elisp literate library

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

Jingtao Xu

October 12, 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)
```

So we can use print debug meesage with this function:

```
(defun literate-elisp-debug (format-string &rest args)
  "Print debug messages if switch is on.
Argument FORMAT-STRING: same argument of Emacs function 'message',
Argument ARGS: same argument of Emacs function 'message'."
  (when literate-elisp-debug-p
    (apply 'message format-string args)))
```

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 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)))
```

3.3.3 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-emacs-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))
```

So we can ensure that `literate-elisp-emacs-read` will always use the original `read` function,which will not be altered when we want to byte compile the org file by function `literate-elisp-byte-compile-file`.

3.3.4 basic read routine for org mode syntax.

It's time to implement the main routine to read literate org file. The basic idea is simple, ignoring all lines out of elisp source block, and be careful about the special character `#`.

On the other side, 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`,but we can't determine whether the original `read` function arrive there after a complete parsing or incomplete parsing, to avoid such condition, we will filter all comments out to ensure original `read` can always have a form to read.

```
(defun literate-elisp-read-datum (in)
  "Read and return a Lisp datum from the input stream.
Argument IN: input stream."

  (literate-elisp-ignore-white-space in)
  (let ((ch (literate-elisp-peek in)))
    (literate-elisp-debug "literate-elisp-read-datum to character '%c'(position:%s)."
                          ch (literate-elisp-position in))

    (cond
      ((not ch)
       (signal 'end-of-file nil))
      ((or (and (not literate-elisp-org-code-blocks-p)
                (not (eq ch ?\#)))
           (eq ch ?\;))
       (let ((line (literate-elisp-read-until-end-of-line in)))
         (literate-elisp-debug "ignore line %s" line))
       nil)
      ((eq ch ?\#)
       (literate-elisp-next in)
       (literate-elisp-read-after-sharpsign in))
      (t
       (literate-elisp-debug "enter into original Emacs read.")
       (funcall literate-elisp-emacs-read in)))))
```

3.3.5 how to handle when meet

We have to be careful when meeting the character `#` and handle different conditions that may occur:

```
(defun literate-elisp-read-after-sharpsign (in)
  "Read after #.
Argument IN: input stream."
  ;;     if it is not inside an elisp syntax
  (cond ((not literate-elisp-org-code-blocks-p)
         ;; check if it is `#+begin_src'
         (if (or (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-elisp-next in)
                          with case-fold-search = t
                          thereis (not (char-equal c1 c2)))
                 (while (memq (literate-elisp-peek in) '(?\s ?\t))
                   (literate-elisp-next in)) ; skip tabs and spaces, return nil
                 ;; followed by `elisp' or `emacs-lisp'
                 (cl-loop with lang = ; this inner loop grabs the language specifier
                          (cl-loop while (not (memq (literate-elisp-peek in) '(?\s ?\t
                              ↪ ?\n)))
                                   with rtn
                                   collect (literate-elisp-next in) into rtn
                                   finally return (apply 'string rtn))
                          for id in literate-elisp-lang-ids
                          never (string-equal (downcase lang) id)))
           ;; if it is not, continue to use org syntax and ignore this line
           (progn (literate-elisp-read-until-end-of-line in)
                  nil)
           ;; if it is, read source block header arguments for this code block and
               ↪ check if it should be loaded.
           (cond ((literate-elisp-load-p (literate-elisp-get-load-option in))
                  ;; if it should be loaded, switch to elisp syntax context
                  (literate-elisp-debug "enter into a elisp code block")
                  (setf literate-elisp-org-code-blocks-p t)
                  nil)
                 (t
                  ;; if it should not be loaded, continue to use org syntax and ignore
                      ↪ this line
                  nil))))
        (t
         ;; 2. if it is inside an elisp syntax
          (let ((c (literate-elisp-next in)))
            (literate-elisp-debug "found #%c inside a org block" c)
            (cl-case c
              ;; check if it is ~#+~, which has only legal meaning when it is equal `#+
                  ↪ end_src'
              (?\+
               (let ((line (literate-elisp-read-until-end-of-line in)))
                 (literate-elisp-debug "found org elisp end block:%s" line))
              ;; if it is, then switch to org mode syntax.
               (setf literate-elisp-org-code-blocks-p nil)
               nil)
              ;; if it is not, then use original elisp reader to read the following
                  ↪ stream
              (t (funcall literate-elisp-emacs-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-elisp-read-internal (&optional in)
  "A wrapper to follow the behavior of original read function.
Argument IN: input stream."
  (cl-loop for form = (literate-elisp-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-elisp-peek in))
          do (cl-return nil)))
```

label:literate-elisp-read Now we define the literate read function which will bind to Emacs variable load-read-function.

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

And the main exported function to do literate load.

```
(defun literate-elisp-load (path)
  "Literate load function.
Argument PATH: target file to load."
  (let ((load-read-function (symbol-function 'literate-elisp-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-elisp-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-elisp-load (pop command-line-args-left))
    (error "No argument left for `literate-elisp-batch-load'")))
```

3.4.2 an interactive command to load a literate org file from Emacs

```
(defun literate-elisp-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-elisp-load (expand-file-name file)))
```

3.4.3 a function to byte compile a literate org file

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-elisp-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-elisp-read-internal))
    (unwind-protect
        (byte-compile-file file load)
      (fset 'read original-read))))
```

After byte compiling an literate org file, it will be compiled to a file with suffix `.org.elc`, after loading such compiled file, Emacs will fail to find the variable or function definition because function `find-library-name` don't treat org file as a source file, so we have to add an advice function to `find-library-name` to fix this issue.

```
(defun literate-elisp-find-library-name (orig-fun &rest args)
  "An advice to make `find-library-name' can recognize org source file.
Argument ORIG-FUN: original function of this advice.
Argument ARGS: the arguments to original advice function."

  (when (string-match "\\(\\.org\\.el\\)" (car args))
    (setf (car args) (replace-match ".org" t t (car args)))
    (literate-elisp-debug "fix literate compiled file in find-library-name :%s" (car
        ↪ args)))
  (apply orig-fun args))
(advice-add 'find-library-name :around #'literate-elisp-find-library-name)
```

3.5 compatibility with other libraries

Our next job is to make `literate-elisp` work with your favorite package. First, we define a function and a macro useful for adding `literate-elisp` support for other libraries.

```
(defun literate-elisp--file-is-org-p (file)
  "Return t if file at FILE is an Org-Mode document, otherwise nil."
  ;; Load FILE into a temporary buffer and see if 'set-auto-mode' sets
  ;; it to 'org-mode' (or a derivative thereof).
  (with-temp-buffer
    (insert-file-contents file t)
    (delay-mode-hooks (set-auto-mode))
    (derived-mode-p 'org-mode)))

(defmacro literate-elisp--replace-read-maybe (test &rest body)
  "A wrapper which temporarily redefines 'read' (if necessary).
If form TEST evaluates to non-nil, then the function slot of 'read'
will be temporarily set to that of 'literate-elisp-read-internal'
\(by wrapping BODY in a 'cl-flet' call)."
  (declare (indent 1)
           (debug (form body)))
  `(cl-letf (((symbol-function 'read)
              (if ,test
                  (symbol-function 'literate-elisp-read-internal)
                ;; 'literate-elisp-emacs-read' holds the original function
                ;; definition for 'read'.
                literate-elisp-emacs-read)))
     ,@body))
```

Then, we implement support for other libraries. These generally take the form of `:around` advice to functions that use `read` in some way (or which call functions that use `read`), so in those cases we will want to use the `literate-elisp--replace-read-maybe` macro to change `read`'s function definition when necessary.

3.5.1 support for Elisp-Refs

```
(with-eval-after-load 'elisp-refs
  (defun literate-elisp-refs--read-all-buffer-forms (orig-fun buffer)
    ":around advice for 'elisp-refs--read-all-buffer-forms',
to make the 'literate-elisp' package comparible with 'elisp-refs'."
    (literate-elisp--replace-read-maybe
        (literate-elisp--file-is-org-p
         (with-current-buffer buffer elisp-refs--path))
      (funcall orig-fun buffer)))
  (advice-add 'elisp-refs--read-all-buffer-forms :around #'
    ↪ literate-elisp-refs--read-all-buffer-forms)

  (defun literate-elisp-refs--loaded-paths (rtn)
    ":filter-return advice for 'elisp-refs--loaded-paths',
to prevent it from ignoring Org files."
    (append rtn
            (delete-dups
             (cl-loop for file in (mapcar #'car load-history)
                      if (string-suffix-p ".org" file)
                         collect file
```

```
                    ;; handle compiled literate-elisp files
                    else if (and (string-suffix-p ".org.elc" file)
                                 (file-exists-p (substring file 0 -4)))
                       collect (substring file 0 -4)))))
  (advice-add 'elisp-refs--loaded-paths :filter-return #'
      ↪ literate-elisp-refs--loaded-paths))
```

3.5.2 support for Helpful

The above support for `elisp-refs` does most of the necessary work for supporting `helpful`; the following is for the edge case of when `helpful` starts expanding macros in a source file to find a definition.

```
  (with-eval-after-load 'helpful
    (defun literate-elisp-helpful--find-by-macroexpanding (orig-fun &rest args)
      ":around advice for `helpful--find-by-macroexpanding',
  to make the `literate-elisp' package comparible with `helpful'."
      (literate-elisp--replace-read-maybe
          (literate-elisp--file-is-org-p
           (with-current-buffer (car args) buffer-file-name))
        (apply orig-fun args)))
    (advice-add 'helpful--find-by-macroexpanding :around #'
        ↪ literate-elisp-helpful--find-by-macroexpanding))
```

3.6 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 to an Emacs lisp file(.el).

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 block without changing them until it reach `#+end_src`.

label:literate-elisp-tangle-reader

```
(defun literate-elisp-tangle-reader (&optional buf)
  "Tangling codes in one code block.
Argument BUF: source buffer."
  (with-output-to-string
      (with-current-buffer buf
        (when (not (string-blank-p
                    (buffer-substring (line-beginning-position)
                                      (point))))
          ;; 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 (string-trim (downcase line)) "#+end_src"))
              do (loop for c across line
```

```
                    do (write-char c))
                 (literate-elisp-debug "tangle elisp line %s" line)
                 (write-char ?\n)
                 (forward-line 1)))))
```

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

```
(cl-defun literate-elisp-tangle (&optional (file (or org-src-source-file-name (
    ↪ buffer-file-name)))
                                 &key (el-file (concat (file-name-sans-extension file)
                                     ↪ ".el"))
                                 header tail
                                 test-p)
  "Literate tangle
Argument FILE: target file"
  (interactive)
  (let* ((source-buffer (find-file-noselect file))
         (target-buffer (find-file-noselect el-file))
         (org-path-name (concat (file-name-base file) "." (file-name-extension file)))
         (literate-elisp-emacs-read 'literate-elisp-tangle-reader)
         (literate-elisp-test-p test-p)
         (literate-elisp-org-code-blocks-p nil))
    (with-current-buffer target-buffer
      (delete-region (point-min) (point-max))
      (when header
        (insert header "\n"))
      (insert ";;; Code:\n\n"
              ";; The code is automatically generated by function `
                  ↪ literate-elisp-tangle' from file `" org-path-name "'.\n"
              ";; It is not designed to be readable by a human.\n"
              ";; It is generated to load by Emacs directly without depending on `
                  ↪ literate-elisp'.\n"
              ";; you should read file `" org-path-name "' to find out the usage and
                  ↪ implementation detail of this source file.\n\n"
              "\n"))

    (with-current-buffer source-buffer
      (save-excursion
        (goto-char (point-min))
        (cl-loop for obj = (literate-elisp-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))))
```

4 Release current library

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

```
(literate-elisp-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: ((cl-lib \"0.6\") (emacs \"24.4\"))

;; This program is free software; you can redistribute it and/or modify
;; it under the terms of the GNU General Public License as published by
;; the Free Software Foundation, either version 3 of the License, or
;; (at your option) any later version.

;; This program is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;; GNU General Public License for more details.

;; You should have received a copy of the GNU General Public License
;; along with this program.  If not, see <http://www.gnu.org/licenses/>.

;;; Commentary:

;; Literate-elisp is an Emacs Lisp library to provide an easy way to use literate
    ↪ programming in Emacs Lisp.
;; It extends the Emacs load mechanism so Emacs can load Org files as Lisp source files
    ↪  directly.
"
                 :tail "(provide 'literate-elisp)
;;; literate-elisp.el ends here
")
```

The head and tail lines require by MELPA repository.
  Now let's check the elisp file to meet the requirement of MELPA.

```
(require 'package-lint)
(with-current-buffer (find-file "literate-elisp.el")
  (checkdoc)
  (package-lint-current-buffer))
```

5 How to insert code block in org file

There are various ways to do it, for example you can extend the org mode's Easy templates to fit your needs.
  I wrote a small Emacs interactive command so it can insert header arguments based on current org properties automatically. Because properties can be inherited from parent sections or whole file scope, so different default value of header arguments can be used.

Let's define a language list we want to support

```
(defvar literate-elisp-language-candidates
    '("lisp" "elisp" "axiom" "spad" "python" "C" "sh" "java" "js" "clojure" "
        ↪ clojurescript" "C++" "css"
      "calc" "asymptote" "dot" "gnuplot" "ledger" "lilypond" "mscgen"
      "octave" "oz" "plantuml" "R" "sass" "screen" "sql" "awk" "ditaa"
      "haskell" "latex" "lisp" "matlab" "ocaml" "org" "perl" "ruby"
      "scheme" "sqlite"))
```

Let's determine the current literate language before inserting a code block

```
(defun literate-elisp-get-language-to-insert ()
  "Determine the current literate language before inserting a code block."
  (or (org-entry-get (point) "literate-lang" t) ;get it from an org property at current
      ↪  point.
      ;; get it from a candidates list.
      (completing-read "Source Code Language: " literate-elisp-language-candidates)))
```

So you can define org property `literate-lang` in a file scope like this in the beginning of an org file

```
#+PROPERTY: literate-lang elisp
```

Or define it in a separate org section with a different default value

```
The is a section for another literate language
:PROPERTIES:
:literate-lang: lisp
:END:
```

Let's define the valid load types for a code block

```
(defvar literate-elisp-valid-load-types '("yes" "no" "test"))
```

Let's determine the current literate load type before inserting a code block

```
(defun literate-elisp-get-load-type-to-insert ()
  "Determine the current literate load type before inserting a code block."
  (or (org-entry-get (point) "literate-load" t) ;get it from an org property at current
      ↪  point.
      (completing-read "Source Code Load Type: " literate-elisp-valid-load-types)))
```

So you can define org property `literate-load` in a file scope like this in the beginning of org file

```
#+PROPERTY: literate-load yes
```

Or define it in a separate org section with a different default value, for example for demo section

```
This is a demo section so don't load codes inside it
#+PROPERTY:
:PROPERTIES:
:literate-load: no
:END:
```

Now it's time to implement the insert command

```
(defun literate-elisp-insert-org-src-block ()
  "Insert the source code block in 'org-mode'."
  (interactive)
  (let ((lang (literate-elisp-get-language-to-insert))
        (load-type (literate-elisp-get-load-type-to-insert)))
    (when lang
      (insert (format "#+BEGIN_SRC %s" lang)
              (if (or (null load-type) (string= "yes" load-type))
                  ""
                (format " :load %s" load-type))
              "\n")
      (newline)
      (insert "#+END_SRC\n")
      (forward-line -2))))
```

You can bind this command to a global key in Emacs like this

```
(global-set-key [f2] 'literate-elisp-insert-org-src-block)
```

6 Tests

6.1 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.

6.2 test cases

6.2.1 test the empty code block

label:test-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 read it gracefully.

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

6.2.2 test code block with prefix space.

Some code blocks have white spaces before `#+begin_src elisp`, let's test whether `literate-elisp` can read it normally.

```
(defvar literate-elisp-a-test-variable 10)
```

Let's write a test case for above code block.

```
(ert-deftest literate-elisp-read-code-block-with-prefix-space ()
  "A spec of code block with prefix space."
  (should (equal literate-elisp-a-test-variable 10)))
```

6.2.3 test code block with lowercase block delimiters

Some code blocks have `#+begin_src elisp` and `#+end_src` in lowercase; let's test whether `literate-elisp` can match it case-insensitively.

```
(defvar literate-elisp-test-variable-2 20)
```

Let's write a test case for above code block.

```
(ert-deftest literate-elisp-read-lowercase-code-block ()
  "A spec of code block with lowercase block delimiters."
  (should (equal literate-elisp-test-variable-2 20)))
```

6.2.4 test code block with `emacs-lisp` instead of `elisp`

Some code blocks use `emacs-lisp` instead of the shortened `elisp` as the language specifier; let's test if `literate-elisp-read-after-sharpsign` matches it properly.

```
(defvar literate-elisp-test-variable-3 30)
```

Let's write a test case for the above code block.

```
(ert-deftest literate-elisp-read-block-with-lang-emacs-lisp ()
  "A spec of code block with the language specifier `emacs-lisp'
instead of `elisp'."
  (should (equal literate-elisp-test-variable-3 30)))
```

6.2.5 test code block with indentation

Some code blocks have indentation on the first line; let's test whether `literate-elisp` can read them normally.

```
(defvar literate-elisp-test-variable-4 40)
```

Let's write a test case for the above code block.

```
(ert-deftest literate-elisp-read-block-with-indentation ()
  "A spec of code block with indentation on the first line."
  (should (equal literate-elisp-test-variable-4 40)))
```

6.2.6 test literate-elisp-read-header-arguments

label:test-literate-elisp-read-header-arguments

```
(ert-deftest literate-elisp-read-header-arguments ()
  "A spec of function to read org header-arguments."
  (should (equal (literate-elisp-read-header-arguments " :load yes") '((:load . "yes"))
      ↪ ))
  (should (equal (literate-elisp-read-header-arguments " :load no  ") '((:load .  "no")
      ↪ )))
  (should (equal (literate-elisp-read-header-arguments ":load yes") '((:load . "yes")))
      ↪ ))
```

6.2.7 test the `:load` header argument

```
(ert-deftest literate-elisp-test-load-argument ()
  (cl-flet ((test-header-args (string)
              (let ((tempbuf (generate-new-buffer " *temp*")))
                (unwind-protect
                    (progn
                      (with-current-buffer tempbuf
                        (insert string)
                        (goto-char 0))
                      (literate-elisp-load-p
                       (literate-elisp-get-load-option tempbuf)))
                  (kill-buffer tempbuf)))))
    (should (test-header-args " :load yes"))
    (should-not (test-header-args " :load no  "))
    (should (test-header-args ":load yes"))))
```

6.2.8 report error message when load incomplete code block

```
(ert-deftest literate-elisp-test-incomplete-code-block ()
  (let ((file (make-temp-file "literate-elisp" nil ".org")))
    (with-current-buffer (find-file-noselect file)
      (insert "# start of literate syntax\n"
              "#+BEGIN_SRC elisp\n"
              "(defn test ()\n"
              " (let \n"
              ")\n"
              "#+END_SRC\n")
      (save-buffer))
    (should-error (literate-elisp-load "test/incomplete-code-block.org"))))
```

7 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