elisp literate library

a literate programming tool to write Emacs lisp codes in org mode.

Jingtao Xu

August 4, 2019

Contents

1 Introduction

An Emacs library or configuration file can write in org mode then tangle to an elisp file later, here is one example: Emacs configurations written in Org mode .

But What if I want to write a library or a configuration file in org file and load it to Emacs directly? If it can, then we will have an uniform development environment without keeping multiple copies of codes. Furthermore, we can jump to the elisp definition in an org file directly when required. That will be a convenient way for our daily development.

So is this library, which extends the Emacs load mechanism so Emacs can load org files as lisp source files directly.

2 How to do it?

In org mode, the Emacs lisp codes surround by lines between `#+begin_src elisp` and `#+end_src` (see org manual).

```
#+BEGIN_SRC elisp :load no
(message "this is a test.~%")
#+END_SRC
```

So to let Emacs lisp can read an org file directly, all lines out of surrounding by `#+begin_src elisp` and `#+end_src` should mean nothing, and even codes surrounding by them should mean nothing if the header arguments in a code block request such behavior.

Here is a trick, a new Emacs lisp reader function get implemented (by binding elisp variable load-read-function) to replace original read function when using elisp function load to load a org file.

The new reader will make elisp reader enter into org mode syntax, which means it will ignore all lines until it meet #+BEGIN_SRC elisp.

When #+begin_src elisp occur, header arguments for this code block will give us a chance to switch back to normal Emacs lisp reader or not.

And if it switch back to normal Emacs lisp reader, the end line #+END_SRC should mean the end of current code block, if it occur, then the reader will switch back to org mode syntax. if not, then the reader will continue to read subsequent stream as like the original Emacs lisp reader.

3 Implementation

3.1 Preparation

We use common lisp macros, along with ob-core and subr-x functions, in this library

```
(require 'cl-lib)
(require 'ob-core)
(require 'subr-x)
```

There is a debug variable to switch on/off the log messages for this library.

```
(defvar literate-elisp-debug-p nil)
```

There is also a dynamic Boolean variable bounded by our read function while parsing is in progress. It'll indicate whether org mode syntax or elisp mode syntax is in use.

```
(defvar literate-elisp-org-code-blocks-p nil)
```

And the code block begin/end identifiers:

```
(defvar literate-elisp-begin-src-id "#+BEGIN_SRC")
(defvar literate-elisp-end-src-id "#+END_SRC")
(defvar literate-elisp-lang-ids (list "elisp" "emacs-lisp"))
```

This library uses alist-get, which was first implemented in Emacs 25.1.

```
(unless (fboundp 'alist-get)
  (defun alist-get (key alist)
    "A minimal definition of 'alist-get', for compatibility with Emacs < 25.1"
    (let ((x (assq key alist)))
      (when x (cdr x)))))
```

3.2 stream read functions

To give us the ability of syntax analysis, stream read actions such as peek a character or read and drop next character should get implemented.

The input streams are the same streams used by the original elisp read function.

3.2.1 literate-elisp-peek

```
(defun literate-elisp-peek (in)
  "Return the next character without dropping it from the stream.
Argument IN: input stream."
  (cond ((bufferp in)
         (with-current-buffer in
           (when (not (eobp))
             (char-after))))
        ((markerp in)
         (with-current-buffer (marker-buffer in)
           (when (< (marker-position in) (point-max))
             (char-after in))))
        ((functionp in)
         (let ((c (funcall in)))
           (when c
             (funcall in c))
           c))))
```

3.2.2 literate-elisp-next

```
(defun literate-elisp-next (in)
  "Given a stream function, return and discard the next character.
Argument IN: input stream."
  (cond ((bufferp in)
         (with-current-buffer in
           (when (not (eobp))
             (prog1
               (char-after)
               (forward-char 1)))))
        ((markerp in)
         (with-current-buffer (marker-buffer in)
           (when (< (marker-position in) (point-max))
             (prog1
               (char-after in)
               (forward-char 1)))))
        ((functionp in)
         (funcall in))))
```

3.2.3 literate-elisp-position

This functions is a helpful function to debug our library.

```
(defun literate-elisp-position (in)
  "Return the current position from the stream.
Argument IN: input stream."
  (cond ((bufferp in)
         (with-current-buffer in
           (point)))
        ((markerp in)
         (with-current-buffer (marker-buffer in)
           (marker-position in)))
        ((functionp in)
         "Unknown")))
```

3.2.4 literate-elisp-read-until-end-of-line

when read org file character by character, if current line determines as an org syntax, then the whole line should ignore, so there should exist such a function.

Before then, let's implement an abstract method to `read characters repeatly while a predicate is met`.

The ignored string return from this function because it may be useful sometimes,for example when reading header arguments after `#+begin_src elisp`.

```
(defun literate-elisp-read-while (in pred)
  "Read and return a string from the input stream, as long as the predicate.
Argument IN: input stream.
Argument PRED: predicate function."
  (let ((chars (list)) ch)
    (while (and (setq ch (literate-elisp-peek in))
                (funcall pred ch))
      (push (literate-elisp-next in) chars))
    (apply #'string (nreverse chars))))
```

Now reading until end of line is easy to implement.

```
(defun literate-elisp-read-until-end-of-line (in)
  "Skip over a line (move to `end-of-line').
Argument IN: input stream."
  (prog1
    (literate-elisp-read-while in (lambda (ch)
                              (not (eq ch ?\n))))
    (literate-elisp-next in)))
```

3.3 handle org mode syntax

3.3.1 code block header argument `load`

There are a lot of different elisp codes occur in one org file, some for function implementation, some for demo, and some for test, so an org code block header argument `load` to decide to read them or not should define,and it has two meanings:

- yes
  It means that current code block should load normally, it is the default mode when the header argument `load` is not provided.

- no
  It means that current code block should ignore by elisp reader.
- test
  It means that current code block should load only when variable literate-elisp-test-p is true.

```
(defvar literate-elisp-test-p nil)
```

Now let's implement above rule.

```
(defun literate-elisp-load-p (flag)
  "Load current elisp code block or not.
Argument FLAG: flag symbol."
  (cl-case flag
    ((yes nil) t)
    (test literate-elisp-test-p)
    (no nil)
    (t nil)))
```

Let's also implement a function to read header arguments after #+BEGIN_SRC elisp, and convert every key and value to a elisp symbol(test is here:ref:test-literate-elisp-read-header-arguments).

```
(defun literate-elisp-read-header-arguments (arguments)
  "Read org code block header arguments as an alist.
Argument ARGUMENTS: a string to hold the arguments."
  (org-babel-parse-header-arguments (string-trim arguments)))
```

Let's define a convenient function to get load flag from the input stream.

```
(defun literate-elisp-get-load-option (in)
  "Read load option from input stream.
Argument IN: input stream."
  (let ((rtn (alist-get :load
                        (literate-elisp-read-header-arguments
                         (literate-elisp-read-until-end-of-line in)))))
    (when (stringp rtn)
      (intern rtn))))
```

3.3.2 fix of invalid-read-syntax

Emacs original read function will try to skip all comments until it can get a valid elisp form, so when we call original read function and there are no valid elisp form left in one code block, it may reach #+end_src, as it don't know how to read it, it will signal an error description (invalid-read-syntax "#"). So when such error occur, we have to handle it(test is here:ref:test-empty-code-block).

Please note that the stream position is just after the character # when above error occur.

```
(defmacro literate-elisp-fix-invalid-read-syntax (in &rest body)
  "Fix read error `invalid-read-syntax'.
Argument IN: input stream.
Argument BODY: body codes."
  (declare (indent 1)
           (debug ([&or bufferp markerp symbolp stringp "t"] body)))
  (let ((ex (make-symbol "ex")))
    `(condition-case ,ex
         ,@body
       (invalid-read-syntax
        (when literate-elisp-debug-p
          (message "reach invalid read syntax %s at position %s"
                   ,ex (literate-elisp-position in)))
        (if (equal "#" (second ,ex))
            ;; maybe this is #+end_src
            (literate-elisp-read-after-sharpsign in)
          ;; re-throw this signal because we don't know how to handle it.
          (signal (car ,ex) (cdr ,ex)))))))
```

3.3.3 handle prefix spaces.

Sometimes #+begin_src elisp and #+end_src may have prefix spaces, let's ignore them carefully.

If it is not processed correctly, the reader may enter into an infinite loop, especially when using a custom reader to tangle codes.

```
(defun literate-elisp-ignore-white-space (in)
  "Skip white space characters.
Argument IN: input stream."
  (while (cl-find (literate-elisp-peek in) '(?\n ?\ ?\t))
    ;; discard current character.
    (literate-elisp-next in)))
```

3.3.4 alternative elisp read function

When tangling org file, we want to tangle elisp codes without changing them(but Emacs original `read` will), so let's define a variable to hold the actual elisp reader used by us then it can be changed when tangling org files(see ref:literate-elisp-tangle-reader).

```
(defvar literate-elisp-read (symbol-function 'read))
```

We don't use the original symbol `read` in `literate-elisp-read` because sometimes function `read` can be changed by the following elisp code

```
(fset 'read (symbol-function 'literate-elisp-read-internal))
```