A program for literate Prolog

Richard de Rozario

17 July 2019

Introduction

The SWI Prolog environment enables literate programming in the form of program comments. However, there is another form of literate programming that embeds the program code as specially marked chunks in a text document. A prominent example of this is supported in the R language environment – especially in RStudio, in the form of .Rmd files. These files are documents that are written in the markdown format, with the program embedded in specially marked chunks throughout the text. The document is processed by a program that formats the text and code into a final output like a pdf file. The lit.pro program discussed here supports all these functions for the Prolog language (in particular SWI Prolog).

An additional benefit is that code chunks like examples can be executed during the processing and the results shown in the document. In essence, this supports "programmatic" documents, where part of the text (including diagrams, tables, etc.) is generated by code. This is useful for documents like tutorials or reports. These types of documents are the primary motivation for writing this program.

Lastly, the program can also extract the code into a "sourcecode only" file that can be executed by the language interpreter or compiler. The extraction process is flexible through the use of "tags" in the code chunks, which enable extraction of chunks that have particular tags, like "example" or "main_code". Also, if the name of a code block is a full file/path name, then that code block will be extracted to that specific file, rather than appended to the default file.

This document is itself the literate program (lit.pro) that performs these functions. In other words, the text documents the source code, which is presented in chunks throughout the document. In essence, this document is the reference guide to the program.[1]

[1] In order to avoid predicate naming clashes, all the predicates in the actual lit.pro source code have been renamed with an _4lit suffix. This enables lit to process documents without accidentally running into predicate name clashes from embedded code chunks. In this document, the suffixes are not present for readability.

A brief usage example

To demonstrate the main features, I'll write a small program and show how to process it with lit.pro. First, a small "hello world" program, written in a literate Prolog document. The text shown in Listing 1 would be saved in a file called, for example, hello.pmd, which is basically a markdown text file, with chunks of prolog code amongst the text.

Listing 1: hello world literate Prolog example

```
---
title: Example prolog markdown
author: RdR
date: 2019-07-07

---

# Introduction
This is example Prolog markdown text that is
processed by the lit.pro program.  The main text
is formatted in markdown and the code is demarcated
by double-percent signs.


%% mycode "hello world"

main :- hello( world ).
hello(X):- write('**'),write('hello '), write(X), writeln('**').

:- main.

%%

The code shown in Listing \ref{mycode} is a
simple Prolog program.  The code is executed when this
document is processed by the lit.pro program.
```

If the text in Listing 1 exists in a file called "hello.pmd", then on the commandline the following would process the file and produce a nicely formatted pdf document.[2]

```
./lit hello.pmd
```

The text in the pdf will appear formatted much like the document you're reading now. The code block would appear as follows:

[2] Assuming that you have pandoc installed.

Listing 2: hello world

```
1
2 main :- hello( world ).
3 hello(X):- write('**'),write('hello '), write(X), writeln('**').
4
5 :- main.
```

hello world

Notice that the output of the program ("hello world") appears just after the code block. That is, the output of the program (if there is any) is injected into the text of the document. By default, lit.pro will provide line numbers in the margins for code listings. To turn that off, you can include the tag "nonum" like the following:

```
%% mycode "hello world" nonum
```

The syntax of the line that marks the start of a code block is very simple:

```
%% LABEL [CAPTION] [TAG]*
```

In other words, the double-percent marker followed by a label of the code chunk and optionally a caption for the listing and zero or more tags. Using the pandoc format of markdown, the label of the code chunk can also be used for references to the listing, for instance the use of \ref{mycode} in Listing 1.

Tags can be any alphanumeric (including underscores) word, starting with a letter. The following are reserved tags:

- nonum — list code without line numbers
- nolist — don't list the code (but still evaluate it)
- noeval — list the code, but don't evaluate it
- skip — do not list or evaluate the code

Other than the reserved tags, you're free to use any word to tag your code. Aside from searching for relevant code, tags also enable you to process or extract only certain chunks. For example, let's say that you have both the main program code, as well as example code in your document. You might use tags "main" and "example" to distinguish between them. Then, you could extract all the main code from the document into a separate .pro file from the commandline as follows:

```
./lit -x -t main myprogram.pmd
```

The annotated source code of lit.pro

The following sections will discuss the source code of the lit.pro program.

The "main" section

There are two global declarations:

- the gensym library is loaded, because we need to generate temporary unique filenames when processing chunks of code. The chunks are stored into a temporary file and then consulted into the Prolog environment.
- b/2 is a dynamic predicate used for temporarily storing lines of code (via Prolog's assertz), before writing them out to a file.

The main predicate provides the top-level program starting point. It sets up the error handling and then calls args(Args), submain(Args), which grabs the commandline arguments and further processes them in submain.

Submain handles three things: process the syntax of the commandline arguments, call the core routine for processing the text in the document (lit/5), or print the help message if processing cannot occur for some reason.

Listing 3: program main section

```
% lit.pro
%   Enable literate Prolog programs, where the file is a markdown text,
%   interspersed with chunks of code.  Small snippets of code can also
%   be embedded in the text inside backquotes.  The program will interpret
%   the code, insert any output and display code as directed.
%
%   To compile: swipl -o lit -g main -c lit.pro
%

:- ensure_loaded(library(gensym)).
:- dynamic(b/2).

main:-
    catch( (args(Args), submain(Args)), Err, (writeln(Err),fail)),
    halt.
main:- halt(1).

args(Args):-
    current_prolog_flag(os_argv,[_,_,_,_|Args]),!.
args([]):-!.

%! submain(-Args:list) is det.
%   Submain extracts the arguments from the command line
%   and calls the core program
submain(Args):-
    ( argline(Infile,Outfile,CodeOnly,Tags,Postproc,Args,_) ->
        lit(Infile,Outfile,CodeOnly,Tags,Postproc)
    ;   print_arghelp   % no args at all
```

29) ,!.

The commandline arguments

The commandline arguments appear as a list of atoms, which are handled with a DCG grammar. The intent of the grammar is to extract the variables Infile, Outfile, CodeOnly, Tags and Postproc, which are explained below. The grammar essentially handles each of the dash-options (which may occur in any order):

- -h or –help — these simply print out the help message, although they are superfluous because just executing the program without any arguments will also print the help text.
- -x — this option will extract the code into a separate sourcecode file with the same name, but with an extension of .pro (unless a specific output filename is provided, or unless the codeblock specifies a filepath/filename). This option also interacts with tags (if they are provided) and will only extract code that contains one or more of the tags specified. Programmatically, the grammar handles the option by setting the CodeOnly parameter to true.
- -p — this option enables the user to set their own post-processing command. The lit program only transforms a .pmd program (markdown text + Prolog code) into a regular .md file. That .md file can then be further processed into, say, a .pdf file. By default, we use pandoc for this post-processing, but the user may wish to use some other program. Programmatically, the post-processing command is a string on the commandline that is stored as an atom in the Postproc variable of the grammar.
- -t — this option enables the user to provide a (comma separated) list of tags. Some tags are reserved words (like "nonum" which lists code without line numbers), but most are used to select which code chunks should be processed. Any code chunk that has one or more of the tags specified on the commandline will be processed.
- -o — this option specifies an outputfile. If the option is not specified, the program will use the input file name, but with the .md extension for the output file name (unless the -x option is used, in which case the .pro extension is used by default).
- inputfilename — the input file name is required for any processing to occur. The naming convention for the input file name is a file with the .pmd extension. To prevent accidental overwriting of files, we don't allow input files with either a .md or a .pro extension.

The grammar is straigthforward. One minor special case is the handling of comma-separated tags. Since by convention most people don't put a space before a comma, the comma ends up attached to the atom that is the tag. We separate the two with the commaed predicate.

Listing 4: the argument line grammar

```
%! argline(?Infile,?Outfile,?CodeOnly,Tags:list,Toks:list,Rest:list) is semidet.
%   The command line argument grammar.
argline(_,_,_,_,_) --> ['-h'], {print_arghelp}, !.
argline(_,_,_,_,_) --> ['--help'], {print_arghelp}, !.
argline(Infile,Outfile,true,Tags,Postproc) -->    % extract code only
    ['-x'],
    !, arglinetail(Infile, Outfile, true, Tags,Postproc).
argline(Infile,Outfile,CodeOnly,Tags,Postproc) -->
    ['-p',Postproc],
    !, arglinetail(Infile, Outfile,CodeOnly,Tags,Postproc).

% process code with any of these tags only
argline(Infile,Outfile,CodeOnly,Tags,Postproc) -->
    ['-t'],[T],
    {   T\='-x',T\='-o',T\='-h',T\='--help',
        file_name_extension(T,'',T),
        (commaed(T,Tag) -> Tag=Tag; Tag=T )
    },
    more_tags(Ts), {append([Tag],Ts,Tags)},
    !, arglinetail(Infile, Outfile, CodeOnly, Tags,Postproc).

argline(Infile,Outfile,CodeOnly,Tags,Postproc) -->
    ['-o',Outfile],
    {    not(access_file(Outfile,write)) ->
            throw('Error: cannot access output file');!
    },!, arglinetail(Infile, Outfile,CodeOnly,Tags,Postproc).

argline(Infile,Outfile,CodeOnly,Tags,Postproc) -->
    [Infile],
    {    not(access_file(Infile,read)) ->
            throw('Error: cannot access input file');!
    },!, arglinetail(Infile, Outfile,CodeOnly,Tags,Postproc).

arglinetail(_,_,_,_,_,[],[]):-!.
arglinetail(In,Out,Code,Tags,Post,S,F):-
    argline(In,Out,Code,Tags,Post,S,F).

more_tags(Tags) --> [','], more_tags(Tags),!.
more_tags([Tag|Tags]) -->
    [T],
    {   T\='-x',T\='-o',T\='-p',T\='-h',T\='--help',
        file_name_extension(T,'',T),
        (commaed(T,Tag) -> Tag=Tag; Tag=T )
    },!,
```

```
45      more_tags(Tags).
46  more_tags([]) --> {}.
47
48  commaed(Word,Atom):-
49      atom_chars(Word,Cs),
50      append(As,[',','],Cs),
51      atom_chars(Atom,As),!.
52
53  %! print_arghelp is det.
54  %   display the command line help
55  print_arghelp:-
56      nl,
57      writeln('Literate Prolog programs using markdown files.'),
58      writeln('usage: lit [-x ] [-p "post processing command"] [-t tag1[, tag2, ..
59      writeln('-x  will extract code only into outputfile'),
60      writeln('-p "..." is the postprocessing command. default="pandoc --listing -
61      writeln('-t tag1[, tag2, ..., tagn]  will only process code blocks with any
62      writeln('-o  specify outputfile name. Default is inputfile, but with .md ext
63      !.
```

The core process of "lit"

The lit predicate is the start of the core processing of lines from the input document. Lit is called after the commandline arguments have been processed, so we first have to ensure that all the parameters have appropriate values. For example, it may be that the user did not specify an output file name, which means we have to set Outfile to some default value. The lit predicate handles all the initiation of these key variables.

After that, we iteratively read a line from the input file and hand that line off to the line grammar. The process_lines predicate handles this.

The remainder of the core program are the grammar for each document line, and the evaluation rules to accompany the grammar.

Ensure that all parameters are set

Listing 5: the core program 'lit'

```
1  %! lit(+Infile:atom, ?Outfile:atom, ?CodeOnly:atom) is det.
2  %   process the commandline arguments (like opening file handles) and pass to
3  %   the core program.
4  lit(Infile, Outfile, CodeOnly, Partags,Postproc):-
5      ( ground(CodeOnly) -> true; CodeOnly = false),
6      ( \+ground(Partags) -> Partags=[]; true),
```

```
7       ( \+ground(Postproc) ->
8           (    member(nonum,Partags) ->
9                   Postproc='pandoc -o %w.pdf %w.md';
10                  Postproc='pandoc --listing -o %w.pdf %w.md'
11          ); true
12      ),
13      ( \+ground(Outfile) ->
14          (    file_name_extension(N,E,Infile),
15               ( (E==pro';E==md') ->
16                  throw('Error: input file cannot be .pro or .md to avoid accident
17                  true
18               ),
19               (CodeOnly -> Ext='pro'; Ext='md'),
20               file_name_extension(N,Ext,Outfile)
21          ); true
22      ),
23      open(Outfile,write, Out,[]),
24
25      (ground(Infile) ->
26          (    open(Infile, read, In,[]),
27               current_output(Serr),        % user current output for errors and war
28               with_output_to(Out,process_lines(Serr,In,CodeOnly,Partags,text,1)),
29               close(Out),
30               close(In),
31
32               % post processing
33               ( \+CodeOnly ->
34                   (    file_name_extension(Base,_,Outfile),
35                        swritef(Cmd,Postproc,[Base,Base]),
36                        shell(Cmd,[])
37                   ); true
38               )
39          );
40          print_arghelp
41      ),!.
```

Iteratively process lines

Listing 6: iteratively process lines

```
1  %! process_lines(+In:handle,+CodeOnly:atom,+Partags:list,+Lnum:integer) is det.
2  %   Process: read line and process according to type of line and state of proces
3  process_lines(Serr,In,CodeOnly,Partags,State,Lnum):-
4      (\+ at_end_of_stream(In)),
5      read_line_to_codes(In, Codes),
6      aline(Serr,CodeOnly,Partags,Lnum,Codes,State,Newstate),!,
```

```
7       Lnext is Lnum + 1,
8       process_lines(Serr,In,CodeOnly,Partags,Newstate,Lnext).
9   process_lines(_Serr,_In,_CodeOnly,_Partags,_State,_Lnum).
```

The grammar of document lines

Individual lines in the document are essentially of two types: a codeblock start (or end), or a text line. However, the codeblock start can imply a number of different "processing states". Namely, we may wish to list or evaluate the code (or a combination of the two), depending on the tags given in the codeblock header or the commandline parameters.

So, the grammar also keeps track of the processing state according to the following state transition rules:

- we start in the "text" state
- any state can transition into the "code" state via the codeblock opening line, unless the tags in the codeblock line indicate another state (e.g. a "skip" state)
- any state can transition into the "skip" state via the codeblock opening line, if that line contains the "skip" tag
- any state can transition into the "noeval" state via the codeblock opening line, if that line contains the "noeval" tag
- any state can transition into the "nolist" state via the codeblock opening line, if that line contains the "nolist" tag
- any state can transition into the "text" state via the codeblock closing line (a line with just the two-percent signs)
- any state persists if the input line is not a codeblock line

Note that text lines are also evaluated according to a detailed grammar (see eval_textline</code) that looks for inline code. Also, in the "code" and "nolist" states (i.e. any state where code is supposed to be evaluated), the code lines are stored in the Prolog database with the b/2 predicate. Then, when the code block finishes, the lines are written out to a temporary file, and consulted back into the Prolog environment (i.e. evaluated). This is done at the end of the codeblock, using the call to eval_codeblock.

Listing 7: grammar of a document line

```
1  %! aline(+CodeOnly:atom,+Lnum:integer,+Chars:list,+State:atom,-Newstate:atom) is
2
3  % codeblock start
4  aline(Serr,CodeOnly,Partags,_Lnum,Codes,_,Outstate):-
5       codeblock_start(CodeOnly,Partags,Outstate,Codes,Rest),
6       (    (length(Rest,L), L>0) ->
7                (    write(Serr,'Line␣'),
8                     write(Serr,'.␣Unknown␣syntax␣ignored:␣'),
9                     string_codes(S,Rest),
```

```
10                      write(Serr,S)
11                  ); true
12      ).
13
14  % text line
15  aline(_Serr,false,_Partags,_Lnum,Codes,text,text):-
16      \+codeblock_intro(Codes,_),
17      eval_textline(Cs,Codes,_),
18      writef('%s\n',[Cs]).
19
20  % skipped code line
21  aline(_Serr,_CodeOnly,_Partags,_Lnum,Codes,skip,skip):-
22      \+codeblock_intro(Codes,_).
23
24  % not listed codeline
25  aline(_Serr,_CodeOnly,_Partags,Lnum,Codes,nolist,nolist):-
26      \+codeblock_intro(Codes,_),
27      append_codeline(Lnum,Codes).
28
29  % listed & evaluated codeline
30  aline(_Serr,_CodeOnly,_Partags,Lnum,Codes,code,code):-
31      \+codeblock_intro(Codes,_),
32      writef('%s\n',[Codes]),
33      append_codeline(Lnum,Codes).
34
35  % not evaluated codeline
36  aline(_Serr,_CodeOnly,_Partags,_Lnum,Codes,noeval,noeval):-
37      \+codeblock_intro(Codes,_),
38      writef('%s\n',[Codes]).
39
40  % end of codeblock -- evaluate if needed
41  aline(_Serr,CodeOnly,_Partags,_Lnum,Codes,Instate,text):-
42      codeblock_intro(Codes,[]),
43      ( (\+CodeOnly, Instate\=skip, Instate\=nolist) -> writef('\n~~~\n\n'); true)
44      ( (Instate\=skip, Instate\=noeval) -> eval_codeblock; true).
45
46  aline(_Serr,_CodeOnly,_Partags,_Lnum,_Codes,State,State).
```

The grammar for start and end of codeblocks

The line that opens a codeblock has a simple grammar of double-percent signs, a label and optional caption and tags. That grammar is translated into Pandoc's grammar for codeblocks (which uses, for example, triple tildes instead of double percent).

Listing 8: grammar for start and end of a codeblock

```
% grammar for line that starts a codeblock
% Grammar: %% mylabel ["mycaption"] [ tag1[, tag2[, tag3...]]]
% Reserved tags: nolist noeval skip nonum
codeblock_start(CodeOnly,Partags,Outstate) -->
    codeblock_intro,
    label(Label),
    whitespace,
    caption(Caption),
    whitespace,
    tags(Tags),
    {   codeblock_state(Partags,Label,Tags,Outstate),
        ( (\+CodeOnly, Outstate \= skip, Outstate \= nolist) ->
            (   writef('~~~{ .prolog label=%w caption="%w" numbers=',[Label,Capt
                ( (\+member(nonum,Tags),\+member(nonum,Partags)) -> write('left
                maplist(write_dottag,Tags),
                writeln('}')
            );
            true
        )
    },!.

% codeblock line start or end
codeblock_intro --> [37,37], whitespace.        % 37='%'

% Note codeblock_state are listed in order for efficiency.
codeblock_state(Partags,Label,Tags,skip):-
    (member(skip,Tags);nopartags(Partags,[Label|Tags])),!.
codeblock_state(_,_,Tags,nolist):-
    member(nolist,Tags),!.
codeblock_state(Partags,_,Tags,noeval):-
    (member(noeval,Tags);member(noeval,Partags)),!.
codeblock_state(_,_,_,code):-!.

write_dottag(A):- write('.'), write(A), write(' ').
```

Detailed grammar for labels, captions and tags

A line of text from the document presents itself as a list of character codes. The detailed grammar below chunks those character lists into grammar elements like labels, captions and tags.

Listing 9: grammar details

```
% whitespace is spaces or tabs.  End-of-line is handled separately
```

```
2  whitespace --> [C], {C=9;C=32}, whitespace.      % 9=tab, 32=space
3  whitespace --> {}.
4
5  % label is a sequence of a letter, followed by alphanumerics or underscores
6  label(Label) --> [A], {letter(A)}, label_tail(As), {atom_chars(Label,[A|As])},!.
7  label_tail([A|As]) --> [A], {alpha_num(A)}, label_tail(As).
8  label_tail([]) --> {}.
9
10 lowercase(C):- C > 96, C < 123.      %  'a' <= C <= 'z'
11 uppercase(C):- C > 64, C < 91.       %  'A' <= C <= 'Z'
12 letter(C) :- (lowercase(C); uppercase(C)),!.
13 digit(C):-  C > 47, C < 58.          %  '0' <= C <= '9'
14 alphanum(C):- (digit(C); letter(C)),!.
15 alpha_num(C):- (C=95; alphanum(C)),!.    % '_'=95
16
17 % caption is any text inside double quotes
18 caption(Caption) --> [34], caption_text(Cap), [34], {atom_chars(Caption,Cap)},!.
   % 34='"'
19 caption('-') --> {}.
20 caption_text([A|As]) --> [A], {A\=34}, caption_text(As).
21 caption_text([]) --> {}.
22
23 % tags is a sequence of multiple labels, perhaps separated by commas, semicolons
24 tags([T|Tags]) --> label(T), separator, tags(Tags).
25 tags([]) --> {}.
26
27 % separator
28 separator --> [C], {C=32;C=9;C=59;C=44}, separator.
   % 32=space, 9=tab, 59=semicolon, 44=comma
29 separator --> {},!.
30
31 % nopartags succeeds if partags is not empty, but no partag is in Tags
32 nopartags([P],Tags):- \+member(P,[nolist,noeval,nonum]),\+member(P,Tags).
33 nopartags([P|Ps],Tags):-
34     (member(P,[nolist,noeval,nonum]) -> true; \+member(P,Tags)),
35     nopartags(Ps,Tags).
```

Actions on various parts of the line grammar

There are two main actions that require supporting predicates: evaluating inline code snippets and evaluating multi-line code chunks. The former are evaluated with call(...) and the latter by writing the code to a temporary file and then consulting the file back in.

Listing 10: evaluation an inline clause

```
% evaluate an inline clause
% e.g. "The X value is `foo(X), write(X)`" will transform into "The X value is 4
%      on the assumption that foo(X) evaluates to foo(42).
eval_textline([C|Cs]) --> [C], {C\=96}, !, eval_textline(Cs).
eval_textline(Cs) -->
    [C], {C=96},                    % 96 = '`' (backtick)
    eval_codechunk(CC),
    {   string_codes(S,CC),
        term_string(T,S),
        with_output_to_codes(call(T),Result)
    },
    eval_textline(Ctail),!,
    {append(Result,Ctail,Cs)}.
eval_textline([]) --> {}.

eval_codechunk([C|Cs]) --> [C], {C\=96}, eval_codechunk(Cs).
eval_codechunk([]) -->  [C], {C=96}.
```

Listing 11: storing and evaluating a code block

```
% append line from codeblock to temporary storage
append_codeline(Lnum,Codes):-
    string_codes(S,Codes),
    assertz(b(Lnum,S)).

% evaluate current block of code in temp storage
eval_codeblock:-
    findall(S,b(_,S),Lines),
    gensym(tmp,ID),
    file_name_extension(ID,pro,Tempfile),
    open(Tempfile,write,F,[]),
    writelnf(F,Lines),
    close(F),
    consult(Tempfile),
    delete_file(Tempfile),
    retractbeyond(0).

% write a list of lines to a file with handle F
writelnf(_,[]):-!.
writelnf(F,[L|Ls]):-
    writeln(F,L),!,
    writelnf(F,Ls).
```

Utilities

Listing 12: retracting lines from temporary storage

```
1 %! retractbeyond(+H:integer) is semidet
2 %   remove all b/2 database entries beyond sequence number H.
3 retractbeyond(H):-
4       b(K,_),
5       K > H,
6       retract(b(K,_) ),
7       fail.
8 retractbeyond(_).
```