Running and debugging in Allegro CL 6.0

Myroslava Dzikovska

CS244/444
Fall 2002

1 Starting Lisp and loading files

You can run the lisp interpreter from the command line by typing `lisp`. However, a much more comfortable way to do it is to run lisp from emacs. It gives you key bindings and editing capabilities missing from the command line.
The simplest way to start lisp within emacs is to download the "ecll" file from the class website, available in resources session. Put it in your bin directory, make sure it is executable with `chmod +x /bin/ecll`, then change to the directory where all your files are located and run ecll on the command line. It will start an emacs window with lisp running in it.
To start lisp from emacs already running, do `M-x allegro`. Lisp will ask you several questions in the emacs input buffer (such as host:). They are of no importance for small programs we do - simply hit enter until you see a lisp buffer and a prompt to input lisp commands.
Once you started lisp in emacs, by switching between buffers you can edit the files and debug in the same program, very convenient in itself. Additionally, if you make a typo, you can use go back to that line in the buffer, use the normal editing keys to fix it, press enter - and the edited line will be executed.
Emacs gives you the following key bindings for lisp [1]
C-c C-y OR C-c C-p - previous command in history
C-c C-n - next command in history
C-d - exit debugger mode
If you want to see additional bindings, do `M-x describe-bindings` while running lisp in Emacs.
When running Allegro lisp, the following convenience commands are available

- `:cl filename` - compile and load a file. Will check the source file for changes, recompile as necessary, and load the file. Preferred method for loading files, because compilation will uncover many errors easily. You need not use quotes, and can skip the extension if it is "lisp". The only disadvantage is that compiled files come with many optimizations and sometimes cannot be debugged easily. If you need to load an interpreted file for debugging after a compiled file was loaded, do `:ld filename.lisp`
- `:ld filename` - load file. If no extension is specified, will try ".fasl" - compiled file. If no compiled file, will try ".lisp".
- `:cf filename` - forced recompile of the file. Will need to be loaded with `:ld` or `:cf` after that
- `:pwd` - show the working directory from which files are loaded
- `:cd directory` - change the working directory
- `:exit` - exit lisp

[1] For those unfamiliar with emacs key notation, see the appendix at the end of this document.

2 Debugging techniques

First, a little setup. Create the file .clinit.cl in your home directory and put the following code in it. The file .clinit.cl is loaded every time you start lisp, and you can put additional settings you want in your every session there.

```
(require :composer)
(use-package :composer)
(proclaim '(optimize (safety 3) (space 0) (speed 0) (debug 3)))
(setq tpl::*print-dead-locals* t)
```

The first two lines are needed to use composer, an environment that gives you some window debugging functions. The last two lines set the optimization levels for maximum convenience in debugging. You can download this file from the class website.
If you're running in emacs, click on *Composer-start composer* in the menu. This will start a more user-friendly composer environment for debugging.

2.1 Examining stack and local variables

There are several useful techniques when things go wrong. Below is a function that tries to add its arguments

```
(defun plus-fun (x y)
  (let ((c 5))
    (+ x y c)
    ))
```

It is called by another function, `caller-fun`.

```
(defun caller-fun ()
  (let ((v2 'a) (v2 '1))
    (setq v1 'a)
    (setq v1 (plus-fun v1 v2))
    (plus-fun v2 v1)
    ))
```

The call to `plus-fun` in `caller-fun` is wrong, because the variable `v1` was set to a symbol `a`, and symbols cannot be added. When you call `caller-fun`, here's what happens

```
CL-USER(44): (caller-fun)
Error: `A' is not of the expected type `NUMBER'
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this process.
[1] CL-USER(45):
```

You can ignore the message about restart actions, it is not necessary for simple debugging. The two things to note here are the error message "got A instead of the number" and `[1]` before the prompt - it indicates that lisp entered debugger mode. You need to determine first where the error occured. To do it, use `:bt` to see the execution stack

```
[1] CL-USER(45): :bt
Evaluation stack:

+ <-
  LET <- PLUS-FUN <- SETQ <- LET <- CALLER-FUN <- EVAL <-
  TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP <-
  TPL:START-INTERACTIVE-TOP-LEVEL
```

You can easily see the the sequence of functions called: `caller-fun` called `plus-fun`, `plus-fun` called `+`, and that's where the error happened. Now you can look in your files to find out the exact location based on this call sequence.
Once you're done checking the stack, you want to exit the debugger mode by typing `:pop` or pressing C-d in emacs. You may need to do it several times if multiple errors happened. You can continue running your programs in debugger mode, but generally speaking it is not a good idea, because if new errors occur on top of debug mode for older errors, lisp will eventually become confused and print some really weird error messages.
To do a more detailed examination of the stack, local variables etc, two optionas are available: a composer window debugger and a command-line debugger.

2.1.1 Using the window debugger

When a program breaks on an error (or as a result of `(break)` statement in the code which induces a breakpoint), type `:wdebug` on the command line. A debugger window appears with a backtrace of the program execution. Moving the cursor over a stack frame and pressing $< Shift - rightclick >$ shows a menu with the following options:
Show entire frame representation - shows the entire frame if portions were not printed
Show locals shows the values of local variables at the point when a frame is executed.
Right-clicking on a variable name brings up a menu with *inspect* option, which lets you examine the value in more detail if it is a complex structure.

2.1.2 Using the command-line debugger

To examine the values of local variables at the time of the call do `:loc` in debugger mode:

```
[1] CL-USER(46): :loc
Interpreted lexical environment:
X: A
Y: B
C: 5
Compiled lexical environment:
0(REST): EXCL::ARGS: (A B 5)
1(LOCAL): X: A
2(LOCAL): Y: B
3(LOCAL): C: 5
4(LOCAL): EXCL::|local-0|: (A B 5)
5(LOCAL): EXCL::|local-1|: A
6(LOCAL): EXCL::|local-2|: (B 5)
7(LOCAL): EXCL::|local-3|: (5)
8(UNKNOWN): :UNKNOWN: NIL
9(UNKNOWN): :UNKNOWN: NIL
[1] CL-USER(47):
```

If you need to see local variables of one of the caller functions, use `:dn` to move down the stack until you're one level below the caller, then call `:loc`. In the example below, to see the local variables of `caller-fun` at the time when it called `plus-fun`, I used `:dn` 3 times to get to the point in `caller-fun` where `plus-fun` was called, and then use `:loc` to print local variables.

```
[1] CL-USER(131): :dn
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
   (+ A B ...)
 ->(LET (#) (+ X Y ...))
   (PLUS-FUN A B)
   (SETQ V1 (PLUS-FUN V1 V2))
   (LET (# #) (SETQ V1 'A) ...)
   (CALLER-FUN)
   (EVAL (CALLER-FUN))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
```

```
[1] CL-USER(132): :dn
Evaluation stack:

   (ERROR TYPE-ERROR :DATUM ...)
   (+ A B ...)
   (LET (#) (+ X Y ...))
 ->(PLUS-FUN A B)
   (SETQ V1 (PLUS-FUN V1 V2))
   (LET (# #) (SETQ V1 'A) ...)
   (CALLER-FUN)
   (EVAL (CALLER-FUN))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)

[1] CL-USER(133): :dn
Evaluation stack:
   (ERROR TYPE-ERROR :DATUM ...)
   (+ A B ...)
   (LET (#) (+ X Y ...))
   (PLUS-FUN A B)
 ->(SETQ V1 (PLUS-FUN V1 V2))
   (LET (# #) (SETQ V1 'A) ...)
   (CALLER-FUN)
   (EVAL (CALLER-FUN))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
[1] CL-USER(134): :loc
Interpreted lexical environment:
V1: A
V2: B
```

Unfortunately, local variables will get sometimes optimized out regardless of the setup. But walking down the stack looking for the frame where they were optimized out will still give you an idea which functions were called and what went wrong.

2.2 Breakpoints

You can put a breakpoint at any place in the lisp file by putting a call to `(break)` function at the point where you want the program to stop. When it stops, you see the following message:

```
Break: call to the ‘break’ function.

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this process.
```

Now you can either use the window debugger with `:wdebug` or the command-line debugger as explained above.
If you want to continue execution after a break point, type `:continue 0` on the debug prompt.

2.3 Tracing and stepping

If your function finishes running without errors but gives wrong results, or if it goes into infinite recursion (causing stack overflow errors), a helpful technique is to trace the execution of the function and other functions it calls. Tracing is enabled by calling `(trace function-name1 ... function-namen)` or `:tr function-name1 ... function-namen` . To untrace a function, call `(untrace function-name)` or `:untr function-name`. To untrace all functions currently traced, call `(untrace)` or `:untr` without arguments. Below is an example of tracing a function reverse-recursive we defined in the tutorial

```
CL-USER(24): :trace reverse-recursive
(REVERSE-RECURSIVE)
CL-USER(25): (reverse-recursive '(a b c d e))
 0: (REVERSE-RECURSIVE (A B C D E))
   1: (REVERSE-RECURSIVE (B C D E))
     2: (REVERSE-RECURSIVE (C D E))
       3: (REVERSE-RECURSIVE (D E))
         4: (REVERSE-RECURSIVE (E))
           5: (REVERSE-RECURSIVE NIL)
           5: returned NIL
         4: returned (E)
       3: returned (E D)
     2: returned (E D C)
   1: returned (E D C B)
 0: returned (E D C B A)
(E D C B A)
CL-USER(26): :untr
NIL
```

You can examine the trace to see which functions were called with which parameters to find the source of error.
If you really need to see how your function is executed step by step, you can use stepping mode. To enter the step mode, you have to recompile and load the comiled file first. Never do stepping on the interpreted files! You will see too much garbage to be useful at all. After compiling and loading the file, call `:step (function-call)` to enter step mode. Once in step mode, pressing ENTER advances on every step. If you want to exit the step mode before function evaluation is finished, call `:pop` or *C-d* if running in Emacs.
Below is a stepping example.

```
CL-USER(31): :cf examples
;;; Compiling file examples.lisp
gc: E=90% N=101040 O+=0 pfu=38+232 pfg=1+31
;;; Writing fasl file examples.fasl
Warning: No IN-PACKAGE form seen in /home/butler/u19/myros/classes/cs444/tex/examples.lisp.
         (Allegro Presto will be ineffective when loading a file having no IN-PACKAGE form.)
;;; Fasl write complete
CL-USER(32): :cl examples
; Fast loading /home/butler/u19/myros/classes/cs444/tex/examples.fasl
CL-USER(34): :step (reverse-recursive '(c b a))
 1: (SYSTEM::FUNCTION-INFORMATION REVERSE-RECURSIVE NIL)
[STEP] CL-USER(35):
 result 1: NIL
 1: (FBOUNDP REVERSE-RECURSIVE)
[STEP] CL-USER(35):
 result 1: #<Function REVERSE-RECURSIVE>
 1: (EXCL::SCAN-KEYWORD REVERSE-RECURSIVE)
[STEP] CL-USER(35):
 result 1: NIL
 1: (EXCL::%INVOKES REVERSE-RECURSIVE ('(C B A)))
[STEP] CL-USER(35):
  2: (EXCL::%EVAL '(C B A))
```

```
[STEP] CL-USER(35):
   3: (SYSTEM::FUNCTION-INFORMATION QUOTE NIL)
[STEP] CL-USER(35):
   result 3: NIL
   3: (FBOUNDP QUOTE)
[STEP] CL-USER(35):
   result 3: #<special operator QUOTE @ #x710107c2>
   3: (EXCL::SCAN-KEYWORD QUOTE)
[STEP] CL-USER(35):
   result 3: NIL
   3: (EXCL::SPEC-CLOSURE-OBJECT-P #<special operator QUOTE @ #x710107c2>)
[STEP] CL-USER(35):
   result 3: T
   3: '((C B A))
[STEP] CL-USER(35):
   result 3: (C B A)
  result 2: (C B A)
  2: (REVERSE-RECURSIVE (C B A))
[STEP] CL-USER(35):
   3: (EXCL::APPEND_2OP NIL (A))
[STEP] CL-USER(35):
   result 3: (A)
   3: (EXCL::APPEND_2OP (A) (B))
[STEP] CL-USER(35):
    4: (EXCL::COPY-LIST-PARTIAL (A))
[STEP] CL-USER(35):
    result 4: (A)  (A)  NIL
   result 3: (A B)
   3: (EXCL::APPEND_2OP (A B) (C))
[STEP] CL-USER(35):
    4: (EXCL::COPY-LIST-PARTIAL (A B))
[STEP] CL-USER(35):
    result 4: (A B)  (B)  NIL
   result 3: (A B C)
  result 2: (A B C)
 result 1: (A B C)
CL-USER(35):
```

If you look through the printout carefully, you will see that on a number of steps, especially in the beginning, LISP exposes some internal function calls it makes, and it also shows the fact that recursion was optimized into iteration. Even though the sequence of calls is not completely identical to the original program, the step mode is very useful if you have branching statements or iteration in your function and need to figure out exactly what happened. While in step mode, you can also use `:loc` to find out the values of any local variables.

3 Appendix 1: Emacs key bindings

If you don't know how to use emacs, learn to use it from the emacs tutorial by clicking on Help-Emacs tutorial in the menu.

Emacs commands generally involve the CONTROL key or the META key (labeled ALT on PC's). The following abbreviations are used.

C-$< chr >$ means hold the CONTROL key while typing the character $< chr >$ Thus, C-f would be: hold the CONTROL key and type f.

M-$< chr >$ means hold the META or ALT key down while typing $< chr >$.

C-x C-x means "Type x twice while holding control key, or Type CTRL-x CTRL-x in succession"

C-x s means "Type x while holding CONTROL key, then type s without CONTROL key"

M-x is a special prefix command. After typing M-x, the cursor switches to minibuffer where it expects you to type the command and press ENTER.

Allegro keys

M-x allegro	start lisp in emacs
C-c C-y	previous command in history
C-c C-p	previous command in history
C-c C-n	next command in history
C-d	exit debugger mode
C-x ;	Comment region in lisp file if editing in the same emacs that allegro is running

General emacs keys

C-x C-s	Save current file
C-x s	Save all files
C-x C-c	exit emacs
C-x 2	split the current frame into 2 subwindows horizontally
C-x 1	split the current frame into 2 subwindows vertically
C-x 1	return to a single subwindow
C-x 5 2	Show a new frame for the same emacs, 2 different independent windows in separate frames

4 Appendix 2: summary of debugging commands

Top level commands

:ld *filename*	load a file
:cl *filename*	compile and load a file
:cf *filename*	recompile a file
:*n*	execute the command which was executed at prompt *n* in this session
:tr *function-name*	trace a function
:tr	list currently traced functions
:untr *function-name*	untrace a function
:untr	untrace all traced functions
:exit	exit lisp

Debugger commands

:bt	show execution stack
:wdebug	show a graphical debugger window (must do *Composer-Start Composer* first)
:dn	move down the stack
:up	move up the stack
:loc	show local variables
(break)	put in the code to place a breakpoint at that position
:contiue 0	continue execution after a breakpoint
:step *(function-call)*	step over the function call
ENTER in step-mode	advance a step when in debugger step mode
:pop	exit debugger mode