Each Parameter is either a built in parameter, a private parameter or a future parameter. We use a **CHOICE**:

```
Parameter ::= CHOICE {      builtinparameter   Builtinparametetertype,
                            privateparameter [1] Newparametertype,
                            futureparameter  [2] Newparametertype }
```

The tags are necessary, since no two elements in a **CHOICE** can have the same type.

The Builtinparametertype is an indication of one of three possible values, thus **ENUMERATED** is suitable:

```
Builtinparametertype ::= ENUMERATED { noshow (0), show (1), repost (3) }
```

The private and future parameters have a name and an optional value:

```
Newparametertype ::= SEQUENCE {     name VisibleString,
                                    value ANY OPTIONAL }
```

ANY is a placeholder where you can put any kind of data. Since this data is in text string format in ABNF, we might use for example **VisibleString** instead of **ANY** above.

We are ready

So now we are ready. Just collect the ASN.1 together:

```
HeaderFields ::= SEQUENCE OF Header

Header ::= SEQUENCE {       headername Headernametype,
                            headerdata Headerdatatype }

Headernametype ::= VisibleString

Headerdatatype ::= SEQUENCE {       identifierlist    Identifierlisttype,
                                    parameterlist     Parameterlisttype OPTIONAL }

Identifierlisttype ::= SEQUENCE OF VisibleString

Parameterlisttype ::= SEQUENCE OF Parameter

Parameter ::= CHOICE {      builtinparameter   Builtinparametetertype,
                            privateparameter [1] Newparametertype,
                            futureparameter  [2] Newparametertype }

Builtinparametertype ::= ENUMERATED { noshow (0), show(1), repost(3) }

Newparametertype ::= SEQUENCE {     name VisibleString,
                                    value ANY OPTIONAL }
```

A Layman's Guide to a Subset of ASN.1, BER, and DER

An RSA Laboratories Technical Note
Burton S. Kaliski Jr.
Revised November 1, 1993*

Abstract. *This note gives a layman's introduction to a subset of OSI's Abstract Syntax Notation One (ASN.1), Basic Encoding Rules (BER), and Distinguished Encoding Rules (DER). The particular purpose of this note is to provide background material sufficient for understanding and implementing the PKCS family of standards.*

1. Introduction

It is a generally accepted design principle that abstraction is a key to managing software development. With abstraction, a designer can specify a part of a system without concern for how the part is actually implemented or represented. Such a practice leaves the implementation open; it simplifies the specification; and it makes it possible to state "axioms" about the part that can be proved when the part is implemented, and assumed when the part is employed in another, higher-level part. Abstraction is the hallmark of most modern software specifications.

One of the most complex systems today, and one that also involves a great deal of abstraction, is Open Systems Interconnection (OSI, described in X.200). OSI is an internationally standardized architecture that governs the interconnection of computers from the physical layer up to the user application layer. Objects at higher layers are defined abstractly and intended to be implemented with objects at lower layers. For instance, a service at one layer may require transfer of certain abstract objects between computers; a lower layer may provide

*Supersedes June 3, 1991 version, which was also published as NIST/OSI Implementors' Workshop document SEC-SIG-91-17. PKCS documents are available by electronic mail to <pkcs@rsa.com>.

003-903015-110-000-000

transfer services for strings of ones and zeroes, using encoding rules to transform the abstract objects into such strings. OSI is called an open system because it supports many different implementations of the services at each layer.

OSI's method of specifying abstract objects is called ASN.1 (Abstract Syntax Notation One, defined in X.208), and one set of rules for representing such objects as strings of ones and zeros is called the BER (Basic Encoding Rules, defined in X.209). ASN.1 is a flexible notation that allows one to define a variety data types, from simple types such as integers and bit strings to structured types such as sets and sequences, as well as complex types defined in terms of others. BER describes how to represent or encode values of each ASN.1 type as a string of eight-bit octets. There is generally more than one way to BER-encode a given value. Another set of rules, called the Distinguished Encoding Rules (DER), which is a subset of BER, gives a unique encoding to each ASN.1 value.

The purpose of this note is to describe a subset of ASN.1, BER and DER sufficient to understand and implement one OSI-based application, RSA Data Security, Inc.'s Public-Key Cryptography Standards. The features described include an overview of ASN.1, BER, and DER and an abridged list of ASN.1 types and their BER and DER encodings. Sections 2—4 give an overview of ASN.1, BER, and DER, in that order. Section 5 lists some ASN.1 types, giving their notation, specific encoding rules, examples, and comments about their application to PKCS. Section 6 concludes with an example, X.500 distinguished names.

Advanced features of ASN.1, such as macros, are not described in this note, as they are not needed to implement PKCS. For information on the other features, and for more detail generally, the reader is referred to CCITT Recommendations X.208 and X.209, which define ASN.1 and BER.

Terminology and notation. In this note, an octet is an eight-bit unsigned integer. Bit 8 of the octet is the most significant and bit 1 is the least significant.

The following meta-syntax is used for in describing ASN.1 notation:

`BIT`	monospace denotes literal characters in the type and value notation; in examples, it generally denotes an octet value in hexadecimal
$\boldsymbol{n_1}$	bold italics denotes a variable
[]	bold square brackets indicate that a term is optional
{}	bold braces group related terms
|	bold vertical bar delimits alternatives with a group

... bold ellipsis indicates repeated occurrences

= bold equals sign expresses terms as subterms

2. Abstract Syntax Notation One

Abstract Syntax Notation One, abbreviated ASN.1, is a notation for describing abstract types and values.

In ASN.1, a type is a set of values. For some types, there are a finite number of values, and for other types there are an infinite number. A value of a given ASN.1 type is an element of the type's set. ASN.1 has four kinds of type: simple types, which are "atomic" and have no components; structured types, which have components; tagged types, which are derived from other types; and other types, which include the `CHOICE` type and the `ANY` type. Types and values can be given names with the ASN.1 assignment operator (`::=`), and those names can be used in defining other types and values.

Every ASN.1 type other than `CHOICE` and `ANY` has a tag, which consists of a class and a nonnegative tag number. ASN.1 types are abstractly the same if and only if their tag numbers are the same. In other words, the name of an ASN.1 type does not affect its abstract meaning, only the tag does. There are four classes of tag:

Universal, for types whose meaning is the same in all applications; these types are only defined in X.208.

Application, for types whose meaning is specific to an application, such as X.500 directory services; types in two different applications may have the same application-specific tag and different meanings.

Private, for types whose meaning is specific to a given enterprise.

Context-specific, for types whose meaning is specific to a given structured type; context-specific tags are used to distinguish between component types with the same underlying tag within the context of a given structured type, and component types in two different structured types may have the same tag and different meanings.

The types with universal tags are defined in X.208, which also gives the types' universal tag numbers. Types with other tags are defined in many places, and are always obtained by implicit or explicit tagging (see Section 2.3). Table 1 lists some ASN.1 types and their universal-class tags.

Type	Tag number (decimal)	Tag number (hexadecimal)
`INTEGER`	2	`02`
`BIT STRING`	3	`03`
`OCTET STRING`	4	`04`
`NULL`	5	`05`
`OBJECT IDENTIFIER`	6	`06`
`SEQUENCE` and `SEQUENCE OF`	16	`10`
`SET` and `SET OF`	17	`11`
`PrintableString`	19	`13`
`T61String`	20	`14`
`IA5String`	22	`16`
`UTCTime`	23	`17`

Table 1. Some types and their universal-class tags.

ASN.1 types and values are expressed in a flexible, programming-language-like notation, with the following special rules:

- Layout is not significant; multiple spaces and line breaks can be considered as a single space.
- Comments are delimited by pairs of hyphens (`--`), or a pair of hyphens and a line break.
- Identifiers (names of values and fields) and type references (names of types) consist of upper- and lower-case letters, digits, hyphens, and spaces; identifiers begin with lower-case letters; type references begin with upper-case letters.

The following four subsections give an overview of simple types, structured types, implicitly and explicitly tagged types, and other types. Section 5 describes specific types in more detail.

2.1 Simple types

Simple types are those not consisting of components; they are the "atomic" types. ASN.1 defines several; the types that are relevant to the PKCS standards are the following:

`BIT STRING`, an arbitrary string of bits (ones and zeroes).

`IA5String`, an arbitrary string of IA5 (ASCII) characters.

`INTEGER`, an arbitrary integer.

NULL, a null value.

OBJECT IDENTIFIER, an object identifier, which is a sequence of integer components that identify an object such as an algorithm or attribute type.

OCTET STRING, an arbitrary string of octets (eight-bit values).

PrintableString, an arbitrary string of printable characters.

T61String, an arbitrary string of T.61 (eight-bit) characters.

UTCTime, a "coordinated universal time" or Greenwich Mean Time (GMT) value.

Simple types fall into two categories: string types and non-string types. BIT STRING, IA5String, OCTET STRING, PrintableString, T61String, and UTCTime are string types.

String types can be viewed, for the purposes of encoding, as consisting of components, where the components are substrings. This view allows one to encode a value whose length is not known in advance (e.g., an octet string value input from a file stream) with a constructed, indefinite-length encoding (see Section 3).

The string types can be given size constraints limiting the length of values.

2.2 Structured types

Structured types are those consisting of components. ASN.1 defines four, all of which are relevant to the PKCS standards:

SEQUENCE, an ordered collection of one or more types.

SEQUENCE OF, an ordered collection of zero or more occurrences of a given type.

SET, an unordered collection of one or more types.

SET OF, an unordered collection of zero or more occurrences of a given type.

The structured types can have optional components, possibly with default values.

2.3 Implicitly and explicitly tagged types

Tagging is useful to distinguish types within an application; it is also commonly used to distinguish component types within a structured type. For instance, optional components of a SET or SEQUENCE type are typically given distinct context-specific tags to avoid ambiguity.

There are two ways to tag a type: implicitly and explicitly.

Implicitly tagged types are derived from other types by changing the tag of the underlying type. Implicit tagging is denoted by the ASN.1 keywords [***class number***] IMPLICIT (see Section 5.1).

Explicitly tagged types are derived from other types by adding an outer tag to the underlying type. In effect, explicitly tagged types are structured types consisting of one component, the underlying type. Explicit tagging is denoted by the ASN.1 keywords [***class number***] EXPLICIT (see Section 5.2).

The keyword [***class number***] alone is the same as explicit tagging, except when the "module" in which the ASN.1 type is defined has implicit tagging by default. ("Modules" are among the advanced features not described in this note.)

For purposes of encoding, an implicitly tagged type is considered the same as the underlying type, except that the tag is different. An explicitly tagged type is considered like a structured type with one component, the underlying type. Implicit tags result in shorter encodings, but explicit tags may be necessary to avoid ambiguity if the tag of the underlying type is indeterminate (e.g., the underlying type is CHOICE or ANY).

2.4 Other types

Other types in ASN.1 include the CHOICE and ANY types. The CHOICE type denotes a union of one or more alternatives; the ANY type denotes an arbitrary value of an arbitrary type, where the arbitrary type is possibly defined in the registration of an object identifier or integer value.

3. Basic Encoding Rules

The Basic Encoding Rules for ASN.1, abbreviated BER, give one or more ways to represent any ASN.1 value as an octet string. (There are certainly other ways to represent ASN.1 values, but BER is the standard for interchanging such values in OSI.)

There are three methods to encode an ASN.1 value under BER, the choice of which depends on the type of value and whether the length of the value is known. The three methods are primitive, definite-length encoding; constructed, definite-length encoding; and constructed, indefinite-length encoding. Simple non-string types employ the primitive, definite-length method; structured types employ either of the constructed methods; and simple string types employ any of the methods, depending on whether the length of the value is known. Types derived by implicit tagging employ the method of the underlying type and types derived by explicit tagging employ the constructed methods.

In each method, the BER encoding has three or four parts:

Identifier octets. These identify the class and tag number of the ASN.1 value, and indicate whether the method is primitive or constructed.

Length octets. For the definite-length methods, these give the number of contents octets. For the constructed, indefinite-length method, these indicate that the length is indefinite.

Contents octets. For the primitive, definite-length method, these give a concrete representation of the value. For the constructed methods, these give the concatenation of the BER encodings of the components of the value.

End-of-contents octets. For the constructed, indefinite-length method, these denote the end of the contents. For the other methods, these are absent.

The three methods of encoding are described in the following sections.

3.1 Primitive, definite-length method

This method applies to simple types and types derived from simple types by implicit tagging. It requires that the length of the value be known in advance. The parts of the BER encoding are as follows:

Identifier octets. There are two forms: low tag number (for tag numbers between 0 and 30) and high tag number (for tag numbers 31 and greater).

Low-tag-number form. One octet. Bits 8 and 7 specify the class (see Table 2), bit 6 has value "0," indicating that the encoding is primitive, and bits 5—1 give the tag number.

Class	Bit 8	Bit 7
universal	0	0
application	0	1
context-specific	1	0
private	1	1

Table 2. Class encoding in identifier octets.

High-tag-number form. Two or more octets. First octet is as in low-tag-number form, except that bits 5–1 all have value "1." Second and following octets give the tag number, base 128, most significant digit first, with as few digits as possible, and with the bit 8 of each octet except the last set to "1."

Length octets. There are two forms: short (for lengths between 0 and 127), and long definite (for lengths between 0 and 2^{1008}–1).

Short form. One octet. Bit 8 has value "0" and bits 7–1 give the length.

Long form. Two to 127 octets. Bit 8 of first octet has value "1" and bits 7–1 give the number of additional length octets. Second and following octets give the length, base 256, most significant digit first.

Contents octets. These give a concrete representation of the value (or the value of the underlying type, if the type is derived by implicit tagging). Details for particular types are given in Section 5.

3.2 Constructed, definite-length method

This method applies to simple string types, structured types, types derived simple string types and structured types by implicit tagging, and types derived from anything by explicit tagging. It requires that the length of the value be known in advance. The parts of the BER encoding are as follows:

Identifier octets. As described in Section 3.1, except that bit 6 has value "1," indicating that the encoding is constructed.

Length octets. As described in Section 3.1.

Contents octets. The concatenation of the BER encodings of the components of the value:

- For simple string types and types derived from them by implicit tagging, the concatenation of the BER encodings of consecutive substrings of the value (underlying value for implicit tagging).

- For structured types and types derived from them by implicit tagging, the concatenation of the BER encodings of components of the value (underlying value for implicit tagging).
- For types derived from anything by explicit tagging, the BER encoding of the underlying value.

Details for particular types are given in Section 5.

3.3 Constructed, indefinite-length method

This method applies to simple string types, structured types, types derived simple string types and structured types by implicit tagging, and types derived from anything by explicit tagging. It does not require that the length of the value be known in advance. The parts of the BER encoding are as follows:

Identifier octets. As described in Section 3.2.

Length octets. One octet, `80`.

Contents octets. As described in Section 3.2.

End-of-contents octets. Two octets, `00 00`.

Since the end-of-contents octets appear where an ordinary BER encoding might be expected (e.g., in the contents octets of a sequence value), the 00 and 00 appear as identifier and length octets, respectively. Thus the end-of-contents octets is really the primitive, definite-length encoding of a value with universal class, tag number 0, and length 0.

4. Distinguished Encoding Rules

The Distinguished Encoding Rules for ASN.1, abbreviated DER, are a subset of BER, and give exactly one way to represent any ASN.1 value as an octet string. DER is intended for applications in which a unique octet string encoding is needed, as is the case when a digital signature is computed on an ASN.1 value. DER is defined in Section 8.7 of X.509.

DER adds the following restrictions to the rules given in Section 3:

1. When the length is between 0 and 127, the short form of length must be used

2. When the length is 128 or greater, the long form of length must be used, and the length must be encoded in the minimum number of octets.
3. For simple string types and implicitly tagged types derived from simple string types, the primitive, definite-length method must be employed.
4. For structured types, implicitly tagged types derived from structured types, and explicitly tagged types derived from anything, the constructed, definite-length method must be employed.

Other restrictions are defined for particular types (such as `BIT STRING`, `SEQUENCE`, `SET`, and `SET OF`), and can be found in Section 5.

5. Notation and encodings for some types

This section gives the notation for some ASN.1 types and describes how to encode values of those types under both BER and DER.

The types described are those presented in Section 2. They are listed alphabetically here.

Each description includes ASN.1 notation, BER encoding, and DER encoding. The focus of the encodings is primarily on the contents octets; the tag and length octets follow Sections 3 and 4. The descriptions also explain where each type is used in PKCS and related standards. ASN.1 notation is generally only for types, although for the type `OBJECT IDENTIFIER`, value notation is given as well.

5.1 Implicitly tagged types

An implicitly tagged type is a type derived from another type by changing the tag of the underlying type.

Implicit tagging is used for optional `SEQUENCE` components with underlying type other than `ANY` throughout PKCS, and for the `extendedCertificate` alternative of PKCS #7's `ExtendedCertificateOrCertificate` type.

ASN.1 notation:

[**[*class*]** ***number***] `IMPLICIT` ***Type***

class = `UNIVERSAL` | `APPLICATION` | `PRIVATE`

where ***Type*** is a type, ***class*** is an optional class name, and ***number*** is the tag number within the class, a nonnegative integer.

In ASN.1 "modules" whose default tagging method is implicit tagging, the notation [**[*class*] *number***] ***Type*** is also acceptable, and the keyword `IMPLICIT` is implied. (See Section 2.3.) For definitions stated outside a module, the explicit inclusion of the keyword `IMPLICIT` is preferable to prevent ambiguity.

If the class name is absent, then the tag is context-specific. Context-specific tags can only appear in a component of a structured or `CHOICE` type.

Example: PKCS #8's `PrivateKeyInfo` type has an optional `attributes` component with an implicit, context-specific tag:

```
PrivateKeyInfo ::= SEQUENCE {
  version Version,
  privateKeyAlgorithm PrivateKeyAlgorithmIdentifier,
  privateKey PrivateKey,
  attributes [0] IMPLICIT Attributes OPTIONAL }
```

Here the underlying type is `Attributes`, the class is absent (i.e., context-specific), and the tag number within the class is 0.

BER encoding. Primitive or constructed, depending on the underlying type. Contents octets are as for the BER encoding of the underlying value.

Example: The BER encoding of the `attributes` component of a `PrivateKeyInfo` value is as follows:

- the identifier octets are `80` if the underlying `Attributes` value has a primitive BER encoding and `a0` if the underlying `Attributes` value has a constructed BER encoding
- the length and contents octets are the same as the length and contents octets of the BER encoding of the underlying `Attributes` value

DER encoding. Primitive or constructed, depending on the underlying type. Contents octets are as for the DER encoding of the underlying value.

5.2 Explicitly tagged types

Explicit tagging denotes a type derived from another type by adding an outer tag to the underlying type.

Explicit tagging is used for optional `SEQUENCE` components with underlying type `ANY` throughout PKCS, and for the `version` component of X.509's `Certificate` type.

ASN.1 notation:

[**[*class*] *number***] `EXPLICIT` ***Type***

class **=** `UNIVERSAL` **|** `APPLICATION` **|** `PRIVATE`

where ***Type*** is a type, ***class*** is an optional class name, and ***number*** is the tag number within the class, a nonnegative integer.

If the class name is absent, then the tag is context-specific. Context-specific tags can only appear in a component of a `SEQUENCE`, `SET` or `CHOICE` type.

In ASN.1 "modules" whose default tagging method is explicit tagging, the notation [**[*class*] *number***] ***Type*** is also acceptable, and the keyword `EXPLICIT` is implied. (See Section 2.3.) For definitions stated outside a module, the explicit inclusion of the keyword `EXPLICIT` is preferable to prevent ambiguity.

Example 1: PKCS #7's `ContentInfo` type has an optional `content` component with an explicit, context-specific tag:

```
ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content
    [0] EXPLICIT ANY DEFINED BY contentType OPTIONAL }
```

Here the underlying type is `ANY DEFINED BY contentType`, the class is absent (i.e., context-specific), and the tag number within the class is 0.

Example 2: X.509's `Certificate` type has a `version` component with an explicit, context-specific tag, where the `EXPLICIT` keyword is omitted:

```
Certificate ::= ...
  version [0] Version DEFAULT v1988,
...
```

The tag is explicit because the default tagging method for the ASN.1 "module" in X.509 that defines the `Certificate` type is explicit tagging.

BER encoding. Constructed. Contents octets are the BER encoding of the underlying value.

Example: the BER encoding of the `content` component of a `ContentInfo` value is as follows:

- identifier octets are `a0`
- length octets represent the length of the BER encoding of the underlying `ANY DEFINED BY contentType` value
- contents octets are the BER encoding of the underlying `ANY DEFINED BY contentType` value

DER encoding. Constructed. Contents octets are the DER encoding of the underlying value.

5.3 ANY

The `ANY` type denotes an arbitrary value of an arbitrary type, where the arbitrary type is possibly defined in the registration of an object identifier or associated with an integer index.

The `ANY` type is used for content of a particular content type in PKCS #7's `ContentInfo` type, for parameters of a particular algorithm in X.509's `AlgorithmIdentifier` type, and for attribute values in X.501's `Attribute` and `AttributeValueAssertion` types. The `Attribute` type is used by PKCS #6, #7, #8, #9 and #10, and the `AttributeValueAssertion` type is used in X.501 distinguished names.

ASN.1 notation:

`ANY [DEFINED BY` ***identifier***`]`

where ***identifier*** is an optional identifier.

In the `ANY` form, the actual type is indeterminate.

The `ANY DEFINED BY` ***identifier*** form can only appear in a component of a `SEQUENCE` or `SET` type for which ***identifier*** identifies some other component, and that other component has type `INTEGER` or `OBJECT IDENTIFIER` (or a type derived from either of those by tagging). In that form, the actual type is determined by the value of the other component, either in the registration of the object identifier value, or in a table of integer values.

Example: X.509's `AlgorithmIdentifier` type has a component of type `ANY`:

```
AlgorithmIdentifier ::= SEQUENCE {
  algorithm OBJECT IDENTIFIER,
  parameters ANY DEFINED BY algorithm OPTIONAL }
```


Here the actual type of the `parameter` component depends on the value of the `algorithm` component. The actual type would be defined in the registration of object identifier values for the `algorithm` component.

BER encoding. Same as the BER encoding of the actual value.

Example: The BER encoding of the value of the `parameter` component is the BER encoding of the value of the actual type as defined in the registration of object identifier values for the `algorithm` component.

DER encoding. Same as the DER encoding of the actual value.

5.4 BIT STRING

The `BIT STRING` type denotes an arbitrary string of bits (ones and zeroes). A `BIT STRING` value can have any length, including zero. This type is a string type.

The `BIT STRING` type is used for digital signatures on extended certificates in PKCS #6's `ExtendedCertificate` type, for digital signatures on certificates in X.509's `Certificate` type, and for public keys in certificates in X.509's `SubjectPublicKeyInfo` type.

ASN.1 notation:

```
BIT STRING
```

Example: X.509's `SubjectPublicKeyInfo` type has a component of type `BIT STRING`:

```
SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  publicKey BIT STRING }
```

BER encoding. Primitive or constructed. In a primitive encoding, the first contents octet gives the number of bits by which the length of the bit string is less than the next multiple of eight (this is called the "number of unused bits"). The second and following contents octets give the value of the bit string, converted to an octet string. The conversion process is as follows:

1. The bit string is padded after the last bit with zero to seven bits of any value to make the length of the bit string a multiple of eight. If the length of the bit string is a multiple of eight already, no padding is done.

2. The padded bit string is divided into octets. The first eight bits of the padded bit string become the first octet, bit 8 to bit 1, and so on through the last eight bits of the padded bit string.

In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the bit string, where each substring except the last has a length that is a multiple of eight bits.

Example: The BER encoding of the `BIT STRING` value "011011100101110111" can be any of the following, among others, depending on the choice of padding bits, the form of length octets, and whether the encoding is primitive or constructed:

```
03 04 06 6e 5d c0                        DER encoding

03 04 06 6e 5d e0                        padded with "100000"

03 81 04 06 6e 5d c0                     long form of length octets

23 09                    constructed encoding: "0110111001011101" + "11"
   03 03 00 6e 5d
   03 02 06 c0
```

DER encoding. Primitive. The contents octects are as for a primitive BER encoding, except that the bit string is padded with zero-valued bits.

Example: The DER encoding of the `BIT STRING` value "011011100101110111" is

```
03 04 06 6e 5d c0
```

5.5 CHOICE

The `CHOICE` type denotes a union of one or more alternatives.

The `CHOICE` type is used to represent the union of an extended certificate and an X.509 certificate in PKCS #7's `ExtendedCertificateOrCertificate` type.

ASN.1 notation:

```
CHOICE {
  [identifier1] Type1,
  ...,
  [identifiern] Typen }
```

where ***identifier*$_1$** , ..., ***identifier*$_n$** are optional, distinct identifiers for the alternatives, and ***Type*$_1$**, ..., ***Type*$_n$** are the types of the alternatives. The identifiers are primarily for documentation; they do not affect values of the type or their encodings in any way.

The types must have distinct tags. This requirement is typically satisfied with explicit or implicit tagging on some of the alternatives.

Example: PKCS #7's `ExtendedCertificateOrCertificate` type is a `CHOICE` type:

```
ExtendedCertificateOrCertificate ::= CHOICE {
  certificate Certificate, -- X.509
  extendedCertificate [0] IMPLICIT ExtendedCertificate
}
```

Here the identifiers for the alternatives are `certificate` and `extendedCertificate`, and the types of the alternatives are `Certificate` and `[0] IMPLICIT ExtendedCertificate`.

BER encoding. Same as the BER encoding of the chosen alternative. The fact that the alternatives have distinct tags makes it possible to distinguish between their BER encodings.

Example: The identifier octets for the BER encoding are `30` if the chosen alternative is `certificate`, and `a0` if the chosen alternative is `extendedCertificate`.

DER encoding. Same as the DER encoding of the chosen alternative.

5.6 IA5String

The `IA5String` type denotes an arbtrary string of IA5 characters. IA5 stands for International Alphabet 5, which is the same as ASCII. The character set includes non-printing control characters. An `IA5String` value can have any length, including zero. This type is a string type.

The `IA5String` type is used in PKCS #9's electronic-mail address, unstructured-name, and unstructured-address attributes.

ASN.1 notation:

```
IA5String
```

BER encoding. Primitive or constructed. In a primitive encoding, the contents octets give the characters in the IA5 string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the IA5 string.

Example: The BER encoding of the `IA5String` value "test1@rsa.com" can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
16 0d 74 65 73 74 31 40 72 73 61 2e 63 6f 6d       DER encoding

16 81 0d                                          long form of length octets
   74 65 73 74 31 40 72 73 61 2e 63 6f 6d

36 13                          constructed encoding: "test1" + "@" + "rsa.com"
   16 05 74 65 73 74 31
   16 01 40
   16 07 72 73 61 2e 63 6f 6d
```

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

Example: The DER encoding of the `IA5String` value "test1@rsa.com" is

```
16 0d 74 65 73 74 31 40 72 73 61 2e 63 6f 6d
```


5.7 INTEGER

The `INTEGER` type denotes an arbitrary integer. `INTEGER` values can be positive, negative, or zero, and can have any magnitude.

The `INTEGER` type is used for version numbers throughout PKCS, cryptographic values such as modulus, exponent, and primes in PKCS #1's `RSAPublicKey` and `RSAPrivateKey` types and PKCS #3's `DHParameter` type, a message-digest iteration count in PKCS #5's `PBEParameter` type, and version numbers and serial numbers in X.509's `Certificate` type.

ASN.1 notation:

`INTEGER` **[**{ ***identifier*$_1$**(***value*$_1$**)** ... ***identifier*$_n$**(***value*$_n$**)** }**]**

where ***identifier*$_1$**, ..., ***identifier*$_n$** are optional distinct identifiers and ***value*$_1$**, ..., ***value*$_n$** are optional integer values. The identifiers, when present, are associated with values of the type.

Example: X.509's `Version` type is an `INTEGER` type with identified values:

```
Version ::= INTEGER { v1988(0) }
```

The identifier `v1988` is associated with the value 0. X.509's `Certificate` type uses the identifier `v1988` to give a default value of 0 for the `version` component:


```
Certificate ::= ...
  version Version DEFAULT v1988,
...
```

BER encoding. Primitive. Contents octets give the value of the integer, base 256, in two's complement form, most significant digit first, with the minimum number of octets. The value 0 is encoded as a single `00` octet.

Some example BER encodings (which also happen to be DER encodings) are given in Table 3.

Integer value	BER encoding
0	`02 01 00`
127	`02 01 7F`
128	`02 02 00 80`
256	`02 02 01 00`
-128	`02 01 80`
-129	`02 02 FF 7F`

Table 3. Example BER encodings of `INTEGER` values.

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

5.8 NULL

The `NULL` type denotes a null value.

The `NULL` type is used for algorithm parameters in several places in PKCS.

ASN.1 notation:

`NULL`

BER encoding. Primitive. Contents octets are empty.

Example: The BER encoding of a `NULL` value can be either of the following, as well as others, depending on the form of the length octets:

```
05 00

05 81 00
```

DER encoding. Primitive. Contents octets are empty; the DER encoding of a `NULL` value is always `05 00`.

5.9 OBJECT IDENTIFIER

The `OBJECT IDENTIFIER` type denotes an object identifier, a sequence of integer components that identifies an object such as an algorithm, an attribute type, or perhaps a registration authority that defines other object identifiers. An `OBJECT IDENTIFIER` value can have any number of components, and components can generally have any nonnegative value. This type is a non-string type.

`OBJECT IDENTIFIER` values are given meanings by registration authorities. Each registration authority is responsible for all sequences of components beginning with a given sequence. A registration authority typically delegates responsibility for subsets of the sequences in its domain to other registration authorities, or for particular types of object. There are always at least two components.

The `OBJECT IDENTIFIER` type is used to identify content in PKCS #7's `ContentInfo` type, to identify algorithms in X.509's `AlgorithmIdentifier` type, and to identify attributes in X.501's `Attribute` and `AttributeValueAssertion` types. The `Attribute` type is used by PKCS #6, #7, #8, #9, and #10, and the `AttributeValueAssertion` type is used in X.501 distinguished names. `OBJECT IDENTIFIER` values are defined throughout PKCS.

ASN.1 notation:

```
OBJECT IDENTIFIER
```

The ASN.1 notation for values of the `OBJECT IDENTIFIER` type is

{ [***identifier***] ***component*$_1$** ... ***component*$_n$** }

component*$_i$** = ***identifier*$_i$** | ***identifier*$_i$** (value*$_i$**) | ***value*$_i$**

where ***identifier***, ***identifier*$_1$**, ..., ***identifier*$_n$** are identifiers, and ***value*$_1$**, ..., ***value*$_n$** are optional integer values.

The form without ***identifier*** is the "complete" value with all its components; the form with ***identifier*** abbreviates the beginning components with another object identifier value. The identifiers ***identifier*$_1$**, ..., ***identifier*$_n$** are intended primarily for documentation, but they must correspond to the integer value when both are present. These identifiers can appear without integer values only if they are among a small set of identifiers defined in X.208.

Example: The following values both refer to the object identifier assigned to RSA Data Security, Inc.:


```
{ iso(1) member-body(2) 840 113549 }
{ 1 2 840 113549 }
```

(In this example, which gives ASN.1 value notation, the object identifier values are decimal, not hexadecimal.) Table 4 gives some other object identifier values and their meanings.

Object identifier value	Meaning
{ 1 2 }	ISO member bodies
{ 1 2 840 }	US (ANSI)
{ 1 2 840 113549 }	RSA Data Security, Inc.
{ 1 2 840 113549 1 }	RSA Data Security, Inc. PKCS
{ 2 5 }	directory services (X.500)
{ 2 5 8 }	directory services—algorithms

Table 4. Some object identifier values and their meanings.

BER encoding. Primitive. Contents octets are as follows, where $value_1$, ..., $value_n$ denote the integer values of the components in the complete object identifier:

1. The first octet has value $40 \times value_1 + value_2$. (This is unambiguous, since $value_1$ is limited to values 0, 1, and 2; $value_2$ is limited to the range 0 to 39 when $value_1$ is 0 or 1; and, according to X.208, n is always at least 2.)

2. The following octets, if any, encode $value_3$, ..., $value_n$. Each value is encoded base 128, most significant digit first, with as few digits as possible, and the most significant bit of each octet except the last in the value's encoding set to "1."

Example: The first octet of the BER encoding of RSA Data Security, Inc.'s object identifier is $40 \times 1 + 2 = 42 = 2a_{16}$. The encoding of $840 = 6 \times 128 + 48_{16}$ is `86 48` and the encoding of $113549 = 6 \times 128^2 + 77_{16} \times 128 + d_{16}$ is `86 f7 0d`. This leads to the following BER encoding:

```
06 06 2a 86 48 86 f7 0d
```

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

5.10 OCTET STRING

The `OCTET STRING` type denotes an arbitrary string of octets (eight-bit values). An `OCTET STRING` value can have any length, including zero. This type is a string type.

The OCTET STRING type is used for salt values in PKCS #5's PBEParameter type, for message digests, encrypted message digests, and encrypted content in PKCS #7, and for private keys and encrypted private keys in PKCS #8.

ASN.1 notation:

OCTET STRING **[**SIZE (**{*size* | *size*$_1$ . . *size*$_2$}**)]**

where ***size***, ***size***$_1$, and ***size***$_2$ are optional size constraints. In the OCTET STRING SIZE (***size***) form, the octet string must have ***size*** octets. In the OCTET STRING SIZE (***size***$_1$ . . ***size***$_2$) form, the octet string must have between size1 and size2 octets. In the OCTET STRING form, the octet string can have any size.

Example: PKCS #5's PBEParameter type has a component of type OCTET STRING:

```
PBEParameter ::= SEQUENCE {
  salt OCTET STRING SIZE(8),
  iterationCount INTEGER }
```

Here the size of the salt component is always eight octets.

BER encoding. Primitive or constructed. In a primitive encoding, the contents octets give the value of the octet string, first octet to last octet. In a constructed encoding, the contents octets give the concatenation of the BER encodings of substrings of the OCTET STRING value.

Example: The BER encoding of the OCTET STRING value 01 23 45 67 89 ab cd ef can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
04 08 01 23 45 67 89 ab cd ef                    DER encoding

04 81 08 01 23 45 67 89 ab cd ef          long form of length octets

24 0c                    constructed encoding: 01 ... 67 + 89 ... ef
   04 04 01 23 45 67
   04 04 89 ab cd ef
```

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

Example: The BER encoding of the OCTET STRING value 01 23 45 67 89 ab cd ef is

```
04 08 01 23 45 67 89 ab cd ef
```


5.11 PrintableString

The PrintableString type denotes an arbitrary string of printable characters from the following character set:

```
A, B, ..., Z
a, b, ..., z
0, 1, ..., 9
(space) ' ( ) + , - . / : = ?
```

This type is a string type.

The PrintableString type is used in PKCS #9's challenge-password and unstructuerd-address attributes, and in several X.521 distinguished names attributes.

ASN.1 notation:

```
PrintableString
```

BER encoding. Primitive or constructed. In a primitive encoding, the contents octets give the characters in the printable string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the string.

Example: The BER encoding of the PrintableString value "Test User 1" can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
13 0b 54 65 73 74 20 55 73 65 72 20 31                DER encoding

13 81 0b 54 65 73 74 20 55 73 65 72 20 31
                                              long form of length octets

33 0f                              constructed encoding: "Test " + "User 1"
   13 05 54 65 73 74 20
   13 06 55 73 65 72 20 31
```

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

Example: The DER encoding of the PrintableString value "Test User 1" is

```
13 0b 54 65 73 74 20 55 73 65 72 20 31
```

5.12 SEQUENCE

The SEQUENCE type denotes an ordered collection of one or more types.

The SEQUENCE type is used throughout PKCS and related standards.

ASN.1 notation:

```
SEQUENCE {
  [identifier1] Type1 [{OPTIONAL | DEFAULT value1}],
  ...,
  [identifiern] Typen [{OPTIONAL | DEFAULT valuen}]}
```

where ***identifier*$_1$** , ..., ***identifier*$_n$** are optional, distinct identifiers for the components, ***Type*$_1$**, ..., ***Type*$_n$** are the types of the components, and ***value*$_1$**, ..., ***value*$_n$** are optional default values for the components. The identifiers are primarily for documentation; they do not affect values of the type or their encodings in any way.

The OPTIONAL qualifier indicates that the value of a component is optional and need not be present in the sequence. The DEFAULT qualifier also indicates that the value of a component is optional, and assigns a default value to the component when the component is absent.

The types of any consecutive series of components with the OPTIONAL or DEFAULT qualifier, as well as of any component immediately following that series, must have distinct tags. This requirement is typically satisfied with explicit or implicit tagging on some of the components.

Example: X.509's Validity type is a SEQUENCE type with two components:

```
Validity ::= SEQUENCE {
  start UTCTime,
  end UTCTime }
```

Here the identifiers for the components are start and end, and the types of the components are both UTCTime.

BER encoding. Constructed. Contents octets are the concatenation of the BER encodings of the values of the components of the sequence, in order of definition, with the following rules for components with the OPTIONAL and DEFAULT qualifiers:

- if the value of a component with the OPTIONAL or DEFAULT qualifier is absent from the sequence, then the encoding of that component is not included in the contents octets

- if the value of a component with the DEFAULT qualifier is the default value, then the encoding of that component may or may not be included in the contents octets

DER encoding. Constructed. Contents octets are the same as the BER encoding, except that if the value of a component with the DEFAULT qualifier is the default value, the encoding of that component is not included in the contents octets.

5.13 SEQUENCE OF

The SEQUENCE OF type denotes an ordered collection of zero or more occurrences of a given type.

The SEQUENCE OF type is used in X.501 distinguished names.

ASN.1 notation:

```
SEQUENCE OF Type
```

where ***Type*** is a type.

Example: X.501's RDNSequence type consists of zero or more occurences of the RelativeDistinguishedName type, most significant occurrence first:

```
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName
```

BER encoding. Constructed. Contents octets are the concatenation of the BER encodings of the values of the occurrences in the collection, in order of occurence.

DER encoding. Constructed. Contents octets are the concatenation of the DER encodings of the values of the occurrences in the collection, in order of occurence.

5.14 SET

The SET type denotes an unordered collection of one or more types.

The SET type is not used in PKCS.

ASN.1 notation:

```
SET {
  [identifier1] Type1 [{OPTIONAL | DEFAULT value1}],
```

…,
[*identifier*$_n$] *Type*$_n$ [{OPTIONAL | DEFAULT *value*$_n$}]}

where ***identifier*$_1$**, …, ***identifier*$_n$** are optional, distinct identifiers for the components, ***Type*$_1$**, …, ***Type*$_n$** are the types of the components, and ***value*$_1$**, …, ***value*$_n$** are optional default values for the components. The identifiers are primarily for documentation; they do not affect values of the type or their encodings in any way.

The OPTIONAL qualifier indicates that the value of a component is optional and need not be present in the set. The DEFAULT qualifier also indicates that the value of a component is optional, and assigns a default value to the component when the component is absent.

The types must have distinct tags. This requirement is typically satisfied with explicit or implicit tagging on some of the components.

BER encoding. Constructed. Contents octets are the concatenation of the BER encodings of the values of the components of the set, in any order, with the following rules for components with the OPTIONAL and DEFAULT qualifiers:

- if the value of a component with the OPTIONAL or DEFAULT qualifier is absent from the set, then the encoding of that component is not included in the contents octets
- if the value of a component with the DEFAULT qualifier is the default value, then the encoding of that component may or may not be included in the contents octets

DER encoding. Constructed. Contents octets are the same as for the BER encoding, except that:

1. If the value of a component with the DEFAULT qualifier is the default value, the encoding of that component is not included.
2. There is an order to the components, namely ascending order by tag.

5.15 SET OF

The SET OF type denotes an unordered collection of zero or more occurrences of a given type.

The SET OF type is used for sets of attributes in PKCS #6, #7, #8, #9 and #10, for sets of message-digest algorithm identifiers, signer information, and recipient information in PKCS #7, and in X.501 distinguished names.

ASN.1 notation:

SET OF ***Type***

where ***Type*** is a type.

Example: X.501's RelativeDistinguishedName type consists of zero or more occurrences of the AttributeValueAssertion type, where the order is unimportant:

```
RelativeDistinguishedName ::=
  SET OF AttributeValueAssertion
```

BER encoding. Constructed. Contents octets are the concatenation of the BER encodings of the values of the occurrences in the collection, in any order.

DER encoding. Constructed. Contents octets are the same as for the BER encoding, except that there is an order, namely ascending lexicographic order of BER encoding. Lexicographic comparison of two different BER encodings is done as follows: Logically pad the shorter BER encoding after the last octet with dummy octets that are smaller in value than any normal octet. Scan the BER encodings from left to right until a difference is found. The smaller-valued BER encoding is the one with the smaller-valued octet at the point of difference.

5.16 T61String

The T61String type denotes an arbtrary string of T.61 characters. T.61 is an eight-bit extension to the ASCII character set. Special "escape" sequences specify the interpretation of subsequent character values as, for example, Japanese; the initial interpretation is Latin. The character set includes non-printing control characters. The T61String type allows only the Latin and Japanese character interepretations, and implementors' agreements for directory names exclude control characters [NIST92]. A T61String value can have any length, including zero. This type is a string type.

The T61String type is used in PKCS #9's unstructured-address and challenge-password attributes, and in several X.521 attributes.

ASN.1 notation:

T61String

BER encoding. Primitive or constructed. In a primitive encoding, the contents octets give the characters in the T.61 string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the T.61 string.

Example: The BER encoding of the `T61String` value "clés publiques" (French for "public keys") can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
14 0f                                                  DER encoding
   63 6c c2 65 73 20 70 75 62 6c 69 71 75 65 73

14 81 0f                                       long form of length octets
   63 6c c2 65 73 20 70 75 62 6c 69 71 75 65 73

34 15                          constructed encoding: "clés" + " " + "publiques"
   14 05 63 6c c2 65 73
   14 01 20
   14 09 70 75 62 6c 69 71 75 65 73
```

The eight-bit character `c2` is a T.61 prefix that adds an acute accent (´) to the next character.

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

Example: The DER encoding of the `T61String` value "clés publiques" is

```
14 0f 63 6c c2 65 73 20 70 75 62 6c 69 71 75 65 73
```

5.17 UTCTime

The `UTCTime` type denotes a "coordinated universal time" or Greenwich Mean Time (GMT) value. A `UTCTime` value includes the local time precise to either minutes or seconds, and an offset from GMT in hours and minutes. It takes any of the following forms:

YYMMDDhhmm`Z`
YYMMDDhhmm`+`*hh'mm'*
YYMMDDhhmm`-`*hh'mm'*
YYMMDDhhmmss`Z`
YYMMDDhhmmss`+`*hh'mm'*
YYMMDDhhmmss`-`*hh'mm'*

where:

YY is the least significant two digits of the year

MM is the month (01 to 12)

DD is the day (01 to 31)

hh is the hour (00 to 23)

mm are the minutes (00 to 59)

ss are the seconds (00 to 59)

`Z` indicates that local time is GMT, `+` indicates that local time is later than GMT, and `-` indicates that local time is earlier than GMT

hh' is the absolute value of the offset from GMT in hours

mm' is the absolute value of the offset from GMT in minutes

This type is a string type.

The `UTCTime` type is used for signing times in PKCS #9's signing-time attribute and for certificate validity periods in X.509's `Validity` type.

ASN.1 notation:

```
UTCTime
```

BER encoding. Primitive or constructed. In a primitive encoding, the contents octets give the characters in the string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the string. (The constructed encoding is not particularly interesting, since `UTCTime` values are so short, but the constructed encoding is permitted.)

Example: The time this sentence was originally written was 4:45:40 p.m. Pacific Daylight Time on May 6, 1991, which can be represented with either of the following `UTCTime` values, among others:

"910506164540-0700"

"910506234540Z"

These values have the following BER encodings, among others:

```
17 0d 39 31 30 35 30 36 32 33 34 35 34 30 5a

17 11 39 31 30 35 30 36 31 36 34 35 34 30 2D 30 37 30
      30
```

DER encoding. Primitive. Contents octets are as for a primitive BER encoding.

6. An example

This section gives an example of ASN.1 notation and DER encoding: the X.501 type `Name`.

6.1 Abstract notation

This section gives the ASN.1 notation for the X.501 type `Name`.

```
Name ::= CHOICE {
  RDNSequence }

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::=
  SET OF AttributeValueAssertion

AttributeValueAssertion ::= SEQUENCE {
   AttributeType,
   AttributeValue }

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY
```

The `Name` type identifies an object in an X.500 directory. `Name` is a `CHOICE` type consisting of one alternative: `RDNSequence`. (Future revisions of X.500 may have other alternatives.)

The `RDNSequence` type gives a path through an X.500 directory tree starting at the root. `RDNSequence` is a `SEQUENCE OF` type consisting of zero or more occurences of `RelativeDistinguishedName`.

The `RelativeDistinguishedName` type gives a unique name to an object relative to the object superior to it in the directory tree. `RelativeDistinguishedName` is a `SET OF` type consisting of zero or more occurrences of `AttributeValueAssertion`.

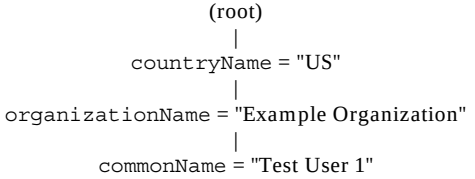
The `AttributeValueAssertion` type assigns a value to some attribute of a relative distinguished name, such as country name or common name. `AttributeValueAssertion` is a `SEQUENCE` type consisting of two components, an `AttributeType` type and an `AttributeValue` type.

The `AttributeType` type identifies an attribute by object identifier. The `AttributeValue` type gives an arbitrary attribute value. The actual type of the attribute value is determined by the attribute type.

6.2 DER encoding

This section gives an example of a DER encoding of a value of type `Name`, working from the bottom up.

The name is that of the Test User 1 from the PKCS examples [Kal93]. The name is represented by the following path:



Each level corresponds to one `RelativeDistinguishedName` value, each of which happens for this name to consist of one `AttributeValueAssertion` value. The `AttributeType` value is before the equals sign, and the `AttributeValue` value (a printable string for the given attribute types) is after the equals sign.

The `countryName`, `organizationName`, and `commonUnitName` are attribute types defined in X.520 as:

```
attributeType OBJECT IDENTIFIER ::=
  { joint-iso-ccitt(2) ds(5) 4 }

countryName OBJECT IDENTIFIER ::= { attributeType 6 }
organizationName OBJECT IDENTIFIER ::=
  { attributeType 10 }
commonUnitName OBJECT IDENTIFIER ::=
  { attributeType 3 }
```

6.2.1 AttributeType

The three `AttributeType` values are OCTET STRING values, so their DER encoding follows the primitive, definite-length method:

```
06 03 55 04 06                                    countryName

06 03 55 04 0a                                 organizationName
```

```
06 03 55 04 03                                          commonName
```

The identifier octets follow the low-tag form, since the tag is 6 for OBJECT IDENTIFIER. Bits 8 and 7 have value "0," indicating universal class, and bit 6 has value "0," indicating that the encoding is primitive. The length octets follow the short form. The contents octets are the concatenation of three octet strings derived from subidentifiers (in decimal): $40 \times 2 + 5 = 85 = 55_{16}$; 4; and 6, 10, or 3.

6.2.2 AttributeValue

The three AttributeValue values are PrintableString values, so their encodings follow the primitive, definite-length method:

```
13 02 55 53                                                  "US"

13 14                                          "Example Organization"
   45 78 61 6d 70 6c 65 20 4f 72 67 61 6e 69 7a 61
   74 69 6f 6e

13 0b                                                   "Test User 1"
   54 65 73 74 20 55 73 65 72 20 31
```

The identifier octets follow the low-tag-number form, since the tag for PrintableString, 19 (decimal), is between 0 and 30. Bits 8 and 7 have value "0" since PrintableString is in the universal class. Bit 6 has value "0" since the encoding is primitive. The length octets follow the short form, and the contents octets are the ASCII representation of the attribute value.

6.2.3 AttributeValueAssertion

The three AttributeValueAssertion values are SEQUENCE values, so their DER encodings follow the constructed, definite-length method:

```
30 09                                        countryName = "US"
   06 03 55 04 06
   13 02 55 53

30 1b                    organizationName = "Example Organizaiton"
   06 03 55 04 0a
   13 14 ... 6f 6e

30 12                                    commonName = "Test User 1"
   06 03 55 04 0b
   13 0b ... 20 31
```


The identifier octets follow the low-tag-number form, since the tag for SEQUENCE, 16 (decimal), is between 0 and 30. Bits 8 and 7 have value "0" since SEQUENCE is in the universal class. Bit 6 has value "1" since the encoding is constructed. The length octets follow the short form, and the contents octets are the concatenation of the DER encodings of the attributeType and attributeValue components.

6.2.4 RelativeDistinguishedName

The three RelativeDistinguishedName values are SET OF values, so their DER encodings follow the constructed, definite-length method:

```
31 0b
   30 09 ... 55 53

31 1d
   30 1b ... 6f 6e

31 14
   30 12 ... 20 31
```

The identifier octets follow the low-tag-number form, since the tag for SET OF, 17 (decimal), is between 0 and 30. Bits 8 and 7 have value "0" since SET OF is in the universal class Bit 6 has value "1" since the encoding is constructed. The lengths octets follow the short form, and the contents octets are the DER encodings of the respective AttributeValueAssertion values, since there is only one value in each set.

6.2.5 RDNSequence

The RDNSequence value is a SEQUENCE OF value, so its DER encoding follows the constructed, definite-length method:

```
30 42
   31 0b ... 55 53
   31 1d ... 6f 6e
   31 14 ... 20 31
```

The identifier octets follow the low-tag-number form, since the tag for SEQUENCE OF, 16 (decimal), is between 0 and 30. Bits 8 and 7 have value "0" since SEQUENCE OF is in the universal class. Bit 6 has value "1" since the encoding is constructed. The lengths octets follow the short form, and the contents octets are the concatenation of the DER encodings of the three RelativeDistinguishedName values, in order of occurrence.

6.2.6 Name

The `Name` value is a `CHOICE` value, so its DER encoding is the same as that of the `RDNSequence` value:

```
30 42
   31 0b
      30 09
         06 03 55 04 06                      attributeType = countryName
         13 02 55 53                               attributeValue = "US"
   31 1d
      30 1b
         06 03 55 04 0a                attributeType = organizationName
         13 14                        attributeValue = "Example Organization"
            45 78 61 6d 70 6c 65 20 4f 72 67 61 6e 69 7a 61
            74 69 6f 6e
   31 14
      30 12
         06 03 55 04 03                       attributeType = commonName
         13 0b                               attributeValue = "Test User 1"
            54 65 73 74 20 55 73 65 72 20 31
```

References

PKCS #1 RSA Laboratories. *PKCS #1: RSA Encryption Standard.* Version 1.5, November 1993.

PKCS #3 RSA Laboratories. *PKCS #3: Diffie-Hellman Key-Agreement Standard.* Version 1.4, November 1993.

PKCS #5 RSA Laboratories. *PKCS #5: Password-Based Encryption Standard.* Version 1.5, November 1993.

PKCS #6 RSA Laboratories. *PKCS #6: Extended-Certificate Syntax Standard.* Version 1.5, November 1993.

PKCS #7 RSA Laboratories. *PKCS #7: Cryptographic Message Syntax Standard.* Version 1.5, November 1993.

PKCS #8 RSA Laboratories. *PKCS #8: Private-Key Information Syntax Standard.* Version 1.2, November 1993.

PKCS #9 RSA Laboratories. *PKCS #9: Selected Attribute Types.* Version 1.1, November 1993.

PKCS #10 RSA Laboratories. *PKCS #10: Certification Request Syntax Standard.* Version 1.0, November 1993.

X.200 CCITT. *Recommendation X.200: Reference Model of Open Systems Interconnection for CCITT Applications.* 1984.

X.208 CCITT. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1).* 1988.

X.209 CCITT. *Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* 1988.

X.500 CCITT. *Recommendation X.500: The Directory—Overview of Concepts, Models and Services.* 1988.

X.501 CCITT. *Recommendation X.501: The Directory—Models.* 1988.

X.509 CCITT. *Recommendation X.509: The Directory—Authentication Framework.* 1988.

X.520 CCITT. *Recommendation X.520: The Directory—Selected Attribute Types.* 1988.

[Kal93] Burton S. Kaliski Jr. *Some Examples of the PKCS Standards.* RSA Laboratories, November 1993.

[NIST92] NIST. *Special Publication 500-202: Stable Implementation Agreements for Open Systems Interconnection Protocols.* Part 11 (Directory Services Protocols). December 1992.

Revision history

June 3, 1991 version

The June 3, 1991 version is part of the initial public release of PKCS. It was published as NIST / OSI Implementors' Workshop document SEC-SIG-91-17.

November 1, 1993 version

The November 1, 1993 version incorporates several editorial changes, including the addition of a revision history. It is updated to be consistent with the following versions of the PKCS documents:

PKCS #1: RSA Encryption Standard. Version 1.5, November 1993.

PKCS #3: Diffie-Hellman Key-Agreement Standard. Version 1.4, November 1993.

PKCS #5: Password-Based Encryption Standard. Version 1.5, November 1993.

PKCS #6: Extended-Certificate Syntax Standard. Version 1.5, November 1993.

PKCS #7: Cryptographic Message Syntax Standard. Version 1.5, November 1993.

PKCS #8: Private-Key Information Syntax Standard. Version 1.2, November 1993.

PKCS #9: Selected Attribute Types. Version 1.1, November 1993.

PKCS #10: Certification Request Syntax Standard. Version 1.0, November 1993.

The following substantive changes were made:

Section 5: Description of `T61String` type is added.

Section 6: Names are changed, consistent with other PKCS examples.

Author's address

Burton S. Kaliski Jr., Ph.D.
Chief Scientist

RSA Laboratories (415) 595-7703
100 Marine Parkway (415) 595-4126 (fax)
Redwood City, CA 94065 USA burt@rsa.com

Övningsuppgifter på ASN.1 och BER

Övningsuppgift 1

Du skall definiera ett protokoll för kommunikation mellan en automatisk våg och en förpackningsmaskin.

Vågen avläser varans vikt i gram som ett flytande tal, och varans typnummer som ett heltal.

Definiera en datatyp **ScaleReading** med vilken vågen kan rapportera detta till förpackningsmaskinen.

Övningsuppgift 2

Som ett alternativ till metersystemet använder en del länder ett måttsystem som baseras på inches, feet och yards. Definera en datatyp **Measurement** som ger ett värde i detta måttsystem, och en datatyp **Box** som ger höjd, längd och bredd hos ett objekt i det nya måttsystemet. feet och yards är heltal, inches ett decimalbråk.

Övningsuppgift 3

Ändra definitionen av **Measurement** i övningsuppgift 2 så att feet bara kan ha värdena 0, 1 eller 2 (3 feet är ju en yard), och så att inches är ett heltal mellan 0 och 1199 som anger längden i hundradels tum. (1200 motsvarar ju 12 tum eller en fot).

Övningsuppgift 4

I en opinionsundersökning som görs utanför vallokalen registrerar man för varje intervjuad person vilket parti denne röstade på. Tillåtna värden är v, s, mp, c, fp, m, kds, nd eller ö. Vidare registreras ålder som ett positivt heltal större än eller lika med 18 och kön som kan vara man eller kvinna. Definiera en datatyp för att överföra dessa uppgifter från intervjustationen till den värddator som bearbetar värdet.

Övningsuppgift 5

I den lokala valundersökningen i Stockholm vill man separat registrera även rösterna för Stockholmspartiet (sp) och Bilistpartiet (bp). Ange, utgående från att lösningsförslag 1 till övningsuppgift 4 är given, en modifierad datatype **Stockholmvoter** där dessa nya partier ingår.

Övningsuppgift 6

I det militära har man tre sekretessgrader: Kvalificerat hemligt, hemligt och Öppet. Föreslå en lämplig datatyp för att ange sekretessgraden för ett dokument som överförs elektroniskt.

Övningsuppgift 7

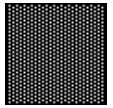
Givet lösningen till övningsuppgift 6, antag att man inom försvarsstaben vill definiera en ny sekretessgrad Extra Kvalificerat Hemligt. Kan man definiera detta i en utvidgad version av ett protokoll definierat som i lösningen av övningsuppgift 6.

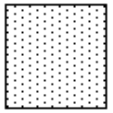
Övningsuppgift 8

Antag att man vill definiera olika mönster för att täcka en yta på en monokrom bildskärm. Varje punkt på skärmen kan vara antingen svart eller vit. Mönstren bildas genom att en ruta på N gånger M pixels upprepas över hela ytan. Möjliga mönster är t.ex.

Bas

Exempel på yta





Ange en ASN.1 datatyp med vilken man kan beskriva olika möjliga sådana täckningsmönster för ytor.

Övningsuppgift 9

Ett lager saluför papper i formaten A3, A4, A5 och A6. En beställare vill veta om lagret har papper i lager av vart och ett av de fyra formaten. Ange en datatyp med vilken butiken kan rapportera detta till beställaren.

Övningsuppgift 10

I X.400-standarden kan ett namn bestå av flera delfält. Ett av delfälten kallas för **OrganizationName** och kan ha ett värde mellan 1 och 64 tecken ur teckenmängden **PrintableString**. Ange ett förslag till definition av detta i ASN.1.

Övningsuppgift 11

I ett protokoll för överföring av personuppgifter mellan två datorer överförs alltid personnummer, bestående av enbart siffror, blanka och bindestreck. Namn (ej uppdelat på för- och efternamn, max 40 tecken) kan överföras, om det är känt, och beräknad årsinkomst kan överföras, om det är känt. Båda dessa uppgifter kan utelämnas, endast personnummer är obligatoriskt. Ange med användning av **SET** en datatyp som kan användas för att överföra denna information.

Övningsuppgift 12

Antag att namnet skall överföras som två fält, ett för förnamn och ett för efternamn. Hur kan man ändra lösningen till övningsuppgift 11 för att passa detta fall.

Övningsuppgift 13

Givet följande ASN.1-typ:

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

                    }
```

Definiera en subtyp till typen ovan som bara inkluderar tal i positiva kvadranten (där både x och y är ³ 0).

Övningsuppgift 14

Givet följande ASN.1-typ:

```
Message             ::= SET
                    {       author Name OPTIONAL,
                            textbody IA5String }
```

Definiera en subtyp till denna, kallad **AnonymousMessage**, i vilken **author** inte uppges.

Övningsuppgift 15

Definiera en datatyp **FullName** som omfattar en serie av tre element i given ordning: Förnamn, initialer och efternamn, där förnamn och initialier kan utelämnas men efternamn alltid måste anges.

Övningsuppgift 16

Definiera en datatyp **BasicFamily** bestående av 0 eller 1 **husband**, 0 eller 1 **wife** och 0, 1 eller flera **children**. Var och en av dessa komponenter anges med sitt namn som en **IA5String**.

Övningsuppgift 17

Definiera en datatyp **ChildLessFamily**, utgående från **BasicFamily**.

Övningsuppgift 18

Givet datatyperna **Aircraft**, **Ship**, **Train** och **MotorCar**, definiera en datatyp **Vessel** vars värde kan vara vilket som helst av dessa fyra datatyper med sina värden.

Övningsuppgift 19

Vad är skillnaden mellan de två datatyperna:

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

och

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

Hur kan man i båda fallen utgående från ovan givna definitioner definiera en ny datatyp **GeneralNameList** som bara kan innehålla **GeneralString**-fält?

Övningsuppgift 20

I en förening tillåts två slag av omröstningar:

(a) De röstande får välja ut ett och endast ett av ett antal alternativ 1 .. N och rösta för detta. Det alternativ som får flest röster vinner.

(b) De röstande får ange en poäng mellan 0 och 10 för vart och ett av alternativen 1 .. N. Det alternativ som får högst sammanlagd poäng vinner.

Ange en ASN.1 datatyp som kan användas för att rapportera en persons röstande till omröstningsräknaren och som kan användas för båda typerna av omröstningar. Den röstandes namn skall ingå i rapporten som en **IA5String**.

Övningsuppgift 21

Antag att en ASN.1-modul ser ut så här:

```
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 WeaterhReporting
```

Skriv om denna ASN.1-modul så att den ger exakt samma kodning, men är definierad med tag default **IMPLICIT** istället för **EXPLICIT**.

Övningsuppgift 22

Vilka tags kan tas bort nedan, med bibehållen korrekt ASN.1?

```
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
                }
```


Övningsuppgift 23

Följande ASN.1-konstruktion är hämtad ur X.500-standarden (88 års version, macros som **OPTIONALLY-SIGNED** togs bort i 1994 års version av 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 }
```

(a) Skriv om konstruktionen så att indenteringarna tydligare visar strukturen.

Övningsuppgift 24

Givet följande ASN.1-modul:

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

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

END
```

Definiera en ASN.1-modul **CarDriving**, som importerar **MainOperation** från modulen ovan, och definierar en ny datatyp **FullOperation** som utöver **MainOperation** också innefattar på och avkoppling av blinkers åt vänster eller höger, och inställning av lyset på släckt, parkeringsljus, halvljus eller helljus.

Övningsuppgift 25

Givet ASN.1-definitionen

```
Surname                 ::= [APPLICATION 1] IA5String

hername Surname         ::= "Mary"
```

Ange dess kodning i BER.

Övningsuppgift 26

Givet ASN.1-definitionen

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

daylight Light  ::= halflight
```

Ange detta värdes kodning i BER

Övningsuppgift 27

Givet följande ASN.1-definitioner och 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
                            }
                }
    }
```

Ange kodningen av firstorder med BER.

Övningsuppgift 28 (på RFC822)

En identifierare i Fortran skall bestå av mellan 1 och 6 bokstäver och siffror. Första tecknet måste vara en bokstav. Skriv en specifikation i ABNF för syntaxen för sådana identifierare.

Övningsuppgift 29 (på RFC822)

I övningsuppgift 20 uppgavs att en förening tillåts två slag av omröstningar:

(a) De röstande får välja ut ett och endast ett av ett antal alternativ 1 .. N och rösta för detta. Det alternativ som får flest röster vinner.

(b) De röstande får ange en poäng mellan 0 och 10 för vart och ett av alternativen 1 .. N. Det alternativ som får högst sammanlagd poäng vinner.

Ange ett förslag till specifikation av dessa data med textmässig kodning och ABNF som specifikationsspråk.

Lösningsförslag till övningsuppgifter på ASN.1 och BER

Övningsuppgift 1

Lösningsförslag 1:

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

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

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

Lösningsförslag 2:

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

Övningsuppgift 2

```
Box             ::= [APPLICATION 0] SEQUENCE
                {       height      Measurement,
                        width       Measurement,
                        length      Measurement
                }

Measurement     ::= [APPLICATION 1] SEQUENCE
                {       yards       INTEGER,
                        feet        INTEGER,
                        inches      REAL
                }
```

Övningsuppgift 3

```
Measurement     ::= [APPLICATION 1] SEQUENCE
                {       yards       INTEGER,
                        feet        INTEGER (0 .. 2),
                        inches      INTEGER (0 .. 1199)
                }
```

Övningsuppgift 4

```
Voter           ::= [APPLICATION 1] SEQUENCE
                {       vote        Vote,
                        age         Age,
                        male        Male
                }
```



```
Vote            ::= [APPLICATION 2] INTEGER
                {       v (0), - - Vänsterpartiet
                        s (1), - - Socialdemokraterna
                        mp (2), - - Miljöpartiet
                        c (3), - - Centerpartiet
                        fp (4), - - Folkpartiet
                        m (5), - - Moderaterna
                        kds (6), - - Kristdemokraterna
                        nd (7), - - Ny demokrati
                        o  (8) - - Övriga
                } (0 .. 8)

Age             ::= [APPLICATION 3] INTEGER ( 18 .. MAX )

Male            ::= [APPLICATION 4] BOOLEAN - - TRUE for male,
                False for female
```

Övningsuppgift 5

```
Stockholmvoter  ::= [APPLICATION 5) SEQUENCE
                {       sthvote     Sthvote,
                        age         Age,
                        sex         Sex
                }

Sthvote         ::= [APPLICATION 6] INTEGER
                {       v (0), - - Vänsterpartiet
                        s (1), - - Socialdemokraterna
                        mp (2), - - Miljöpartiet
                        c (3), - - Centerpartiet
                        fp (4), - - Folkpartiet
                        m (5), - - Moderaterna
                        kds (6), - - Kristdemokraterna
                        nd (7), - - Ny demokrati
                        o  (8), - - Övriga,
                        sp (9), - - Stockholmspartiet,
                        bp (10) - - Bilistpartiet
                }
                ( INCLUDES Vote | 9 | 10 )
```

Övningsuppgift 6:

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

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

Övningsuppgift 7

Lösningsförslag 1:

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

Lösningsförslag 2:

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

Anmärkning: Man måste upprepa **open, secret** och **qsecret** på nytt i definitionen av **StabSecrecy**. Om man vill slippa denna upprepning, får man först deklarare **StabSecrecy** och sedan deklarera **Secrecy** som en subtyp till **StabSecrecy**.

Övningsuppgift 8

Lösningsförslag 1:

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

Lösningsförslag 2:

```
Row                  ::= [APPLICATION 0] BIT STRING

Pattern              ::= [APPLICATION 1] SEQUENCE
                         {       height      INTEGER,
                                 width       INTEGER,
                                 SEQUENCE OF Row
                         }
```

Övningsuppgift 9

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

Övningsuppgift 10 (hämtat ur X.411)

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

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


Övningsuppgift 11

Lösningsförslag 1:

```
PersonRecord         ::= [APPLICATION 0] 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             ::= [APPLICATION 2] GraphicString (SIZE (1 .. 40))

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

Lösningsförslag 2:

```
PersonRecord         ::= [APPLICATION 0] 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             ::=  GraphicString (SIZE (1 .. 40))

Incometype           ::=  INTEGER (0 .. MAX)
```

Lösningsförslag 3:

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

PersonRecord         ::= [APPLICATION 0] SET
                         {       pnumber     Pnumber1 (SIZE (13))

                                 name        GraphicString (SIZE (1 .. 40))
OPTIONAL,
                                 income      INTEGER (0 .. MAX) OPTIONAL
                         }
```

Övningsuppgift 12

Lösningsförslag 1:

```
PersonRecord        ::= [APPLICATION 0] SET
                    {       pnumber     Pnumber,
                            gname       GNametype OPTIONAL,
                            sname       SNametype OPTIONAL,
                            income      Incometype OPTIONAL
                    }

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

Pnumber             ::= Pnumber1 (SIZE (13))

GNametype           ::= [APPLICATION 2] GraphicString (SIZE (1 .. 40))

SNametype           ::= [APPLICATION 3] GraphicString (SIZE (1 .. 40))

Incometype          ::= [APPLICATION 4] INTEGER (0 .. MAX)
```

Lösningsförslag 2:

```
PersonRecord        ::= [APPLICATION 0] 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            ::= [APPLICATION 2] SEQUENCE
                    {       sName       GraphicString (SIZE (1 .. 40)),
                            gName       GraphicString (SIZE(1 .. 40))
                    }

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

Fråga: Varför är nedanstående lösning felaktig?

```
PersonRecord        ::= [APPLICATION 0] SET
                    {       pnumber     Pnumber,
                            gname       Nametype OPTIONAL,
                            sname       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            ::= [APPLICATION 2] GraphicString (SIZE (1 .. 40))

Incometype          ::= [APPLICATION 4] INTEGER (0 .. MAX)
```

Övningsuppgit 13

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

Övningsuppgift 14

Lösningsförslag 1:

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

Lösningsförslag 2:

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

Övningsuppgift 15

```
FullName            ::= SEQUENCE
                    {       givenName [APPLICATION 0] IA5String
OPTIONAL,
                            initials [APPLICATION 1] IA5String OPTIONAL,
                            surname [APPLICATION 2] IA5STring
                    }
```

Fråga: Kan APPLICATION-etiketterna i lösningsförslaget ovan utelämnas?

Svar: Nej, ty elementen måste ha olika tags för att kunna skilja dem åt. Om det inte hade stått OPTIONAL så hade dessa etiketter kunnat utelämnas, ty då hade elementen kunna skiljas åt genom ordningsföljden.

Övningsuppgift 16

```
BasicFamily         ::= SEQUENCE
                    {       husband [0] IA5String OPTIONAL,
                            wife [1] IA5STring OPTIONAL,
                            children [2] SEQUENCE OF IA5String OPTIONAL
                    }
```

Fråga: Är det lämpligt att använda **SEQUENCE OF** eller **SET OF** ovan?

Svar: Om man vill ange barnens ordningsföljd, är **SEQUENCE OF** bättre.

Övningsuppgift 17

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

Övningsuppgift 18

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

Övningsuppgift 19

Lösningsförslag 1:

```
GeneralNameListA    ::= gs < NameListA

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

Lösningsförslag 2:

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

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


Övningsuppgift 20

Lösningsförslag 1:

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

Lösningsförslag 2:

(Utgår från att den röstande i fallet (b) alltid räknar upp samtliga alternativ i nummerordning, alternativnumret kan då utelämnas.)

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

Övningsuppgift 21

```
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
```

Övningsuppgift 22

Tags som kan tas bort anges i kursiv stil nedan

```
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
                        }
```

Övningsuppgift 23a

```
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 }
```

Övningsuppgift 23b

Svar: Såvitt jag kan förstå fattas det två komma-tecken, se nedan:

```
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 }
```


Övningsuppgift 23c

Svar: **COMPONENTS OF** är ingen egen typ, och kan alltså inte ha någon identifierare. Den syftar till att kopiera in en serie separat definierade typelement. Den är praktisk om man har en serie standardelement, t.ex. **CommonResults**, som skall kopieras in på flera olika ställen.

Övningsuppgift 23d

Svar: I en **SET** måste alla elementen ha olika typ. Det innebär att det räcker med att ange tag för alla utom ett av elementen.

Övningsuppgift 24

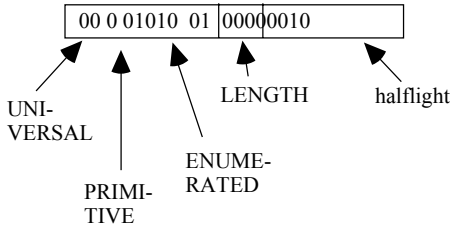
```
CarDriving { 1 2 4711 18 } DEFINITIONS EXPLICIT TAGS ::=
BEGIN
                IMPORTS MainOperation FROM Driving {1 2 4711 17}

                FullOperation ::= [APPLICATION 1] SEQUENCE
                {       COMPONENTS OF MainOperation,
                        blink       SEQUENCE
                                    {       on   BOOLEAN,
                                            left  BOOLEAN
                                    },
                        light       ENUMERATED
                                    {       dark(0),
                                            parkingLight (1),
                                            halfLight (2),
                                            fullLight (3)
                                    }
                }
END - - of module CarDriving
```

Anmärkning: Eftersom det inte fanns något EXPORTS i Driving, betyder detta att alla objekt specificerade i Driving är exporterade.

Övningsuppgift 25

APPLI-CATION	CON-STRUCTED	Tag nr.	Length	UNI-VER-SAL	PRIMI TIVE	IA5-STRING	Length				
01	1	00001	6	00	0	10110	4	M	a	r	y
61			06	16			04	M	a	r	y

Övningsuppgift 26



Övningsuppgift 27

(Jag lovar inte att det är rätt!)

		Antal oktetter
beverage	**(context explicit tag)101 00001 (ENUMERATED) 000 01010**	**2**
tea	**(length) 1 (value) 00000001**	**2**
jam	**(contextexplicit 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**

Övningsuppgift 28

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"

fortran-identifier = ALPHA *5(ALPHA / DIGIT)

Övningsuppgift 29

Vote = Voter-name "," (One-choice / Choice-list)

Voter-name = <"> Name <">

Name =1*Namechar

Namechar = <any printable ASCII character except <"> and "\">
/ "\\" / "\""

One-choice = "Single:" 1*DIGIT

Choice-list = "Multiple:" 1#(Alternative Score)

Alternative = 1*DIGIT

Score = = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" / "10"

Jämför med ASN.1-lösningen av övningsuppgift 20 (lösningsförslag 1), se nedan:

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

Index

This is a quick and easy HTML guide for beginners and can be a good reference tool for novice HTML writers. It is not a complete listing of all the HTML language, but should be an excellent spot for you to start. Some of the HTML below may not work on all browsers, but should work with Microsoft, Netscape and any other browser supporting some of the latest HTML language.

Compendium 6 page 99

How to Start an HTML Document

If you have an HTML Editor, many come with a start-up template you can use. However, if you do not, you can still actually create HTML Documents with a basic Windows' Notepad! I do suggest that you find a regular editor, which will speed up many of the processes of creating and updating your pages.

The Start-Up Template

Every HTML Document should include the following commands:

<html>
<head> <title> </title> </head>
<body>

</body>
</html>
Start your new page by typing the above commands in. Notice that the HTML commands start with a "<" and end with a ">". Also, for about every starting command (<body>, for example) there is always an ending command for that function (</body>).

Now, you are ready to start entering the basic text for your page. Name a title that is appropriate to the information you are going to provide on that page.

<html>
<head> <title>**Your Title Goes Here** </title> </head>
<body>

Your Text Goes Here

</body>
</html>
Back to top of page

Basic HTML Extensions

Now you are ready to start laying out your page. Have some idea, first, on what order and placement you want the text and images displayed.

<h1> </h1>

The first command you will generally use is the Headline command.

<h1>Your Main Headline</h1> This diplays the biggest headline size. **You MUST include the closing command, or the headline function will continue to work on any following text!** To change the size of the headline, simply change the number with the headline command accordingly.

<h1> Headline </h1>

<h2> Headline </h2>

<h3> Headline </h3>

and so on, down to level 6

 Line Break

HTML documents will continue to keep wrapping text together until it comes to a command for it to do otherwise. The <br> command DOES NOT need an ending statement. It is simply used to start a new line of text.

New Paragraph

Used in the same way as Line Break. However, this function also inserts a blank line to better seperate the paragraphs. It also does NOT need a closing command ( )

<hr> Horizontal Line

This inserts a horizontal line across the page to help divide or break up sectionns of text. It does not need a closing arguement. However, there are a few additional options you can add to this command:

<hr width=50%>
This tells the line to only go across 50% of the page.

  Unordered List

Nice function to use if you have a list to display. Start with the <ul> command to tell the document to expect a list. Then, for each listing, you must use the <li> command. Simply place the <li> in front of each item. There is no need to use a line break, since this function will automatically move to the next line when it comes to the next item.

Item #1
Item #2

The <ul> DOES REQUIRE a closing arguement! At the end of your list, insert to tell the document that is the end of the list. It is really easy to add to the list later. Just start with the <li> inside your list commands (<ul>) and insert your text!

<img src="imagename.gif"> Display Images

Insert this command where you want to display an image. Substitute "imagename.gif" with the path to your image. If the image is located in the same directory as you HTML document, then simply use the image's filename. However, if you keep all your images in another directory, such as "images", then you must input the path to the file:
<img src="images/filename.gif">

You can also align your images along with your text. If you want the image to be displayed on the left, and your text to flow around it on the right, then use the following **align** command:
<img **align=left** src="images/filename.gif">

The best way to learn this is to experiment with your page and see what the different options do to it. You can also change the **border** of an image. If you want no border around the image, use the border=0 command as in the following: <img align=left **border=0** src="images/filename.gif">

To increase the border width around the image, simply increase the value of the border command.

Back to top of page

Background Colors and Images

Colors

You can also change the color or look of the background for your HTML page! This is done inside the <body> command.

If you want a solid color background instead of the default gray, add the following command to the <body> statement:

<body **bgcolor="#ffffff"**>

This will create a solid white background. Reload your page and see what it looks like. To change the background color, simply change the value of the bgcolor statement. As in the above example, the ffffff makes all the RGB red, green, blue) values full. Try inserting numbers into some of these and see what you get! For exampe:

<body bgcolor="#B2E2FF"> creates a sky-blue color. See Color Chart Codes

Images

If you want more of a textured look, or something besides a solid color for background, then the background image is what you want.

First you must have a .gif file of the background you want for your page. The image does not need to be very big at all. In fact, you may want it down to about 100x100 pixels. The browser automatically copies it until it covers the entire page.

Once you have your file, simply (as with the bgcolor command) add the following statement to the <body> call:

<body background="imagename.gif">

That should do it! Just make sure the image is in the same directory as the page that is calling it, or that you have the correct path to the file.

Back to top of page

Creating Links

This a nice place to have an HTML editor, but it still can be done relatively simply. As with most of these, you do not necessarily have to type all these commands out by hand if you have an editor. However, it is greatly important that you understand what the editor is doing, so that in the event of an error on your page - you know what to look for and can fix it.

To create a link, either to an outside site or to another page on yours, find the word or words that you want to be highlighted as the link. Immediately right before the linkable word, type:

****
Where "link-to-site" is the path to an outside site or a document on your site. For outside sites, you must include the full path: "http://www.domain.com/page.html"

and after the linkable word: ****

For example, if I want the word "NASA" to take me to NASA'a site, my link would read:
NASA

You can also make an image a link to something by including it inside the and the .

Example: <img src="nasa.gif">

It is VERY important that you have the immediately following the linkable item. Otherwise, all the remaining text will be included in that link.

Back to top of page

Tables

Tables are used to create rows and columns of text and images. Simular to a newspaper layout.

To create a table in HTML, you must start with **<table>** and end with the **</table>** commands. You then begin a row of data by using **<tr>**. To now insert data into this row, add **<td>** and go ahead with your normal text and/or images.

If you want to start a new column, use the **<td>** again, with the text following. It does not matter how much information you have after the **<td>**. The table will continue to wrap the text.

For example:

<table border=1>
<tr><td>Your first column of text
<td>Your second column of text
</table>
Displays this:

Your first column of text	Your second column of text

If you want to see the borders of the table, which will probably help in the beginning and then you can always get rid of them, add **border=1** to the **<table>** function as above. **1** being the thickness of the border.

The **</td>** is optional at the end of the data cell (the same for </tr>), but SHOULD be used when using tables within other tables.

Some of the text and images may be centered vertically on your table. If you want them to start at the top, add **valign=top** to the **<td>** call for that text. (**<td valign=top>**)

To start a new horizontal row, use **<tr>** (and **<td>** for the data). Starting to get the picture now?

<table border=1>
<tr><td>Your first column of text
<td>Your second column of text
<tr><td>Second row
</table>

Your first column of text	Your second column of text
Second row	

You could then add another **<td>** to move over to the next column, or you can stretch the existing table cell over the next column by using **<td colspan=2>**.

<table border=1>
<tr><td>Your first column of text
<td>Your second column of text
<tr><td colspan=2>Second row
</table>

Your first column of text	Your second column of text
Second row	

The same can be done with spanning rows. Use **<td rowspan=#>**, replacing # with the number of horizontal rows you wish to span.

Hope this gets you started with tables. Try experimenting around with what all you can do with tables and view the different results with your browser.

Back to top of page

Java Scripts & Codes

You can Cut and Paste these Scripts into your page.

Frame On Frame Off:
This script lets the user choose which version of your page they want to see. Framepage.html is the your page with frames. Regualerpage.html is an alternate you have created

Frames: On Off
```
<form>
<b>Frames:</b>
<INPUT TYPE="Button" VALUE="On" onClick="parent.location='framepage.html'">
<INPUT TYPE="Button" VALUE="Off" onClick="parent.location='regularpage.html'">
</form>
```

This shows the user the last date this page was modified:
```
<SCRIPT LANGUAGE="JavaScript">
//Modified by Coffeecup.com
function initArray()
{
this.length = initArray.arguments.length
for (var i = 0; i < this.length; i++)
this[i+1] = initArray.arguments[i]
}
var DOWArray = new initArray("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday");
var MOYArray = new
initArray("January","February","March","April","May","June","July","August","September","October","November","December");

var LastModDate = new Date(document.lastModified);
document.write("This page was last updated on ");
document.write(DOWArray[(LastModDate.getDay()+1)],", ");
document.write(MOYArray[(LastModDate.getMonth()+1)]," ");
document.write(LastModDate.getDate(),", ",(LastModDate.getYear()+1900));
document.write(".");
</SCRIPT>
```

Script produces a randomly generated graphic to use as ad or cool effect

```
<Script Language ="JavaScript">
//Modified by user
//produces a randomly generated graphic to use as ad or cool effect
function RandomNumber()
{
var today = new Date();
var num= Math.abs(Math.sin(today.getTime()/1000));
return num;
}
function RandomGraphics()
{
var x = RandomNumber();
if (x > .77) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > .66) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > .55) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > .44) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > .33) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > .22) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > .11) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
if (x > 0) {document.write("<A HREF='URL'><img src='image here' align=center hspace=10></a>"); return; }
}
RandomGraphics();
//End Script
</SCRIPT>
```

Real Audio Tag will begin to stream live. ***This is not a script***

```
<EMBED SRC="*.rpm" WIDTH=40 HEIGHT=20 CONTROLS = PlayButton AUTOSTART=TRUE>
```

Wave Background Script designed to work with I.E. and Netscape Formats.

```
<!--Begin Wave File Script--><script>
if(navigator.userAgent.indexOf("MSIE") != -1)
document.writeln ('<bgsound src="*.wav">');else
document.writeln ('<embed src="*.wav" height=55 width=146 autostart=true hidden=true>');
</script><!--End Wave File Script-->
```

Java Script Archive

Back to top of page

Other HTML Reference Sources

- **Wilbur HTML 3.2 Reference**
- **Microsoft HTML Reference**
- **HyperText Markup Language**
  - **HTML2.0**
  - **HTML3.0**

Advanced HTML | Adding a touch of style

Getting started with HTML

***Dave Raggett*, revised 29th August 2000.**

This is a short introduction to writing HTML. Many people still write HTML by hand using tools such as NotePad on Windows, or SimpleText on the Mac. This guide will get you up and running. Even if you don't intend to edit HTML directly and instead plan to use an HTML editor such as Netscape Composer, or W3C's Amaya, this guide will enable you to understand enough to make better use of such tools and how to make your HTML documents accessible on a wide range of browsers. Once you are comfortable with the basics of authoring HTML, you may want to learn how to add a touch of style using CSS, and to go on to try out features covered in my page on advanced HTML

A convenient way to automatically fix markup errors is to use the HTML Tidy utility. This also tidies the markup making it easier to read and easier to edit. I recommend you regularly run Tidy over any markup you are editing. Tidy is very effective at cleaning up markup created by authoring tools with sloppy habits.

p.s. a good way to learn is to look at how other people have coded their html pages. To do this, click on the "View" menu and then on "Source". Try it with this page to see how I have applied the ideas I explain below. You will find yourself developing a critical eye as many pages look rather a mess under the hood!

This page will teach you how to:

- start with a title
- add headings and paragraphs
- add emphasis to your text
- add images
- add links to other pages
- use various kinds of lists

If you are looking for something else, try the advanced HTML page.

Start with a title

Every HTML document needs a title. Here is what you need to type:

```
<title>My first HTML document</title>
```

Change the text from "My first HTML document" to suit your own needs. The title text is preceded by the start tag <title> and ends with the matching end tag </title>. The title should be placed at the beginning of your document.

Add headings and paragraphs

If you have used Microsoft Word, you will be familiar with the built in styles for headings of differing importance. In HTML there are six levels of headings. H1 is the most important, H2 is slightly less important, and so on down to H6, the least important.

Here is how to add an important heading:

```
<h1>An important heading</h1>
```

and here is a slightly less important heading:

```
<h2>A slightly less important heading</h2>
```

Each paragraph you write should start with a tag. The is optional, unlike the end tags for elements like headings. For example:

```
<p>This is the first paragraph.</p>

<p>This is the second paragraph.</p>
```

Adding a bit of emphasis

You can emphasise one or more words with the tag, for instance:

```
This is a really <em>interesting</em> topic!
```

Adding interest to your pages with images

Images can be used to make your Web pages distinctive and greatly

help to get your message across. The simple way to add an image is using the <img> tag. Let's assume you have an image file called "peter.jpg" in the same folder/directory as your HTML file. It is 200 pixels wide by 150 pixels high.

```
<img src="peter.jpg" width="200" height="150">
```

The src attribute names the image file. The width and height aren't strictly necessary but help to speed the display of your Web page. Something is still missing! People who can't see the image need a description they can read in its absence. You can add a short description as follows:

```
<img src="peter.jpg" width="200" height="150"
alt="My friend Peter">
```

The alt attribute is used to give the short description, in this case "My friend Peter". For complex images, you may need to also give a longer description. Assuming this has been written in the file "peter.html", you can add one as follows using the longdesc attribute:

```
<img src="peter.jpg" width="200" height="150"
alt="My friend Peter" longdesc="peter.html">
```

You can create images in a number of ways, for instance with a digital camera, by scanning an image in, or creating one with a painting or drawing program. Most browsers understand GIF and JPEG image formats, newer browsers also understand the PNG image format. To avoid long delays while the image is downloaded over the network, you should avoid using large image files.

Generally speaking, JPEG is best for photographs and other smoothly varying images, while GIF and PNG are good for graphics art involving flat areas of color, lines and text. All three formats support options for progressive rendering where a crude version of the image is sent first and progressively refined.

Adding links to other pages

What makes the Web so effective is the ability to define links from one page to another, and to follow links at the click of a button. A single click can take you right across the world!

Links are defined with the <a> tag. Lets define a link to the page defined in the file "peter.html":


```
This a link to <a href="peter.html">Peter's page</a>.
```

The text between the <a> and the is used as the caption for the link. It is common for the caption to be in blue underlined text.

To link to a page on another Web site you need to give the full Web address (commonly called a URL), for instance to link to www.w3.org you need to write:

```
This is a link to <a href="http://www.w3.org/">W3C</a>.
```

You can turn an image into a hypertext link, for example, the following allows you to click on the company logo to get to the home page:

```
<a href="/"><img src="logo.gif" alt="home page"></a>
```

Three kinds of lists

HTML supports three kinds of lists. The first kind is a bulletted list, often called an ***unordered list***. It uses the <ul> and <li> tags, for instance:

```
<ul>
  <li>the first list item</li>

  <li>the second list item</li>

  <li>the third list item</li>
</ul>
```

Note that you always need to end the list with the end tag, but that the is optional and can be left off. The second kind of list is a numbered list, often called an ***ordered list***. It uses the <ol> and <li> tags. For instance:

```
<ol>
  <li>the first list item</li>

  <li>the second list item</li>

  <li>the third list item</li>
</ol>
```

Like bulletted lists, you always need to end the list with the end tag, but the end tag is optional and can be left off.

The third and final kind of list is the definition list. This allows you to list terms and their definitions. This kind of list starts with a <dl> tag and ends with </dl> Each term starts with a <dt> tag and each definition starts with a <dd>. For instance:

```
<dl>
  <dt>the first term</dt>
  <dd>its definition</dd>

  <dt>the second term</dt>
  <dd>its definition</dd>

  <dt>the third term</dt>
  <dd>its definition</dd>
</dl>
```

The end tags </dt> and </dd> are optional and can be left off. Note that lists can be nested, one within another. For instance:

```
<ol>
  <li>the first list item</li>

  <li>
    the second list item
    <ul>
      <li>first nested item</li>
      <li>second nested item</li>
    </ul>
  </li>

  <li>the third list item</li>
</ol>
```

You can also make use of paragraphs and headings etc. for longer list items.

Getting Further Information

If you are ready to learn more, I have prepared some accompanying material on advanced HTML and adding a touch of style.

W3C's Recommendation for HTML 4.0 is the authorative specification for HTML. However, it is a technical specification. For a less technical source of information of information you may want to purchase one of the many books on HTML, for example "Raggett on HTML 4", published 1998 by Addison Wesley. See also ***"Beginning XHTML"***, published 2000 by Wrox Press, which introduces W3C's reformulation of HTML as an application of XML. XHTML 1.0 is now a W3C Recommendation.

Best of luck and get writing!

Dave Raggett <*dsr@w3.org*>

Adding a touch of style

Back to Basics

***Dave Raggett*, 26th November 1999.**

This is a short guide to styling your Web pages. It will show you how to use W3C's Cascading Style Sheets language (CSS) as well as alternatives using HTML itself. The route will steer you clear of most of the problems caused by differerences between different brands and versions of browsers.

Getting started

Let's start with setting the color of the text and the background. You can do this by using the STYLE element to set style properties for the document's tags:

```
<style type="text/css">
  body { color: black; background: white; }
</style>
```

The stuff between the <style> and </style> is written in special notation for style rules. Each rule starts with a tag name followed by a list of style properties bracketed by { and }. In this example, the rule matches the ***body*** tag. As you will see, the body tag provides the basis for setting the overall look and feel of your Web page.

Each style property starts with the property's name, then a colon and lastly the value for this property. When there is more than one style property in the list, you need to use a semicolon between each of them to delimit one property from the next. In this example, there are two properties - "color" which sets the color of the text, and "background" which sets the color of the page background. I recommend always adding the semicolon even after the last property.

Colors can be given as names or as numerical values, for instance `rgb(255, 204, 204)` which is a fleshy pink. The 3 numbers correspond to red, green and blue respectively in the range 0 to 255. You can also use a hexadecimal notation, the same color can also be written as `#FFCCCC`. More details on color is given in a later section.

Note that the style element must be placed in the document's head along with the title element. It shouldn't be placed within the body.

Linking to a separate style sheet

If you are likely to want to use the same styles for several Web pages it is worth considering using a separate style sheet which you then link from each page. You can do this as follows:

```
<link rel="stylesheet" href="style.css">
```

The LINK tag should be placed in the document's head. The ***rel*** attribute must be set to the value "stylesheet" to allow the browser to recognize that the ***href*** attribute gives the Web address (URL) for your style sheet.

Setting the page margins

Web pages look a lot nicer with bigger margins. You can set the left and right margins with the "margin-left" and "margin-right" properties, e.g.

```
<style type="text/css">
  body { margin-left: 10%; margin-right: 10%; }
</style>
```

This sets both margins to 10% of the window width, and the margins will scale when you resize the browser window.

Setting left and right indents

To make headings a little more distinctive, you can make them start within the margin set for the body, e.g.

```
<style type="text/css">
  body { margin-left: 10%; margin-right: 10%; }
  h1 { margin-left: -8%;}
  h2,h3,h4,h5,h6 { margin-left: -4%; }
</style>
```

This example has three style rules. One for the body, one for h1 (used for the most important headings) and one for the rest of the headings (h2, h3, h4, h5 and h6). The margins for the headings are additive to the margins for the body. Negative values are used to move the start of the headings to the left of the margin set for the body.

In the following sections, the examples of particular style rules will need to be placed within the style element in the document's head (if present) or in a linked style sheet.

Controlling the white space above and below

Browsers do a pretty good job for the white space above and below headings and paragraphs etc. Two reasons for taking control of this yourself are: when you want a lot of white space before a particular heading or paragraph, or when you need precise control for the general spacings.

The "margin-top" property specifies the space above and the "margin-below" specifies the space below. To set these for all h2 headings you can write:

```
h2 { margin-top: 8em; margin-bottom: 3em; }
```

The em is a very useful unit as it scales with the size of the font. One em is the height of the font. By using em's you can preserve the general look of the Web page independently of the font size. This is much safer than alternatives such as pixels or points, which can cause problems for users who need large fonts to read the text.

Points are commonly used in word processing packages, e.g. 10pt text. Unfortunately the same point size is rendered differently on different browsers. What works fine for one browser will be illegible on another! Sticking with em's avoids these problems.

To specify the space above a particular heading, you should create a named style for the heading. You do this with the ***class*** attribute in the markup, e.g.

```
<h2 class="subsection">Getting started</h2>
```

The style rule is then written as:

```
h2.subsection { margin-top: 8em; margin-bottom: 3em; }
```

The rule starts with the tag name, a dot and then the value of the class attribute. Be careful to avoid placing a space before or after the dot. If you do the rule won't work. There are other ways to set the styles for a particular element but the class attribute is the most flexible.

When a heading is followed by a paragraph, the value for margin-bottom for the heading isn't added to the value for margin-top for the paragraph. Instead, the maximum of the two values is used for the spacing between the heading and paragraph. This subtlety applies to margin-top and margin-bottom regardless of which tags are involved.

First-line indent

Sometimes you may want to indent the first line of each paragraph. The following style rule emulates the traditional way paragraphs are rendered in novels:

```
p { text-indent: 2em; margin-top: 0; margin-bottom: 0; }
```

It indents the first line of each paragraph by 2 em's and suppresses the inter-paragraph spacing.

Controlling the font

This section explains how to set the font and size, and how to add italic, bold and other styles.

Font styles

The most common styles are to place text in italic or bold. Most browsers render the ***em*** tag in italic and the ***strong*** tag in bold. Let's assume you instead want em to appear in ***bold italic*** and strong in **BOLD UPPERCASE**:

```
em { font-style: italic; font-weight: bold; }
strong { text-transform: uppercase;  font-weight: bold; }
```

If you feel so inclined, you can fold headings to lower case as follows:

```
h2 { text-transform: lowercase; }
```

Setting the font size

Most browsers use a larger font size for more important headings. If you override the default size, you run the risk of making the text too small to be legible, particularly if you use points. You are therefore recommended to specify font sizes in relative terms.

This example sets heading sizes in percentages relative to the size used for normal text:

```
h1 { font-size: 200%; }
h2 { font-size: 150%; }
h3 { font-size: 100%; }
```

Setting the font family

It is likely that your favorite font won't be available on all browsers. To get around this, you are allowed to list several fonts in preference order. There is a short list of generic font names which are garanteed to be available, so you are recommended to end your list with one of these: serif, sans-serif, cursive, fantasy, or monospace, for instance:

```
body { font-family: Verdana, sans-serif; }
h1,h2 { font-family: Garamond, Times New Roman, serif; }
```

In this example, important headings would preferably be shown in Garamond, failing that in Times New Roman, and if that is unavailable in the browsers default serif font. Paragraph text would appear in Verdana or if that is unavailable in the browser's default sans-serif font.

The legibility of different fonts generally depends more on the height of lower case letters than on the font size itself. Fonts like Verdana are much more legible than ones like Times New Roman and are therefore recommended for paragraph text.

Adding borders and backgrounds

You can easily add a border around a heading, list, paragraph or a group of these enclosed with a ***div*** element. For instance:

```
div.box { border: solid; border-width: thin; }
```

which can be used with markup such as:

```
<div class="box">
The content within this DIV element will be enclosed
in a box with a thin line around it.
</div>
```

There are a limited choice of border types: dotted, dashed, solid, double, groove, ridge, inset and outset. The border-width property sets the width. Its values include thin, medium and thick as well as a specified width e.g. 0.1em. The border-color property allows you to set the color.

A nice effect is to paint the background of the box with a solid color or with a tiled image. To do this you use the background property. You can fill the box enclosing a div as follows:

```
div.color {
  background: rgb(204,204,255);
  padding: 0.5em;
  border: none;
}
```

Without an explicit definition for border property some browsers will only paint the background color under each character. The padding property introduces some space between the edges of the colored region and the text it contains.

You can set different values for padding on the left, top, right and bottom sides with the padding-left, padding-top, padding-right and padding-bottom properties, e.g. `padding-left: 1em`.

Suppose you only want borders on some of the sides. You can control the border properties for each of the sides independently using the border-left,

border-top, border-right and border-bottom family of properties together with the appropriate suffix: style, width or color, e.g.

```
p.changed {
  padding-left: 0.2em;
  border-left: solid;
  border-right: none;
  border-top: none;
  border-bottom: none;
  border-left-width: thin;
  border-color: red;
}
```

which sets a red border down the left hand side only of any paragraph with the class "changed".

What about browsers that don't support CSS?

Older browsers, that is to say before Netscape 4.0 and Internet Explorer 4.0, either don't support CSS at all or do so inconsistently. For these browsers you can still control the style by using HTML itself.

Setting the color and background

You can set the color using the BODY tag. The following example sets the background color to white and the text color to black:

```
<body bgcolor="white" text="black">
```

The BODY element should be placed before the visible content of the Web page, e.g. before the first heading. You can also control the color of hypertext links. There are three attributes for this:

- **LINK** for unvisited links
- **VLINK** for visited links
- **ALINK** for the color used when you click the link

Here is an example that sets all three:

```
<body bgcolor="white" text="black"
  link="navy" vlink="maroon" alink="red">
```

You can also get the browser to tile the page background with an image using the background attribute to specify the Web address for the image, e.g.

```
<body bgcolor="white" background="texture.jpeg" text="black"
```

```
  link="navy" vlink="maroon" alink="red">
```

It is a good idea to specify a background color using the bgcolor attribute in case the browser is unable to render the image. You should check that the colors you have chosen don't cause legibility problems. As an extreme case consider the following:

```
<body bgcolor="black">
```

Most browsers will render text in black by default. The end result is that the page will be shown with black text on a black background! Lots of people suffer from one form of color blindness or another, for example olive green may appear brown to some people.

A separate problem appears when you try to print the Web page. Many browsers will ignore the background color, but will obey the text color. Setting the text to white will often result in a blank page when printed, so the following is not recommended:

```
<body bgcolor="black" text="white">
```


You can also use the bgcolor attribute on table cells, e.g.

```
<table border="0" cellpadding="5">
 <tr>
  <td bgcolor="yellow">colored table cell</td>
 </tr>
</table>
```

Tables can be used for a variety of layout effects and have been widely exploited for this. In the future this role is likely to be supplanted by style sheets, which make it practical to achieve precise layout with less effort.

Named colors

The standard set of color names is: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. These 16 colors are defined in HTML 3.2 and 4.0 and correspond to the basic VGA set on PCs. Most browsers accept a wider set of color names but use of these is not recommended.

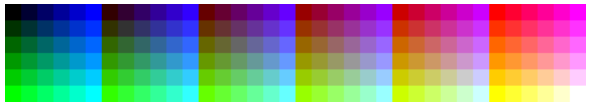
Hexadecimal color values

Values like "#FF9999" represent colors as hexadecimal numbers for red, green and blue. The first two characters following the # give the number for red, the next two for green and the last two for blue. These numbers are always in the range 0 to 255 decimal. If you know the values in decimal, you can convert to hexadecimal using a calculator, like the one that comes as part of Microsoft Windows.

Browser safe colors

Many older color systems can only show up to 256 colors at a time. To cope with this, browsers make do with colors from a fixed palette. The effect of this is often visible as a speckling of colors as the browser tries to approximate the true color at any point in the image.

Most browsers use the same so called "browser safe" palette. This uses 6 evenly spaced gradations in red, green and blue and their combinations. If you select image colors from this palette, you can avoid the speckling effect. This is particularly useful for background areas of images.



These are constructed from colors where red, green and blue are restricted to the values:

RGB	00	51	102	153	204	255
Hex	00	33	66	99	CC	FF

The page background uses the nearest color in the palette. If you set the page background to a color which isn't in the browser safe palette, you run the risk that the background will have different colors depending on whether the computer is using indexed or true-color.

How do you find browser safe colors? There are several ways: you can load the browser safe palette into an image editor, you can use an image viewer to select a color from an image showing all colors in the palette, or you can use a tool that shows the system palette, and read off the color you like.

Setting the font, its size and color

The FONT tag can be used to select the font, to set its size and the color. This example just sets the color:

```
This sentance has a <font color="yellow">word</font> in yellow
```

The ***face*** attribute is used to set the font. It takes a list of fonts in preference order, e.g.

```
<font face="Garamond, Times New Roman">some text ...</font>
```

The ***size*** attribute can be used to select the font size as a number from 1 to 6. If you place a - or + sign before the number it is interpreted as a relative

value. Use size="+1" when you want to use the next larger font size and size="-1" when you want to use the next smaller font size, e.g.

```
<font size="+1" color="maroon"
  face="Garamond, Times New Roman">some text ...</font>
```

Getting Further Information

For further information you may want to purchase one of the many books on HTML that cover CSS, for example "Raggett on HTML 4", published 1998 by Addison Wesley. For a more detailed explanation, "Cascading Style Sheets" by Håkon Wium Lie and Bert Bos, pub. 1997 by Addison Wesley, provides an in-depth look at CSS as seen by the architects of CSS themselves.

I plan to extend this guide with additional pages explaining CSS positioning, printing and aural style sheets.

Best of luck and get writing!

Version 4.0 Formatted -- February 1999

The latest version of this document is available at http://werbach.com/barebones/, where you will also find the text version, translations, and background materials.

The Bare Bones Guide to HTML lists all the tags that current browsers are likely to recognize. I have included all the elements in the official HTML 4.0 recommendation with common attributes, as well as Netscape and Microsoft extensions. This document is a quick reference, not a complete specification; for official information about HTML and its development, see the World Wide Web Consortium site at http://www.w3.org/MarkUp/.

The Guide is designed to be as concise as possible, and therefore it doesn't go into any detail about how to use the various tags. A few tags link to notes that address frequently-asked questions. If you're looking for more detailed step-by-step information, see my WWW Help Page.

Table of Contents

Important: If you are not clear about the differences between the various versions of HTML, I suggest that you read my discussion of the development of HTML, or the World Wide Web Consortium HTML activity statement.

BASIC ELEMENTS

Document Type	<HTML></HTML>	(beginning and end of file)
Title	<TITLE></TITLE>	(must be in header)
Header	<HEAD></HEAD>	(descriptive info, such as title)
Body	<BODY></BODY>	(bulk of the page)

STRUCTURAL DEFINITION

	Heading	<H?></H?>	(the spec. defines 6 levels)
	Align Heading	<H? ALIGN=LEFT|CENTER|RIGHT></H?>	
	Division	<DIV></DIV>	
	Align Division	<DIV ALIGN=LEFT|RIGHT|CENTER|JUSTIFY></DIV>	
4.0	**Defined Content**	<SPAN></SPAN>	
	Block Quote	<BLOCKQUOTE></BLOCKQUOTE>	(usually indented)
4.0	**Quote**	<Q></Q>	(for short quotations)
4.0	**Citation**	<Q CITE="URL"></Q>	
	Emphasis	<EM></EM>	(usually displayed as italic)
	Strong Emphasis	<STRONG></STRONG>	(usually displayed as bold)
	Citation	<CITE></CITE>	(usually italics)
	Code	<CODE></CODE>	(for source code listings)
	Sample Output	<SAMP></SAMP>	
	Keyboard Input	<KBD></KBD>	
	Variable	<VAR></VAR>	
	Definition	<DFN></DFN>	(not widely implemented)
	Author's	<ADDRESS></ADDRESS>	

	Address		
	Large Font Size	<BIG></BIG>	
	Small Font Size	<SMALL></SMALL>	
4.0	**Insert**	<INS></INS>	(marks additions in a new version)
4.0	**Time of Change**	<INS DATETIME=":::"></INS>	
4.0	**Comments**	<INS CITE="URL"></INS>	
4.0	**Delete**	<DEL></DEL>	(marks deletions in a new version)
4.0	**Time of Change**	<DEL DATETIME=":::"></DEL>	
4.0	**Comments**	<DEL CITE="URL"></DEL>	
4.0	**Acronym**	<ACRONYM></ACRONYM>	
4.0	**Abbreviation**	<ABBR></ABBR>	

PRESENTATION FORMATTING

	Bold	<B></B>	
	Italic	<I></I>	
4.0*	**Underline**	<U></U>	(not widely implemented)
	Strikeout	<STRIKE></STRIKE>	(not widely implemented)
4.0*	**Strikeout**	<S></S>	(not widely implemented)
	Subscript		
	Superscript		
	Typewriter	<TT></TT>	(displays in a monospaced font)
	Preformatted	<PRE></PRE>	(display text spacing as-is)
	Width	<PRE WIDTH=?></PRE>	(in characters)
	Center	<CENTER></CENTER>	(for both text and images)
N1	**Blinking**	<BLINK></BLINK>	(the most derided tag ever)
	Font Size	<FONT SIZE=?></FONT>	(ranges from 1-7)
	Change Font Size	<FONT SIZE="+|-?"></FONT>	
	Font Color	<FONT COLOR="#$$$$$$"></FONT>	
4.0*	**Select Font**	<FONT FACE="***"></FONT>	
N4	**Point size**	<FONT POINT-SIZE=?></FONT>	
N4	**Weight**	<FONT WEIGHT=?></FONT>	
4.0*	**Base Font Size**	<BASEFONT SIZE=?>	(from 1-7; default is 3)
MS	**Marquee**	<MARQUEE></MARQUEE>	

POSITIONING

N3	**Multi-Column**	<MULTICOL COLS=?></MULTICOL>
N3	**Column Gutter**	<MULTICOL GUTTER=?></MULTICOL>
N3	**Column Width**	<MULTICOL WIDTH=?></MULTICOL>
N3	**Spacer**	<SPACER>
N3	**Spacer Type**	<SPACER TYPE=HORIZONTAL|VERTICAL|BLOCK>
N3	**Size**	<SPACER SIZE=?>
N3	**Dimensions**	<SPACER WIDTH=? HEIGHT=?>
N3	**Alignment**	<SPACER ALIGN=LEFT|RIGHT|CENTER>
N4	**Layer**	<LAYER></LAYER>
N4	**Name**	<LAYER ID="***"></LAYER>
N4	**Location**	<LAYER LEFT=? TOP=?></LAYER>
N4	**Rel. Position**	<LAYER PAGEX=? PAGEY=?></LAYER>
N4	**Source File**	<LAYER SRC="***"></LAYER>
N4	**Stacking**	<LAYER Z-INDEX=?></LAYER>
N4	**Stack Position**	<LAYER ABOVE="***" BELOW="***"></LAYER>
N4	**Dimensions**	<LAYER HEIGHT=? WIDTH=?></LAYER>
N4	**Clipping Path**	<LAYER CLIP=,,,></LAYER>
N4	**Visible?**	<LAYER VISIBILITY=SHOW|HIDDEN|INHERIT></LAYER>
N4	**Background**	<LAYER BACKGROUND="$$$$$$"></LAYER>
N4	**Color**	<LAYER BGCOLOR="$$$$$$"></LAYER>
N4	**Inline Layer**	<ILAYER></ILAYER> (takes same attributes as LAYER)
N4	**Alt. Content**	<NOLAYER></NOLAYER>

LINKS, GRAPHICS, AND SOUNDS

	Link	

	Link to Location	<A HREF="URL#***"></A>	(if in another document)
		<A HREF="#***"></A>	(if in current document)
4.0*	**Target Window**	<A HREF="URL" TARGET="***"></A>	
4.0*	**Action on Click**	<A HREF="URL" ONCLICK="***"></A>	(Javascript)
4.0*	**Mouseover Action**	<A HREF="URL" ONMOUSEOVER="***"></A>	(Javascript)
4.0*	**Mouse out Action**	<A HREF="URL" ONMOUSEOUT="***"></A>	(Javascript)
	Link to Email	<A HREF="mailto:@"></A>	
N, MS	**Specify Subject**	<A HREF="mailto:@?SUBJECT=***"></A>	(use a real question mark)
	Define Location	<A NAME="***"></A>	
	Display Image	<IMG SRC="URL">	
	Alignment	<IMG SRC="URL" ALIGN=TOP|BOTTOM|MIDDLE|LEFT|RIGHT>	
N1	**Alignment**	<IMG SRC="URL" ALIGN=TEXTTOP|ABSMIDDLE|BASELINE|ABSBOTTOM>	
	Alternate	<IMG SRC="URL" ALT="***">	(if image not displayed)
	Dimensions	<IMG SRC="URL" WIDTH=? HEIGHT=?>	(in pixels)
		<IMG SRC="URL" WIDTH=% HEIGHT=%>	(as percentage of page width/height)
	Border	<IMG SRC="URL" BORDER=?>	(in pixels)
	Runaround Space	<IMG SRC="URL" HSPACE=? VSPACE=?>	(in pixels)
N1	**Low-Res Proxy**	<IMG SRC="URL" LOWSRC="URL">	
	Imagemap	<IMG SRC="URL" ISMAP>	(requires a script)
	Imagemap	<IMG SRC="URL" USEMAP="URL">	
MS	**Movie Clip**	<IMG DYNSRC="***" START="***" LOOP=?>	
MS	**Background Sound**	<BGSOUND SRC="***" LOOP=?|INFINITE>	
	Client-Side Map	<MAP NAME="***"></MAP>	(describes the map)
	Map Section	<AREA SHAPE="DEFAULT|RECT|CIRCLE|POLY" COORDS=",,," HREF="URL"|NOHREF>	

N1	**Client Pull**	<META HTTP-EQUIV="Refresh" CONTENT="?; URL=URL">	
N2	**Embed Object**	<EMBED SRC="URL">	(insert object into page)
N2	**Object Size**	<EMBED SRC="URL" WIDTH=? HEIGHT=?>	
4.0	**Object**	<OBJECT></OBJECT>	
4.0	**Parameters**	<PARAM>	

DIVIDERS

	Paragraph	<P></P>	(closing tag often unnecessary)
	Align Text	<P ALIGN=LEFT|CENTER|RIGHT></P>	
N	**Justify Text**	<P ALIGN=JUSTIFY></P>	
	Line Break	 	(a single carriage return)
	Clear Textwrap	<BR CLEAR=LEFT|RIGHT|ALL>	
	Horizontal Rule	<HR>	
	Alignment	<HR ALIGN=LEFT|RIGHT|CENTER>	
	Thickness	<HR SIZE=?>	(in pixels)
	Width	<HR WIDTH=?>	(in pixels)
	Width Percent	<HR WIDTH="%">	(as a percentage of page width)
	Solid Line	<HR NOSHADE>	(without the 3D cutout look)
N1	**No Break**	<NOBR></NOBR>	(prevents line breaks)
N1	**Word Break**	<WBR>	(where to break a line if needed)

LISTS

Unordered List	<UL><LI></UL>	(before each list item)
Compact	<UL COMPACT></UL>	
Bullet Type	<UL TYPE=DISC|CIRCLE|SQUARE>	(for the whole list)
Bullet Type	<LI TYPE=DISC|CIRCLE|SQUARE>	(this & subsequent)
Ordered List	<OL><LI></OL>	(before each list item)
Compact	<OL COMPACT></OL>	
Numbering Type	<OL TYPE=A|a|I|i|1>	(for the whole list)
Numbering Type	<LI TYPE=A|a|I|i|1>	(this & subsequent)
Starting Number	<OL START=?>	(for the whole list)
Starting Number	<LI VALUE=?>	(this & subsequent)
Definition List	<DL><DT><DD></DL>	(<DT>=term, <DD>=definition)
Compact	<DL COMPACT></DL>	
Menu List	<MENU><LI></MENU>	(before each list item)
Compact	<MENU COMPACT></MENU>	
Directory List	<DIR><LI></DIR>	(before each list item)
Compact	<DIR COMPACT></DIR>	

BACKGROUNDS AND COLORS

	Tiled Bkground	<BODY BACKGROUND="URL">	
MS	**Watermark**	<BODY BGPROPERTIES="FIXED">	
	Bkground Color	<BODY BGCOLOR="#$$$$$$">	(order is red/green/blue)
	Text Color	<BODY TEXT="#$$$$$$">	
	Link Color	<BODY LINK="#$$$$$$">	
	Visited Link	<BODY VLINK="#$$$$$$">	
	Active Link	<BODY ALINK="#$$$$$$">	

(More info at http://werbach.com/web/wwwhelp.html#color)

Compendium 6 page 113

SPECIAL CHARACTERS

Special Character	&#?;	(where ? is the ISO 8859-1 code)
<	<	
>	>	
&	&	
"	"	
Registered TM	®	
Registered TM	®	
Copyright	©	
Copyright	©	
Non-Breaking Space		

(Complete list at http://www.uni-passau.de/%7Eramsch/iso8859-1.html)

FORMS

	Define Form	<FORM ACTION="URL" METHOD=GET|POST></FORM>	
4.0*	**File Upload**	<FORM ENCTYPE="multipart/form-data"></FORM>	
	Input Field	<INPUT TYPE="TEXT|PASSWORD|CHECKBOX|RADIO|FILE|BUTTON|IMAGE|HIDDEN|SUBMIT|RESET">	
	Field Name	<INPUT NAME="***">	
	Field Value	<INPUT VALUE="***">	
	Checked?	<INPUT CHECKED>	(checkboxes and radio boxes)
	Field Size	<INPUT SIZE=?>	(in characters)
	Max Length	<INPUT MAXLENGTH=?>	(in characters)
4.0	**Button**	<BUTTON></BUTTON>	
4.0	**Button Name**	<BUTTON NAME="***"></BUTTON>	
4.0	**Button Type**	<BUTTON TYPE="SUBMIT|RESET|BUTTON"></BUTTON>	
4.0	**Default Value**	<BUTTON VALUE="***"></BUTTON>	
4.0	**Label**	<LABEL></LABEL>	
4.0	**Item Labelled**	<LABEL FOR="***"></LABEL>	
	Selection	<SELECT></SELECT>	

	List		
	Name of List	<SELECT NAME="***"></SELECT>	
	# of Options	<SELECT SIZE=?></SELECT>	
	Multiple Choice	<SELECT MULTIPLE>	(can select more than one)
	Option	<OPTION>	(items that can be selected)
	Default Option	<OPTION SELECTED>	
	Option Value	<OPTION VALUE="***">	
4.0	**Option Group**	<OPTGROUP LABEL="***"></OPTGROUP>	
	Input Box Size	<TEXTAREA ROWS=? COLS=?></TEXTAREA>	
	Name of Box	<TEXTAREA NAME="***"></TEXTAREA>	
N2	**Wrap Text**	<TEXTAREA WRAP=OFF|HARD|SOFT></TEXTAREA>	
4.0	**Group elements**	<FIELDSET></FIELDSET>	
4.0	**Legend**	<LEGEND></LEGEND>	(caption for fieldsets)
4.0	**Alignment**	<LEGEND ALIGN="TOP|BOTTOM|LEFT|RIGHT"></LEGEND>	

TABLES

	Define Table	<TABLE></TABLE>	
4.0*	**Table Alignment**	<TABLE ALIGN=LEFT|RIGHT|CENTER>	
	Table Border	<TABLE BORDER></TABLE>	(either on or off)
	Table Border	<TABLE BORDER=?></TABLE>	(you can set the value)
	Cell Spacing	<TABLE CELLSPACING=?>	
	Cell Padding	<TABLE CELLPADDING=?>	
	Desired Width	<TABLE WIDTH=?>	(in pixels)
	Width Percent	<TABLE WIDTH=%>	(percentage of page)
4.0*	**Table Color**	<TABLE BGCOLOR="$$$$$$"></TABLE>	
4.0	**Table Frame**	<TABLE FRAME=VOID|ABOVE|BELOW|HSIDES|LHS|RHS|VSIDES|BOX|BORDER></TABLE>	
4.0	**Table Rules**	<TABLE RULES=NONE|GROUPS|ROWS|COLS|ALL></TABLE>	
MS	**Border Color**	<TABLE BORDERCOLOR="$$$$$$"></TABLE>	

MS	**Dark Border**	<TABLE BORDERCOLORDARK="$$$$$$"></TABLE>	
MS	**Light Border**	<TABLE BORDERCOLORLIGHT="$$$$$$"></TABLE>	
	Table Row	<TR></TR>	
	Alignment	<TR ALIGN=LEFT|RIGHT|CENTER|MIDDLE|BOTTOM>	
	Table Cell	<TD></TD>	(must appear within table rows)
	Alignment	<TD ALIGN=LEFT|RIGHT|CENTER VALIGN=TOP|MIDDLE|BOTTOM>	
	No linebreaks	<TD NOWRAP>	
	Columns to Span	<TD COLSPAN=?>	
	Rows to Span	<TD ROWSPAN=?>	
4.0*	**Desired Width**	<TD WIDTH=?>	(in pixels)
N3	**Width Percent**	<TD WIDTH="%">	(percentage of table)
4.0*	**Cell Color**	<TD BGCOLOR="#$$$$$$">	
	Header Cell	<TH></TH>	(same as data, except bold centered)
	Alignment	<TH ALIGN=LEFT|RIGHT|CENTER|MIDDLE|BOTTOM>	
	No Linebreaks	<TH NOWRAP>	
	Columns to Span	<TH COLSPAN=?>	
	Rows to Span	<TH ROWSPAN=?>	
4.0*	**Desired Width**	<TH WIDTH=?>	(in pixels)
N3	**Width Percent**	<TH WIDTH="%">	(percentage of table)
4.0*	**Cell Color**	<TH BGCOLOR="#$$$$$$">	
4.0	**Table Body**	<TBODY>	
4.0	**Table Footer**	<TFOOT></TFOOT>	(must come before THEAD>
4.0	**Table Header**	<THEAD></THEAD>	
	Table Caption	<CAPTION></CAPTION>	
	Alignment	<CAPTION ALIGN=TOP|BOTTOM|LEFT|RIGHT>	
4.0	**Column**	<COL></COL>	(groups column attributes)

4.0	**Columns Spanned**	<COL SPAN=?></COL>	
4.0	**Column Width**	<COL WIDTH=?></COL>	
4.0	**Width Percent**	<COL WIDTH="%"></COL>	
4.0	**Group columns**	<COLGROUP></COLGROUP>	(groups column structure)
4.0	**Columns Spanned**	<COLGROUP SPAN=?></COLGROUP>	
4.0	**Group Width**	<COLGROUP WIDTH=?></COLGROUP>	
4.0	**Width Percent**	<COLGROUP WIDTH="%"></COLGROUP>	

FRAMES

4.0*	**Frame Document**	<FRAMESET></FRAMESET>	(instead of <BODY>)
4.0*	**Row Heights**	<FRAMESET ROWS=,,,></FRAMESET>	(pixels or %)
4.0*	**Row Heights**	<FRAMESET ROWS=*></FRAMESET>	(* = relative size)
4.0*	**Column Widths**	<FRAMESET COLS=,,,></FRAMESET>	(pixels or %)
4.0*	**Column Widths**	<FRAMESET COLS=*></FRAMESET>	(* = relative size)
4.0*	**Borders**	<FRAMESET FRAMEBORDER="yes|no"></FRAMESET>	
4.0*	**Border Width**	<FRAMESET BORDER=?></FRAMESET>	
4.0*	**Border Color**	<FRAMESET BORDERCOLOR="#$$$$$$"></FRAMESET>	
N3	**Frame Spacing**	<FRAMESET FRAMESPACING=?></FRAMESET>	
4.0*	**Define Frame**	<FRAME>	(contents of an individual frame)
4.0*	**Display Document**	<FRAME SRC="URL">	
4.0*	**Frame Name**	<FRAME NAME="***"|_blank|_self|_parent|_top>	
4.0*	**Margin Width**	<FRAME MARGINWIDTH=?>	(left and right margins)
4.0*	**Margin Height**	<FRAME MARGINHEIGHT=?>	(top and bottom margins)
4.0*	**Scrollbar? Not**	<FRAME SCROLLING="YES|NO|AUTO">	

4.0*	**Resizable**	<FRAME NORESIZE>	
4.0*	**Borders**	<FRAME FRAMEBORDER="yes|no">	
4.0*	**Border Color**	<FRAME BORDERCOLOR="#$$$$$$">	
4.0*	**Unframed Content**	<NOFRAMES></NOFRAMES>	(for non-frames browsers)
4.0	**Inline Frame**	<IFRAME></IFRAME>	(takes same attributes as FRAME)
4.0	**Dimensions**	<IFRAME WIDTH=? HEIGHT=?></IFRAME>	
4.0	**Dimensions**	<IFRAME WIDTH="%" HEIGHT="%"></IFRAME>	

SCRIPTS AND JAVA

	Script	<SCRIPT></SCRIPT>	
	Location	<SCRIPT SRC="URL"></SCRIPT>	
	Type	<SCRIPT TYPE="***"></SCRIPT>	
	Language	<SCRIPT LANGUAGE="***"></SCRIPT>	
4.0*	**Other Content**	<NOSCRIPT></NOSCRIPT>	(if scripts not supported)
	Applet	<APPLET></APPLET>	
	File Name	<APPLET CODE="***">	
	Parameters	<APPLET PARAM NAME="***">	
	Location	<APPLET CODEBASE="URL">	
	Identifier	<APPLET NAME="***">	(for references)
	Alt Text	<APPLET ALT="***">	(for non-Java browsers)
	Alignment	<APPLET ALIGN="LEFT|RIGHT|CENTER">	
	Size	<APPLET WIDTH=? HEIGHT=?>	(in pixels)
	Spacing	<APPLET HSPACE=? VSPACE=?>	(in pixels)
N4	**Server Script**	<SERVER></SERVER>	

MISCELLANEOUS

	Comment	<!-- *** -->	(not displayed by the browser)
	Prologue	<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">	
	Searchable	<ISINDEX>	(indicates a searchable index)
	Prompt	<ISINDEX PROMPT="***">	(text to prompt input)
	Send Search	<A HREF="URL?***">	(use a real question mark)
	URL of This File	<BASE HREF="URL">	(must be in header)
4.0*	**Base Window Name**	<BASE TARGET="***">	(must be in header)
	Relationship	<LINK REV="***" REL="***" HREF="URL">	(in header)
N4	**Linked File**	<LINK TYPE="***" SRC="***"></LINK>	
	Meta Information	<META>	(must be in header)
	Style Sheets	<STYLE></STYLE>	(implementations vary)
4.0	**Bidirect Off**	<BDO DIR=LTR|RTL></BDO>	(for certain character sets)

Top Ten Mistakes in Web Design

By Jakob Nielsen, SunSoft Distinguished Engineer, May 1996

1. Using Frames

Splitting a page into frames is very confusing for users since frames break the fundamental user model of the web page. All of a sudden, you cannot bookmark the current page and return to it (the bookmark points to another version of the frameset), URLs stop working, and printouts become difficult. Even worse, the predictability of user actions goes out the door: who knows what information will appear where when you click on a link?

2. Gratuitous Use of Bleeding-Edge Technology

Don't try to attract users to your site by bragging about use of the latest web technology. You may attract a few nerds, but mainstream users will care more about useful content and your ability to offer good customer service. Using the latest and greatest before it is even out of beta is a sure way to discourage users: if their system crashes while visiting your site, you can bet that many of them will not be back. Unless you are in the business of selling Internet products or services, it is better to wait until some experience has been gained with respect to the appropriate ways of using new techniques. When desktop publishing was young, people put twenty fonts in their documents: let's avoid similar design bloat on the Web.

As an example: Use VRML if you actually have information that maps naturally onto a three-dimensional space (e.g., architectural design, shoot-them-up games, surgery planning). Don't use VRML if your data is N-dimensional since it is usually better to produce 2-dimensional overviews that fit with the actual display and input hardware available to the user.

3. Scrolling Text, Marquees, and Constantly Running Animations

Never include page elements that move incessantly. Moving images have an overpowering effect on the human peripheral vision. A web page should not emulate Times Square in New York City in its constant attack on the human senses: give your user some peace and quiet to actually read the text!

Of course, <BLINK> is simply evil. Enough said.

4. Complex URLs

Even though machine-level addressing like the URL should never have been exposed in the user interface, it is there and we have found that users actually try to decode the URLs of pages to infer the structure of web sites. Users do this because of the horrifying lack of support for navigation and sense of location in current web browsers. Thus, a URL should contain human-readable directory and file names that reflect the nature of the information space.

Also, users sometimes need to type in a URL, so try to minimize the risk of typos by using short names with all lower-case characters and no special characters (many people don't know how to type a ~).

5. Orphan Pages

Make sure that all pages include a clear indication of what web site they belong to since users may access pages directly without coming in through your home page. For the same reason, every page should have a link up to your home page as well as some indication of where they fit within the structure of your information space.

6. Long Scrolling Pages

Only 10% of users scroll beyond the information that is visible on the screen when a page comes up. All critical content and navigation options should be on the top part of the page.

Note added December 1997: More recent studies show that users are more willing to scroll now than they were in the early years of the Web. I still recommend minimizing scrolling on navigation pages, but it is no longer an absolute ban.

7. Lack of Navigation Support

Don't assume that users know as much about your site as you do. They always have difficulty finding information, so they need support in the form of a strong sense of structure and place. Start your design with a good understanding of the structure of the information space and communicate this structure explicitly to the user. Provide a site map and let users know where they are and where they can go. Also, you will need a good search feature since even the best navigation support will never be enough.

8. Non-Standard Link Colors

Links to pages that have not been seen by the user are blue; links to previously seen pages are purple or red. Don't mess with these colors since the ability to understand what links have been followed is one of the few navigational aides that is standard in most web browsers. Consistency is key to teaching users what the link colors mean.

9. Outdated Information

Budget to hire a web gardener as part of your team. You need somebody to root out the weeds and replant the flowers as the website changes but most people would rather spend their time creating new content than on maintenance. In practice, maintenance is a cheap way of enhancing the content on your website since many old pages keep their relevance and should be linked into the new pages. Of course, some pages are better off being removed completely from the server after their expiration date.

10. Overly Long Download Times

I am placing this issue last because most people already know about it; not because it is the least important. Traditional human factors guidelines indicate 10 seconds as the maximum response time before users lose interest. On the web, users have been trained to endure so much suffering that it may be acceptable to increase this limit to 15 seconds for a few pages.

Even websites with high-end users need to consider download times: many B2B customers access websites from home computers in the evening because they are too busy to surf the Web during working hours.

Other related information

Added by Jacob Palme, not part of Nielsens paper.

For this document with links to related information, see URL http://www.useit.com/alertbox/9605.html

See also Ten Quick Tips for Better Site Design

See also the following two papers in the June 1997 issue of the Internet World:

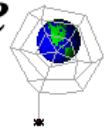
Space: The first Frontier - Keeping text and graphics in their place, by Wayne Bremser

and

Avoid the Five Cardinal Graphical Sins Say no to lazy images and bizarre colors, by David Busch

En stilguide för väven

av Karl-Johan Norén



Skriven av Karl-Johan Norén, kjnoren@hem3.passagen.se, maj 1997.

Detta är allt-i-ett versionen av min En stilguide för väven, lämplig för t ex utskrifter. Vill någon använda denna guide som kurs- eller undervisningsmaterial går det bra, förutsatt att den återges i sin helhet. Kopior för personligt bruk är också tillåtet. Vill någon använda det för annat ändamål, var vänlig kontakta mig.

Skriver du ut den här filen kan du också skriva ut Appendix A som innehåller samtliga URL:er till källorna som nämns i denna guide. Egentligen borde det vara vävläsarnas sak att ordna det, men då de flesta inte klarar av denna mycket enkla sak får ni hålla tillgodo med denna inte helt perfekta lösning.

Den senaste versionen av denna stilguide finns tillgänglig från <http://hem3.passagen.se/kjnoren/stilguide/>.

Introduktion

Detta är ett försök att skriva en någorlunda utförlig och komplett stilguide på svenska för vävsidor. Det är alltså inte något dokument för dig som vill lära dig HTML, utan snarare för dig som vill *använda* HTML på bästa sätt på väven.

Om du har läst andra stilguider, främst Tim Berners-Lees eller WDGs så kommer du säkert att känna igen dig, men jag har försökt att undvika att göra den här guiden till en översättning, även om mycket av det som nämns där återfinns här. Jag har också använt mina egna erfarenheter av drygt två års vävsidessnickrande, liksom mina erfarenheter som "surfare" på olika plattformar och med olika vävläsare.

Notera dock att det här är en *guide*, det är inte någon form av anvisningar om hur saker och ting måste göras. Se det som goda råd och idéer, men det är till syvende og sidst du som måste bygga och skriva dina sidor, och se till att de har en form som är anpassad till innehållet.

Innehållsförteckning

Några kommentarer

Jag använder konsekvent "väv" och sammansättningar med det istället för "webb", som i mina öron är oerhört ful svengelska. Likaså använder jag "rutor" istället för "ramar" (frames) eftersom analogin med vanliga fönster då blir tydligare. Ett fönster (dvs vävläsaren) kan ha flera rutor, men flera ramar?

Som så mycket annat på väven är den här stilguiden visserligen färdig, men den är inte klar och den lär aldrig bli det heller. Jag tar gärna emot kommentarer på saker som kan göras tydligare eller är felaktiga, liksom tips om nya ämnen eller frågor. Exempel saknas i många fall, men det ska förhoppningsvis bättra sig med tiden.

Att arbeta *med* HTML

Den troligen viktigaste, men samtidigt svåraste, lektionen om HTML är hur man arbetar *med* HTML, istället för *emot* HTML. Jag talar alltså inte om vilka verktyg man använder, eller vad de olika koderna står för, utan från vilken utgångspunkt man använder språket och vad man vill uppnå med det.

HTML är ett verktyg bland många andra, och precis som andra verktyg har det styrkor och svagheter - det är bra på vissa saker och dåligt på andra. För att förstå vad som är HTMLs styrkor och svagheter måste man förstå filosofin *bakom* HTML.

Struktur kontra presentation

HTML är ett språk för att beskriva *strukturen* i en text, vilken roll de olika delarna spelar i dokumentet som helhet. Man anger vad som är rubriker, vad som ingår i de olika styckena, vilka ord som är betonade osv utan att beskriva exakt *hur* rubrikerna ska synas eller styckena ska skiljas från varandra. Istället få HTML-tolken (som vanligen är en vävläsare) presentera sidan på ett sätt som passar mediet och förhoppningsvis besökaren (läsaren).

Fördelen med detta är att ett HTML-dokument kan presenteras i en mängd olika situationer: med en fullgrafisk vävläsare, på en ren textterminal, med en röstläsare osv. Det gör det också lätt att automatiskt skapa innehållsförteckningar, dispositioner och index från en vävsida, vilket bland annat sökmaskinerna drar nytta av.

Nackdelen är att man som författare ger upp kontrollen över exakt hur presentationen ska bli. Ansvaret för detta ligger istället på vävläsaren, som förhoppningsvis ska presentera innehållet på sidan i enlighet med besökarens önskemål, men tyvärr är de flesta vävläsare bedrövliga på detta.

Är layout möjlig?

Därmed inte sagt att det saknas layoutelement i HTML. En del fungerar som attribut till andra element (t ex ALIGN), en del kan användas för layout (t ex TABLE), en del har inget annat syfte (t ex FONT) - men försöker man uppnå en exakt kontroll över en sidas utseende med HTML så kommer man oundvikligen att slå huvudet i väggen. Det finns *inga som helst* garantier att sidan kommer att presenteras i enlighet med författarens önskemål - typsnittet kan saknas, skärmen kan vara för liten eller ha för få färger osv osv.

I det absolut värsta fallet gör man sidan oanvändbar för en större eller mindre grupp besökare. Börja därför med en god strukturell beskrivning av dina sidor, och använd sedan layoutdirektiv *utöver* detta. På så vis kan man behålla flexibilititen och anpassningsbarheten hos HTML.

Annars finns det en bättre lösning än layoutdirektiv direkt i HTML-koden: *style sheets*, som är betydligt mer kraftfulla för att beskriva en layout, kan anpassas för olika situationer och miljöer, och är gjorda med målet att både författaren och besökaren kan ange sina önskemål om presentationen. Dessa sidor använder för övrigt style sheets. Men precis som layoutdirektiv direkt i HTML-koden kan style sheets missbrukas.

Anpassningsbarhet

Mycket av det som nämns ovan och som jag tar upp senare i den här stilguiden syftar mot en och samma sak: *anpassningsbarhet*. En välgjord vävsida anpassar sig efter den plattform den visas på och besökarens behov. Om man tror att man behöver ha en separat kopia av alla sidor med enbart text så har man inte förstått poängen med HTML. Om man behöver använda en sådan behärskar man oftast inte HTML.

Eller som Tim Berners-Lee, skaparen av *World Wide Web*, sa i *Technology Review* i juli 1996:

> Anyone who slaps a "this page is best viewed with Browser X" label on a Web page appears to be

yearning for the bad old days, before the Web, when you had very little chance of reading a document written on another computer, another word processor, or another network.

Jag vill poängtera att klassiska grafiska kunskaper från layout eller reklamskapande inte är värdelösa på väven. Men väven är ett nytt och eget medium, skilt från de pappersbaserade. Att försöka använda klassiska grafiska kunskaper direkt på väven utan att först förstå vad väven har för egenskaper, möjligheter och begränsningar är förkastligt. Man måste först förstå vad väven innebär, vilka särskilda krav den ställer och vilka möjligheter den ger.

Fortsatt läsning

- What is HTML? An Opinion
  En sida av Michael H. Kelsey om filosofin bakom HTML och vad den innebär.
- The Telephone is the Best Metaphor for the Web
  Jakob Nielson ger sin motivering varför väven har mer gemensamt med telefonen än med TVn.
- Hints for Web Authors
  Warren Steel ger sin filosofi för hur han använder HTML.
- Publishing on the Web Is Different
  Jukka Korpela beskriver skillnaderna mellan att publicera information med vanliga medier och att göra det på väven.

Vävplatsen som helhet

En vävplats är en samling vävsidor som är grupperade tillsammans och vanligen behandlar ett gemensamt ämne. Sidorna i den här stilguiden kan sägas utgöra en (mycket liten) vävplats, som ingår i min egen personliga vävplats. Mina sidor är i sin tur en del i DSVs vävplats.

Genom att bygga upp en klar struktur över din egen vävplats gör du det enkelt för de som besöker en del av den att titta på andra delar med, och det blir också enklare att uppdatera och ändra sidorna.

Trädmodellen

Detta är den absolut vanligaste modellen för att bygga upp en vävplats, och också en av de enklaste. Sidorna ordnas hierarkiskt, med en sida som *rot* (hem-, topp- eller välkomstsida). Rotsidan har sedan länkar till de underliggande sidorna. Lägger man ett antal underliggande sidor i samma katalog (mapp eller *directory*) har man skapat en gren som i sig har en egen rotsida och underliggande sidor. Alla sidor har en länk till sidan direkt ovanför.

Även om man tycker att man bara har en eller ett fåtal sidor om ett ämne så är det ofta en bra idé att lägga dem i en egen katalog. På så vis ger man utrymme för framtida utökningar och nya sidor, och slipper krånglet med att flytta sidor som det finns massor med länkar till utifrån.

Hur grenarna ska organiseras är inte någon lätt fråga, särskilt inte om man har en flerspråkig vävplats. Är det bara en eller ett fåtal sidor som ska vara tillgängliga på två språk kan man samla dem i samma katalog, annars kan det löna sig att lägga dem var sin katalog. Se också till att namnen på sidorna är konsekventa, t ex att alla engelska sidor i en i övrigt svensk katalog börjar med `e-`.

Sidorna inom en gren kan också innehålla länkar till de andra sidorna inom samma gren, eller bara sidorna direkt före och efter. På så vis kan man läsa dem i sekvens, utan att behöva backa tillbaka till rotsidan.

Ge vävplatsen en identitet

Det ska märkas att sidorna i din vävplats hör ihop med varandra, och det ska inte vara någon tvekan om när man har lämnat den. Det viktigaste medlet för detta är att ha en konsekvent stil på sidorna. Sidhuvuden och sidfötter ska vara

gemensamma för sidorna, och fungera på samma sätt i dem. Gemensam information som länkar till resten av platsen, kontaktinformation till författaren, när sidan senast uppdaterades osv ska återfinnas på samma plats i alla sidorna. En annan god effekt av detta är att sidorna blir lätta att uppdatera och navigera i.

Gör sidorna lätta att navigera

Att sidorna är länkade till varandra löser bara halva problemet. Man måste också kommunicera *vart* länkarna går, och vilken *roll* de spelar i sidan som helhet. Oftast kan detta göras direkt i länktexten, men ibland kan det vara lämpligt att bygga upp en fast sekvens eller en "guidad tur" genom sidorna. Likaså bör man göra det möjligt för återkommande besökare eller personer med erfarenhet i ämnet att snabbt nå den information han eller hon är på jakt efter.

En användbar tumregel är trestegsprincipen: man ska kunna nå godtycklig sida i vävplatsen från vilken annan sida i den i tre steg, dvs via högst två mellanliggande sidor. Väl valda indexsidor gör det möjligt att uppnå detta mål även i större vävplatser utan särskilda hjälpmedel. Nästa steg kan vara en samlad innehållsförteckning för alla vävplatsens sidor eller en lokal sökmaskin.

Det vara möjligt att nå platsens välkomstsida, eventuella sökmaskiner och innehållsförteckningar direkt från alla sidor i vävplatsen. Ordlistor, copyright-meddelanden och andra gemensamma resurser kan hanteras på samma sätt.

Man uppnår ytterligare en fördel med detta. Det finns *inga som helst* garantier att besökaren kommer till en viss sida på det sätt du har förutsett eller väntar dig. Han eller hon kan ha använt en sökmaskin eller en länk utifrån. Länkarna till dina kringliggande sidor gör det då enkelt att nå resten av din plats.

Det är också en bra idé att ge information om vad som är nytt eller har ändrats. Man kan ha en blänkare på rotsidan, en särskild sida med nyheter eller både och. Sidan med nyheter kan också användas för att visa hela utvecklingen av vävplatsen.

Dokumentstorlek

I många fall kan ett ämne eller ett hypertextdokument svara mot flera sidor (den här stilguiden är ett exempel). Det finns både fördelar och nackdelar med att dela upp ett dokument i flera delar, men man bör definitivt sträva mot att varje sida behandlar ett avgränsat ämne. Att slå samman flera skilda ämnen ger lätt sidorna växtvärk, och kan förvirra besökare.

En nackdel med att ha flera små sidor är att varje sida måste hämtas för sig, och varje ny förbindelse över nätverket tar tid. Å andra sidan har stora sidor också nackdelar. De tar tid att hämta och det besökaren söker kan vara var som helst i sidan. Informationen blir också mer svåröverskådlig, och man tvingar besökaren att bläddra runt över stora sjok av information.

Oavsett om man delar upp dokumentet eller samlar det i en stor sida är det därför en fördel att först ha en översikt eller en innehållsförteckning. På så sätt kan en besökare enkelt avgöra om det du erbjuder är av intresse, och kan också snabbt nå det han eller hon är på jakt efter. I vissa fall kan man erbjuda både flera mindre ihoplänkade sidor och en stor sida med all information i ett. Den senare är användbar för t ex utskrifter.

I vilket fall bör man alltid se till att välkomstsidan för en vävplats är högst 60 kB stor, *inklusive* bilder, länkade style sheets, Java applets osv. Helst bör den inte vara mer än 30 kB stor, gärna mindre.

Kraven kan lättas upp för mer ämnesspecifika sidor, men 60 till 100 kB är i många fall en praktisk övre gräns för enbart HTML-koden och texten. Mer än så och sidan blir oöverskådlig. Man bör också ge en varning om en länkad sida (dvs text och bild) är över en viss storlek, t ex 40 kB. Att ge sidans storlek i kB inom parentes efter länken är fullt tillräckligt.

Fortsatt läsning

- Sub-Site Structure
  Jakob Nielsen tittar på hur stora vävplatser kan delas upp i många mindre för enklare navigering.

Innehållet i vävsidan

Innehållet är det som är värdefullt i din vävsida, och det du ska lägga ner mest arbete på. I den här diskussionen definierar jag innehåll i dess bredaste bemärkelse, både den information som ska kommuniceras, i viss mån hur den presenteras och metainformation som vem som skrev den, när den skrevs, sammanfattningar osv.

Metainformation

Det är en självklarhet att varje sida ska ge information om vem som skrev den, hur man kontaktar författaren och när den senast ändrades. Är en del av sidan inte helt färdig så bör man också tala om det, liksom eventuell copyright och andra juridiska aspekter.

Mycket av denna information kan med fördel samlas med ADDRESS-elementet, på samma plats i alla sidorna och med samma layout. Det gör den lätt att uppdatera och hitta. Länkar till relaterade sidor kan också samlas i anslutning med metainformationen.

Om man behöver använda längre copyright-meddelanden eller meddela en specifik policy är det en bra idé att bryta ut den till en egen sida, och länka till den från ett kort copyright-meddelande. På så vis belastar man inte varje sida med långa identiska texter, och uppdateringen blir också enklare.

Informationen om hur man kontaktar författaren kan med fördel göras som en mailto-länk till författarens e-postadress, eller ett formulär för att skicka e-post. Som länktext bör e-postadressen själv användas. Då undviker man att kontaktinformationen försvinner om sidan skrivs ut eller sparas i textformat, och någon som vill använda ett separat e-postprogram kan använda klippa och klistra för adressen.

Använd HTML rätt

En viktig egenskap i HTML är att man gör strukturen i en sida *explicit*. Istället för att använda ett radindrag eller en blankrad för att säga att nu är det ett nytt stycke, så använder man elementet `P`. Istället för att använda t ex Helvetica i 24 punkter för en rubrik så använder man elementet `H1`. På samma sätt markerar man längre citat, rubriker, listor osv.

Detta har flera trevliga egenskaper. En sida kan presenteras på ett adekvat sätt på en stor mängd plattformar och miljöer. En röstläsare kan säga `nytt stycke` eller göra en paus för att meddela styckebyte. En grafisk läsare kan använda radindrag. En textläsare kan använda blankrad, osv.

Ett annat viktigt resultat är att sökmaskiner kan använda rubriker och andra element för att indexera din sida korrekt. Undvik definitivt att använda t ex `BLOCKQUOTE` för att skapa en indentering. Visserligen gör många vävläsare en indentering, men vävläsaren är i sin fulla rätt att säga `jag citerar` eller sätta `>` före varje rad i citatet, som är den standard som används i e-post och i nys.

Låt sidan stå på egna ben

Samtidigt som länkar, sökmaskiner och bokmärken gör väven så rik som den är så medför de också krav. Du kan aldrig vara säker på att en besökare har läst en sida som föregår den de är på. Sidan måste vara användbar även i dessa fall - den måste stå på egna ben.

Först och främst ska sidan ha en användbar och tydlig titel. `Introduktion` säger inte speciellt mycket, men `Projekt Runeberg: Introduktion` gör det. En väl vald titel är också värdefull för sökmaskiner och personer som länkar till din sida.

Låt heller inte texten på sidan utgå från ett underförstått sammanhang. Se istället till att brödtexten presenterar sig själv, utan hjälp av rubriker och tidigare sidor. Se också till att det finns länkar till de omkringliggande sidorna. På så vis spelar det ingen roll till vilken sida en besökare kommer först - han eller hon kan fortfarande avgöra om dina sidor är användbara och kan få hela ditt budskap.

Fortsatt läsning

- A Web Site is a Harsh Mistress
  Diane Wilson tittar på hur en vävsida och en besökare fungerar ihop.
- Writing for the Web
  Ytterligare en sida i Jakob Nielsens serie *The Alertbox*. Denna studerar de krav på språket väven ställer.

Att hantera länkarna

Om själva sidorna i din vävplats är brödet, så är länkarna smöret. Länkarna är det som binder samman sidorna och ser till att din vävplats inte är en isolerad ö skild från omvärlden. Det är ingen större överdrift att påstå att länkarna *är* väven. HTML-dialekter och sidor kommer och går, men länkarna består.

Namnge dem rätt

En av de viktigaste principerna när man konstruerar ett informationssystem är att den relevanta informationen ska vara lätt att identifiera, samtidigt som den inte ska störa den övriga informationen runtomkring.

Länkar med namnet `klicka här` eller liknande bryter mot båda principerna ovan. Det finns två intressanta informationsbitar för en länk: att det finns en länk till någonting, och vad länken berör, men dessa två är inte i direkt anslutning till varandra. Man måste titta på området före och efter själva länken för att få dess funktion.

Dessutom blir meningarna ofta klumpigare och mer svårlästa genom användning av `klicka här`. Visst, det är lätt att skriva med "klicka här", men målet är väl knappast att ha en lättskriven vävplats, utan en *lättläst och lättanvänd* en? Eller hur?

Länknamn bör vara korta och deskriptiva. Tre eller fyra ord räcker i många fall. Om sidan man länkar till är stor kan man ange dess storlek (text och bild tillsammans) inom parentes efter länken. En vanligt minimivärde som nämns för detta är 40 kB, men man bör definitivt märka alla länkar till sidor på 100 kB eller mer.

Ord man ska försöka undvika i länknamn är bla a `information`, `mer`, `här`, `bakåt`, `tillbaka`, `framåt` och `hem`. De fyra första säger egentligen ingenting om vad länken handlar om, de fyra senare är i praktiken reserverade för funktioner i vävläsaren. Sidan som `tillbaka` pekar på är ofta inte densamma som den sida som besökaren var på senast.

Se till att de pekar rätt

Döda länkar är trista länkar. Det enda man får se är ett litet meddelande med `HTTP 404 File Not Found` på grå bakgrund. Förhoppningsvis står och faller inte din sida med dess länkar, men man bör definitivt ha för vana att kontrollera dem regelbundet och fixa alla som pekar fel på något sätt.

Men en länk kan också vara felaktig och fortfarande peka på någonting. Två fall är väldigt vanliga här. Det första är att det finns kvar en sida med en notis om att den nya adressen är si-och-så, komplett med en länk dit. Ordna dessa med, du kan aldrig veta när en sådan sida försvinner, och du tvingar dina besökare till att följa ytterligare en länk.

Det andra fallet är mindre tydligt, men nog så vanligt. Det är att en del av URL:en pekar fel, men servern som sidan finns på korrigerar felet med en s k *redirect*. Vanliga sådana fel är att länken pekar på en maskin som sidan tidigare låg på, eller att en avslutande "/" saknas i URL:en. Visserligen korrigeras felet, men det tar tid och nätverksresurser helt i onödan. Vid minsta tvekan, klipp och klistra in URL:en direkt från vävläsaren!

index.html eller inte index.html

`index.html` `index.htm`

komplicerad. När sökvägen i en URL avslutas med "/" pekar den på en katalog, och vad vävservern skickar tillbaka beror på hur den är inställd. Oftast tittar den efter om det finns en fil som heter t ex `index.html`, `default.html` eller `Welcome.html`, och skickar iväg den, om den finns. Denna fil kan kallas för en *indexfil*.

Finns det inte en fil med ett sådant namn, eller om listan över standardfilnamn är tom, så kan vävläsaren antingen skicka tillbaka en lista över alla filer i katalogen, eller helt enkelt meddela att man inte får titta i den. För att veta exakt på vilket sätt din vävserver fungerar och vilka standardnamn den har så får kontrollera med din vävansvarige eller dokumentationen, eller helt enkelt testa själv.

I vilket fall bör du vara konsekvent vad gäller dina URL:er. Antingen använder du URL:er som pekar på katalogen, eller som pekar direkt på indexfilen. Använder du båda finns det risk för att dina besökare blir förvirrade, eller tvingas ladda ner *exakt* samma sida *två* gånger. Du kan använda en länk i formen `<A HREF="./">länktext</A>` för att peka på den aktuella katalogen.

Länklistor

Det är lätt att göra jättelistan Allan över all världens länkar, men att hålla den uppdaterad är definitivt *inte* lätt. Dessutom finns det redan stora tjänster för detta i form av Yahoo, olika ämnesspecifika index eller sökmaskiner. Att ens försöka konkurrera med dem är dödfött.

Då är det en bättre idé att göra en relativt kort och personlig lista över länkar - sidor som du själv besöker relativt ofta. På så vis kan du också frigöra din lista med bokmärken i vävläsaren för t ex sidor du ska ta en närmare titt på lite senare. Ännu bättre blir det om du läggger till dina egna kommentarer om sidorna.

Om det saknas en bra lista med länkar till sidor inom ett område du är intresserad av kan du naturligtvis slå slag i saken och försöka göra en mer eller mindre komplett sådan - men tänk på att det är *mycket* jobb.

Bilder på väven

Hittills har vi mest behandlat text, men bilder fyller också en viktig funktion på väven. De används som prydnader för att göra en sida mer attraktiv, som navigationshjälpmedel eller som en integrerad del av en sidas innehåll.

Bilder i allmänhet

Det första du måste tänka på är att bilder är *stora*. Även en liten ikon motsvarar lätt hundra ord eller mer. Visserligen säger en bild lika mycket som tusen ord, men i en dator är är den ofta många gånger större. Det är heller inte alla vävläsare som kan visa bilder överhuvudtaget, eller inte kan visa dem direkt på sidan. De kan också göra att sidan tar längre tid att presenteras för mottagaren.

Samtidigt är de ofta också ovärderliga. De gör sidan mer attraktiv och de kan enkelt förklara eller visa saker som inte går att säga i ord. Men hanteringen av bilder på väven kräver planering, noggrannhet och att man tänker efter före.

Det viktigaste är att göra bilderna så utrymmessnåla som möjligt. För att göra detta kan man reducera antalet färger, klippa bort delar som inte behövs eller minska bildens fysiska storlek. Har man ett galleri eller behöver en stor bild skapar man miniatyrer av som länkar till motsvarande fullstora bild. Miniatyerna ger en snabb översikt av vad du har att erbjuda, laddas ner snabbare och låter besökaren själv välja ut vilka bilder han eller hon vill se.

Nästa del är återanvändning. Lägg dina bilder samlat i en katalog, och länka sedan in bilderna därifrån för alla dina sidor. Om du använder samma bild för samma funktion på dina sidor ritas dels sidan upp snabbare eftersom vävläsaren redan har laddat ner bilden en gång, och du ger också dina sidor en starkare identitet. Om det finns ett centralt bildbibliotek i din vävserver ska du definitivt utnyttja det.

Vävläsaren måste också veta hur stor bilden är innan den kan rita upp texten runtomkring. Du kan reservera

`HEIGHT` `WIDTH`

bilden med de här värdena - dels får du *alltid* bättre resultat om du gör det själv, dels är inte vävläsaren tvungen att skala om bilden. Den kan rita om sidan istället.

Sist, men definitivt inte minst, kommer ALT-texten. Ange *alltid* en ALT-text till *alla* dina bilder. Jag tar upp mer om ALT-texten nedan.

Logotyper

Om ett företag eller en organisation har en logotyp ska den naturligtvis återfinnas på alla sidorna. Om man dessutom gör den till en länk till rotsidan fyller den ytterligare en funktion. Ofta kan man också ersätta den vanliga rubriktexten, t ex `Välkommen till XYZ` med logotypen. Om man då placerar bilden *i* rubriken och ger den en bra ALT-text kommer den också att hanteras rätt av sökmaskiner och (förhoppningsvis) vävläsare som inte laddar ner bilderna. De här sidorna använder för övrigt den metoden.

Ikoner för navigation

Det finns flera skäl till att använda ikoner istället för text för de grundläggande gemensamma länkarna. En ikon syns oftast bättre än en text, vilket är en fördel. Detta kräver dock att ikonerna används konsekvent, dvs samma ikon används för `Nästa`, `Topp` osv på alla sidor. Konsekvensen har naturligtvis andra fördelar, som att ge sidorna en identitet och att vävläsaren behöver ladda ner färre bilder. Om man bygger upp en knapprad kan man låta en ikon som pekar på sidan man är på vara kvar, men utan en länk.

ALT-texterna är extra viktiga här - det är inte roligt att se en räcka med `[LINK]` över hela sidans bredd. Ett problem här är att ikonerna ofta är relativt små, vilket innebär att om du ger ikonen attributen `HEIGHT` och `WIDTH` så kan det hända att ALT-texten inte får plats i en del vävläsare. En lösning är att slopa `HEIGHT` och `WIDTH` i dessa fall, en annan att skriva extremt korta ALT-texter, t ex `>` istället för `Nästa`. Den förra lösningen kan fungera bra om ikonerna är på botten av sidan. Den senare lösningen ger problem för röstläsare. Attributet `REL` kan troligen användas för att åtgärda detta, men det är endast en handfull vävläsare som stöder `REL`.

Prydnader

I många fall används bilder bara eftersom de gör sidan snyggare och mer attraktiv. I många fall är det inga problem så länge man tänker på ALT-texter och storleken på bilderna, men tyvärr finns det inga bra och effektiva sätt att använda bilder istället för punkter i listor eller som avdelare. Style sheets går att använda, men stödet för detta fortfarande dåligt.

ALT-texten

ALT-texten är ett av dina viktigaste verktyg för att skapa anpassningsbara vävsidor. Det är långtifrån endast blinda personer eller personer vars vävläsare inte kan visa bilder överhuvudtaget som drar nytta av dem. En stor andel användare av grafiska vävläsare surfar normalt med avstängd automatisk nedladdning av bilder (de flesta uppskattningar pekar på ungefär en tredjedel). En textläsare som Lynx kan hämta och spara bilder, eller visa dem i en separat bildläsare - som oftast är *bättre* på detta än en grafisk vävläsare.

Tyvärr är ALT-texten begränsad på många sätt, främst beroende på hur IMG-elementet är konstruerat. Det kan innehålla max 1024 tecken och, vad värre är, inte heller några HTML-element. Du kan däremot lägga *hela* bilden i ett eget element, t ex en rubrik, och ALT-texten kommer då att "ärva" det elementet. Det nya elementet `OBJECT` i Cougar löser de problemen, men det stöds mycket dåligt.

I vilket fall ska ALT-texten i nästan samtliga fall *ersätta* bildens *funktion*, inte beskriva bilden. För rent dekorativa bilder är `ALT=""` effektivt. För avdelare är `ALT="----"` bra (repetera -- tills du har 60 eller så), osv.

Fortsatt läsning

- Text-friendly authoring
  Alan Flavell diskuterar hur ALT-texten ska hanteras samt för- och nackdelarna med klickbara bilder.

- Image Use on the Web
  En kort text från WDG om bildformat och bildhantering.
- On images, especially in the Web context
  En grundlig genomgång av hur bilder ska användas på väven.

Objekt - ljud, Java mm

Ett stort användningsområde för HTML och väven som uppstått nyligen är som bärare av andra medier: ljud, animationer, filmer, Java-applets, JavaScript, ActiveX-kontroller osv osv. Jag kallar alla dessa gemensamt för *objekt*. Mycket av det som gäller för vanliga bilder gäller också för dessa, men jag ska poängtera några saker som blir extra viktiga.

Begränsat stöd

Många typer av objekt är beroende av insticksprogram (plug-ins), hjälpapplikationer eller så är de begränsade till endast ett fåtal antal plattformar. Många stänger dessutom av tekniker som Java och JavaScript av olika skäl. Du bör därför aldrig göra sidan *beroende* av de här teknikerna. Om ditt innehåll kräver att du använder objekt av något slag så bör sidan i övrigt meddela det på ett hövligt sätt. Att endast ha en text med `Din vävläsare är skit` eller motsvarande skadar i slutändan bara dig.

Gör istället ditt bästa för att *övertyga* besökaren att ditt innehåll behöver objektet. Många har t ex möjlighet att köra Java-applets, men har stängt av det av olika skäl, eller måste starta upp en annan vävläsare. Om du meddelar vad din applet gör så ger du dem anledning att sätta på det just för din sida. Samma angreppssätt kan också användas för andra typer av objekt.

Animationer och annat som rör sig

Vår hjärna är programmerad för att lägga märke till saker som rör sig. En sak som blinkar, ändrar form eller rör sig på något annat sätt får automatiskt en högre prioritet än något som inte gör det. En *mycket* högre prioritet.

Visst, det drar uppmärksamheten till sig, men din vävsida är inte en neonskylt som man ska lägga märke till när man går förbi. Din besökare *tittar* redan på din sida. Ditt mål är inte att dra uppmärksamheten till den, utan att kommunicera. Om då din animation eller blinkande text *stör* besäkaren och hindrar honom eller henne från att läsa texten i lugn och ro motverkar den snarast sitt syfte!

Animationer är inte fel, men ofta är en knappt märkbar animering eller en som bara går en gång effektivare. Kittla intresset och nyfikenheten, dränk det inte!

Ljud

Precis som bilder har ljud sin plats på väven. En vävsida som behandlar musikinstrument är knappast komplett utan exempel på hur instrumenten låter, en sida om en musikgrupp kan ha korta bitar ur gruppens låtar, och ofta kan man ha en enkel välkomsthälsning.

Men det är alltid besökaren som ska ha valet om han eller hon vill lyssna på ljudet! Om man surfar mitt i natten medan resten av familjen sover vill man knappast höra plötsliga eller höga ljud, likaså inte ifall man sitter i en gemensam labsal i en skola. Ge istället en enkel länk till ljudet eller ljuden.

Det finns naturligtvis fall då ljud som laddas ner automatiskt kan vara effektfulla. T ex kan en vävsida om *The Beatles* spela upp inledningsackordet från *Help!*. Men tänk då på att ljudet *måste* vara litet, så att det dyker upp kvickt. Ju kortare det är desto bättre. Sådana här exempel är också så gott som alltid undantagsfall.

Saker man bör undvika

Det jag sagt tidigare om att det här är en *guide* och inte en anvisning gäller extra mycket för denna del. De tekniker och metoder jag tar upp här är snarare sådana att de antingen kan göras bättre på andra sätt, eller så är de kända för att orsaka problem. Vill du använda dem så får du göra det, men mitt råd är att du vet exakt *varför* du gör det, och varför du *behöver* göra det, innan du sätter dig ner och gör det.

Rutor

Rutor (eller ramar som det vanligen kallas) har ett antal stora problem med dagens implementering, både designmässigt och hur de hanteras av vävläsarna. Det är inget fel på själva idén att ha flera samtidiga *vyer* av en datamängd, tvärtom, men det sätt som den realiserades på av Netscape är ett klassiskt fall av BAD (Broken As Designed).

Det största problemet är att URL:er slutar fungera - de pekar inte på den aktuella vyn utan den *ursprungliga* vyn - den som definierade rutorna. *Det finns ingen relation från de enskilda sidorna tillbaka till den sida som definierade rutorna*. Hanteringen av rutorna är också helt inriktad på visuell presentation helt utan plattformsoberoende.

Häftig färghantering

Svarta bakgrunder är ofta snygga, men är det lätt att läsa texten på dem? Använder man dessutom färgad text i någon mån blir det hela ännu mer svårläst, och personer som lider av någon sorts färgblindhet kan få svart text på svart bakgrund. Närmare tio procent av alla män lider av någon rödgrön färgblindhet i någon grad.

En vävsida ska kunna vara användbar i svartvitt precis som en färgfilm kan visas på en svartvit TV. Detta oavsett om begränsningen ligger i människan eller i maskinen.

Det man särskilt måste uppmärksamma är att kontrasten mellan texten och bakgrunden ska vara stor, att länkarna är lätta att urskilja och att du inte riskerar att göra någon del av texten osynlig. Det sista är en risk om man inte anger *alla* färger som sidan använder. Ange antingen *inga* färger, eller *alla* färger. Om du använder en bakgrund ska du se till att texten fortfarande är lättläst på bakgrunden, och att dina valda färger matchar den.

Föråldrad information

Som jag nämnde tidigare i den här stilguiden så finns det få saker som är så trista som länkar som inte leder någonstans. Men döda länkar är bara en liten del av problemet - en sida med föråldrad information är minst lika illa.

Det är minst lika viktigt att sidorna hålls uppdaterade och fräscha som att de finns överhuvudtaget. Tänk också på att hålla dem uppdaterade är ett stort arbete som tar tid. Det är ytterligare ett skäl till att starta smått - genom att utveckla vävplatsen i lugn takt inser man hur mycket underhåll den behöver, och sitter inte med en mängd sidor som ingen ordnar ta hand om.

Hårdkodade sidor

Redan tidigt med NCSA Mosaic började det dyka upp sidor som var noggrant grafiskt designade för att uppnå en viss presentation. De utgick ofta från att alla använde samma bredd på vävläsarens fönster, samma vävläsare och exakt samma typsnitt. Stämde inte detta blev allt en enda röra. I takt med att vävläsare som Netscape införde mer element för layout och att olika HTML-verktyg började marknadsföras som WYSIWYG har det här blivit allt vanligare.

Vanliga exempel på detta är att använda FONT-elementet för att skapa rubriker och bestämma typsnitt, och att använda tabeller för att skapa en specifik layout av sidan. Problemet med dessa sidor är att de slänger bort den viktigaste egenskapen en vävsida har: anpassningsbarheten.

Istället för att låta vävläsaren presentera sidan och dess innehåll i enlighet med de möjligheter den har så blir sidan i många fall endast användbar i specifika situationer. En hårdkodning innebär inte bara att man förutsätter en specifik vävläsare, utan vanligen också att automatisk laddning av bilder är på, att vissa typsnitt är installerade, en viss storlek på vävläsarens fönster, ett visst färgdjup på skärmen, en viss storlek på texten osv osv.

Resultatet är en sida som dels blir svårare att hålla aktuell och uppdaterad, dels inte blir användbar för många. Tänk också på att även om *din* vävläsare inte presenterar ett visst element som du vill ha det, så kanske en annan vävläsare presenterar det rätt, och några besökare gillar det sätt du ogillar.

Istället för att använda `<BR>` och en liten gif-bild för att skapa ett radindrag så klaga hos tillverkarna av vävläsaren att deras vävläsare inte kan anpassas för att visa `<P>` som du vill ha det! De flesta vävläsare är faktiskt bedrövliga på att presentera sidor, och att fixa designen i HTML-koden är en återvändsgränd.

Stora tabeller

En vanlig variant av de hårdkodade sidorna är de som utnyttjar en eller ett par stora tabeller för att visa innehållet. Detta innebär inte bara de vanliga nackdelarna med hårdkodade sidor, utan medför också problem som är inbyggda i hur tabeller hanteras i HTML.

Om man går till en sida som har en enda stor tabell, t ex Pagina med en grafisk vävläsare som stöder tabeller så ser du att även när en stor del av sidan har laddats ner, så syns det knappast någonting på skärmen! Orsaken är att tabellen i sin helhet måste vara tillgänglig för vävläsaren innan den kan rita upp den. Ingår det bilder kan det dröja ännu längre om de saknar attributen `HEIGHT` och `WIDTH`.

Det här är något som definitivt inte är lyckat. Visserligen är det ok om det dröjer med att sidan dyker upp i sin helhet, men det bör åtminstone dyka upp *någon* text inom tio sekunder. Kan din sida inte klara att få fram någon rubrik och lite text på den tiden är det risk att besökaren helt enkelt tröttnar.

I HTML 4.0 finns det en ny och förbättrad tabellhantering som låter vävläsaren rita upp tabellen inkrementellt, dvs allteftersom den kommer in. Men knappast någon vävläsare stöder detta idag, och en sådan tabell blir knappast heller lika snygg som en som ritas upp på konventionellt sätt. Den är därför knappast ett alternativ för layout idag.

Ett annat vanligt problem med att göra sidan som en tabell är att man tvingar besökaren att bläddra horisontellt för att läsa innehållet, något som är *helt* förkastligt. Låt din besökare *själv* bestämma vad som är lämplig fönsterbredd för honom eller henne!

Fortsatt läsning

- This page optimized for ...
  Jahn Rantmeister går igenom varför hårdkodade sidor är en dålig idé.
- Color Perception Issues
  Diane Wilson tittar på färger och färgseendet.
- Top Ten Mistakes in Web Design
  Ytterligare ett dokument i Jakob Nielsens Alertbox-serie.

Kontrollera sidorna

Det finns få saker som är så enerverande som att upptäcka att man måste göra om en vävsida när en ny version av ens vävläsare dyker upp. Till skillnad från människor är också datorer dumma, så gör du misstag i din HTML-kod kan du lätt göra hela sidan oläslig.

Validera din sida

Att validera en HTML-sida innebär att man kontrollerar om sidans HTML-kod är syntaktiskt korrekt och rätt stavat. Enkelt uttryckt gör man en grammatisk kontroll av HTML-koden i enlighet med ett antal formella regler.

Varje standard av HTML har sin egen uppsättning formella regler. Vilken standard som ska användas görs med en

doctype-deklaration allra först i sidan - före `<HTML>`. Vilken dokumenttyp (DTD) du ska använda bestäms av vilka koder du använder, men det vanligaste är nog den för HTML 3.2 Wilbur: `<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">`.

Om man ska rätta alla fel eller ej är en samvetsfråga. I vissa fall kan felaktig HTML vara önskvärd eller till och med nödvändig. En god tumregel är att om du *vet* vad som kommer att ge ett fel, *varför* det ger ett fel och *varför* du använder konstruktionen ifråga i förväg så kan det passera. Annars inte.

Låt en kompis kolla sidorna

Man blir lätt blind inför sina egna alster. Det är en gammal sanning för alla författare, och minst lika sann på väven. På det här sättet kan du undvika alla sorters fel i innehållet i dina sidor, både språkliga och faktamässiga.

Ett annat sätt är att lägga undan texten i några dagar och sedan gå tillbaka till den för att läsa om den och rätta felen. Det här är inte lika bra som att låta en kompis kolla dem, men det fungerar. Likaså är det ofta önskvärt med en stavningskontroll (det ska sägas att jag själv är en slarver vad gäller detta). HTML-koderna kan ställa till problem, men det går i nödfall att öppna sidan i sin vävläsare och sedan spara texten i en vanlig textfil.

Använd mer än en vävläsare

Olika vävläsare reagerar olika på olika konstruktioner. Detta gäller särskilt då de råkar ut för felaktig HTML, men också i andra sammanhang. I vilket fall är "Det fungerar bra i Netscape!" en helt oacceptabel ursäkt för att en besökare som använder en annan vävläsare inte kan använda sidan! Tänk också på att inte bara vävläsarnas namn skiljer dem åt. Olika versioner av samma vävläsare eller samma vävläsare på olika plattformar skiljer sig också, i vissa fall minst lika mycket.

Hur många vävläsare man ska använda är en svår fråga, men att testa dem på Netscape och Internet Explorer är som att säga "vi spelar båda sorters musik: country and western". Både Netscape och Internet Explorer bygger i hög grad på gamla NCSA Mosaic, och beteendet i Internet Explorer har till stora delar skapats efter det i Netscape.

Som minimum bör man ha kollat sidorna dels i en ren textläsare och en grafisk läsare, förhoppningsvis också med automatisk laddning av bilder avslaget. Opera är särskilt värdefull här då den dels har ett riktigt textläge, dels att buggarna i den är helt olika de som finns i Netscape och Internet Explorer, men den finns bara till Windows. Lynx är en ren text-läsare som däremot finns till i stort sett alla plattformar, även MacOS.

Någonstans mellan en kontroll i en vävläsare och en riktig validering kommer olika typer av heuristiska verktyg. De kallas ofta för *linters* efter programmet **`lint`** för att kontrollera C-kod. Det finns flera verktyg, men det vanligaste heter rätt och slätt Weblint. Tyvärr förekommer det en hel del förvirring om vad som är en validator och vad som är ett heuristiskt verktyg, och många del heuristiska verktyg marknadsförs även som validatorer. Det innebär inte att de inte är värdefulla, men man bör vara klar över skillnaden.

Fortsatt läsning

- The Kinder, Gentler Validator
  Ett oumbärligt verktyg för att kontrollera din HTML-kod. Denna online-validator ger hjälpsamma felmeddelanden och har tillgång till ett stort antal DTD-er.
- Weblint
  Hemsidan för Weblint, ett Perl-program för att göra enklare kontroller av HTML-kod.
- Lynx
  Lynx är en textbaserad men mycket kraftfull vävläsare, tillgänglig till de flesta plattformar, även MacOS och Windows.
- Opera En relativt ny vävläsare till Windows, med många intressanta finesser.

Källor och fortsatt läsning

Det här är en lista på några av de sidor jag använt som källmaterial och som stilguider för mina egna sidor. Av naturliga skäl är det bara ett axplock, men jag har försökt att plocka russinen ur kakan.

Artiklar, referenser och essäer

- The Web Design Group
  WDG har en av de absolut bästa platserna med resurser för författare på väven, med utmärkta referenser för HTML 3.2 Wilbur och CSS1, FAQer, stilguider med mera. *Rekommenderas!*

- The Alertbox
  The Alertbox: Current Issues in Web Usability är en samling essäer av Jakob Nielsen, SunSoft. De tar bland annat upp trender, design i dess vidaste bemärkelse och väven som medium.

- Designing For the Web
  Diane Wilson tar bl a upp frågor kring färger och färgseendet - viktigt då uppemot tio procent av alla män lider av någon sorts färgblindhet.

- Referensböcker
  Under denna inte speciellt upphetsande rubrik har Eva von Pepel den troligen största och bästa samligen sidor om HTML och style sheets som finns på svenska. Tyvärr är det ofta rätt tekniskt och illa skrivet.

- The World Wide Web Consortium
  Standardiseringsorganet för HTML och andra vävrelaterade tekniker och standarder.

Last modified/Senast ändrad: 09 Jan 1999

Font Size Comparisons as shown on Screen

Jacob Palme <jpalme@dsv.su.se>
Last change: 2001-02-01

Fonts marked with an asterisk in the table below are unreadable with some web browsers on some platforms.

With Explorer 5.0,
10 pt corresponds to 12 px and
12 pt corresponds to 16 px.

The Multipart/Related Content Type

The Multipart/related content type is designed when you are sending several files, which are related by URL-links. It is used, for example, to send HTML, SGML and XML with embedded pictures or applets as separate files.

Each file is a separate body parts. Each body part is labelled by either Content-ID or Content-Location. The URL referring to the body part from another body part, is of the URL type "cid:" to refer to a Content-ID, or can be any kind of URL (absolute or relative) to refer to a Content-Location with the same content.

Example (abbreviated):

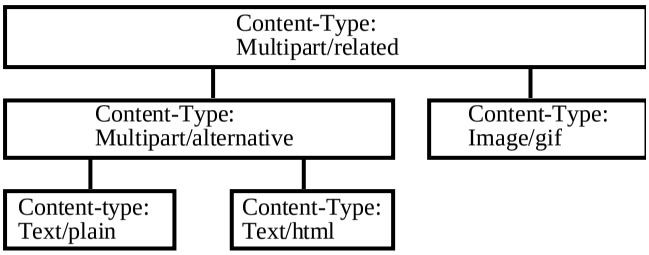
`Content-type: Multipart/related`	The compound object of the HTML text and the embedded message.

`Content-Type: Text/html`
 	
`<IMG SRC="cid:1*foo@bar.net>`
 	
`<IMG SRC="picture.gif">`
	The main text in HTL format.
	
	Link to an embedded image using a "cid:" type URL.
	Link to an embedded image using a relative URL.

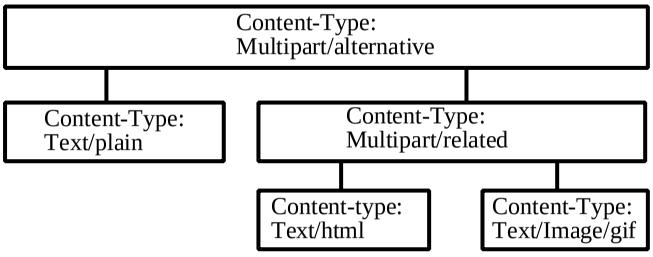
`Content-Type: Image/gif` `Content-ID: 1*foo@bar.net`	The first embedded image, identified by a Content-ID.
`Content-Type: Image/gif` `Content-Location: picture.gif`	The second embedded image, identified by a Content-Locdation URL.

Since some mailers do not support this, messages are usually sent using multipart/alternative, with plain text in the first branch and HTML in the second branch. This can be done in two ways:

With the multipart/alternative inside the multipart/related:



With the multipart/alternative outside the multipart/related:



Some mailers send messages using each of these methods, so a good mailer will have to be able to receive messages in both formats.

Why Bitmapped Screen Dumps used as OHs Sometimes get Ugly

By professor Jacob Palme,
Department of Computer and Systems Sciences
KTH Technical University

Abstract

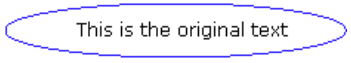
Computer screen dumps will sometimes look very ugly when you view then in manuals and help texts. This document analyzes why and suggest how to avoid this problem.

The Problem

When you dump a computer screen content in a bitmapped file, this file will usually get the same resolution as is used for the computer screen, 72 pixels/ inch or 96 pixels/inch.

If you then use this bitmapped picture in an overhead or in a document, which is to be read on a computer, the picture looks OK if it is shown at the same resolution as it was produced. If, however, the resolution is changed, the picture can sometimes become very ugly.

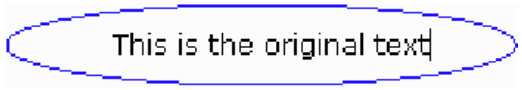
If, for example, the original bitmapped picture looked like this:



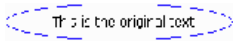
The original bitmapped picture used in all the examples. Verdana text, 12 pt, 72 dots/inch resolution.

It looks OK if shown with the original size as above. If, however, it is shown in reduced or enlarged size, it will

look very ugly:



The same image increased to 1,5 times original size.



The same image reduced to 2/3 of original size.

This problem will especially occur if you show the document with a program which allows the increase or reduction of the size of the document on the screen, such as Adobe Acrobat or Microsoft Powerpoint. Powerpoint, using anti-aliasing.

The explanation can be seen if you increase the size of the letter "h" from the original and the two revised pictures. In the table below, the size of these characters has been increased 8 times, so that you can see them more clearly.

Original "h"

The oriignal "h" was a bitmapped picture 6 pixels wide and 10 pixels high.

"h" increased 1,5 times

As you can see here, there is no good way to increase bitmapped picture by 50 %. The only choices are to either leave the line widths as they are, or double the line widths. The choice between these alternatives seems to have been done rather randomly.

"h" reduced to 2/3

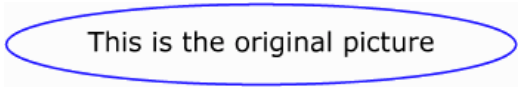
When you reduce by 2/3, it means that you have to remove 1/3 of the pixels. Since the original "h" was composed of lines 1 pixel width, some of them became 0 length after reduction. Not very good either.

Solution 1: Show in Original Size

The simplest solution to this problem is to ensure that the pictures are always shown in their original size. This is the method used by web browsers, and web browsers are, because of that, sometimes better when displaying documents containing screen dumps.

Solution 2a: Produce in very High Resolution

Another solution is to produce the picture in very high resolution. If, for example, the picture in the example above is produced with 600 pixels/inch, and its size is then reduced, it will still look fairly good. Example

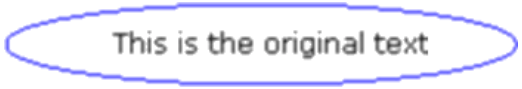


High-resolution bitmapped picture reduced to 72 pixels/ inch

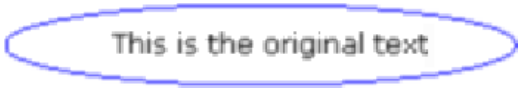
Solution 2b: Convert to very High Resolution and Blur

This is variant of Solution 2a, where you have to start with the original picture in only a low resolution format. You can convert the low resolution format to a high resolution format and then blur the picture, which will make it behave a little better when reduced to low resolution while viewing.

Below is an example, where the original picture used in all the examples above was converted in this way.



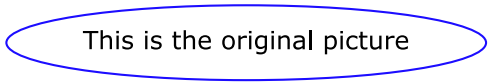
The picture to the left was first converted from 72 to 720 pixels/inch, then blurred, then converted back to 72 pixels/inch, and then actually shown in 1,5 times enlargement:



A simpler variant, the original picture was blurred as is, and then shown in 1,5 times enlargement.

Solution 3: Store Picture in Object Format instead of Bitmapped Format

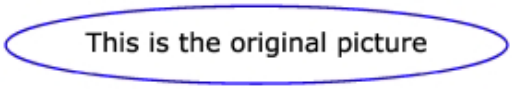
The best solution should be to store the picture in an object oriented format instead of a bitmapped format. With this method, the picture should always get the best possible rendering on the screen. Example of how it should look like (my experients, see "solution 4" below were not so positive, however, when trying to embed PDF and Flash in HTML):



Object picture

Solution 4: Embed objects in other formats handled by plug-ins

I also experimented with inserting the picture using plugins. Before this, I converted the text to paths, so that the result should not rely on the right fonts available. Here are the results:

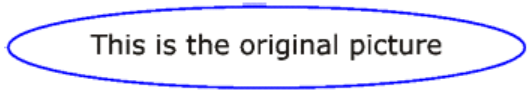


Embedding PDF in web page. This will only work with Explorer, not with Netscape 4.7.

This does not look as neat as it should, probably because the functionality in Explorer for embedding PDF in HTML has some problems, or because I did not understand correctly how to use it.

The HTML code used for the picture above was:

```
<object type="application/pdf" codetype="application/pdf" height=114
width=348 data="original-paths.pdf">
```



Embedding Flash in a web page.

This embedding was done using the following :-) neat and simple code, which Dreamweaver produced automatically for me:

```
<object  classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=5,0,0,0"
width="348"  height="57">
<param  name=movie  value="original-paths-150-perc-inc.swf">
<param  name=quality  value=high>
<embed  src="original-paths-150-perc-inc.swf"  quality=high
pluginspage="http://www.macromedia.com/shockwave/download/index.cgi?P1_Prod_Version=ShockwaveFlash"
type="application/x-shockwave-flash"  width="348"  height="57">
</embed>
</object>
```

Extensible Markup Language

Extensible Markup Language is a technology that is now being developed under the auspices of the World Wide Web Consortium (W3C). XML complements HTML and is based on Standard Generalized Markup Language (SGML). Whereas HTML describes a set of commands that define how data is laid out on a page, XML allows the data on an HTML page to be described by the type of information it represents. One direct consequence of this distinction is that search engines can return more meaningful hits. For example, using XML, a search engine might be able to distinguish whether the word "cookie" in an HTML page applies to the Internet or to a tasty dessert.

XML allows browser clients to dowrtload an HTML page just once and then manipulate the page off line, without referring to the server. The client can view the data in any meaningful way desired and can manipulate the data; for example, XML would, ideally, allow a client to perform actions such as extracting a hotel's name and address from the result of a search and then feeding the data into a mapping program that would print out driving directions.

XML Syntax Details

XML uses the Unicode character set for encoding. This means that XML can be used with a wide variety of international languages. Unicode has associated encodings for purposes of transmission. XML uses Universal Transformation Format-B (UTF-8) by default. (For information about Unicode and UTF-8, see Chapter 4, 'Encoding Standards.")

Unlike HTML, XML is case sensitive and white space is relevant.
XML has some reserved characters that have special meaning. For example, the characters "<'" and ">'" need to be encoded as "&lt" and "&gt" respectively. The sentence "25 > 24 and 25 < 26" would thus be encoded as "25 &gt 24 and 25 &lt 26" in XML.

XML syntax is very similar to that of HTML and consists of a series of tags and annotated text. The major difference is that the XML tags indicate what the data represents, rather than how the data should be represented. Another difference is that the tags are limited in HTML, whereas XML has unlimited potential because users can define their own sets of tags.

XML syntax consists of a series of elements. Each element consists of a beginning tag, contents, and an ending tag, as in this example:

<PERSON>

<LASTNAME>Gates</LASTNAME>

<FIRSTNAME>Bill</FIRSTNAME>

</PERSON>

In this example, the element PERSON begins on the first line and ends on the last line of the example. As shown, elements can contain other elements. Simple XML documents can be self-contained and self-describing. Complex XML documents are described using an external file called a *Document Type Definition file*.

Document Type Definition File

A Document Type Definition (DTD) file specifies the valid syntax for an XML document. In particular, a DTD file enumerates the elements that can appear in a particular XML document and the relationships among different elements. An XML document may or may not have a DTD associated with it.

XML and SGML

As previously stated, XML is based on Standard Generalized Markup Language (SGML). XML is a simplified form of SGML that lends itself readily to transmission across networks by reducing complexity while promoting ease of use and interoperability across platforms and applications. A form of XML document called "well4ormed" does not need an associated DTD file and can thus be transmitted over networks more easily. SGML has no equivalent to a well-formed XML document. A different form of XML document called "valid" does require an associated DTD file and does have an SGML-equivalent concept.

XML Vocabularies

Simply put, an XML DTD file constitutes a *vocabulary*. A vocabulary is the collection of elements defined by a DTD file and the rules for constructing valid instances of those elements. The following sections describe some XML vocabularies. More are expected to be defined in the future.