Pure Scheme internationalization

January 2025

This is the manual for po-lib version 0.1, a R7RS library for internationalization, last updated 25 January 2025.

Table of Contents

1 Mark strings for internationalization

Let us look at an example:

```
(define (main)
  "Say the line."
  (display "Hello, world!")
  (newline))
```

In this example, you may want to translate the greeting message in the user's language. Import the `(po-lib)` module, and mark the line:

```
(import (po-lib))

(define (main)
  "Say the line."
  (display (gettext "Hello, world!"))
  (newline))
```

Please note that if you are running your code on the Hoot platform, you will need to provide additional webassembly imports. See Chapter 6 [Hoot], page 7.

`gettext` *msgid options ...* [Function]

Translate *msgid* into the current locale. The behavior can be modified with the *options* syntax, according to the following example:

```
(gettext "This is the message to print."
  (domain "my library")
  (context "disambiguation string")
  (plural "These are the ~a messages to print." n-messages)
  (comments "Dear translators, I address this message to you \
regarding the example translation.")
  (source "example-file.scm" 42)
  (range minimum-possible-n-messages maximum-possible-n-messages)
  (extra-flags "scheme-format"))
```

`domain` If you are defining a library, you most likely don't want your translations to clash with those of a dependent library. In order to prevent that from happening, you should make sure to use a domain for your internationalized code. See Section "Solving Ambiguities" in *GNU 'gettext' utilities*.

`context` English can use the same word to mean two different things, where other languages will want to use different strings. The context is not visible to the user, but it will help choose the correct translation. See Section "Using contexts for solving ambiguities" in *GNU 'gettext' utilities*.

`plural` If the message can be singular or plural depending on a run-time value (here, *n-messages*), then the correct translation should be picked according to the plural forms for the current language. The first argument is the alternative plural form in English, and the second argument is the argument of the thing marked plural. See Section "Additional functions for plural forms" in *GNU 'gettext' utilities*.

`comments` It might be a good idea to indicate to the translator some information about how the message will be used. See Section "The Format of PO Files" in *GNU 'gettext' utilities*.

`source` You may want to indicate the source location as a filename and line number, but you maybe should only do that as a macro expansion side-effect.

`range`
`extra-flags`
By default, the plural case can handle any number for the plural argument. However, in certain cases, it is known that the plural argument may be constrained into a range, which can help the translators. Other flags may also be passed. See Section "The Format of PO Files" in *GNU 'gettext' utilities*.

Sometimes you do not want to actually perform the translation, but still mark the string so that it can be translated.

`mark-string` *`msgid options ...`* [Function]
Return *msgid*. The *options* are the same as those for *gettext*, but the run-time plural argument is discarded at the syntax level.

2 Extract marked strings in your code

With the previous example, you may extract the translations:

```
(import (po-lib))

(define (main)
  "Say the line."
  (display (gettext "Hello, world!"))
  (newline))

(dynamic-xgettext
  #f
  "myproject"
  "0.0"
  "msgid-bugs@myproject"
  main)
```

What this will do is return a list of template translations, starting with a header with all the information, and then all translations that were used when calling `main` with no argument.

`dynamic-xgettext` *domain project-name project-version bug-report-address f* [Function]

Return multiple values: a list of template translations with a header first, and all values returned by *f* called with no arguments.

The extraction is dynamic, because strings within dead code will not be detected. It may be difficult to exercise all the translations in a complex program however.

`with-marked-strings` *domain code ...* [Macro]

Prematurely register all the occurences of calls to `gettext` or `mark-string` and then expand *code* in the same lexical scope. Even the messages in deep rarely used code branches will thus be marked for translations, provided the lexical scope of the expansion of `with-marked-strings` is reached during the dynamic-xgettext invocation. For instance,

```
(dynamic-xgettext #f "example" "0.0.0" "test@example.org"
  (lambda ()
    (when #f
      (display (gettext "Oh no, a rare condition occured!")))))
```

will return a translation with only the header template, because the useful branch is never reached, while

```
(dynamic-xgettext #f "example" "0.0.0" "test@example.org"
  (lambda ()
    (with-marked-strings #f
      (when #f
        (display (gettext "Oh no, a rare condition occured!"))))))
```

will return 2 translations templates: the header template, and the message “Oh no, a rare condition occured!”.

3 Save the translation template

Extracting the strings will give you a list of translation templates, which are translations with an empty translated message.

`translations->po` *`translations port?`* [Function]

Write a list of *translations* to *port* in the PO syntax.

4 Maintain translations with GNU Gettext

Once you have a PO template file, you can start translating. See Section "The Translator's View" in *GNU 'gettext' utilities*, for how to do that with GNU Gettext. You need not compile your PO file to MO.

5 Load translations

Once you are done translating, you should have a PO file. Open it as an input port, and read all translations, as a list.

`read-translation` *port?* [Function]
: Read a new translation entry from *port*, or the current input port. If *port* is at the end, return the end-of-file object.

In order to be usable, the translations must be registered to the run-time translator.

`current-po-file-index` [Variable]
: This parameter controls which PO files are used. Provided that you get multiple PO files, one for each language, you will have a list of list of translations. You can parameterize this with such a value. The PO files will be indexed in a binary search tree for efficient retrieval.

`current-languages` [Variable]
: This parameter holds a list of languages to try in order. They can have a regional component, separated from the language by an underscore character. It is initialized with the value of the `LANG` environment variable.

`detect-current-languages` [Function]
: When loading the po-lib library, the `current-languages` variable is initialized by the environment. In most cases, it is the content of the `LANG` environment variable, and on Hoot, it is based on the `navigator.languages` variable.

  Return the list of language specified by the environment.

6 How to use po-lib on Hoot

Hoot is a compiler from Scheme to WebAssembly. In order to process the `navigator.languages` variable, the WebAssembly module where po-lib is included will require the following imports:

`document.navigatorLanguagesRef`
: This function takes no argument and returns an array of strings. Presumably you will want to return `navigator.languages`.

`document.arrayLength`
: Given an array as its sole argument, this function should return its length as a 32-bit integer.

`document.stringArrayRef`
: Given an array *languages* and a 32-bit integer index *i*, this function should return the string *`languages`*`[`*`i`*`]`.

7 Testing the library

The library comes with unit tests.

`run-unit-tests` *runner?* [Function]
: Run the stand-alone po-lib test suite with a SRFI-64 *runner*, or if missing, a new simple test runner.

`unit-tests` [Function]
: Run the unit tests as part of a wider test suite.

Index