elisp literate library

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

Jingtao Xu

December 6, 2018

Contents

1 Introduction

An emacs library or configuation file can be written in org mode then is tangled 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 loaded by emacs directlly? If it can be done, then we

will have an uniform development environment without keep multiple copies of codes. Furturemore, we can jump to the definition into the org file directlly to change them. That will be an convinient way for our daily development.

2 How to do it?

In org mode, the comment line start with character # (see org manual), and the emacs lisp codes are surrounded by lines between `#+begin_src elisp` and `#+end_src` (see org manual).

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

So to let emacs lisp can read an org file directlly, all lines out of surrounding by `#+begin_src elisp` and `#+end_src` should be ignored, and even codes surrounding by them should be ignored if the options in a code block request such behaviour.

Here is a trick, a new emacs lisp reader function is specified (by binding elisp variable load-read-function) to replace original `read` function when using elisp function `load` to load this org file.

It will make elisp reader enter into org mode syntax, then ignore all lines after that until it meet `#+BEGIN_SRC lisp`.

When `#+begign_src elisp` is met, all org options for this code block will be read and it 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 be checked, if it is, then emacs lisp reader will switch back to org mode syntax. if it is not, emacs lisp reader will continue to read subsequent stream as like the original emacs lisp reader.

3 Implementation

3.1 Preparation

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

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

a dynamic boolean variable to be bound by our read functions while parsing is in progress. It'll indicate whether org mode syntax is used or elisp mode syntax is used in an elisp code block.

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

3.2 stream read functions

stream read functions such as character peek or next are required to read from input streams, which is the same input stream used by the original elisp read function.

3.2.1 literate-peek

```
(defun literate-peek (in)
  "return the next character without dropping it from the 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-next

```
(defun literate-next (in)
  "Given a stream function, return and discard the next character."
  (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-read-while

```
(defun literate-read-while (in pred)
  "Read and return a string from the input stream, as long as the
predicate--which will be called for each character--returns
```

```
true."
  (let ((chars (list)) ch)
    (while (and (setq ch (literate-peek in))
                (funcall pred ch))
      (push (literate-next in) chars))
    (apply #'string (nreverse chars))))
```

3.2.4 literate-skip-to-end-of-line

```
(defun literate-skip-to-end-of-line (in)
  "Skip over a comment (move to end-of-line)."
  (prog1
    (literate-read-while in (lambda (ch)
                              (not (eq ch ?\n))))
    (literate-next in)))
```

3.3 handle org mode syntax

3.3.1 source code block option `tangle`

There are many different elisp codes are written in one org file, some for function implementation, some for demo, and some for test, so an org code block option is defined to decide to read them or not. For example, if one elisp code block is used for demo, then it should be ignored when loading this org file.

a new org code block option `tangle` is defined after `#+BEGIN_SRC elisp`,and it has three meanings:

- yes
  It means that current code block should be read normally, it is the default mode when the option `tangle` is not provided.
- no
  It means that current code block should be ignored by lisp reader.

```
(defun literate-tangle-p (flag)
  (case flag
    (no nil)
    (t t)))
```

Let's implement a function to read options after `#+BEGIN_SRC`, and convert every key and value to a elisp symbol.

```
(defun literate-read-org-options (options)
  (loop for token in (split-string options)
        collect (intern token)))
```

3.3.2 basic read routine for org mode syntax.

Let's define the main read routine to read an org mode stream. the basic idea is very simple, ignore all lines out of elisp source block, and be careful about some special characters.

```
(defun literate-read-datum (in)
  "Read and return a Lisp datum from the input stream."
  (let ((ch (literate-peek in)))
    (cond
      ((not ch)
       (error "End of file during parsing"))
      ((eq ch ?\n)
       (literate-next in)
       nil)
      ((and (not literate-inside-org-code-blocks-p)
            (not (eq ch ?\#)))
       (let ((line (literate-skip-to-end-of-line in)))
         (when literate-reader-debug-p
           (message "ignore line %s" line)))
       nil)
      ((eq ch ?\#)
       (literate-read-after-sharpsign in))
      (t (read in)))))
```

3.3.3 how to handle when meet

```
(defvar org-elisp-begin-src-id "#+BEGIN_SRC elisp")
(defun literate-read-after-sharpsign (in)
  (literate-next in)
  (cond ((not literate-inside-org-code-blocks-p)
         (if (loop for i from 1 below (length org-elisp-begin-src-id)
                   for c1 = (aref org-elisp-begin-src-id i)
                   for c2 = (literate-next in)
                   thereis (not (char-equal c1 c2)))
           (progn (literate-skip-to-end-of-line in)
                  nil)
           (let ((org-options (literate-read-org-options (literate-skip-to-end-of-line
               ↪ in))))
             (when literate-reader-debug-p
               (message "found org elisp src block, options:%s" org-options))
             (cond ((literate-tangle-p (getf org-options :tangle))
                    (when literate-reader-debug-p
                      (message "enter into a elisp code block"))
                    (setf literate-inside-org-code-blocks-p t)
                    nil)))))
        (literate-inside-org-code-blocks-p
         (let ((c (literate-next in)))
           (when literate-reader-debug-p
             (message "found #%c inside a org block" c))
           (case c
             (?\+
              (let ((line (literate-skip-to-end-of-line in)))
                (when literate-reader-debug-p
                  (message "found org elisp end block:%s" line)))
              (setf literate-inside-org-code-blocks-p nil))
             (t (read in)))))
        (t
```

```
          (read in))))
```

3.3.4 use the literate reader when load org file

```
(defun literate-read (&optional in)
  (if (and load-file-name
           (string-match "\\.org\\'" load-file-name))
    (literate-read-datum in)
    (read in)))

(defun literate-load (path)
  (let ((load-read-function (symbol-function 'literate-read))
        (literate-inside-org-code-blocks-p nil))
    (load path)))
```

3.3.5 provide a command to load literate org file directlly from emacs

```
(defun literate-load-file (file)
  "Load the Lisp file named FILE."
  ;; This is a case where .elc and .so/.dll make a lot of sense.
  (interactive (list (read-file-name "Load org file: " nil nil 'lambda)))
  (literate-load (expand-file-name file)))
```

3.3.6 byte compile an literate org file (TODO)

```
(defun literate-byte-compile-file (file)
  "byte compile an org file."
  )
```

3.3.7 tangle org file to elisp file

A function is provided to build an emacs lisp file from an org file.

```
(cl-defun literate-tangle (file &optional (el-file (concat (file-name-sans-extension
    ↪ file) ".el")))
  (let* ((source-buffer (find-file-noselect file))
         (target-buffer (find-file-noselect el-file))
         (load-read-function (symbol-function 'literate-read))
         (literate-inside-org-code-blocks-p nil))
    (with-current-buffer target-buffer
      (delete-region (point-min) (point-max))
      (insert ";;; This file is automatically generated by `literate-tangle' from file
          ↪ `"
              (pathname-name file) "." (pathname-type file) "'\n")
      (insert
       (with-output-to-string
           (with-current-buffer source-buffer
             (goto-char (point-min))
             (loop for obj = (literate-read-datum source-buffer)
```

```
                    if obj
                      do (pp obj)
                         (princ "\n")
                    until (eobp)))))
      (save-buffer)
      (kill-current-buffer))))
```

So when a new version of ./literate-elisp.el can be released from this file, the following code should be executed.

```
(literate-tangle "literate-elisp.org")
```