literate lisp

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

Jingtao Xu

February 14, 2019

Contents

1 Introduction

This is a common lisp project to show a way how to use literate programming in common lisp.

It extends the common lisp reader syntax so a common lisp vendor can read org files as common lisp source files.

[literate programming examples](#) show the reason why use org mode, and there are also another lisp project [papyrus](#) to do the similar thing but it use markdown file format.

By using common lisp package [literate-lisp](#) , Emacs [org mode](#) and elisp library [polymode](#), literate programming can be easy in one org file containing both documentation and source codes, and this org file works well with [SLIME](#).

2 How to do it?

In org mode, the comment line start with character # (see [org manual](#)), and the lisp codes exists between #+begin_src lisp and #+end_src (see [org manual](#)).

```
#+BEGIN_SRC lisp :load no
(format t "this is a test.~%")
#+END_SRC
```

So to let lisp can read an org file directly, all lines out of surrounding by #+begin_src lisp 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 [lisp reader](#) syntax for "# "([Sharpsign Whitespace](#)) will have a meaning to make lisp reader enter into org mode syntax, then ignore all lines after that until it meet #+BEGIN_SRC lisp.

When #+begign_src lisp occurs, org [header arguments](#) for this code block give us a chance to switch back to normal lisp reader or not.

And if it switch back to normal lisp reader, the end line #+END_SRC should mean the end of current code block,so the lisp reader syntax for "#+"([Sharpsign Plus](#))will have an additional meaning to determine if it is #+END_SRC, if it is, then lisp reader will switch back to org mode syntax, if it is not, lisp reader will continue to read subsequent stream as like the original lisp reader.

This workflow restricts the org file starting with a comment character and a space character("# "), but it should not be a problem but indeed a convenient way for us to specify some local variables, for example I often put them in the first line of an org file:

```
# -*- encoding:utf-8 Mode: POLY-ORG;  -*- ---
```

Which make Emacs open file with utf-8 encoding and [poly-org-mode](#).

3 Implementation

3.1 Preparation

Firstly a new lisp package for this library is defined.

```
(in-package :common-lisp-user)
(defpackage :literate-lisp
  (:use :cl)
  (:export :tangle-org-file :with-literate-syntax)
  (:documentation "a literate programming tool to write common lisp codes in org file."
    ↪ ))
(pushnew :literate-lisp *features*)
(in-package :literate-lisp)
```

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

```
(defvar debug-literate-lisp-p nil)
(declaim (type boolean debug-literate-lisp-p))
```

And let's define the org code block identifier.

```
(defvar org-lisp-begin-src-id "#+begin_src lisp")
```

3.2 function to handle reader syntax for "# "(# + Space)

There are a lot of different lisp codes occur in one org file, some for function implementation, some for demo, and some for test, so a new org code block header argument `load` 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 header argument `load` is not provided.

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

- test
  It means that current code block should load only when feature `literate-test` exist.

```
(defun load-p (feature)
  (case feature
    ((nil :yes) t)
    (:no nil)
    (:test (find :literate-test *features* :test #'eq))))
```

Let's implement a function to read header arguments after #+BEGIN_SRC lisp, and convert every key and value to a lisp keyword(Test in here: 4.2.1).

```lisp
(defun read-org-code-block-header-arguments (string begin-position-of-header-arguments)
  (with-input-from-string (stream string :start begin-position-of-header-arguments)
    (let ((*readtable* (copy-readtable nil))
          (*package* #.(find-package :keyword))
          (*read-suppress* nil))
      (loop for elem = (read stream nil)
                 while elem
                 collect elem))))
```

Now it's time to implement the new reader function for syntax "#
"(# + Space).

```lisp
(defun sharp-space (stream a b)
  (declare (ignore a b))
  (loop for line = (read-line stream nil nil nil)
        until (null line)
        for start1 = (loop for c of-type character across line
                           for i of-type fixnum from 0
                           until (not (find c '(#\Tab #\Space)))
                           finally (return i))
        do (when debug-literate-lisp-p
             (format t "ignore line ~a~%" line))
        until (when (equalp start1 (search org-lisp-begin-src-id line :test #'
              ↪ char-equal))
                (let* ((header-arguments (read-org-code-block-header-arguments line
                         ↪ (+ start1 (length org-lisp-begin-src-id)))))
                  (load-p (getf header-arguments :load :yes)))))
  (values))
```

3.3 an implementation of original feature test.

This code block reference from the sbcl source codes with some mi-
nor modifications.

```lisp
;;; If X is a symbol, see whether it is present in *FEATURES*. Also
;;; handle arbitrary combinations of atoms using NOT, AND, OR.
(defun featurep (x)
  (typecase x
    (cons
     (case (car x)
       ((:not not)
        (cond
          ((cddr x)
           (error "too many subexpressions in feature expression: ~S" x))
          ((null (cdr x))
           (error "too few subexpressions in feature expression: ~S" x))
          (t (not (featurep (cadr x))))))
       ((:and and) (every #'featurep (cdr x)))
       ((:or or) (some #'featurep (cdr x)))
       (t
        (error "unknown operator in feature expression: ~S." x))))
    (symbol (not (null (member x *features* :test #'eq))))
    (t
     (error "invalid feature expression: ~S" x))))
```

3.4 function to handle reader syntax for "#+"

The mechanism to handle normal lisp syntax "#+" is also referenced from sbcl source codes.

```lisp
(defun sharp-plus (stream sub-char numarg)
  ;; 1. read into the feature as an keyword.
  (let ((feature (let ((*package* #.(find-package :keyword))
                       ;;(*reader-package* nil)
                       (*read-suppress* nil))
                   (read stream t nil t))))
    ;;      2.1 if the feature is '#+END_SRC', then switch back to org syntax.
    (when debug-literate-lisp-p
      (format t "found feature ~s,start read org part...~%" feature))
    (cond ((eq :END_SRC feature)
           (when debug-literate-lisp-p
             (format t "found #+END_SRC,start read org part...~%"))
           (funcall #'sharp-space stream sub-char numarg))
          ;; 2.2 otherwise test the feature.
          ;;   2.2.1 If the feature exist, read the following object recursively
          ;;     ↪ normally.
          ((featurep feature)
           (read stream t nil t))
          ;;   2.2.1 if the feature doesn't exist, read the following object
          ;;     ↪ recursively and ignore it.
          (t
           (let ((*read-suppress* t))
             (read stream t nil t)
             (values))))))
```

3.5 Install the new reader syntax.

Let's use a new read table to hold the reader for org syntax.

```lisp
(defvar *org-readtable* (copy-readtable))
```

Now install the reader function to this read table.

```lisp
(set-dispatch-macro-character #\# #\space #'sharp-space *org-readtable*)
(set-dispatch-macro-character #\# #\+ #'sharp-plus *org-readtable*)
```

3.6 tangle an org file

To build lisp file from an org file, we implement a function tangle-org-file. Argument org-file is the source org file. Argument keep-test-codes is a Boolean value to indicate whether test codes should load. The basic method is simple here, we use function sharp-space to ignore all lines should be ignored, then export all code lines until we reach #+end_src, this process is repeated to end of org file. This mechanism is good enough because it will not damage any codes in org code blocks.

```
(defun tangle-org-file (org-file &key
                        (keep-test-codes nil)
                        (output-file (make-pathname :defaults org-file
                                                    :type "lisp")))
  (let ((*features* (if keep-test-codes
                        *features*
                        (remove :literate-test *features* :test 'eq))))
    (with-open-file (input org-file)
      (with-open-file (output output-file :direction :output
                             :if-does-not-exist :create
                             :if-exists :supersede)
        (format output
                ";;; This file is automatically generated from file '~a.~a'.
;;; It is not designed to be readable by a human.
;;; It is generated to load by a common lisp vendor directly without depending on '
    ↪ literate-lisp'.
;;; Please read file '~a.~a' to find out the usage and implementation detail of this
    ↪ source file.~%~%"
                (pathname-name org-file) (pathname-type org-file)
                (pathname-name org-file) (pathname-type org-file))
        (block read-org-files
          (loop do
            ;; ignore all lines of org syntax.
            (sharp-space input nil nil)
            ;; start to read codes in code block until reach '#+end_src'
            (loop for line = (read-line input nil nil)
                  do
              (cond ((null line)
                     (return-from read-org-files))
                    ((string-equal "#+end_src" (string-trim '(#\Tab #\Space) line))
                     (when debug-literate-lisp-p
                       (format t "reach end of source code block.~%"))
                     (write-line "" output)
                     (return))
                    (t
                     (when debug-literate-lisp-p
                       (format t "read code line:~s~%" line))
                     (write-line line output)))))))))
```

So when a new version of `./tangle.lisp` can release from this file, the following code should execute.

```
(tangle-org-file
 (format nil "~a/tangle.org"
         (asdf:component-pathname (asdf:find-system :literate-lisp))))
```

Listing 1: a demo code to tangle current org file.

3.7 make ASDF handle org file correctly

Firstly, let's define a macro so org syntax codes can be compiled and loaded.

```
(defmacro with-literate-syntax (&body body)
  `(let ((*readtable* *org-readtable*))
     ,@body))
```

Now let's add literate support to ASDF system.
Firstly a new source file class for org files should define in ASDF package.

```
(defclass asdf::org (asdf:cl-source-file)
  ((asdf::type :initform "org")))
(eval-when (:compile-toplevel :load-toplevel :execute)
  (export '(asdf::org) :asdf))
```

So a new ASDF source file type `:org` can define an org file like this

```
(asdf:defsystem literate-demo
  :components ((:module demo :pathname "./"
                        :components ((:org "readme"))))
  :depends-on (:literate-lisp))
```

Listing 2: a demo code to show how to include org file in ASDF.

And file `readme.org` will load as a lisp source file by ASDF.
Then the new reader syntax for org file installs when ASDF actions perform to every org file.

```
(defmethod asdf:perform :around (o (c asdf:org))
  (literate-lisp:with-literate-syntax
    (call-next-method)))
```

Then after loading this package, one org file can load by ASDF automatically.

4 Test cases

4.1 Preparation

Now it's time to validate some functions. The FiveAM library is used to test.
Web service travis ci will load config file ./.travis.yml automatically every time there is a new git change.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (unless (find-package :fiveam)
    #+quicklisp (ql:quickload :fiveam)
    #-quicklisp (asdf:load-system :fiveam)))
(5am:def-suite literate-lisp-suite :description "The test suite of literate-lisp.")
(5am:in-suite literate-lisp-suite)
```

4.2 test groups

4.2.1 test for reading org code block header-arguments

7

```
(5am:test read-org-code-block-header-arguments
  (5am:is (equal nil (read-org-code-block-header-arguments "" 0)))
  (5am:is (equal '(:load :no) (read-org-code-block-header-arguments " :load no  " 0)))
  (5am:is (equal '(:load :no) (read-org-code-block-header-arguments " :load no" 0))))
```

4.3 run all tests in this library

this function is the entry point to run all tests and return true
if all test cases pass.

```
(defun run-test ()
  (5am:run! 'literate-lisp-suite))
```

4.4 run all tests in demo project

To run all tests in demo project literate-demo, please load it by
yourself.

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

- papyrus A common lisp Literate Programming Tool in markdown file