*:96 Overheads

Part 2a: Encoding, ABNF

More about this course about Internet application protocols can be found at URL:

`http://dsv.su.se/jpalme/internet-course/Int-app-prot-kurs.html`

Last update: 2005-02-09 2:42

Example of data structure declarations in Pascal

```
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;
```

Why encoding and encoding syntax specification?

1. Syntax must be exact. Saying “Parameters are indicated with a parameter name followed by a parameter value” does not specify whether to encode as “increment 5” or “increment:5” or “increment=5” or “increment = 5”.
2. Computer-internal formats like 64-bit floating point are particular to one computer architecture and not portable. Even the storage of octets in 32-bit words is different in different architectures. Sending internal data would thus get “New York” transformed to “ weNkroY” when moved between computers with different “byte order”.
3. A syntax specification language like ASN.1 ABNF or XML ensures that the syntax specification is unambiguous.

   (Or should be, but ABNF has a historical problem with not fully specifying where white space is allowed, i.e. to distinguish between “From: Peter Paul” and “From:Peter Paul”.)
4. Character set must be specified. Defaults are used, but have caused problems.

5. Character set must be specified. Defaults are used, but have caused problems.

Character set	Representation of “Ä” (hexadecimal)
ISO Latin One	C4
Unicode (ISO 10646), UCS-4	000000C4
Unicode, UTF-8 coding	E2C4
CP850 (old MS-DOS)	8E
ISO 6937/1	C861
old Mac OS	80

Character sets

A character set is a rule for encoding a certain set of glyphs onto one or more octets. By a glyph is meant a kind of small picture and a kind of syntactic description of the character. The same glyph need not look exactly identical, different fonts can display the same glyph in somewhat different ways.

Examples of characters and their encoding

Syntactic description	Encoding in some common character sets (hexadecimal representation)				Glyphs
	ISO 646	ISO646-SE	ISO 8859-1	Unicode & ISO 10646	
latin capital letter A with diaeresis	n.a.	5B	C4	00C4	Ä
latin capital letter O with diaeresis	n.a.	5C	DC	00DC	Ö
latin capital letter O with stroke	n.a.	n.a.	D8	00D8	Ø
Reverse Solidus	5C	n.a	5C	005C	\

Swedish character encodings

Glyph	å	ä	ö	ü	Å	Ä	Ö	Ü
Two-char encoding	aa	a:	oe	u:	AA	A:	O:	U:
SEN_850200_B = ISO646-SE	7D	7B	7C	??	4D	5B	5C	??
ISO 646 glyph for the encoding above	}	{	|		]	[	\	
ISO 10646	00E5	00E4	00F6	00FC	00C5	00C4	00D6	00DC
ISO 8859-1	E5	E4	F6	FC	C5	C4	D6	DC
Macintosh	8C	8A	9A	9F	81	80	85	86
Old MS-DOS	86	84	94	91	8F	8E	99	9A
T.61=ISO 6937/1	CA61	C861	C86F	C875	CA41	C861	C86F	C855

How can you put more than 255 different characters into eight bit octets?

Method 1	ISO 6937	Use multiple characters for some encodings, for example é as e´ or o as o¨.
Method 2	ISO 2022	Use several different 255 character sets, and special shift sequences to shift from one set to another set.
Method 3	Unicode, ISO 10646	Use two or four octets for each character, but provide compression techniques to compress them during transmission. UTF-8 is an example of a compression encoding scheme for ISO 10646, which has the property that the most common characters, like a-z and A-Z, have the same one-octet encoding as in ISO 646 and ISO 8859-1.
Method 4	HTML, MIME Quoted-Printable	Use special encodings for special characters, like ` ` for `non-breaking space` or `&ouml;` for ö.

UTF-8 encoding of ISO 10646 and Unicode

The UTF-8 (RFC 2044) is an encoding of Unicode with the very important property that all US-ASCII characters have the same coding in UTF-8 as in US-ASCII. This means that protocols, in which special US-ASCII characters have special significance, will work, also with UTF-8. They start with the two or four-octet encodings of ISO 10646 (UCS-4):

UCS-4 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

Subsets used in some standards

Name	Subset description	Where it is used
specials	“(”, “)”, “<”, “>”, “@”, “,”, “;”, “:”, “\”, “"”, “.”, “[”, “]”	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.

Binary and textual data

Binary data

Examples: Data compressed with various compression algorithms, images in formats like GIF, JPEG or TIFF, application data in a format particular to a certain application, such as Word, Excel, Filemaker Pro, Adobe Acrobat, etc.

Textual data

Data which is textual in character, in that it consists of a sequence of “readable” characters, sometimes organized into lines, such as plain text, HTML source, Postscript documents, source code in a programming language, etc.

There is no sharp limit between binary and textual data. Some properties which sometimes distinguishes textual data are:

- The character sequence to delimit line breaks differs between platforms, and is often modified at transmission from one platform to another. Macintosh usually uses a single Carriage Return (CR), Unix usually uses a single Line Feed (LF), MS Windows usually uses the character sequence CRLF in file storage, but this is often transformed to only LF when data is important into RAM by an application program.
- Sometimes, characters are encoded according to a character set, which is a rule deciding which glyph to show for a certain bit combination. Sometimes, the character set is modified when textual data is moved between computers or between applications.

Marking the end of data

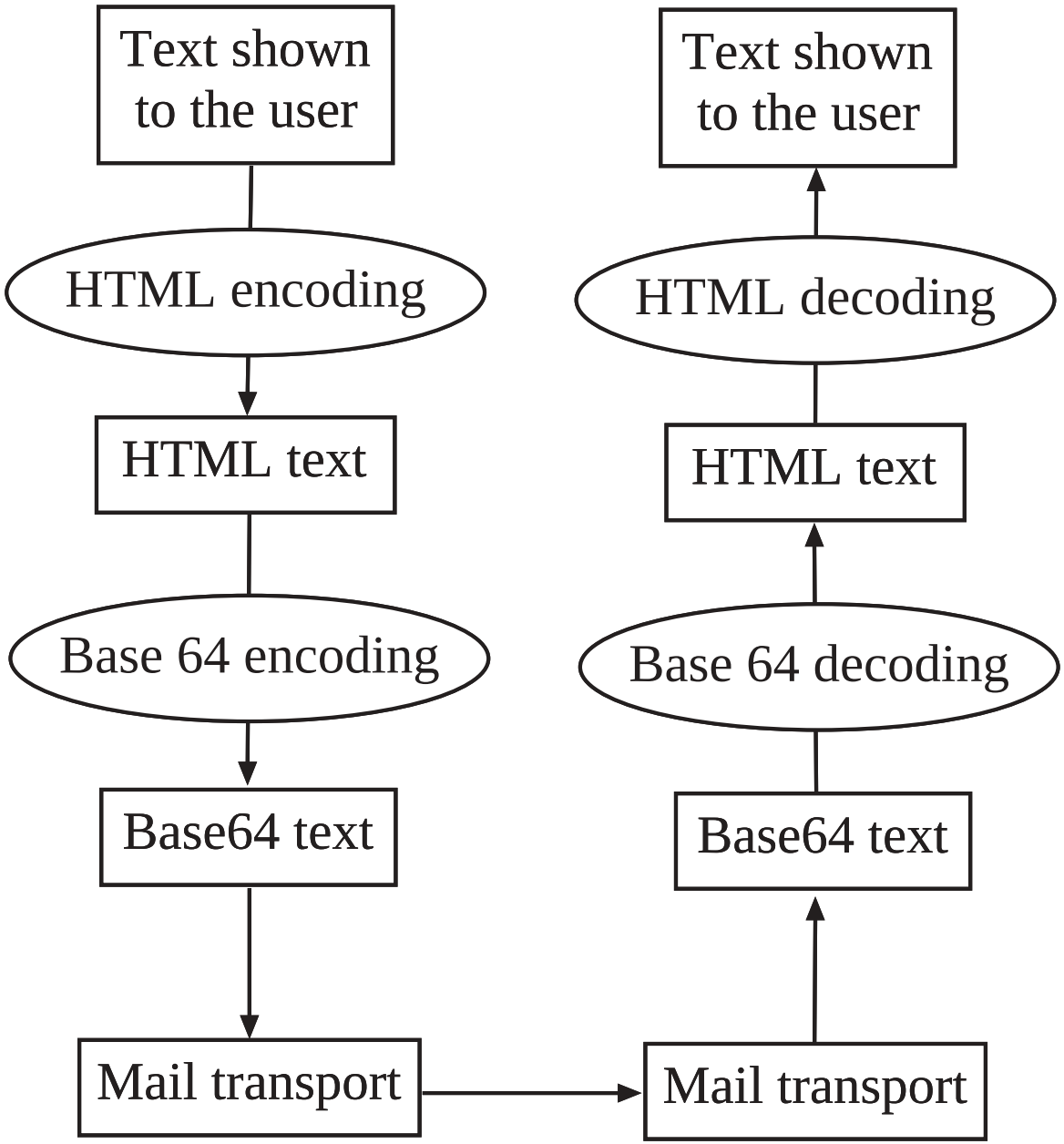
Internet protocols often need to transmit one or several objects of data. The data transmitted is often formatted according to its own encoding rules.

Method	Description	Examples	Used in	Problems
1	Use a special character sequence to mark end of data	`CRLF.CRLF`	SMTP	What to do if this sequence occurs in the data you want to transmit?

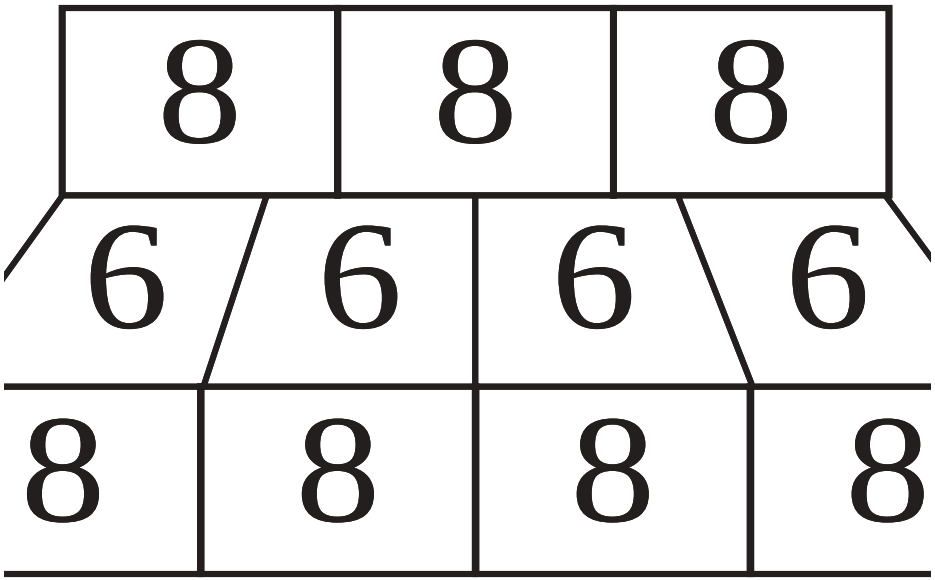
		`boundary: xyzabc`
		`--xyzabc¶`
		`--xyzabc--¶`
			MIME	

2	Indicate length in advance	`10*ABCDEFGHIJ`	HTTP	You may not know the length in advance, for example live broadcasting
3	Chunked transmission	`5*ABCDE5*FGHIJ`	HTTP	
4	Encode in limited character set	`UuRrc232cmflcw`	Base64	Inefficient

Encoding in more than one layer



Base64 encoding of binary data into text

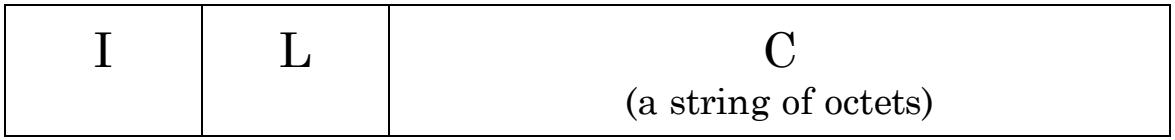


BASE64 is more reliable and works as follows: Take three octets (24 bits), split them into four 6-bit bytes, and encode each 6-bit byte as one character. Since 6-bit bytes can have 64 different values, 64 different characters are needed. These have been chosen to be those 64 ascii characters which are known not to be perverted in transport. Since BASE64 requires 4 octets, 32 bits, to encode 24 bits of binary data, the overhead is 1/3 or 33 %.

Encoding of protocol units

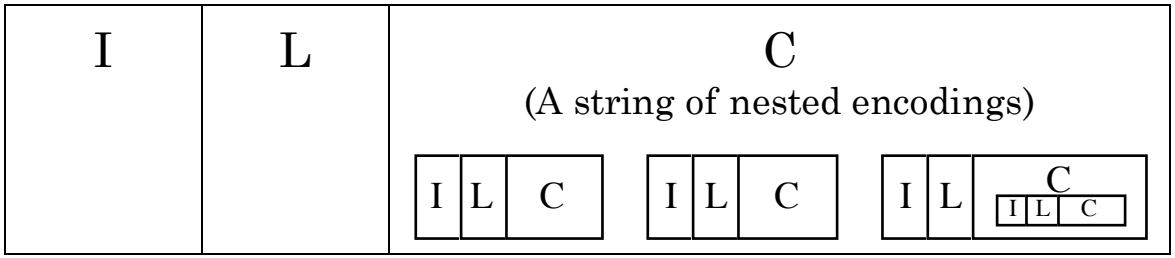
Binary encoding, often in the format: {identifier; length; value}

Primitive:



I = Identifier octets
L = Length octets
C = Contents octets

Constructed:



Binary encodings are often specified using the ASN.1 specification language.

Textual encoding, much like text in a programming language:

Example 1:

```
a002 OK [READ-WRITE] SELECT completed
```

Example 2:

```
EditReplace .Find = "^p ", .Replace = " "
```

Textual encodings are often specified using the ABNF specification language.

Linear White Space

Character name	Real rendering	Notation on this page
Space	A non-printing break with the same width as a single letter.	
Horizontal tab	Moving the printing position to the next print position, usually a wider break than for a space.	→
Line break	Moving the printing position to the next line, using CR, LF or CR+LF.	¶

Acronym	Term	Description	Examples
LWSP	Linear White Space	Sequence of one or more space and horizontal tab characters.	→ →
FWSP	Folding White Space	Linear White Space which also can include line breaks. Continuation lines must begin with tab or space.	→ ¶ →
			¶ →
CFWSP	Comment Folding White Space	Folding White Space which can contain comments in parenthesis.	(Rose) ¶ →(Tulip)

Examples of identical code, in-spite-of CLWSP, in e-mail headers:

```
In-Reply-To: <199807112000.WAA30049@mailbox.hogia.net>
```

```
In-Reply-To: (Your message of 11 July 1998)
             <199807112000.WAA30049@mailbox.hogia.net>
```

```
In-Reply-To: <199807112000.WAA30049@mailbox.hogia.net>
 (Your message of 11 July 1998)
```

Inpreciseness of common usage of where LWSP and CLWSP is allowed and not allowed.

Many different Internet standards use ABNF, but all of them do not use exactly the ABNF notation in the same way. In particular, many Internet standards do not specify where LWSP (Linear White Space) is permitted or required.

Thus, Internet standards often specify things like:

```
Subject = "Subject" ":" "sentence"
```

Is space allowed/required or not between elements here?

The above ABNF specification, when used in older standards, might not clarify if spaces are *allowed* or *required* between the elements.

ABNF syntax elements

A simple ABNF production with an OR ("/") element:

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

This says that when you send an "answer" from one computer to another, you send either the string "`Answer: Yes`" or the string "`Answer: No`".

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 ”
```

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 " ) ;Compare *5 1*
```

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    ”
```

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*.

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 1 (From RFC2822):

```
keywords        =       "Keywords:" phrase *("," phrase) CRLF
phrase          =       1*word / obs-phrase
word            =       atom / quoted-string
atom            =       [CFWS] 1*atext [CFWS]
```

Example 1, value:

```
Keywords: Orchids, Tropical flowers
```

Example 2 (from RFC822):

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

Example 2, value a:

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

Example 2, value b:

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

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" )
```

Summary of ABNF notation

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

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 EXERCISES IN COMPENDIUM 6 PAGES 57-66

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:

"→ →"

"→ ¶
→"

" ¶
→"

Usage:

```
From: John Smith <jsmith@foo.bar>

From: John Smith
      <jsmith@foo.bar>
      (typed by Mary Smith)
```

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

Examples of use of ABNF from RFC 2822

Example 1, ABNF (from RFC 2822):

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

Example 2, ABNF (from RFC2822):

```
mailbox          =       name-addr / addr-spec
name-addr        =       [display-name] angle-addr
angle-addr       =       [CFWS] "<" addr-spec ">" [CFWS]
                         / obs-angle-addr
display-name     =       phrase
addr-spec        =       local-part "@" domain
```

Example 2, value a:

```
jpalme@dsv.su.se
```

Example 2; value b:

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

Example 3 (from RFC2822):

```
fields = *(trace
           *(resent-date /
             resent-from /
             resent-sender /
             resent-to /
             resent-cc /
             resent-bcc /
             resent-msg-id))
         *(orig-date /
         from /
         sender /
         reply-to /
         to /
         cc /
         bcc /
         message-id /
         in-reply-to /
         references /
         subject /
         comments /
         keywords /
         optional-field)
```

Example 4 (from RFC2822)

```
in-reply-to     =       "In-Reply-To:" 1*msg-id CRLF
msg-id          =       [CFWS] "<" id-left "@" id-right ">" [CFWS]
id-left         =       dot-atom-text / no-fold-quote / obs-id-left
id-right        =       dot-atom-text / no-fold-literal /
                        obs-id-right
no-fold-quote   =       DQUOTE *(qtext / quoted-pair) DQUOTE
```

Example 4, value a:

```
In-Reply-To: <12345*jpalme@dsv.su.se>
```

Example 4, value b:

```
In-Reply-To: <12345*jpalme@dsv.su.se> <5678*jpalme@dsv.su.se>
```

Example 4, value c:

```
In-Reply-To: Your message of July 26 <12345*jpalme@dsv.su.se>
```

Examples of use of square brackets ([) and (])

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

Example 5 (from RFC822):

```
 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
```

Example 5, value a:

```
 Received: from mars.dsv.su.se (root@mars.dsv.su.se
        [130.237.158.10])
        by zaphod.sisu.se (8.6.10/8.6.9) with ESMTP
        id MAA29032 for <cecilia@sisu.se>
```

ABNF syntax rules, comments

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

Example 6 (from RFC2822):

```
 specials        =       "(" / ")" /      ; Special characters used in
                         "<" / ">" /      ;  other parts of the syntax
                         "[" / "]" /
                         ":" / ";" /
                         "@" / "\" /
                         "," / "." /
                         DQUOTE
```

Example 7 (from RFC822):

```
obs-zone      =    "UT" / "GMT" /          ; Universal Time
                                           ; North American UT
                                           ; offsets
                   "EST" / "EDT" /         ; Eastern:  - 5/ - 4
                   "CST" / "CDT" /         ; Central:  - 6/ - 5
                   "MST" / "MDT" /         ; Mountain: - 7/ - 6
                   "PST" / "PDT" /         ; Pacific:  - 8/ - 7

                   %d65-73 /               ; Military zones - "A"
                   %d75-90 /               ; through "I" and "K"
                   %d97-105 /              ; through "Z", both
                   %d107-122               ; upper and lower case
```

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`”

Examples:

```
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.

RFC 822 lexical scanner 1


```
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
```

The same with the 1994 version of ABNF


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

%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.