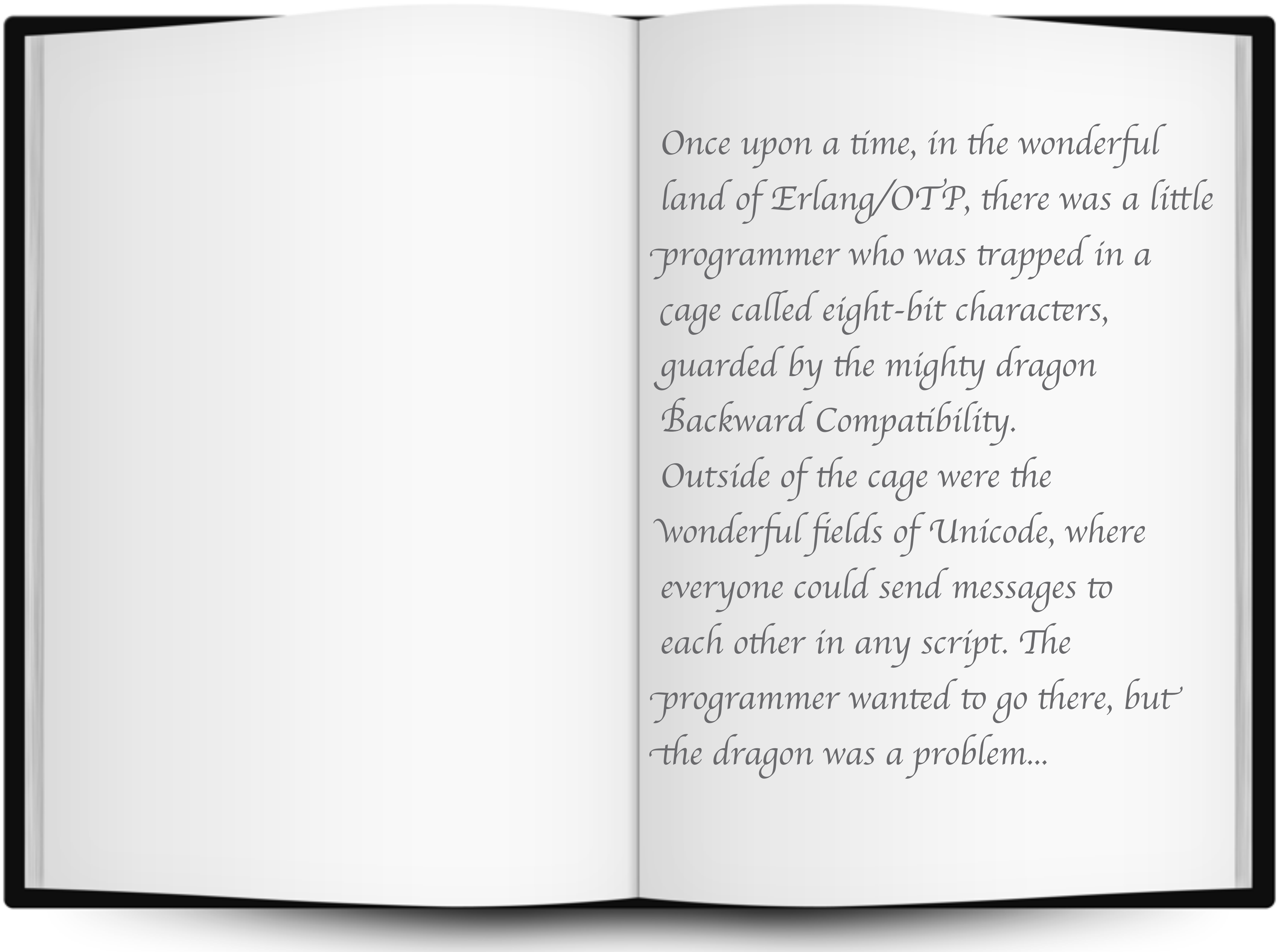
BRING UNICODE TO ERLANG!

THE TALE OF THE PROGRAMMER, THE EIGHT BIT CHARACTER CAVE, THE GREEN FIELDS OF UNICODE AND THE DRAGON OF BACKWARD COMPATIBILITY

PATRIK NYBLOM, ERICSSON AB
pan@erlang.org



Once upon a time, in the wonderful land of Erlang/OTP, there was a little programmer who was trapped in a cage called eight-bit characters, guarded by the mighty dragon Backward Compatibility.

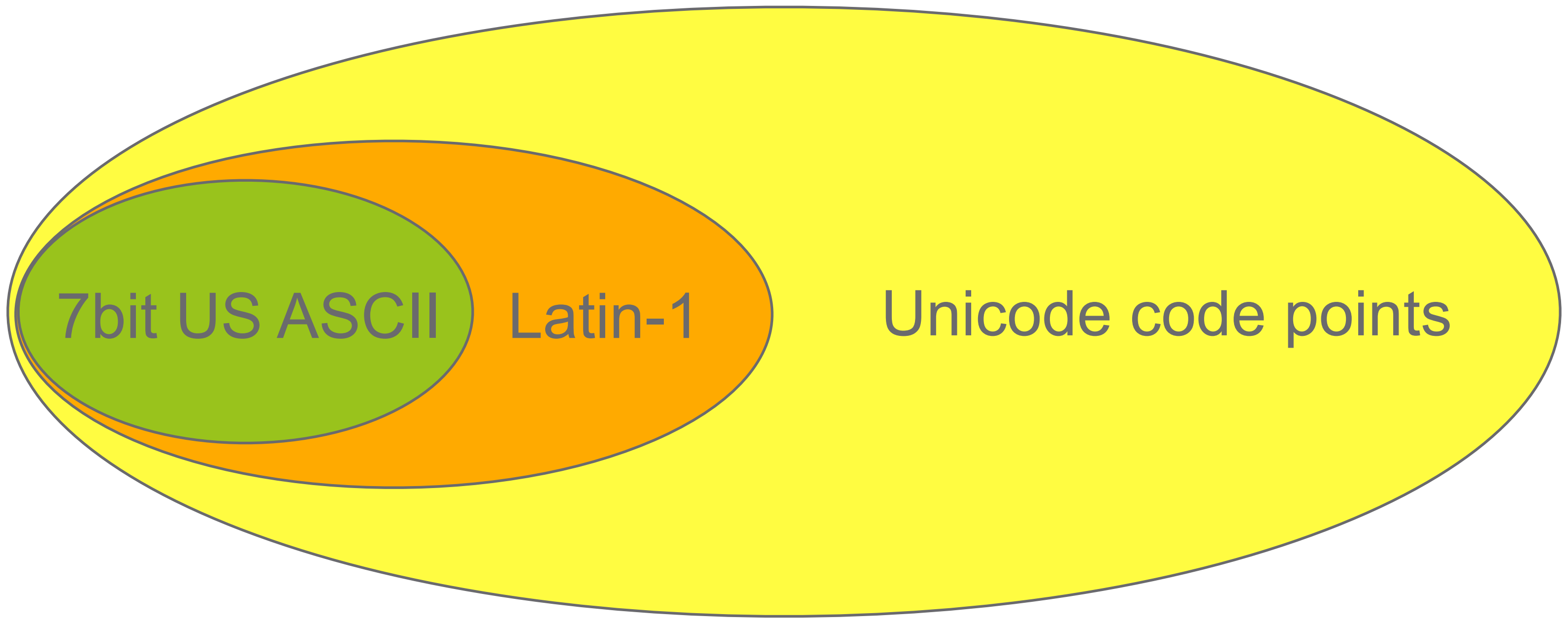
Outside of the cage were the wonderful fields of Unicode, where everyone could send messages to each other in any script. The programmer wanted to go there, but the dragon was a problem...

UNICODE - WHAT IS IT REALLY?

› The Unicode standard defines *code points* for “all” known living and dead scripts

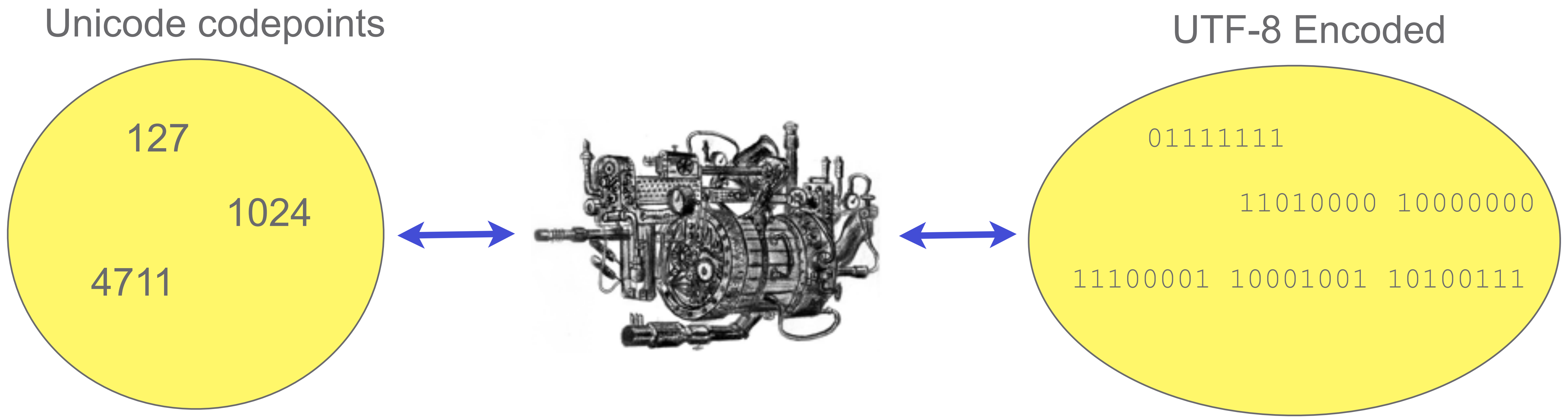
› More or less, every symbol ever used in any language, plus some other symbols, have a codepoint

› The *code points* are compatible with US-ASCII (7bit) and ISO-Latin-1



UNICODE - BUT THERE'S MORE

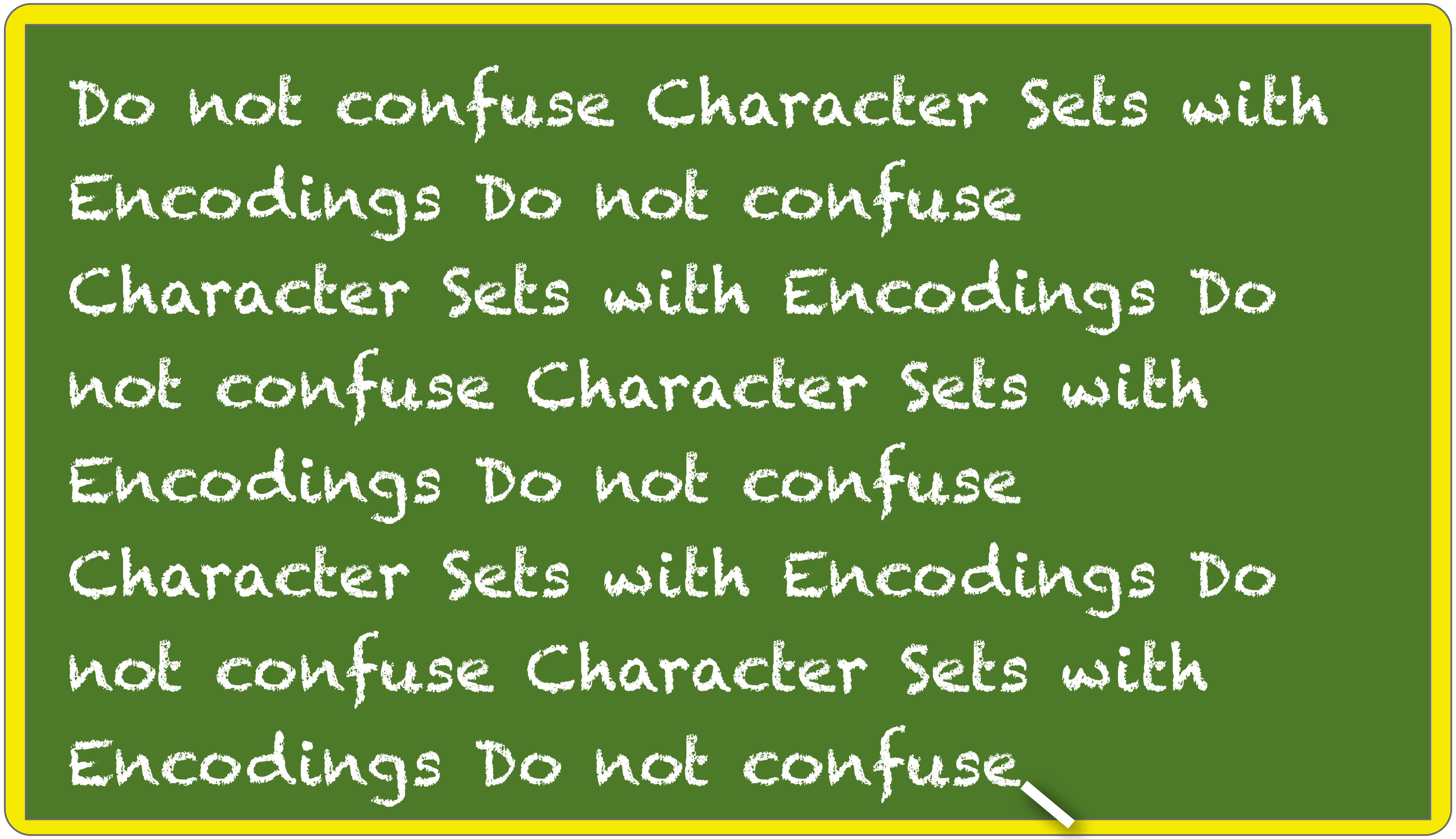
- There are also a couple of encodings available
  - UTF-8
  - UTF-16 (big and little endian)
  - UTF-32 (big and little endian)
  - UCS-4
- Encodings define mappings from code points to any byte oriented media



THE FORGOTTEN ENCODING...

- When we only had ISO-Latin-1 or 7bit ASCII, we never thought of encodings
- Each character could fit in one byte
- We can call that “byte-wise encoding”
- Byte-wise encoding had enormous advantages
  - Compact
  - All characters had the same size
  - Every character could be written to any device (except 8bit unclean ones)
- Life was good, so good that we did not even reflect upon it
  - But was it really that good?

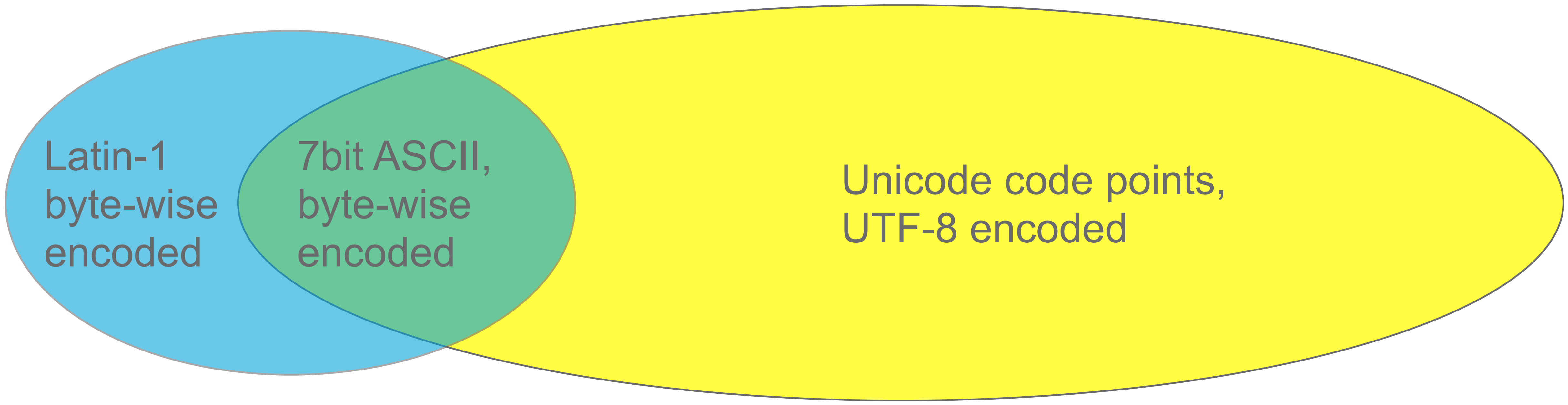
RULE OF UNICODE #1



Do not confuse Character Sets with Encodings Do not confuse Character Sets with Encodings Do not confuse Character Sets with Encodings Do not confuse Character Sets with Encodings Do not confuse Character Sets with Encodings Do not confuse

UTF-8 ENCODING

- A popular encoding, especially in western countries
- Backwards compatible with 7bit ASCII characters byte-wise encoded
- But *NOT* backwards compatible with Latin-1 characters byte-wise encoded


Latin-1 byte-wise encoded
7bit ASCII, byte-wise encoded
Unicode code points, UTF-8 encoded

REPRESENTATION IN ERLANG

- Unicode Strings are represented as lists of integers, one list element per character, the integer being the code point
- When encoded in a binary (byte-oriented), the default encoding in Erlang is UTF-8
  - All Unicode aware modules that handle binaries, handle UTF-8 binaries.
  - Some modules handle other encodings, for communicating with the outside world
- If you mix lists of codepoints and UTF-8 binaries, you get `unicode:chardata()`, a possibly deep list of integers representing codepoints, and UTF-8 binaries.
  - This is *NOT* an `iolist()`
    - But a pure binary with any encoding is of course always valid in i/o
  - Binaries with byte-wise encoded characters 128..255 are not allowed

UNICODE CONVERSION

- Conversion between lists and binaries with a certain encoding, is done with the module `unicode`
- `characters_to_list(Data) -> list()`
  - `Data = unicode:chardata()`
- `characters_to_list(Data,InEncoding) -> list()`
  - `Data = latin1_chardata() | chardata() | external_chardata()`
    - `latin1_chardata() = iolist()`
    - `external_chardata() = as chardata(), but with binaries in other encoding, defined by InEncoding`
  - `InEncoding = latin1| unicode | utf8 | utf16 | {utf16, big | little} | utf32 | {utf32, big | little}`

UNICODE CONVERSION

- Conversion between lists and binaries with a certain encoding, is done with the module `unicode`
- `characters_to_list(Data) -> list()`
  - `Data = unicode:chardata()`
- `characters_to_list(Data,InEncoding) -> list()`
  - `Data = latin1_chardata() | chardata() | external_chardata()`
    - `latin1_chardata() = iolist()`
    - `external_chardata() = as chardata(), but with binaries in other encoding, defined by InEncoding`
  - `InEncoding = latin1| unicode | utf8 | utf16 | {utf16, big | little} | utf32 | {utf32, big | little}`

Confusing! Rule #1...

UNICODE CONVERSION

- `characters_to_binary(Data) -> binary()`
  - `Data = unicode:chardata()`
- `characters_to_binary(Data, InEncoding) -> binary()`
- `characters_to_binary(Data, InEncoding, OutEncoding) -> binary()`
  - `Data = latin1_chardata() | chardata() | external_chardata()`
  - `InEncoding = latin1| unicode | utf8 | utf16 | {utf16, big | little} | utf32 | {utf32, big | little}`
  - `OutEncoding = latin1| unicode | utf8 | utf16 | {utf16, big | little} | utf32 | {utf32, big | little}`

BIT-SYNTAX

- Matching an encoded character:
  - <<X/utf8,_/binary>> = Bin
- Constructing an encoded character
  - <<Ch/utf16-little>>
- Syntactic sugar for literals
  - `<<"Ὀδυσσεύς"/utf8>>`
  - Same as `<<$Ὀ/utf8, $δ/utf8, $υ/utf8, $σ/utf8,`
    `$σ/utf8, $ε/utf8, $ύ/utf8, $ς/utf8>>`
- A string literal, or a character literal, can contain any "literal" unicode character if the input device to the evaluator or the compiler can represent it
- If the device is byte-wise, \x{HHH} can be used to write a character > 255

ERICSSON

UNICODE AND I/O

- The `io` module is Unicode aware
  - So are it’s friends (io_lib, io_lib_format etc)
- The `file` module handles bytes, not characters and is *not* Unicode aware
  - Except `file:open/2`, which can open files for the `io` module to handle
- The actual io_servers then? (like files, standard_io etc)
  - Maybe, maybe not
  - Depends on the ‘encoding’ option

```
1> io:getopts().
[... {encoding,unicode}]
```

- The ‘encoding’ tells you what is in the other end
  - if encoding != ‘latin1’, you can send any (unicode) character to it
    - Perfect for `io`-module, less useful for `file` module
  - if encoding == ‘latin1’, you can only send characters < 256 to it
    - i.e. a plain byte oriented device, perfect for `file` module

REGARDLESS OF ANY ENCODING IN THE IO_SERVER…

- The `io` module handles `unicode:chardata()`
- `io:put_chars/{1,2}` or `io:format/{2,3}` does not change parameter types depending on the encoding of the `io_server` (puh!)
- But you may not be able to output all characters if encoding is ‘latin1’...
- The file module handles bytes, so all data to e.g. `file:write/2` will be `iolist()` or `binary()`
- Mixing any other encoding than ‘latin1’ with `file:write`/`file:read` etc will get you into trouble

RULE OF UNICODE #2

Do not use the file module to read and write files with other encoding than 'latin1'! Do not use the file module to read and write files with other encoding than 'latin1'! Do not use the file module to read and write files with other encoding than 'latin1'! Do not use the

IO:FORMAT AND UNICODE

- The `t` modifier makes `~p` and `~s` handle Unicode
  - **`~s`** means Latin-1 characters in lists and byte-wise encoded characters in binaries
  - **`~ts`** means any Unicode characters in lists and UTF-8 binaries (but will fallback to byte-wise encoded binaries if UTF-8 decoding fails)
  - The upside of using `io_lib:format("~s",...)` is that the return value is guaranteed to be an `iolist()`!
  - **`~p`** takes any data and will try to display "string-data" as literal strings as long as they're in the Latin-1 range and binaries are byte-wise encoded.
  - **`~tp`** takes any data and will try to display "string-data" as literal strings both for byte-wise and UTF-8 encoded binaries.
    - The "printable range" parameter to Erlang controls which characters are considered printable:
      - **`erl +pc latin1`** - Only Latin-1 characters
      - **`erl +pc unicode`** - Any printable Unicode character

IO:FORMAT EXAMPLE

```
$ erl +pc latin1
1> io:format("~p~n",[{<<"Åkerspöke">>,
                      <<"Åkerspöke"/utf8>>,
                      <<"ὈΔΥΣΣΕΎΣ"/utf8>>}]).
{<<"Åkerspöke">>,
 <<195,133,107,101,114,115,112,195,182,107,101>>,
 <<225,189,136,206,148,206,165,206,163,
   206,163,206,149,206,142,206,163>>}
ok
2> io:format("~tp~n",[{<<"Åkerspöke">>,
                       <<"Åkerspöke"/utf8>>,
                       <<"ὈΔΥΣΣΕΎΣ"/utf8>>}]).
{<<"Åkerspöke">>,<<"Åkerspöke"/utf8>>,
 <<225,189,136,206,148,206,165,206,163,
   206,163,206,149,206,142,206,163>>}
ok
```

IO:FORMAT EXAMPLE

```
$ erl +pc unicode
1> io:format("~tp~n",[{<<"Åkerspöke">>,
                       <<"Åkerspöke"/utf8>>,
                       <<"ὈΔΥΣΣΕΎΣ"/utf8>>}]).
{<<"Åkerspöke">>,<<"Åkerspöke"/utf8>>,<<"ὈΔΥΣΣΕΎΣ"/utf8>>}
ok
2> io:format("~tp~n",[{<<"Åkerspöke">>,
                       <<"Åkerspöke"/utf8>>,
                       lists:seq(2710,2720)}]).
{<<"Åkerspöke">>,<<"Åkerspöke"/utf8>>,"ખગઘઙચછજઝઞટઠ"}
ok
```

IO:FORMAT EXAMPLE

```
$ erl +pc unicode
1> io:format("~tp~n",[{<<"Åkerspöke">>,
                       <<"Åkerspöke"/utf8>>,
                       <<"ὈΔΥΣΣΕΎΣ"/utf8>>}]).
{<<"Åkerspöke">>,<<"Åkerspöke"/utf8>>,<<"ὈΔΥΣΣΕΎΣ"/utf8>>}
ok
2> io:format("~tp~n",[{<<"Åkerspöke">>,
                       <<"Åkerspöke"/utf8>>,
                       lists:seq(2710,2720)}]).
{<<"Åkerspöke">>,<<"Åkerspöke"/utf8>>,"ખગઘઙચછજઝઞટઠ"}
ok
```

Wat?

THE SHELL

- The shell will handle Unicode characters > 255 if..
  - Your terminal handles UTF-8 (or you run werl) and..
    - Your `LANG` or `LC_CTYPE` environment indicates UTF-8 and...
    - You use “new shell”
  - You specifically do

    ```
    1> io:setopts([{encoding,utf8}])
    ```

    - Which will make any standard_io server output and input data in UTF-8 (regardless of `-oldshell` or `-noshell`)
- Check with

  ```
  1> io:getopts()
  ```

- Read Stdlib Users Guide: Using Unicode In Erlang for troubleshooting tips!

FILE NAMES

- File names in the full Unicode range comes in at least two flavours:
- Transparent file naming
  - Encoding of file names is by convention (usually UTF-8)
  - The convention can not be safely assumed from LANG, LC_CTYPE or anything else
  - Linux behaves like this
- Mandatory Unicode file names
  - Some encoding is enforced
  - Different solutions on Windows and Mac, but the Erlang programmer should not need to know...

SO...

- When in doubt, Erlang regards all file names to be byte-wise encoded
- Erlang is in doubt when the OS uses transparent file naming
- On Mac and Windows, all file names are considered to be in Unicode
  - But on Mac Erlang also have to handle the graphemes...
- The brave can turn on Unicode file names on any platform:
  - `erl +fnu`
  - `erl +fnuw`
- The semi-brave can determine file name encoding from the environment
  - `erl +fna`
  - `erl +fnai`
- **`+fn{l|a|w}`** also affects environment variables, parameters and `open_port({spawn_executable, ...}, ...)`

WHAT ABOUT MY SOURCE FILES?

- The Erlang source code can be in one of two encodings:
  - UTF-8
  - Byte-wise (latin1)
- UTF-8 encoded source does ***not*** mean that the language accepts the full Unicode character set everywhere...
- ...but you ***can*** write string literals in the full Unicode range if your source is UTF-8 encoded
- Encoding is selected on one of the two first lines in the file with a comment matching:

  `“coding\s*[:=]\s*([-a-zA-Z0-9])+”`
- Examples:
  - `%% coding: UTF-8`
  - `%% I’ve settled for encoding = Latin-1`
  - `%% With coding: Latin-1 I mean byte-wise`

THE LITERALS...

- So, you have UTF-8 encoded source files...
- `"αβ"` works perfect
- `<<"αβ"/utf8>>` also works perfect
- `<<"αβ">>` is horror!
  - The bytes get truncated to eight bits (as always in bit syntax)!

THE LITERALS...

- So, you have UTF-8 encoded source files...
- `"αβ"` works perfect
- `<<"αβ"/utf8>>` also works perfect
- `<<"αβ">>` is horror!
  - The bytes get truncated to eight bits (as always in bit syntax)!

With byte-wise
encoding, this would
have created an UTF-8
binary by accident.

THE LITERALS...

- So, you have UTF-8 encoded source files...
- `"αβ"` works perfect
- `<<"αβ"/utf8>>` also works perfect
- `<<"αβ">>` is horror!
  - The bytes get truncated to eight bits (as always in bit syntax)!

With byte-wise encoding, this would have created an UTF-8 binary by accident.

Do not ever, ever utilise that.

THE LITERALS...

- So, you have UTF-8 encoded source files...
- `"αβ"` works perfect
- `<<"αβ"/utf8>>` also works perfect
- `<<"αβ">>` is horror!
  - The bytes get truncated to eight bits (as always in bit syntax)!

With byte-wise encoding, this would have created an UTF-8 binary by accident.

Do not ever, ever utilise that. Ever!

WHAT'S NOT THERE

- Erlang does not handle graphemes that consist of more than one Unicode code point, except in handling file names on Mac
- Erlang also has no notion of language, why to_upper and to_lower is not implemented

WHAT'S NOT THERE

- Erlang does not handle graphemes that consist of more than one Unicode code point, except in handling file names on Mac
- Erlang also has no notion of language, why to_upper and to_lower is not implemented

Wat?

WHAT'S NOT THERE

- Erlang does not handle graphemes that consist of more than one Unicode code point, except in handling file names on Mac
- Erlang also has no notion of language, why to_upper and to_lower is not implemented

Open source projects to the rescue:

Wat?

WHAT'S NOT THERE

- Erlang does not handle graphemes that consist of more than one Unicode code point, except in handling file names on Mac
- Erlang also has no notion of language, why to_upper and to_lower is not implemented

Wat?

Open source projects to the rescue:

i18n - a mapping to C libraries for localization by Michael Uvarov

WHAT'S NOT THERE

- Erlang does not handle graphemes that consist of more than one Unicode code point, except in handling file names on Mac
- Erlang also has no notion of language, why to_upper and to_lower is not implemented

Wat?

Open source projects to the rescue:

i18n - a mapping to C libraries for localization by Michael Uvarov

Elixir - language independent upcase and downcase in module String

UPCASE AND LOWCASE...

```
1>'Elixir.String':downcase(<<"ὈΔΥΣΣΕΎΣ"/utf8>>).
<<"ὀδυσσεύσ"/utf8>>
2>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:from_utf8(<<"ὈΔΥΣΣΕΎΣ"/utf8>>))).
<<"ὀδυσσεύς"/utf8>>
3>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"Maßen"/utf8>>)))).
<<"massen">>
4> i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çaliştiği"/utf8>>
5> i18n_string:to_utf8(
       i18n_string:to_lower('tr_TR',
           i18n_string:to_upper(
               i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştığı"/utf8>>
```

UPCASE AND LOWCASE...

```
1>'Elixir.String':downcase(<<"ὈΔΥΣΣΕΎΣ"/utf8>>).
<<"ὀδυσσεύσ"/utf8>>
```

Correct, when we have no notion of language!

```
2>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:from_utf8(<<"ὈΔΥΣΣΕΎΣ"/utf8>>))).
<<"ὀδυσσεύς"/utf8>>
3>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"Maßen"/utf8>>)))).
<<"massen">>
4> i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalistiği"/utf8>>
5> i18n_string:to_utf8(
       i18n_string:to_lower('tr_TR',
           i18n_string:to_upper(
               i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştığı"/utf8>>
```

UPCASE AND LOWCASE...

```
1>'Elixir.String':downcase(<<"ὈΔΥΣΣΕΎΣ"/utf8>>).
<<"ὀδυσσεύσ"/utf8>>
```

Correct, when we have no notion of language!

```
2>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:from_utf8(<<"ὈΔΥΣΣΕΎΣ"/utf8>>))).
<<"ὀδυσσεύς"/utf8>>
```

Correct, when we have a notion of language!

```
3>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"Maßen"/utf8>>)))).
<<"massen">>
4> i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalîştiği"/utf8>>
5> i18n_string:to_utf8(
       i18n_string:to_lower('tr_TR',
           i18n_string:to_upper(
               i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştığı"/utf8>>
```

UPCASE AND LOWCASE...

```
1>'Elixir.String':downcase(<<"ὈΔΥΣΣΕΎΣ"/utf8>>).
<<"ὀδυσσεύσ"/utf8>>
```

Correct, when we have no notion of language!

```
2>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:from_utf8(<<"ὈΔΥΣΣΕΎΣ"/utf8>>))).
<<"ὀδυσσεύς"/utf8>>
```

Correct, when we have a notion of language!

```
3>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"Maßen"/utf8>>)))).
<<"massen">>
```

Hopeless...

```
4> i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştiği"/utf8>>
5> i18n_string:to_utf8(
       i18n_string:to_lower('tr_TR',
           i18n_string:to_upper(
               i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştığı"/utf8>>
```

UPCASE AND LOWCASE...

```
1>'Elixir.String':downcase(<<"ὈΔΥΣΣΕΎΣ"/utf8>>).
<<"ὀδυσσεύσ"/utf8>>
```

Correct, when we have no notion of language!

```
2>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:from_utf8(<<"ὈΔΥΣΣΕΎΣ"/utf8>>))).
<<"ὀδυσσεύς"/utf8>>
```

Correct, when we have a notion of language!

```
3>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"Maßen"/utf8>>)))).
<<"massen">>
```

Hopeless...

```
4> i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çaliştiği"/utf8>>
```

Wrong language

```
5> i18n_string:to_utf8(
       i18n_string:to_lower('tr_TR',
           i18n_string:to_upper(
               i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştığı"/utf8>>
```

UPCASE AND LOWCASE…

```
1>'Elixir.String':downcase(<<"ὈΔΥΣΣΕΎΣ"/utf8>>).
<<"ὀδυσσεύσ"/utf8>>
```

Correct, when we have no notion of language!

```
2>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:from_utf8(<<"ὈΔΥΣΣΕΎΣ"/utf8>>))).
<<"ὀδυσσεύς"/utf8>>
```

Correct, when we have a notion of language!

```
3>i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"Maßen"/utf8>>)))).
<<"massen">>
```

Hopeless…

```
4> i18n_string:to_utf8(
      i18n_string:to_lower(
          i18n_string:to_upper(
              i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çaliştiği"/utf8>>
```

Wrong language

```
5> i18n_string:to_utf8(
       i18n_string:to_lower('tr_TR',
           i18n_string:to_upper(
               i18n_string:from_utf8(<<"çalıştığı"/utf8>>)))).
<<"çalıştığı"/utf8>>
```

Correct, we have the right locale!

SUMMARY OF SETTINGS

- **`LANG`** and **`LC_CTYPE`** affects your shell and **`standard_io`**/ **`standard_error`** - should reflect your actual terminals capabilities
- **`+pc {unicode`** | **`latin1}`** affects what letters (code points) you want Erlang heuristics to determine as printable (**`~p`** etc)
- **`+fn{l|a|u}`** [{**`w|i|e`**}] tells Erlang how to interpret file names and what to do if decoding fails - should reflect the file system you are running on
- **`epp:default_encoding/0`** tells you what the default source code encoding is
- **`io:setopts/{1,2}`**, **`-oldshell`**, **`-noshell`** sets the accepted character set and encoding of an **`io_server`**

FUTURE WORK

- Unicode file naming default (at least **`+fna`**)
- More options (ranges) to **`+pc`**
- UTF-8 default for source code
- Support for installation in Unicode path
- open_port({spawn, ...}, ...) to handle Unicode
- upcase/lowcase
  - locale support?
  - Simpler approach?
  - Fix dirty scheduling and integrate i18n?
- Atoms in Unicode
- Variables in Unicode (?)
- String library extended/rewritten
- Simpler literal syntax for UTF-8 binaries?

ERICSSON