elisp literate library

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

Jingtao Xu

December 7, 2018

Contents

1 Introduction **1**

2 How to do it? **1**

3 Implementation **2**
3.1 Preparation 2
3.2 stream read functions 2
3.3 handle org mode syntax 4
3.4 load org file with new syntax 6
3.5 function to tangle org file to elisp file 7

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 load it to emacs directly? If it can be done, then we will have an uniform development environment without keeping multiple copies of codes. Furturemore, we can jump to the elisp definition in an org file directly when required. That will be a 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 directly, 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 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 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 catched, if it is catchd, then the reader will switch back to org mode syntax. if it is not, then the 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-elisp-debug-p nil)
```

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

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

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` are required.

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

```
(defun literate-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))))
```

3.2.1 literate-next

```
(defun literate-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))))
```

3.2.2 literate-read-until-end-of-line

when read org file character by character, if current line is determined as an org syntax, then the whole line should be ignored, so a function to do so should be provied.

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

```
(defun literate-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-peek in))
                (funcall pred ch))
      (push (literate-next in) chars))
    (apply #'string (nreverse chars))))
```

Now reading until end of line is easy to implement, the ignored string is returned because it may be used sometimes,for example when reading options after `#+begin_src elisp`.

```
(defun literate-read-until-end-of-line (in)
  "Skip over a comment (move to 'end-of-line').
Argument IN: input stream."
  (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 elisp reader.

```
(defun literate-tangle-p (flag)
  "Tangle current elisp code block or not
Argument FLAG: flag symbol."
  (case flag
    (no nil)
    (t t)))
```

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

```
(defun literate-read-org-options (options)
  "Read org code block options.
Argument OPTIONS: a string to hold the 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 file input stream. the basic idea is very simple, ignore all lines out of elisp source block, and be careful about the special character #.

```
(defun literate-read-datum (in)
  "Read and return a Lisp datum from the input stream.
Argment IN: input stream."
  (let ((ch (literate-peek in)))
    (cond
      ((not ch)
       (error "End of file during parsing"))
      ((and (not literate-elisp-org-code-blocks-p)
            (not (eq ch ?\#)))
       (let ((line (literate-read-until-end-of-line in)))
         (when literate-elisp-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

1. The basic logic
   a) if it is inside an org syntax, check if it is #+begin_src elisp
      i. if it is not, continue to use org syntax and ignore this line
      ii. if it is, read source block options for this code block
         A. if it should be tangled, switch to elisp syntax context
         B. if it should not be tangled, continue to use org syntax and ignore this line
   b) if it is inside an elisp syntax, then check if it is #+
      i. if it is, then switch to org mode syntax.
      because this is the only legal meaning when it is equal to '#+end$_{src}$', so the test here is a little simple.
      ii. if it is not, then use original elip reader to read the following stream
      it should be some legal elisp expressions such like #'(lambda ()).
2. The implementation

```
(defvar literate-elisp-begin-src-id "#+BEGIN_SRC elisp")
(defun literate-read-after-sharpsign (in)
  "Read after #.
Argument IN: input stream."
  (literate-next in)
        ;; 1. if it is not inside an elisp syntax
  (cond ((not literate-elisp-org-code-blocks-p)
         ;; 1.1 check if it is '#+begin_src elisp'
         (if (loop for i from 1 below (length literate-elisp-begin-src-id)
                   for c1 = (aref literate-elisp-begin-src-id i)
                   for c2 = (literate-next in)
                   thereis (not (char-equal c1 c2)))
         ;; 1.2. if it is not, continue to use org syntax and ignore this line
           (progn (literate-read-until-end-of-line in)
                  nil)
         ;; 1.3 if it is, read source block options for this code block
           (let ((org-options (literate-read-org-options (
               ↪ literate-read-until-end-of-line in))))
             (when literate-elisp-debug-p
               (message "found org elisp src block, options:%s" org-options))
             (cond ((literate-tangle-p (getf org-options :tangle))
         ;; 1.4 if it should be tangled, switch to elisp syntax context
                    (when literate-elisp-debug-p
                      (message "enter into a elisp code block"))
                    (setf literate-elisp-org-code-blocks-p t)
                    nil)))))
         ;; 1.5 if it should not be tangled, continue to use org syntax and
             ↪ ignore this line
        (t
        ;; 2. if it is inside an elisp syntax
         (let ((c (literate-next in)))
           (when literate-elisp-debug-p
             (message "found #%c inside a org block" c))
           (case c
             ;; 2.1 check if it is ~#+~, which has only legal meaning when it is
                 ↪ equal '#+end_src'
             (?\+
              (let ((line (literate-read-until-end-of-line in)))
                (when literate-elisp-debug-p
                  (message "found org elisp end block:%s" line)))
             ;; 2.2. if it is, then switch to org mode syntax.
              (setf literate-elisp-org-code-blocks-p nil))
             ;; 2.3 if it is not, then use original elip reader to read the
                 ↪ following stream
             (t (read in)))))))
```

3.4 load org file with new syntax

3.4.1 literate reader is used when loading a org file

```
(defun literate-read (&optional in)
  "Literate read function.
Argument IN: input stream."
  (if (and load-file-name
           (string-match "\\.org\\'" load-file-name))
    (literate-read-datum in)
    (read in)))
```

```
(defun literate-load (path)
  "Literate load function.
Argument PATH: target file to load."
  (let ((load-read-function (symbol-function 'literate-read))
        (literate-elisp-org-code-blocks-p nil))
    (load path)))
```

3.4.2 an interactive command is provided to load a literate org file directly from emacs

```
(defun literate-load-file (file)
  "Load the Lisp file named FILE.
Argument FILE: target file path."
  ;; 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)))
```

3.4.3 a function to byte compile a literate org file (TODO)

```
(defun literate-byte-compile-file (file)
  "Byte compile an org file.
Argument FILE: file to compile."
  )
```

3.5 function to 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 &key (el-file (concat (file-name-sans-extension file) "
    ↪ .el")) header tail)
  "Literate tangle
Argument FILE: target file"
  (let* ((source-buffer (find-file-noselect file))
         (target-buffer (find-file-noselect el-file))
         (load-read-function (symbol-function 'literate-read))
         (literate-elisp-org-code-blocks-p nil))
    (with-current-buffer target-buffer
      (delete-region (point-min) (point-max))
      (when header
        (insert header "\n"))
      (insert ";; This file is automatically generated by `literate-tangle' from file `
          ↪ "
              (pathname-name file) "." (pathname-type file) "'\n\n"
              ";;; Code:\n\n")
      (insert
       (with-output-to-string
           (with-current-buffer source-buffer
             (goto-char (point-min))
             (loop for obj = (progn
                               ;; ignore whitespace or newline because reader can't
                                   ↪ handle them.
                               (while (find (char-after) '(?\n ?\ ?\t))
```

```
                                  (forward-char 1))
                                (literate-read-datum source-buffer))
                  if obj
                    do (pp obj)
                       (princ "\n")
                  until (eobp)))))
      (when tail
        (insert "\n" tail))
      (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"
                 :header ";;; literate-elisp.el --- literate program to write elisp
                     ↪ codes in org mode

;; Copyright (C) 2018-2019 Jingtao Xu

;; Author: Jingtao Xu <jingtaozf@gmail.com>
;; Created: 6 Dec 2018
;; Version: 0.1
;; Keywords: elisp literate org
;; URL: https://github.com/jingtaozf/literate-elisp

;;; Commentary:
"
                 :tail "(provide 'literate-elisp)
;;; literate-elisp.el ends here
")
```

The head and tail lines are required by MELPA respository.