Demonstration: A stepper for Armed Bear Common Lisp (ABCL)

Alejandro Zamora Fonseca

ale2014.zamora@gmail.com

ABSTRACT

In this paper a stepper tool for the Armed Bear Common Lisp (ABCL) implementation is proposed, describing its features and implementation related details. ABCL does not currently have a stepper and the addition of one can help improve the quality of the code designed to run in the implementation, and also it can be useful to assist in the debugging process of Common Lisp (CL) portable code.

CCS CONCEPTS

• **Software and its engineering** → *Software notations and tools*;

KEYWORDS

Common Lisp, stepper, debugging

ACM Reference Format:

Alejandro Zamora Fonseca. 2023. Demonstration: A stepper for Armed Bear Common Lisp (ABCL). In *Proceedings of the 16th European Lisp Symposium (ELS23).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.7815887

1 INTRODUCTION

A stepper is a tool that allows to control and follow step by step the execution of a subprogram. Many programming languages provide stepping tools as part of its debugging mechanisms. CL is not an exception and includes the `step` macro in its standard to be optionally implemented.

ABCL does not include a stepper and this fact has been mentioned as one of the reasons that detract the implementation from being a proper contemporary CL implementation, see [1].

This paper introduces a stepper to address this issue in ABCL. I will describe its features, examples of use and some implementation details.

ELS23, Apr 24–25 2023, Amsterdam, Netherlands

https://doi.org/10.5281/zenodo.7815887

The code for this tool is shared as open source in the form of a pull request on ABCL's own code in its Github repository (see [2]).

2 RELATED WORK

Recently, João Távora [3] made a great summary of the state of art of stepping in CL and presented a visual and portable stepper module (sly-stepper) for his IDE Sly.

Unfortunately sly-stepper does not support ABCL and the other efforts mentioned in that paper appear to be incomplete or difficult to integrate into any CL implementation.

3 DESIGN AND IMPLEMENTATION

In my opinion, a stepper is needed for ABCL to improve the quality of the implementation, to bring more debugging tools for users of ABCL or other CL systems, and to lay the groundwork for integrations with external IDEs like Sly, Slime and others. Having a simple stepper in text mode shipped with the implementation, would also allow developers to not depend on any external tools for this task, regardless of the environment in which the implementation will run.

In this section I'll describe details in the design and implementation of the stepper and its features.

This step tool is integrated in the evaluation code in ABCL's main evaluator which is, at its core, an interpreter. The evaluator works by traversing the successive subforms in interpreted code after the macroexpansion, but it cannot go inside compiled functions, which are executed from its Java bytecode. For this reason the stepper will not enter either into compiled code.

Internally, when the user runs the `step` macro, the interpreter first sets itself to stepping mode and will allow the user to proceed to step through each sub-form according to her needs. The stepping related code is implemented to be called from selected stages of the evaluator. And finally after the form is executed, the stepping mode is disabled.

The following diagram illustrates an overview of the stepper arquitecture. The stepper hooks created in the middle of the evaluator were used to call the component that implements the logic of the options in the stepper (`Handle Stepping`), based on the result of the component `Step in symbol ?`.

Both components manipulate the internal states of the stepper in the evaluator (Lisp.java), which are mainly two flags to control the `step` and `next` features.

Most of the code for this tool was done in Lisp (abcl-stepper.lisp), taking advantage of the nice API that ABCL provides for developers to interact between Java and Lisp. Namely, the ability to create `Primitive` methods in Java that can be easily called from Lisp as functions was essential. On the other hand, the constructions to call Java objects and methods from Lisp were also useful.

See Figure 1.

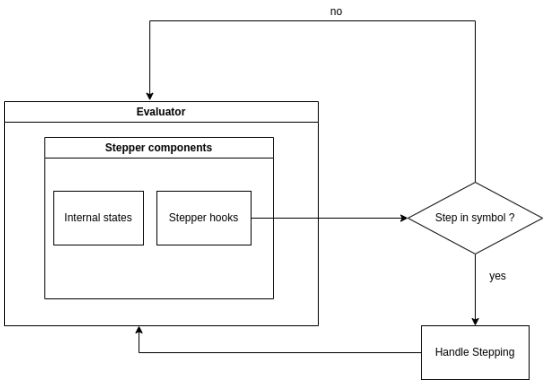

Figure 1: The arquitecture of the stepper

This approach was used due to its simplicity, as opposed to others that need to instrument the subforms of the code to step, in order to perform the stepping. Taking advantage of the built-in evaluator makes it possible to step into any interpreted code without needing to instrumenting it.

On every stage of the stepping process, the user will be prompted with a screen like the one in Listing 1.

```
We are in the stepper mode
Evaluating step 1 -->
(TEST)
Type ':?' for a list of options
```

Listing 1: The head of the prompt.

If the user presses `:?`, the system will show a simplified help with a list of the features present in the stepper, see Listing 2

```
Type ':l' to see the values of bindings on the local environment
Type ':c' to resume the evaluation until the end without the stepper
Type ':n' to resume the evaluation until the next form previously selected
↪  to step in
Type ':s' to step into the form
Type ':i' to inspect the current value of a variable or symbol
Type ':b' to add a symbol as a breakpoint to use with next (n)
Type ':r' to remove a symbol used as a breakpoint with next (n)
Type ':d' to remove all breakpoints used with next (n)
Type ':w' to print the value of a binding in all the steps (watch)
Type ':u' to remove a watched binding (unwatch)
Type ':bt' to show the backtrace
Type ':q' to quit the evaluation and return NIL
Type ':?' for a list of options
```

Listing 2: Minimal help option.

Now the rest of the options will be described in the following subsections.

3.1 Locals bindings

The `:l` option will display the local bindings for variables and functions in the current environment passed to the current form to evaluate, a typical response would look like as described in Listing 3:

```
Showing the values of variable bindings.
From inner to outer scopes:
N=2
Showing the values of function bindings.
From inner to outer scopes:
FLET1=#<FUNCTION #<(FLET FLET1) {3ACE0BC7}> {3ACE0BC7}>
```

Listing 3: Local bindings option.

3.2 Continue to the end

The continue `:c` option will, basically, ignore the stepping process and perform the evaluation of the form without any stop.

3.3 Stop at next marked symbol

The next `:n` feature allows to stop the stepper only when the interpreter is analyzing one of the symbols specified in the list of `stepper::*stepper-stop-symbols*` or any of the exported symbols presented in any of the list of packages specified in `stepper::*stepper-stop-packages*`. These variables will have initially the value `NIL` and, if left unchanged, `next` will behave almost exactly as `continue`. It is useful when we want to step over large or complex code and avoid stepping every form in order to jump only to the interested ones. This feature will be explained more in detail in next sections.

In the middle of the stepping process it is possible to change the value of the variable `stepper::*stepper-stop-symbols*`, using the options `:b`, `:r` and `:d`. The `:b` option allows to add a symbol to `stepper::*stepper-stop-symbols*`, option `:r` will remove a symbol in `stepper::*stepper-stop-symbols*` and the option `:d` will remove all the symbols in the aforementioned variable.

3.4 Step into the form

The step `:s` functionality is the most basic operation in the stepper, it will step into the current form until the evaluation ends. It can even step into ABCL internal functions.

3.5 Inspect variables

This feature (`:i`) allows one to inspect the content of a variable or binding, present in the current environment. It will first ask the user to type the symbol to inspect and proceed to print its value.

Some screens as quick examples (Listing 4)

```
Type ':?' for a list of options
:i
Type the name of the symbol: *some-var*
NIL
Type ':?' for a list of options
:i
Type the name of the symbol: x
3
```

Listing 4: Inspect variable option.

3.6 Show backtrace

The `:bt` option provides the ability to print the current backtrace, which is useful for analyzing the evaluation path until the current stepping point.

3.7 Watch and (un)watch

The feature `watch` allows to follow the values of a variable in all the steps, the user can add a variable to watch by typing `:w` and when prompted, the symbol to watch. After that, the user can remove the variable from being watched by using the `:u` option and entering the same symbol.

3.8 Quit evaluation

The quit `:q` feature will abort the evaluation in the stepper and return NIL. This is useful to avoid running the remaining forms in the code when the user wants to leave the stepper, especially if the rest of the program is performing expensive operations.

3.9 Examples of usage

This subsection explains in details, by using some examples, the features of the current stepper. The examples shown here will be using the pure ABCL's REPL but will behave the same if you use the shell buffer in Emacs for a slightly more comfortable development environment.

First let's examine the `inspect` feature combined with the `step` feature.

In this example we can see how the inspect feature is used and it retrieves correctly the values for the lexical variable `x` and the special variable `*some-var*` which is rebinded in the code. The values are shown using `cl:print`. Listing 5

As a second example the use of the `list locals` feature will be illustrated. Here, we can observe that the list of local bindings include variable and function bindings and they are showed from inner to outer scopes, for that reason the value of variable `a` is first `2` and later `1`. See Listing 6

In the previous example the use of the feature `continue` was shown too, allowing to complete the evaluation of the form turning off the stepper.

The following stepper session will be used to explain the `next` and `quit` features. They allow to stop the execution only in designed symbols. It behaves like if we were adding

```
CL-USER(1): (require :asdf)
NIL
CL-USER(2): (require :abcl-contrib)
NIL
CL-USER(3): (require :abcl-stepper)
NIL
CL-USER(4): (defparameter *some-var* 1)
*SOME-VAR*
CL-USER(5): (defun test ()
  (let ((*some-var* nil)
        (x 3))
    (list *some-var* 3)))
TEST
CL-USER(6): (stepper:step (test))
We are in the stepper mode
Evaluating step 1 -->
(TEST)
Type ':?' for a list of options
:i
Type the name of the symbol: *some-var*
1
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 2 -->
(BLOCK TEST
  (LET ((*SOME-VAR* NIL) (X 3))
    (LIST *SOME-VAR* 3)))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 3 -->
(LET ((*SOME-VAR* NIL) (X 3))
  (LIST *SOME-VAR* 3))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 4 -->
(LIST *SOME-VAR* 3)
Type ':?' for a list of options
:i
Type the name of the symbol: x
3
Type ':?' for a list of options
:i
Type the name of the symbol: *some-var*
NIL
Type ':?' for a list of options
:c
step 4 ==> value: (NIL 3)
step 3 ==> value: (NIL 3)
step 2 ==> value: (NIL 3)
step 1 ==> value: (NIL 3)
(NIL 3)
```

Listing 5: Step and inspect features.

breakpoints for the stepping process. Let's look at the stops in this example. The stepper is stopping in the call with the symbol `'step-next::loop-1` because it was added to `stepper::*stepper-stop-symbols*`. It is also stopping in the call with the symbol `'step-next::loop-3` because that symbol was exported in the package `next-step` and the symbol was added to `stepper::*stepper-stop-packages*`. `'step-next::loop-2` is skipped when using `next` because it is not present in any of the lists of symbols mentioned before.

We can observe as well the use of the feature `quit`. After the use of it, the evaluation was stopped before the initialization of the special variable `step-next::*test-next-var*` and therefore it is not bound yet after complete the stepping process. See Listing 7

If we observe the second stepper call in the `test-next` function, we can see the use of the `:b` and `:r` features. Using the `:b` option adds a breakpoint to the symbol `step-next::loop-2`, the breakpoint to `step-next::loop-1` is removed using the

```
CL-USER(7): (stepper:step (flet ((flet1 (n) (+ n n)))
                    (let ((a 1))
                      (let ((a 2))
                        (+ (flet1 3) a)))))
We are in the stepper mode
Evaluating step 1 -->
(FLET ((FLET1 (N) (+ N N)))
  (LET ((A 1))
    (LET ((A 2))
      (+ (FLET1 3) A))))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 2 -->
(LET ((A 1))
  (LET ((A 2))
    (+ (FLET1 3) A)))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 3 -->
(LET ((A 2))
  (+ (FLET1 3) A))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 4 -->
(+ (FLET1 3) A)
Type ':?' for a list of options
:l
Showing the values of variable bindings.
From inner to outer scopes:
A=2
A=1
Showing the values of function bindings.
From inner to outer scopes:
FLET1=#<FUNCTION #<(FLET FLET1) {238E109B}> {238E109B}>
Type ':?' for a list of options
:c
step 4 ==> value: 8
step 3 ==> value: 8
step 2 ==> value: 8
step 1 ==> value: 8
8
CL-USER(8):
```

Listing 6: Local bindings feature.

option `:r` and, after executing the command `:n`, the stepper stops this time at `step-next::loop-2` instead of `step-next::loop-1`. See Listing 8.

The following example exhibits the use of the `backtrace` feature. This option allows to visualize the full evaluation path to the point of the program being analyzed by the stepper. See Listing 9.

The last example presents the `watch`(`:w`) feature which permits to monitor the values of a variable in the stepping process. In the code sample it can be seen the succesive values of the variable `x` and how they are removed after the `unwatch`(`:u`) option is applied. See Listing 10.

4 CONCLUSION

A first functional stepper for ABCL has been introduced. It can help users to dig into the guts of every complex code to help find the root cause of errors.

I think that even knowing that this is the first version, it can be usable and offer an alternative to debug complex systems built using ABCL or even portable CL code.

```
CL-USER(8): (defpackage step-next (:use :cl))
#<PACKAGE STEP-NEXT>
CL-USER(9): (in-package :step-next)
#<PACKAGE STEP-NEXT>
STEP-NEXT(10): (defun loop-1 (a b)
  (loop :for i :below a
        :collect (list a b)))
LOOP-1
STEP-NEXT(11): (defun loop-2 (a)
  (loop :for i :below a
        :collect i))
LOOP-2
STEP-NEXT(12): (defun loop-3 (n &optional (times 1))
  (loop :for i :below times
        :collect times))
LOOP-3
STEP-NEXT(13): (defun test-next (n)
  (loop-1 (1+ n) n)
  (loop-2 (1- n))
  (loop-3 n 3)
  ;; quit (q) here
  (defparameter *test-next-var* (loop :for i :below (expt 10 6)
                                      :collect i)))
TEST-NEXT
STEP-NEXT(14): (push 'loop-1 stepper::*stepper-stop-symbols*)
(LOOP-1)
STEP-NEXT(15): (export 'loop-3)
T
STEP-NEXT(16): (push 'step-next stepper::*stepper-stop-packages*)
(STEP-NEXT)
STEP-NEXT(17): (stepper:step (test-next 7))
We are in the stepper mode
Evaluating step 1 -->
(TEST-NEXT 7)
Type ':?' for a list of options
:n
We are in the stepper mode
Evaluating step 2 -->
(LOOP-1 (1+ N) N)
Type ':?' for a list of options
:n
step 2 ==> value: ((8 7) (8 7) (8 7) (8 7) (8 7) (8 7) (8 7) (8 7))
We are in the stepper mode
Evaluating step 3 -->
(LOOP-3 N 3)
Type ':?' for a list of options
:q
NIL
STEP-NEXT(18): (assert (not (boundp '*test-next-var*)))
NIL
STEP-NEXT(19):
```

Listing 7: Next and quit features.

```
STEP-NEXT(19): (stepper:step (test-next 7))
We are in the stepper mode
Evaluating step 1 -->
(TEST-NEXT 7)
Type ':?' for a list of options
:b
Type the name of the symbol to use as a breakpoint with next (n): loop-2
Type ':?' for a list of options
:r
Type the name of the breakpoint symbol to remove: loop-1
Type ':?' for a list of options
:n
We are in the stepper mode
Evaluating step 2 -->
(LOOP-2 (1- N))
Type ':?' for a list of options
:n
step 2 ==> value: (0 1 2 3 4 5)
We are in the stepper mode
Evaluating step 3 -->
(LOOP-3 N 3)
Type ':?' for a list of options
:q
NIL
STEP-NEXT(20):
```

Listing 8: Next and quit features.

5 FURTHER WORK

The current stepper is implemented in a way that blocks any remaining threads in the system until the stepping process

```
STEP-NEXT(20): (defun test-backtrace (x)
  (labels ((f1 (x) (f2 (1+ x)))
           (f2 (x) (f3 (* x 3)))
           (f3 (x) (+ x 10)))
    (f1 x)))
TEST-BACKTRACE
STEP-NEXT(21): (stepper:step (test-backtrace 3))
We are in the stepper mode
Evaluating step 1 -->
(TEST-BACKTRACE 3)
Type ':?' for a list of options
:b
Type the name of the symbol to use as a breakpoint with next (n): +
Type ':?' for a list of options
:n
We are in the stepper mode
Evaluating step 2 -->
(+ X 10)
Type ':?' for a list of options
:bt

(#<LISP-STACK-FRAME ((LABELS F3) 12) {758EDEFD}>
 #<LISP-STACK-FRAME ((LABELS F2) 4) {6CD6F8DD}>
 #<LISP-STACK-FRAME ((LABELS F1) 3) {3B96D1EB}>
 #<LISP-STACK-FRAME (TEST-BACKTRACE 3) {6D76BF34}>
 #<LISP-STACK-FRAME (SYSTEM::%EVAL (ABCL-STEPPER:STEP
   (TEST-BACKTRACE 3))) {64C522B}>
 #<LISP-STACK-FRAME (EVAL (ABCL-STEPPER:STEP
   (TEST-BACKTRACE 3))) {387EC7BA}>
 #<LISP-STACK-FRAME (SYSTEM:INTERACTIVE-EVAL
   (ABCL-STEPPER:STEP (TEST-BACKTRACE 3))) {356A40D7}>
 #<LISP-STACK-FRAME (TOP-LEVEL::REPL) {6DBDC651}>
 #<LISP-STACK-FRAME (TOP-LEVEL::TOP-LEVEL-LOOP) {9840CC7}>)
Type ':?' for a list of options
:c
step 2 ==> value: 22
step 1 ==> value: 22
22
STEP-NEXT(22):
```

Listing 9: Show backtrace feature

is finished. This was done by simplicity in the design and to avoid unpleasant race conditions on the internal states. Changing it to a non-blocking version, would be more flexible for users, especially when debugging systems in production.

Include other well known step features in other implementations like step-out and step-next (move to the next form avoiding step-into)

Implement the evaluation of custom expressions in the current environment.

Find a way to integrate it with Sly/Slime. Currently, when called inside Sly/Slime REPL it will only show print a message and return the form without any stepping. Also find a way to abstract the integration with any IDE.

6 ACKNOWLEDGEMENTS

I would like to thank my wife Valeria, who helped me in the general design of the features and lovingly motivated me to complete the implementation and this paper.

Also to the Common Lisp community for providing me all the software, documentation and support to every doubt I had in my learning all these years.

```
STEP-NEXT(22): (defun test-watch ()
  (let ((x 1))
    (setq x 3)
    (setq x 7)
    (setq x 21)
    x))
TEST-WATCH
STEP-NEXT(23): (stepper:step (test-watch))
We are in the stepper mode
Evaluating step 1 -->
(TEST-WATCH)
Type ':?' for a list of options
:w
Type the name of the symbol to watch: x
Type ':?' for a list of options
Watched bindings:
Couldn't find a value for symbol X
:s
We are in the stepper mode
Evaluating step 2 -->
(BLOCK TEST-WATCH
  (LET ((X 1))
    (SETQ X 3)
    (SETQ X 7)
    (SETQ X 21)
    X))
Type ':?' for a list of options
Watched bindings:
Couldn't find a value for symbol X
:s
We are in the stepper mode
Evaluating step 3 -->
(LET ((X 1))
  (SETQ X 3)
  (SETQ X 7)
  (SETQ X 21)
  X)
Type ':?' for a list of options
Watched bindings:
Couldn't find a value for symbol X
:s
We are in the stepper mode
Evaluating step 4 -->
(SETQ X 3)
Type ':?' for a list of options
Watched bindings:
X=1
:s
step 4 ==> value: 3
We are in the stepper mode
Evaluating step 5 -->
(SETQ X 7)
Type ':?' for a list of options
Watched bindings:
X=3
:u
Type the name of the symbol to (un)watch : x
Type ':?' for a list of options
:s
step 5 ==> value: 7
We are in the stepper mode
Evaluating step 6 -->
(SETQ X 21)
Type ':?' for a list of options
:s
step 6 ==> value: 21
step 3 ==> value: 21
step 2 ==> value: 21
step 1 ==> value: 21
21
STEP-NEXT(24):
```

Listing 10: Watch feature

REFERENCES

[1] Abcl manual. URL https://abcl.org/releases/1.9.0/abcl-1.9.0.pdf.
[2] Pr with the stepper code. URL https://github.com/armedbear/abcl/pull/568.
[3] João Távora. A portable, annotation-based, visual stepper for common lisp. 2020. URL https://zenodo.org/record/3742759.