

Richard Carlsson

Klarna

All you wanted to know about Character Encodings but were too afraid to ask

- Why are there so many different encodings?
- What does all the terminology mean?
- What do I need to keep in mind?
- What is all this Unicode stuff anyway?
- How does Unicode work in Erlang?
- How do we switch our system to Unicode?

In the beginning, there was the Teleprinter



- Incoming characters are printed to paper
- Keypress is sent to the remote teleprinter or computer (and printed on the paper, for feedback)
- The first teleprinters were invented in the 1870s
- Baudot code (5-bit, shifted state for letters/symbols)

Baudot code?

- Émile Baudot (1 baud = 1 'signalling event'/second)
- **1874, some 70 years before the first computers**
- 5 bits (32 combinations)
- **A-Z, 0-9, &()=%/-;!'?,:., ERASE, FIGSHIFT, ...**
- ~60 characters in total, thanks to shift state
- Binary, fixed length
 - Only 2 signalling states: on/off
 - Always in groups of 5, e.g., **A = 00001**

Yeah? Why not Morse code, while you're at it?

- Even earlier, invented in 1844
- Varying length of symbols (E = -, Z = - — — - -)
 - As in Huffman coding, common symbols are shorter
- 5 different signalling states
 - Short tone (1 unit length)
 - Long tone (3 units)
 - Gap (1 unit silence between tones within a symbol)
 - Letter space (3 units silence)
 - Word space (7 units silence)

Keep it simple

- **Simple and uniform is good for machines**
- 2 levels (on/off) are easier than 5
- Fixed-length groups is easier than variable-length
- Bad part of Baudot code: *stateful*
 - Last pressed shift key decides if **00001** = **A** or **1**, etc.
 - The human operator used 5 keys at once like a piano chord, and the shift state didn't change very often
 - Still pretty simple to build machines that handle it

Control confusion

- We have already *mixed symbols with control codes*
 - Erase (or "Delete", or "Backspace")
 - Change shift state
 - Line Feed, Bell, and other control codes soon added
- We're talking about **controlling a teleprinter...**
 - position on paper, shift state, alert the operator...
- ...rather than representing "characters"
- And in 1874, nobody thought about data processing

85 years later: ASCII

- American Standard Code for Information Interchange
- 1963, 7-bit code (128 combinations)
- Stateless (no "current shift mode" to remember)
- Both uppercase and lowercase characters
 - Bit 5 works as a Caps Shift, and bit 6 as Ctrl
 - **1000001 = A**
 - **1100001 = a**
 - **0000001 = Ctrl-A**
 - A-Z, a-z, and 0-9 are contiguous, for easy processing

But what about 8-bit bytes?

- The term "byte" has been used since 1954
- Early computers had very varying byte sizes
 - Typically 4-6 bits
- The IBM System/360 started the 8-bit trend, 1964
 - Computer memories were small and expensive
 - Every bit counted – not an easy decision, back then
 - In the long run, 8 was more practical (power of 2)
 - Used IBM's own 8-bit character encoding, EBCDIC
 - ...which everybody hated, but still exists out there

Extended ASCII

- ASCII assumes that only English will be used
- More symbols were needed for other languages
- Many 8-bit "extended ASCII" encodings created
 - Often called "code pages", and numbered
 - E.g., IBM PC standard code page 437 (DOS-US)
 - Countries picked their own preferred encoding
 - Big mess, no interoperability between operating systems and countries

Latin-1



- DEC VT220 terminal, 1983
 - (Only like 100 years after those first teleprinters...)
- "Multinational Character Set" extended ASCII
 - Became the ISO 8859-1 standard, also called Latin-1
 - Variants: 8859-2 – 8859-15, Windows-1252, ...
 - Still a lot of confusion

Not everyone uses Roman

- E.g., Shift-JIS (8-bit) for Japanese
 - 0-127 are as in ASCII, with 2 exceptions:
 - ~ (tilde) → ¯ (overline)
 - \ (backslash) → ¥ (yen)
 - 161-223 are katakana ("alphabetical" symbols)
 - **Variable-length encoding**
 - 129-159 and 224-239 are the first of 2 bytes
 - 6355 common kanji (Chinese) + other symbols
 - Hard to jump in at any point: first or second byte?

Stateful encodings

- E.g., ISCII for Indian languages, or ISO 2022 for Japanese, Korean and Chinese
- Escape codes change the meaning of all the following bytes until the next escape code
 - May also change number of bytes per character
- Even harder to jump in at any point: must scan backwards to find latest escape code

How do you know which encoding a text is using?

- Most software leaves this problem to the user
 - Most file formats have no way to indicate encoding
 - A raw text file has no place to store such metadata
- That it works at all is mainly by convention
 - People on similar systems and in the same country usually use the same encoding
 - If not, you have a problem (yes, you)
 - Converting encodings can destroy information

IANA encoding names

- The IANA (Internet Assigned Numbers Authority) maintains the list of official names of encodings to be used in communication
- The names are not case sensitive
- For example:
 - Name: ANSI_X3.4-1968 [RFC1345,KXS2]
 - Alias: ASCII
 - Alias: US-ASCII (preferred MIME name)
 - Alias: ISO646-US

Charset declarations

- Email/HTTP headers:
 - Content-Type: text/html; charset=utf-8
- XML declaration (at start of XML document)
 - `<?xml version="1.0" encoding="iso-8859-1" ?>`
- If you don't know the charset, how can you read the text that says what charset it is?
 - Email/HTTP headers are always ASCII only
 - Or guess encoding and see if you can read enough to find the declaration that tells you the real encoding
 - Most encodings are downwards ASCII compatible

8 bits can be unsafe

- Some old systems assume that 7-bit ASCII is used
 - The extra bit in each byte was used as a checksum (parity) in data transfers, or as a special flag
 - That bit was often automatically reset to 0
 - Zero bytes, line breaks, etc., could also be mangled
- To send 8-bit data through such systems and be sure that it survives, you have to use only character codes that will not be touched
 - E.g., Base64 (or the older Uuencode) for e-mail
 - [1, 128, 254] => "AYD+"

A few simple rules

- Remember what encoding your strings are in
- Never mix different encodings in the same string
- When you write to a file, socket, or database:
 - What encoding is expected?
 - Do you need to convert before writing?
 - Is it your job to provide a charset declaration?
- When your program gets text from somewhere:
 - Do you know what the encoding is?
 - Do you need to convert to your local encoding?

Never trust your eyes

- There can be several layers of interpretation between your output and what you see
 - Some of these may try to be clever and "help" you
 - There are many system settings that can affect things
- **Do not trust what your editor shows you**
- **Do not trust what you see in the terminal**
- If you want to know what's in a file or data stream, use a low level tool like `'od -c <filename>'`

Terminology

- People use the terms "character encoding", "character set" or "charset", "code page", etc., informally to mean more or less the same thing
 - A set of characters (and control codes)
 - ...and how to represent them as bytes
- But there are many different concepts when it comes to languages and writing systems
- Unicode terminology tries to separate the concepts


What is a "character"?

- It's not the number it's been given in an encoding
- It's not the byte (or bytes) stored in a file
- It's not the graphical shape (which font?)
- It's really more the *idea* of the character
- Unicode talks about *abstract characters* and gives them unique names, such as LATIN CAPITAL LETTER X, or RUNIC LETTER OE (ǰ)

Code points

- Each abstract character is assigned a *code point*
- A natural number between 0 and $10FFFFFF^{\text{hex}}$ (over a million possible code points)
- Divided into "planes" of 2^{16} code points each
 - 000000-00FFFF is called the *Basic Multilingual Plane* and contains most characters in active use today
 - Most planes are currently unused, so plenty of space
- **Just numbers, not bytes or machine words**
- The subset 0-255 is Latin-1 (and 0-127 is ASCII)

But where do fonts come in?

- Encodings (and strings in general) just express the *abstract* characters, not how they should look
- A font maps the abstract code points to *glyphs* (specific graphical shapes): **A**, A, **A**, A, **A**...
- You can usually forget about fonts, **but**:
 - That you can *handle* a character (code point) in your code doesn't mean everybody can *see* the symbol
 - No single font contains *all* Unicode code points
 - Without a suitable font, the user sees  or similar.

Unicode encodings

- 21 bits are needed for full Unicode
 - Unicode was first designed for 16 bits (not enough)
 - But the range 0-10FFFF is final (they promise)
- There is a family of official encodings for Unicode called Unicode Transformation Format (UTF)
 - UTF-8
 - UTF-16
 - UTF-32
 - ... and a couple of others that are not very useful

UTF-8

- Quickly becoming the universally preferred encoding in external representations everywhere
- Downwards ASCII-compatible (0-127 unchanged)
- UTF-8 is not the same thing as Latin-1!
 - Latin-1 is fixed-length (exactly 1 byte per character)
 - UTF-8 is *variable-length* (1-6 bytes per character)
 - Characters in the BMP (0-FFFF) use 1-3 bytes
 - Latin-1 codes 128-255 need 2 bytes (åäö, etc.)
 - ASCII characters use a single byte

UTF-8 details

Bits	Max	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	7F	0.....					
11	7FF	110.....	10.....				
16	FFFF	1110....	10.....	10.....			
21	1FFFFFF	11110...	10.....	10.....	10.....		
26	3FFFFFFF	111110..	10.....	10.....	10.....	10.....	
31	7FFFFFFF	1111110.	10.....	10.....	10.....	10.....	10.....

- First byte says exactly how many bytes are used
- Bytes 2-6 have high bits 10 – easy to find start byte
- If data is corrupted, it is easy to find next start byte
- Preserves sorting order of Unicode code points
- Easy to recognize UTF-8 text; also, FE/FF not valid

UTF-16

- Some systems (e.g., Java and the Windows API) use 16-bit characters – after all, Unicode first said they were going to use no more than 16 bits
- UTF-16 uses 2 bytes per character for 0000-FFFF
- Code points above FFFF are encoded using *two* 16-bit characters (same trick as in Shift-JIS) called *surrogate pairs*
 - D800-DBFF (high) followed by DC00-DFFF (low)
 - Together they give a number from 10000-10FFFF
 - Only valid when used in pairs (and in correct order)

UTF-32

- UTF-32 means that a 32-bit integer is used for each code point in the string
 - Can of course use 64-bit integers or wider as well
 - No surrogate pairs may be used – always expanded
- Mainly used internally in APIs and in chars/strings in programming languages
 - In particular for functions that work on a single character rather than on a string
 - Unix-like systems normally use 32-bit wide characters (`wchar_t`) also in strings

What about "UCS"?

- Universal Character Set (ISO 10646)
 - Mostly the same as Unicode (same code points)
 - Byte encodings that nobody liked (UTF-1, UCS-2) that have been obsoleted by UTF-8 and UTF-16
- Basically, they tried to create a standard too soon
 - Not enough acceptance from software vendors
 - Unicode "won" and UCS was adapted to Unicode
- If you see "UCS", think "Unicode" or "obsolete"

Endianism

- If you send 16-bit or 32-bit data as a stream of bytes, you have to decide in which order to send the bytes of each integer: high or low bytes first?
 - Someone who reads the data must know in which order to reassemble the bytes into integers
- Default byte order in Internet protocols is big-endian (“network order”)
 - But usually it depends on the system, i.e., little-endian is typically used on x86-machines, etc.
- UTF-8 has no endianism; the byte order is fixed

Byte order mark (BOM)

- The code point FEFF can be used to indicate the byte order, by inserting it *first* in the text
 - If found, it is not really part of the text, just a hint
 - "Zero-width, non-breaking space", i.e., invisible, if it occurs anywhere else
 - FE followed by FF implies big-endian
 - FF followed by FE implies little-endian
 - The code point FFFE is reserved as invalid in Unicode, so there can only be one interpretation
- Don't use BOM in UTF-8 (causes confusion)

Not only encodings

- Unicode includes a number of rules and algorithms for normalizing, sorting, and displaying text
- The rules for alphabetic sorting depend on the language (English, Swedish and German differ)
- How do you convert Croatian text to uppercase?
- There are free support libraries that handle this for you, like ICU for C/C++
 - Many programming languages have ICU bindings

Normalization forms

- Regardless of encoding and surrogate pairs, Unicode text is *variable-length* (even in UTF-32)
- There are base characters and combining characters
 - ¨ (776) is a combining character (diaeresis)
 - Ä can be represented by [196] or by [65, 776]
 - It can get much more complicated than that
 - The combining characters follow the base character
- To compare strings, they must be normalized
 - NFC (normal form composed) is usually best

Erlang, Latin-1 and Unicode

Chars? We don't need no steenkin' chars!

Strings are already Unicode

- Erlang strings are just lists of integers
- Erlang integers are not bounded in size
- We already use Latin-1 (0-255) in our strings
- Unicode simply means larger numbers in the lists
 - Technically it's UTF-32 – one integer per code point
- You have to think about other encodings when you read or write text, or convert to or from binaries
- Erlang source code is still always Latin-1! No Russian in source files for now, sorry!

Binaries are chunks of bytes

- The question is: how are the code points of the string represented as bytes in the binary?
- For Latin-1, you use exactly one byte per character
 - `list_to_binary(String) / binary_to_list(Bin)`
- To pack a string as a UTF-8 binary:
 - `unicode:characters_to_binary("Motörhead")`
- To unpack a UTF-8 binary to a string:
 - `unicode:characters_to_list(<<"MotÃ¶rhead">>)`

IO-lists are also just bytes

- An IO-list can be:
 - A binary, containing **any bytes**
 - A list of integers and/or other IO-lists (to any depth)
 - **All integers must be between 0 and 255**
- E.g., [88, [89, 90], [32, <<90,89,88>>], ...]
- Concatenating IO-lists is cheap: $L3 = [L1, L2]$
- IO-lists can be written directly to files/sockets or converted to binary using `iolist_to_binary(List)`
- Latin-1 strings can be used directly as IO-lists

New Unicode type: chardata

- Chardata is similar to an IO-list. Chardata can be:
 - A binary, containing **UTF-8 encoded** characters
 - A list of integers and/or other chardata, to any depth
 - **All integers are Unicode code points (0-10FFFF)**
- E.g., [1234, <<195,132>>] → "ÄÄ"
- Library support for converting to/from other encodings, and for writing to output
- `io:format("~ts", [CharData])` instead of `~s`

I/O stream encodings

- Before Unicode in Erlang, I/O streams assumed the data was in Latin-1, and never modified the bytes
- The `file:open/2` function now takes an option `{encoding, ...}` which by default is `latin1`
- You can change the encoding of an existing stream with `io:setopts/2` or inspect it with `io:getopts/1`
- The shell now uses Unicode I/O by default if the OS language settings (`LANG` etc.) indicate it

Think before you output

- A plain old flat Latin-1 string in Erlang is a charlist and can be printed with `~ts` as well as with `~s`
- A Latin-1 binary cannot be printed with `~ts`, because `~ts` expects binaries to contain UTF-8
- Chardata cannot be written directly to a file or socket – it may contain integers above 255
 - If you know it contains only ASCII integers (0-127) and/or UTF-8 binaries, and the output stream expects UTF-8, you can treat it as an IO-list

Switching to Unicode

How can you gradually change your system to support Unicode all the way through?

Start with the output!

- First of all, make sure you can **output** Unicode, in web pages, mail, PDFs, etc. (Safest, and makes sure you can **see** the results of later changes)
- Start with one page/document at a time and test
 - Generate charset declarations for UTF-8
 - Transform the text to UTF-8 when you output
- Check that result looks good, both for Latin-1 text (e.g., åäö), and full Unicode text (e.g., Cyrillic characters)

Get a grip on internal data

- Start assuming that strings can contain codes > 255
- Most code working on list strings needs no change
- Check all packing/unpacking of binaries
- Use UTF-8 in binaries, not Latin-1, in particular when you store them in the database or on disk
 - Consider converting data in old tables/files to UTF-8
- Prefer UTF-8 encoded binaries rather than lists of characters whenever you are storing text

Don't accept any Unicode input until you're ready!

- **All input** to the system must be limited to Latin-1 until you can handle Unicode all the way through!
- Make sure that web pages do not post UTF-8 encoded text back to you, even if the text on the page is declared as UTF-8
 - `accept-charset` attribute
 - Check the input at runtime for safety
- Test sending in e.g., Cyrillic and see what breaks!

Not all code at once!

- Incrementally – a "big bang" update of all code to suddenly handle full Unicode is not feasible in a large and continuously running system
- Take care of one part of the code at a time
- Convert to/from Latin-1 when you interface with other code that is not Unicode-compliant yet
 - Convert gracefully; don't crash if the conversion fails. Replace chars >255 with '?' instead.

Last: accept Unicode input

- Once you let full Unicode into the system at some point, it will start to spread all over the place
- Entry points: HTTP POST or GET (URL queries), XMLRPC, files via FTP, etc. Etc.
- Ensure all Unicode input gets normalized to NFC (composed normal form) at the entry point!
 - Should already be guaranteed by text sent from web browsers (W3C standard), but you need to make sure

That's all there is to it

What could possibly go wrong?