elisp literate library

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

Jingtao Xu

December 18, 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/compile org file with new syntax 7
3.5 function to tangle org file to elisp file 9

4 Tests **10**
4.1 Introduction 10
4.2 test cases 11

5 References **11**

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.

2 How to do it?

In org mode, the comment line start with character # (see org manual), and the emacs lisp codes surround 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 mean nothing, and even codes surrounding by them should mean nothing if the options 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 #+begign_src elisp occur, org options 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, so 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 in this library

```
(require 'cl-lib)
```

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)
```

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-peek

```
(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))))
```

3.2.2 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))))
```

3.2.3 literate-position

This functions is a helpful function to debug our library.

```
(defun literate-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-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 predication meet`.

```
(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 return from this function because it may be useful sometimes,for example when reading options after `#+begin_src elisp`.

```
(defun literate-read-until-end-of-line (in)
  "Skip over a line (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 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 option `tangle` to decide to read them or not should define,and it has three meanings:

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

- no
  It means that current code block should ignore by elisp reader.

```
(defun literate-tangle-p (flag)
  "Tangle current elisp code block or not.
Argument FLAG: flag symbol."
  (cl-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(test is here:4.2.2).

```
(defun literate-read-org-options (options)
  "Read org code block options.
Argument OPTIONS: a string to hold the options."
  (cl-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 simple, ignoring all lines out of elisp source block, and be careful about the special character `#`.

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 with our literate reader(test is here:4.2.1).

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

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(see 3.5).

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

It's time to implement the main routine to read literate org file.

```
(defun literate-read-datum (in)
  "Read and return a Lisp datum from the input stream.
Argment IN: input stream."
  (let ((ch (literate-peek in)))
    (when literate-elisp-debug-p
      (message "literate-read-datum to character '%c'(position:%s)."
```

```
            ch (literate-position in)))
  (condition-case ex
       (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-next in)
          (literate-read-after-sharpsign in))
         (t (funcall literate-elisp-read in)))
     (invalid-read-syntax
      (when literate-elisp-debug-p
        (message "reach invalid read syntax %s at position %s"
                 ex (literate-position in)))
      (if (equal "#" (second ex))
        ;; maybe this is #+end_src
        (literate-read-after-sharpsign in)
        ;; re-throw this signal because we don't know how to handle it.
        (signal (car ex) (cdr err)))))))
```

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 tangle, switch to elisp syntax context
         B. if it should not tangle, 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 elisp 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."
        ;; 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 (cl-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 (cl-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))
           (cl-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)
              nil)
             ;; 2.3 if it is not, then use original elip reader to read the
                 ↪ following stream
             (t (funcall literate-elisp-read in)))))))
```

3.4 load/compile org file with new syntax

3.4.1 literate reader is in use when loading a org file

original function `read` will read until it can get a valid lisp form, we will try to keep this behavior.

```
(defun literate-read-internal (&optional in)
  "A wrapper to follow the behavior of original read function.
Argument IN: input stream."
  (cl-loop for form = (literate-read-datum in)
        if form
          do (cl-return form)
             ;; if original read function return nil, just return it.
        if literate-elisp-org-code-blocks-p
```

```
          do (cl-return nil)
             ;; if it reach end of stream.
        if (null (literate-peek in))
          do (cl-return nil)))
```

Now we define the literate read function which will bind to emacs variable load-read-function.

```
(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-internal in)
    (read in)))
```

And the main exported function to do literate load.

```
(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)))
```

If you want to literate load file in batch mode, here it is:

```
(defun literate-batch-load ()
  "Literate load file in `command-line' arguments."
  (or noninteractive
      (signal 'user-error '("This function is only for use in batch mode")))
  (if command-line-args-left
    (literate-load (pop command-line-args-left))
    (error "No argument left for `literate-batch-load'")))
```

3.4.2 an interactive command 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

This feature is under development and not work yet

Currently(2018.12.16) Emacs bytecomp library always use function read to read elisp forms, instead of the function specified by variable load-read-function.so we modify the symbol function of read when byte compiling org file.

```
(defun literate-byte-compile-file (file &optional load)
  "Byte compile an org file.
Argument FILE: file to compile.
Arguemnt LOAD: load the file after compiling."
  (interactive
   (let ((file buffer-file-name)
         (file-dir nil))
     (and file
          (derived-mode-p 'org-mode)
          (setq file-dir (file-name-directory file)))
     (list (read-file-name (if current-prefix-arg
                               "Byte compile and load file: "
                               "Byte compile file: ")
                           file-dir buffer-file-name nil)
           current-prefix-arg)))
  (let ((literate-elisp-org-code-blocks-p nil)
        (load-file-name buffer-file-name)
        (original-read (symbol-function 'read)))
    (fset 'read (symbol-function 'literate-read-internal))
    (unwind-protect
        (byte-compile-file file load)
      (fset 'read original-read))))
```

3.5 function to tangle org file to elisp file

To build an emacs lisp file from an org file without depending on `literate-elisp` library, we need tangle an org file.

Firstly, when tangle elisp codes, we don't want to use original emacs `read` function to read them because it will ignore comment lines and it's hard for us to revert them back to a pretty print code, so we define a new reader function and bind it to variable `literate-elisp-read`

This reader will read codes in a code blocks without changing them until it reach `#+end_src`.

```
(defun literate-elisp-tangle-reader (&optional buf)
  "Tangling codes in one code block.
Arguemnt BUF: source buffer."
  (with-output-to-string
      (with-current-buffer buf
        (when (/= (point) (line-beginning-position))
          ;; if reader still in last line,move it to next line.
          (forward-line 1))
        (loop for line = (buffer-substring-no-properties (line-beginning-position) (
            ↪ line-end-position))
              until (or (eobp) (string-equal (downcase line) "#+end_src"))
              do (loop for c across line
                       do (write-char c))
                 (write-char ?\n)
                 (forward-line 1)))))
```

Now we can tangle the elisp code blocks with the following codes.

```
(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))
         (org-path-name (concat (pathname-name file) "." (pathname-type file)))
         (literate-elisp-read 'literate-elisp-tangle-reader)
         (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 function `literate-tangle'
          ↪ from file `" org-path-name "'.\n"
              ";; It is not designed to be readable by a human and is generated to load
                  ↪  by Emacs directly without library `literate-elisp'.\n"
              ";; you should read file `" org-path-name "' to find out the usage and
                  ↪ implementation detail of this source file.\n\n"
              ";;; Code:\n\n"))

    (with-current-buffer source-buffer
      (goto-char (point-min))
      (cl-loop for obj = (literate-read-internal source-buffer)
               if obj
               do (with-current-buffer target-buffer
                    (insert obj "\n"))
               until (eobp)))

    (with-current-buffer target-buffer
      (when tail
        (insert "\n" tail))
      (save-buffer)
      (kill-current-buffer))))
```

And when a new version of ./literate-elisp.el can release from this file, the following code should execute.

```
(literate-tangle
 "literate-elisp.org"
 :header ";;; literate-elisp.el --- literate program to write elisp codes in org mode
     ↪ -*- lexical-binding: t; -*-

;; Copyright (C) 2018-2019 Jingtao Xu

;; Author: Jingtao Xu <jingtaozf@gmail.com>
;; Created: 6 Dec 2018
;; Version: 0.1
;; Keywords: lisp docs extensions tools
;; URL: https://github.com/jingtaozf/literate-elisp
;; Package-Requires: ((emacs \"24\"))

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

The head and tail lines require by MELPA repository.

4 Tests

4.1 Introduction

We use ERT library to define and run tests. Web service travis ci will load config file ./.travis.yml to run these tests automatically every time there is a new git change.

4.2 test cases

4.2.1 test the empty code block

If one code block is empty, we will use emacs original `read` function, which will read `#+end_src` and signal an error, let's test whether `literate-elisp` can handle it correctly.

```
;; This is a comment line to test empty code block.
```

4.2.2 test literate-read-org-options

```
(ert-deftest literate-read-org-options ()
  "A spec of function to read org options."
  (should (equal (literate-read-org-options " :tangle yes") '(:tangle yes)))
  (should (equal (literate-read-org-options " :tangle no  ") '(:tangle no)))
  (should (equal (literate-read-org-options ":tangle yes") '(:tangle yes))))
```

5 References

- Literate. Programming. by Donald E. Knuth
- Literate Programming a site of literate programming
- Literate Programming in the Large a talk video from Timothy Daly,one of the original authors of Axiom.
- literate programming in org babel
- A collection of literate programming examples using Emacs Org mode
- elisp-reader.el customized reader for Emacs Lisp