OCaml-text user manual

Jérémie Dimino

July 16, 2012

Contents

1 Introduction

`ocamlt-text` is a library for manipulation of unicode text. It can replace the `String` module of the standard library when you need to access to strings as sequence of UTF-8 encoded unicode characters.

It also supports encoding (resp. decoding) of unicode text into (resp. from) a lot of different characrter encodings.

2 Character encoding

`ocamlt-text` uses `libiconv` to transcode between variaous character encodings. The `libiconv` itself is quite painfull to use, `ocamlt-text` tries to offer a cleaner interface, which is located in the module `Encoding`.

2.1 Decoding

Decoding means extracting a unicode character from a sequence of bytes. To decode text, the first thing to do is to create a decoder; this is done by the `Encoding.decoder` function:

```
val decoder : Encoding.t -> Encoding.decoder
```

The type `Encoding.t` is the type of character encoding. It is defined as an alias to the type string; in fact it is simply the name of the character encoding, such as ``UTF-8'', ``ASCII'', ...

The decoder allow you to decode characters, by using the decode function:

```
val decode : decoder -> string -> int -> int -> decoding_result
```

It takes as arguments:

- a decoder, of type `Encoding.decoder`
- a buffer *buf*, which contains encoded characters
- an offset *ofs* in the buffer
- a length *len*.

`decode` will read up to *len* bytes in the buffer, starting at the offset *ofs*. If the bytes does not contains a valid multi-byte sequence, it will returns `Dec_error`. If the decoder read *len* bytes without reaching the end of the multi-byte sequence, it returns `Dec_need_more`. If it succeed, it returns `Dec_ok(code_point, num)` where `code_point` is the code-point that has been successfully decoded, ad `num` is the number of bytes consumed.

2.2 Encoding

Encoding means transforming a unicode character into a sequence of bytes, depending of the character encoding. Encoding characters works almost like decoding. The first things is to create a decoder with:

```
val encoder : Encoding.t -> Encoding.encoder
```

then, encoding is done by:

```
val encode : encoder -> string -> int -> int -> code_point -> encoding_result
```

Arguments have the same meaning that for decoding, except that the buffer will be written insteand of read. `encode` will write into the buffer the multi-byte sequence correspoing to the given code-point. On success it returns `Enc_ok num` where `num` is the number of bytes written. If the unicode character cannot be represented in the given encoding, it returns `Enc_error`. If the buffer does not contain enough room for the multi-byte seuqnece, it returns `Enc_need_more`..

2.3 The system encoding

The system character encoding, `Encoding.system` is determined by environment variables. If you print non-ascii text on a terminal, it is a good idea to encode it in the system encoding. You may also use transliteration (see section 2.4) to prevent failling when unicode character cannot be encoded in the system encoding.

2.4 Special encodings

The `libiconv` library allow character encoding names to be suffixed by `//IGNORE` or `//TRANSLIT`. The first one means that character that cannot be represented in given encoding are skipped silently. The secong means that these characters will be approximated. Note that `//TRANSLIT` depends on the current locale settings.

For example, consider the following program:

```
print_endline (Encoding.recode_string
                 ~src:"UTF-8"
                 ~dst:"ASCII//TRANSLIT"
                 "Mon nom est J\xc3\xa9r\xc3\xa9mie")
```

(where `c3a9` is the UTF-8 representation of "é"). According to the current locale settings, the printing will be different:

```
$ LANG=fr_FR.UTF-8 ./a.out
Mon nom est Jeremie
$ LANG=C ./a.out
Mon nom est J?r?mie
```

The advantage of transliteration is that encoding text will never fail, and give an acceptable result.

3 Text manipulation

The `Text` module of `ocamlt-text` is designed to handle unicode text manipulation. By unicode text, we means sequence of unicode characters, and not sequence of bytes. However, to stay compatible with the rest of the ocaml world which uses only standard latin-1 strings, and to keep pattern matching over unicode text, `ocamlt-text` choose to represent text as UTF-8 strings, without using an abstract type. This is OK as long as you respect the following rules:

- **Text is immutable:** never modify in place the contents of a string containing text
- **Never trust inputs:** always check for validity text comming from the outside world
- **Use the right functions:** if you want to iterate over characters of a text, compute the number of characters contained in a text, ... use `UTF-8` aware functions

The module `Text` always assumes that strings it receive contains valid `UTF-8` encoded text. It is your job to ensure it is the case.

3.1 UTF-8 validation

UTF-8 validation consists on verifying whether a string contains valid UTF-8 encoded text. This can be done with one of these two functions:

```
val check : string -> string option
val validate : string -> unit
```

`Text.check` receive a string, and returns `Some error` if the given string does not contains valid UTF-8 encoded text. Otherwise it returns `None`. `Text.validate` does the same thing but raises an exception instead of returning an option.

For example:

```
# Text.check "Hello";;
- : string option = None
# Text.check "\xff";;
- : string option = Some "at offset 0: invalid start of UTF-8 sequence"
# Text.validate "Hello";;
- : unit = ()
# Text.validate "\xff";;
Exception:
Text.Invalid ("at offset 0: invalid start of UTF-8 sequence", "\255").
```

3.2 Iteration

Since UTF-8 encoded character use variable sequence length, iteration over a text can not be done the same way as iteration over a byte array. Indeed, to get the nth character of a text, you need to scan the whole text before the character.

Instead, `ocamlt-text` provides an API to iterate over a text by using pointers (of type `Text.pointer`). A pointer represent the position of a UTF-8 encoded unicode character in a text. You can create a pointer by using one of these three functions:

```
val pointer_l : t -> pointer
  (** Returns a pointer to the left of the given text *)

val pointer_r : t -> pointer
  (** Returns a pointer to the right of the given text *)

val pointer_at : t -> int -> pointer
  (** [pointer_at txt n] returns a pointer to the character at
      position [n] in [txt]. *)
```

Once you have a pointer, you scan text to the right or left. Here is a simple example, where we scan the given text to find a character ".":

```
let search txt =
  let rec loop pointer =
    match Text.next pointer with
      | None ->
          (* End of the text *)
          false
      | Some(".", pointer) ->
          true
      | Some(ch, pointer) ->
          loop pointer
  in
  loop (Text.pointer_l txt)
```

Each call to `Text.next` returns either `None` if the end of the text have been reached, or `Some(ch, pointer)` where:

- `ch` is the character pointed by the pointer
- `pointer` is a pointer to the next character or the end of text

`Text.prev` works in the same way.
Of course, using pointers is the last resort when the functions of the module `Text` are not sufficient.

4 Regular expressions with PCRE

If compiled with support for PCRE, `ocamlt-text` define a syntax extension for writing human readable regular expressions in ocaml sources.

4.1 Enabling the syntax extension

If you are using `ocamlfind`, simply adds the "text.pcre" to the list of packages. For example, to compile a file "foo.ml" using the syntax extension, just type:

```
$ ocamlfind ocamlc -syntax camlp4o -package text.pcre -linkpkg -o foo foo.ml
```

4.2 Syntax of regular expression

Here is the grammar of regular expressions:

- ***string literal*** matches exactly the given string
- _ (underscore) matches any character, except new-line in non multiline mode
- ***regexp regexp*** match the concatenation of the two given regular expression
- ***regexp*** | ***regexp*** matches the first regular expression or the second
- ***regexp{n}*** matches n times the given regular expression
- ***regexp{n-m}*** matches at least n times and up to m times the given regular expression
- ***regexp{n+}*** matches at least n times the given regular expression, and maybe more
- ***regexp**** matches the given regular expression 0 time or more. This is the same as ***regexp{0+}***
- ***regexp+*** matches the given regular expression 1 time or more. This is the same as ***regexp{1+}***
- ***regexp?*** matches the given regular expression 0 time or 1 time. This is the same as ***regexp{0-1}***
- *[* ***character-set*** *]* matches any character of ***character-set***
- *[^* ***character-set*** *]* matches any character that is not a member of ***character-set***
- *(* ***regexp*** *)* matches ***regexp***
- < ***regexp*** does a look behing
- <! ***regexp*** does a negative look behing
- > ***regexp*** does a look ahead (without consuming any character)
- >! ***regexp*** does a negative look ahead (without consuming any character)
- ***regexp as ident*** matches ***regexp*** and bind the result to the variable ***ident***
- ***regexp as ident : type*** matches ***regexp*** and bind the result to the variable ***ident***, mapping it with the function *type_of_string*
- ***regexp as ident := func*** matches ***regexp*** and bind the result to the variable ***ident***, mapping it with the function *func* which may be any ocaml expression
- \ ***ident*** is a backward reference to a previously bounded variable
- ***ident*** matches the regular expression contained in the variable ***ident***
- ***if ident then regexp*** matches ***regexp*** if ***ident*** as been previously matched.
- ***if ident then regexp else regexp*** is the same as the previous contruction but with an else branch.
- &+ ***mode*** enable the given mode
- &− ***mode*** disable the given mode

A *string literal* can be any string, with classic ocaml escape sequence. Moreover, it also support the new escape sequence $\backslash u\{XXXX\}$ where $XXXX$ is a unicode code-point written in hexadecimal. For example $\backslash u\{e9\}$ correspond to the latin-1 character "é".

4.3 Quotations

The syntax extension defines two camlp4 quotations, that might be used in expressions or in patterns. The first one is "`re_text`". It takes a regular expression as defined before and convert it to a string, following the syntax of PCRE.

For example:

```
let re = Pcre.regexp <:re_text< "foo" _* "bar" >>
```

The goal of this quotation is to make regular expression more readable. The second quotation, "`re`" expands into a compiled regular expression, of type `Pcre.regexp`, for examples:

```
let re = <:re< "foo" _* "bar" >>
let f str = Pcre.exec ~rex:<:re< "foo" _* "bar" >> str
```

Note that in both case, the regular expression will be compiled only one time. And in the second example, it will be compiled the first time it is used (by using lazy evaluation).

But the more interesting use of this quotation is in pattern matchings. It is possible to put a regular expression in an arbitrary pattern, and capture variables.

Here is a simple example of what you can do:

```
let rec f = function
  | <:re< "foo" (_* as x) "bar" >> :: _ -> Some x
  | _ :: l -> f l
  | [] -> None
```

If is also possible to use several regular expressions in the same pattern:

```
  match v with
    | <:re< "foo" (_* as x) "bar" >> :: <:re< "a"* " " "b"* >> :: _ -> ...
    ...
```

4.4 Variables

Variables are identifiers, starting with a lower or upper case letter, which are bound to a regular expression. By default `ocamlt-text` defines variables for each posix characters class: `lower`, `upper`, `alpha`, `digit`, `alnum`, `punct`, `graph print`, `blank`, `cntrl`, `xdigit`, `space ascii`, `word`. Each one matches exactly one character. Note that they match only ASCII letters.

`ocamlt-text` also defines variables for unicode properties. For example `Ll`, will match all lowercase letters, including non-ASCII ones. For a list of all unicode properties, look at the manual page `pcresyntax(3)`.

`ocamlt-text` defines variables for each script, such as `Arabic` or `Greek`.

In addition, it defines the following variables:

- `hspace` matching any horizontal space character, including non-ASCII ones
- `vspace` matching any vertical space character, including non-ASCII ones
- `bound` matching any word boundary character, including non-ASCII ones
- `bos` matching the beginning of a subject, whatever the current mode is
- `eos` matching the end of a subject, whatever the current mode is

New variables can be defined by toplevel bindings. For instance:

```
let digit3 = <:re< ["0"-"9"]{3} >>
```

will generate the binding for the `digit3` variable and define the regexp variable `digit3` for the rest of the file.

If the contents of a variable matches text of length 1, it can be used in character set. And if possible, it can be negated by prefixing it with a "`!`". All predefined variables and all character set variables can be negated.

4.5 Modes

Modes may be activated (resp. disabled) by using the syntax "`&+ mode`" (resp. "`&− mode`") in a regular expression. Available modes are:

- *i* or *caseless*: performs case-insentive matching
- *m* or *multiline*: pass into multiline mode; `^` and `$` will match the beginning and end of lines instead of beginning and end of subject
- *s*, *singleline* or *dotall*: the `_` will match any characters, including newline ones.

4.6 Greedy vs possessive vs lazy

The post-operators ?, +, ∗, and more generally {...} may be suffixed with one of ? or + to modify their behaviour. By default regular expressions are greedy, which means that they match the maximum possible they can. Suffixing them with ? will make them lazy, which means the contrary.

For example, consider the following function:

```
let f = function
  | <:re< "a"* as x >> -> Some x
  | _ -> None
```

if we apply it on *aaa* we got:

```
$ f "aaa";;
- : Text.t option = Some "aaa"
```

Now, if we make the matching "*a*"∗ lazy:

```
let f = function
  | <:re< "a"*? as x >> -> Some x
  | _ -> None
```

we got:

```
$ f "aaa";;
- : Text.t option = Some ""
```

Possessive prevents backtracking. For example, with:

```
let f = function
  | <:re< ("a"* as x) ("a" as y) >> -> Some(x, y)
  | _ -> None
```

we got:

```
$ f "aaa";;
- : Text.t option = None
```