A portable, annotation-based, visual stepper for Common Lisp

João Távora
Ravenpack
joaotavora@gmail.com

ABSTRACT

Many programming systems feature a stepping debugger, a tool that lets users execute code, section by section, in steps of their own choosing. Despite many attempts throughout the decades, the Common Lisp language is still lacking in this regard. We propose and describe the workings of a new, portable, visual stepping facility for Common Lisp, realized as an extension to SLY, a cross-implementation Common Lisp IDE for the Emacs editor. This facility is realized as an increment to an existing source code annotation system known as "stickers", whose working principles we also describe in this work. As part of the solution arrived at for the main objective, we also present two reusable software components: (1) a simple, near portable technique for constructing a source-tracking Common Lisp expression reader in terms of a preexisting compliant expression reader and (2) a technique to carry over source-tracking information to the expansion of macro expressions.

CCS CONCEPTS

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

KEYWORDS

Debugging, Stepping, Common Lisp

ACM Reference Format:
João Távora. 2020. A portable, annotation-based, visual stepper for Common Lisp. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3742759

1 INTRODUCTION

1.1 What is stepping

A program stepper lets users execute code, section by section, in steps of their own choosing. A number of hidden control points are inserted along the execution paths of the program. At each point, the system may interrupt the program and wait for user instructions before and/or after executing the next section. Steppers often fall within the category of *program execution monitors*, along with profilers, tracers and other debugging tools. Stepping is the one of the most popular forms of debugging since it allows users to study the evolution of the state of a program by direct inspection, as opposed to combining conjecture and experimentation. A stepper gives users full control over the speed of the actual program and often even allows manipulations of that state. This frequently presents a debugging advantage over guessing a programs' state or monitoring its outputs.

Many programming systems, indeed all of the most popular programming systems provide stepper tools as a part of a debugging tool-chain, i.e. a set of programs often developed and distributed alongside language compilers and interpreters. In languages such as C, the prevalence of a stepping system (such as the popular `gdb` program) is so strong that it becomes synonymous with the term "debugger". This sometimes leads to users migrating to the Lisp family of languages being surprised to find debugging systems that aren't about stepping at all.

Stepper systems usually function through text-based interfaces that are capable of displaying the source code of the relevant code sections. Nevertheless, many users of stepper tools prefer to use them through the more sophisticated user interfaces of editor programs and integrated development environments (IDEs). We shall call these tools *visual* steppers[1]: special-purpose programs running inside the IDE communicate with the stepper tool by means of a protocol and visually annotate the source text of a program with the results of the stepping session. The expression about to be executed may e.g. be marked with a red dot or highlighted in a special color. Furthermore, a modern user's expectations of a stepper system might include the ability to add break points; to "step over" an expression; to "step into" a call to a function defined elsewhere in the source code; to "step out of" the current call; to "continue" to a certain point, and to inspect the values of local and global variables by mouse-clicking on their manifestations in the source code.

1.2 Common Lisp stepping

Common Lisp programmers are so detached from the practice of stepping that some will simply declare that's proof that they don't need one at all. There is a hint of truth to this declaration, as Common Lisp systems have traditionally directed efforts to other types of debugging facilities such as powerful interactive program restarts and function traces. At any rate, it seems undeniable that, stepper or no stepper, Common Lisp programs will be debugged.

Nevertheless, an abundance of Common Lisp steppers have been proposed throughout the decades, even if few have actually enjoyed any adherence. Anecdotal evidence suggests potential newcomers shy away from Common Lisp because of missing stepper functionality. Indeed, this reality appears not to be lost even on the authors of the Common Lisp specification, which have included in their work a provision for a special implementation-defined `CL:STEP` macro.

Some Lisp implementations can and do implement stepping to various levels of ability. SBCL combines its implementation of `CL:STEP` with the restarts system to provide a text-based stepper,

[1] After the nomenclature used in [9]

ELS'20, April 27–28 2020, Zürich, Switzerland

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3742759

while the LispWorks[2] and Allegro CL systems have sophisticated graphical stepping dialogues with views to the source code and program state. Unfortunately, these systems are unavailable to whomever wishes to try portable programs across different Common Lisp implementations. Even if one switches between two implementations that do have steppers, the difference in capability and user interface is often enough to discourage the use of either system.

The problem of interface inconsistency across implementations is not exclusive to the stepper feature: other debugging tools such as the inspector, the debugger or the REPL suffer from it. Thus, many Common Lisp programmers will use a generic text editor such as GNU Emacs[1] together with the SLIME[5] extension. This combination forms a capable, implementation-agnostic Common Lisp IDE that suppresses the problems described above and provides a consistent user interface to many debugging features.

Regrettably, even though Emacs provides visual stepper interfaces for many programming languages, SLIME doesn't provide a portable stepper interface. We believe this to be due to the fact that the technical challenges to be surmounted are greater than for other debugging tools. Among other problems, the portability mandate of SLIME implies it is ultimately only allowed to invoke functionality present in all the Common Lisp systems it connects to. This mandate implies that even if all implementations where to implement some form of `CL:STEP`, that alone wouldn't be powerful enough to, say, communicate source location information to and from Emacs, annotating the program source.

1.3 A portable, visual stepper for Common Lisp

We should note that none of the obstacles listed above are conceptual in nature, so there must still be hope for a portable, visual stepper for Common Lisp. The SLIME/Emacs combination makes it a particularly attractive target for such a tool, given its relevance among Common Lisp users and the flexibility of Emacs's Elisp language.

In fact, the SLY Common Lisp IDE[11], a derivative program of SLIME, has redesigned some of the underpinnings of its predecessor to make the development of new extensions easier. We shall describe how "stickers", a feature that SLY has recently acquired, lets users manually instrument selected Common Lisp forms whose results they are interested in. Stickers are already a "poor man's stepper", in the sense that they have some fundamental semantics of stepping but still encumber the user with work than could be performed automatically. To fill this gap, we shall explore methods of combining stickers with automatic code analysis. We shall then be able to present an innovative stepper tool for the Common Lisp language based on the SLY extension to the Emacs editor, hereafter designated the *SLY/Emacs stepper*.

2 RELATED WORK

"Stepping is an old idea." So go the opening words of this section's namesake in the article "Annotation-Based Program Stepping", written by MIT's G. Parker in 1987[9]. In this article, the author surveys the efforts of the 1970 decade to develop various kinds of stepping tools in the MACLISP environment. Likewise, we shall proceed to evaluate a small sample of Common Lisp stepper systems, focusing on the ones that are portable, visual and intersect our methodology.

In the remainder of Parker's paper[9], a visual stepper system, VisiStep, is described. Its distinguishing characteristics are the integration with the MACLISP system and an *annotation-based* approach, a key difference to other *evaluator-based* techniques.

Annotation-based program stepping is a form of code instrumentation. It comprises the addition of statements to the program shortly before its compilation. By way of a so-called *wrapper* macro, these statements are added before and after each section to be stepped. The addition is transient and invisible: it does not modify the source file. Furthermore, the program cannot itself discern the presence or absence of these additional statements, so its outputs are unmodified. Parker[9] points out that this approach can work with an unmodified evaluator, since the compiler is simply given more instructions to compile. He also asserts this approach to be more efficient, more portable and more selective, the latter meaning that it allows the user to select only those sections of the code that he wishes to step through. However, the author acknowledges the annotation-based stepper's difficulty in handling some macro's non-evaluated syntactic elements (such as the arguments to `COND`), and how it must rely on "code-walking knowledge"[9, I-4.8] to determine the forms where the wrapping may take place.

By contrast, an evaluator-based or *interpretative* approach involves writing a Lisp evaluator or instrumenting the Lisp interpreter. The evaluator itself then becomes responsible for issuing the stepper-enabling statements before and after each evaluation. "UniCStep - a Visual Stepper for COMMON LISP", written by I. Haulsen and A. Sodan in 1989[6], presents such a stepper system, written for an early version of the GNU Emacs editor. The authors reply directly to Parker's contention's of the superiority of the annotation-based approach, asserting the evaluator-based approach to be more comfortable and flexible because the user does not have to specify in advance what to step and where to stop. They assent to one technical disadvantage such as the fact that evaluator-based alternative need a loader to be emulated and more sophisticated ways of remembering the source of the loaded code.

We should note a more recent attempt at a Common Lisp stepper, such as Pascal Bourguignon's work[4]. This consists of a portable, evaluator-based approach that replicates the implementation-defined behavior of Common Lisp's `STEP` macro. Bourguignon's stepper provides a replacement package for the standard `COMMON-LISP` package, through which the user must re-load the code whose forms can then be passed to the `STEP` macro. This stepper has no editor or source-tracking integration as of yet, but it seems to have been in the plans at some point during its development.

Finally, a word should be spared for Emacs's `edebug.el` authored ca. 1988 by Daniel LaLiberte[7]. Edebug is designed to step through Emacs Lisp programs within Emacs itself. Since it executes inside the Lisp machine that is also the editor, the source-tracking integration is very good. `edebug.el` is an annotation-based stepper that deals with the problem of a macro's un-evaluated syntactic elements by skipping macros it knows nothing about. The macros whose expansions the user is interested in can be annotated separately with `edebug.el`-specific declarations.

3 METHODS

Our proposed portable stepper system for SLY/Emacs can be broken down into three main components:

(1) A non-intrusive source code annotation system, called "stickers". This system primarily allows "interesting" Common Lisp forms to be designated by the user. On compilation, the annotated code is transmitted to the Common Lisp compiler, and executes equivalently to non-annotated code;
(2) A *source-tracking reader*, i.e. a process by which a stream of characters containing source code forms is read into a symbolic expression representing the form, whilst recording the positions of the start and end characters of each sub-form;
(3) A *specialized code walker*, a process by which an arbitrary Common Lisp form can be traversed at compilation-time to determine the syntactic value of each of its sub-forms as processed by the compiler after the macro-expanding phase.

It should become apparent that the application of 3. to the results of 2. relieve users in 1. of the need to manually designate forms of interest. They sole job becomes requesting the annotated compilation of arbitrary lengths of source code, leaving the stepper system to automatically annotate all possible forms of interest.

The following subsections detail the workings of each component in this arrangement.

3.1 Stickers

Stickers are a form of code annotation in use in SLY/Emacs. Initially conceived as an alternative to the `PRINT` or `FORMAT` statements introduced by users when debugging programs, this system lets users visually mark individual symbolic or compound forms in whose future execution they're interested in. Crucially, the visual markings exist only in Emacs's memory for as long as the user wishes. They aren't saved in the source code itself. When the compilation of the containing top-level form happens from within SLY/Emacs, SLY will collect those visual markings, enumerate them, and emit for compilation a modified version of the form. This process is called *arming the stickers*.

The modified version is functionally equivalent to the original in the sense that i.e. user programs have no way to detect which one they are executing. The modifications consist of multiple invocations of a special `RECORD` wrapper macro[2], whose definition is presented below in much simplified fashion:

```
(defmacro record (id &body body)
  `(let ((%retval :exited-non-locally)
         (%condition)
         (%sticker (find-sticker ,id)))
     (handler-bind ((condition (lambda (c)
                                 (setq %condition c))))
       (before-sticker %sticker)
       (unwind-protect
            (values-list
             (setq %retval (multiple-value-list
                            (progn ,@body))))
         (after-sticker %sticker %retval %condition)))))
```

E.g., if the user marks the forms `(foo (bar))` and `(bar)` inside the following expression:

```
(let ((baz 42)) (+ (foo (bar)) baz))
```

The sticker system will collect the two markings, label them with the numbers 1 and 2 and emit the following equivalent form for compilation:

```
(let ((baz 42)) (+ (record 1 (foo (record 2 (bar)))) baz))
```

As can be seen, every time the expression above is executed, expansion of the wrapper macro causes the functions `before-sticker` and `after-sticker` to be called with the appropriate `%sticker` object. Depending on the user's preference, these functions may decide to stop execution of the program (by way of `invoke-debugger`), or to simply record the fact that the sticker was traversed. The list of recordings can later be retrieved and replayed later inside SLY/Emacs.

In our simple example, the benevolent user placed stickers on two expressions that are indeed executed, i.e. they exist in places of evaluation as defined by the syntax of the `(let ...)` special form and the `(+ ...)` function form. If a sticker had instead been placed on the expressions `(baz 42)` or `+` – two examples of non-evaluated forms – that would have created a difficulty, since the code would become syntactically invalid and fail compilation. In a worse situation, the code would still be syntactically valid but semantically absurd.

SLY/Emacs's sticker system doesn't have a way to reject these individual stickers, so it proceeds heuristically: it rejects the compilation of the whole top-level form if it determines that arming stickers results in an *increase* to the number of compilation warnings. In that case, the original form is compiled but the stickers *fail to arm*. This strategy works for a vast majority of cases, but it doesn't seem impossible to construct a pathological case where the principle of functional equivalence stated above is violated.

3.2 Source-tracking form reader

At its simplest definition, a source-tracking form reader is a variation of the Common Lisp `CL:READ` function that invokes a hook every time a sub-form is read, and proceeds to pass to this hook a measure of the character distance traveled so far to read it. This enables programs that need both the usual results of `CL:READ` and a table of form-to-source-code pairings.

Since the Common Lisp standard doesn't specify any form of source-tracking reader, some preexisting alternatives were evaluated:

(1) `Eclector`[8], a self-described "portable Common Lisp reader that is highly customizable and can return concrete syntax trees", is a full realization of a Common Lisp code reader that doesn't rely on any preexisting reader. Its distinguishing characteristic are "concrete syntax trees" that aren't represented by `CONS` cells, rather by CLOS objects that mimic the properties of such cells while also keeping "concrete" source file information;
(2) `hu.DWIM.reader`[3], by Pascal Bourguignon, is another full realization of a compliant, portable and programmable Common Lisp reader. Though not a source-tracking reader *per se*,

[2] This macro is very similar to the `WRAP` macro presented in [9], which the exception that for some technical reason that other version was realized as a special form.

it could be used in conjunction with a mechanism to track character counts in streams;

Any alternative could have been used for our purposes (the second one with minor changes). However, since we wish to minimize our program's dependency chain we also searched for simpler alternatives.

We reasoned that our stepper tool already expects a compliant Common Lisp implementation. Therefore it may, by definition, also expect a compliant `CL:READ`. So we set out to design a portable source-tracking reader that completely reuses the implementation's reader instead of replacing its implementation entirely. To achieve this, we settled on an arrangement of two separate techniques:

- A *character counting stream*. This wraps the input `CL:STREAM` object (from which we intend to `CL:READ` from) inside a so-called "gray stream" object. Such objects are not in the standard but are still widely available and used extensions to the Common Lisp standard[3]. Gray streams allow the individual character reading operations to be intercepted and controlled by the user. In this case, our character counting stream is equivalent to the wrapped input stream except for the fact that it keeps count of the number of characters read so far.
- A *substitute read-table*, achieved by rebinding the variable `*READTABLE*`. This table *shadows* each of the entries of the current read-table (using `GET/SET-MACRO-CHARACTER`) with a function that fully controls the influence of each character over the returned symbolic expression. By setting up this function in a particular manner, the resulting table remains functionally indistinguishable from the original one, while gaining the ability to invoke a hook that records source positions.

The inter-operation of the two techniques is summarized by the `READ-TRACKING-SOURCE` function:

```
(defun read-tracking-source
    (&optional (stream *standard-input*)
       (eof-error-p t) eof-value
       recursive-p (observer #'ignore))
  (let* ((ccs (char-counting-stream stream))
         (*readtable*
           (substitution-table
            *readtable*
            (lambda (shadowed-entry)
              (let (;; correct for the fact that
                    ;; one character of the form
                    ;; has already been read.
                    (start (1- (char-count ccs)))
                    (results (multiple-value-list
                              (funcall shadowed-entry)))
                    (end (char-count ccs)))
                (multiple-value-prog1 (apply #'values results)
                  (when results
                    (funcall observer (car results)
                             start end))))))))
    (read ccs eof-error-p eof-value recursive-p)))
```

[3]They are already heavily used in SLY/Emacs, for example

This mechanism provides a simple, reliable[4] and portable[5] way to read source-code. We can demonstrate its use on the simple `(let ...)` form already presented above:

```
(with-input-from-string
    (s "(let ((baz 42)) (+ (foo (bar)) baz))")
  (read-tracking-source
   s t nil nil
   (lambda (form start end)
     (format t "~&(~2,a ~2,a) <=> ~a"
             start end form))))
```

This expression returns the intended symbolic expression representing the form, `(LET ((BAZ 42)) (+ (FOO (BAR)) BAZ))`, while also producing the desired table of source positions:

```
(1  4 ) <=> LET
(7  10) <=> BAZ
(11 13) <=> 42
(6  14) <=> (BAZ 42)
(5  15) <=> ((BAZ 42))
(17 18) <=> +
(20 23) <=> FOO
(25 28) <=> BAR
(24 29) <=> (BAR)
(19 30) <=> (FOO (BAR))
(31 34) <=> BAZ
(16 35) <=> (+ (FOO (BAR)) BAZ)
(0  36) <=> (LET ((BAZ 42))
              (+ (FOO (BAR)) BAZ))
```

3.3 Specialized code walker

Equipped with a correspondence between forms and source code, the Common Lisp side of our stepper system nears a state where it may inform SLY/Emacs of where to place sticker annotations. A final obstacle remains: as we have seen in 3.1, only a subset of these forms may be annotated with the `RECORD` macro, i.e. we are only interested in the ones that are executable.

Clearly, we need an agent that understands the semantics of each Common Lisp special form[6], determines sub-expressions of interest in our source-mapped tree and discards all the others. As was already noted in [9], this seemingly simple task is severely complicated by macros and by the specific constraints of a stepper system.

3.3.1 Mnesic macroexpansion. To reach a state where nothing but special and function forms exist, a macroexpander must remove macro calls by expanding them. However, in doing so, our program *must also remember* whence each macro's expansion came, specifically the source position of the form in its pre-expansion state. This behavior is what we refer to as *mnesic macroexpansion*, the opposite of *amnesic macroexpansion*.

Take the form:

[4]This was tested in SBCL, Allegro CL, CCL and ECL. There are differences to the way that some implementations will construct the standard read-table (for example, SBCL and Allegro CL represent "constituent" characters differently) and `SUBSTITUTION-TABLE` has special provisions for that. The same technique should in theory work with non-standard read-tables, but this has not been tested.

[5]As noted, except for the use of gray streams.

[6]The human programmer *is* such an agent, but he is precisely the one we are trying to relieve of these tasks.

```
(LET ((BAZ 42))
   (COND ((PLUSP BAZ) (FOO)) (T (BAR))))
```

It may be expanded to something like[7]:

```
(LET ((BAZ 42))
  (IF (PLUSP BAZ)
      (FOO)
      (THE T (BAR))))
```

By this point, the system may finally come to the realization that only the forms `42`, `baz`, `(plusp baz)`, `(foo)`, `(bar)`, `(if ...)` and `(let ...)` are in positions of evaluation. Regrettably, it may now have lost track of *where* each form lives in the source.

To recover this information, it is not enough to naively consult the hash-table produced in 3.2, since some forms didn't exist in our original source-tracked version. Even if they did, the macroexpanding facilities are not generally obliged to return the same `CONS` objects for the forms, regardless of whether they expand them or not.

In the `Eclector` library discussed in section 3.2, a `RECONSTRUCT` function attempts to solve this very problem by correlating the fully macro-expanded "raw" tree with the original "concrete syntax tree", returning a mirroring of the former that keeps as much from the latter as possible. After some experimentation with this approach, we noticed it missed many forms in executable positions and so decided it wasn't producing the results we had hoped for. Moreover, the quality of results tended to vary across implementations, possibly due to the aforementioned `CONS`-related problems.

To overcome this obstacle we need a different approach. Instead of trying to recover from a fully macroexpanded tree, we must hook into macroexpansion as soon as it happens. This shall allow us to lose as little source-tracking information as possible. Thus, we conclude that a programmable, portable code-walker is necessary. Such a system shall let us execute a hook at each macroexpansion step shortly before and shortly after each expansion.

After surveying open-source alternatives for code-walkers, we settled on a program called `AGNOSTIC-LIZARD`[10], which fits exactly these requirements[8]. `AGNOSTIC-LIZARD:WALK-FORM`, its main primitive, produces the desired full macroexpansion and can be given a set of callback functions as hooks.

Here's the snippet that illustrates our use of this walker:

```
(defun mnesic-macroexpand-all (form subform-positions)
  (let (stack (expansion-positions (make-hash-table)))
    (values
     (agnostic-lizard:walk-form
      form nil
      :on-every-form-pre
      (lambda (subform env)
        (push (list
               :from subform
              :at (gethash subform subform-positions))
         stack)
        subform)
      :on-every-form
      (lambda (expansion env)
        (push (pop stack)
              (gethash expansion expansion-positions))
        expansion))
     expansion-positions)))
```

As we can see, our `MNESIC-MACROEXPAND-ALL` function uses a stack to take advantage of the manner in which macroexpansion traverses the tree: there may be more than one consecutive call to each of the callbacks `:ON-EVERY-FORM-PRE` and `:ON-EVERY-FORM`. However, in the end, the calls to one and the other perfectly mirror each other. Each element of the stack holds the result looked up in the `SUBFORM-POSITIONS` hash-table for non-expanded forms. That source information is later saved on the output hash-table `EXPANSION-POSITIONS`, whose keys are of expanded forms.

If we give this function the form:

```
(COND ((FOO) (BAR)) ((BAZ) (QUUX)) (T 42))
```

We may obtain something[9] like:

```
(IF (FOO) (PROGN (BAR))
    (IF (BAZ) (PROGN (QUUX))
        (IF T (PROGN 42) NIL)))
```

The resulting hash-table `EXPANSION-POSITIONS`, returned as a second value, has these mappings:

```
(IF (FOO) ..) => (:from (COND ((FOO)..)) :at (0 . 42))
42            => (:from 42               :at (38 . 40))
(BAR)         => (:from (BAR)            :at (13 . 18))
(PROGN 42)    => (:from (PROGN 42)       :at NIL)
(PROGN (BAR)) => (:from (PROGN (BAR))    :at NIL)
T             => (:from T                :at (36 . 37))
(IF T ...)    => (:from (COND (T 42))    :at NIL)
(QUUX)        => (:from (QUUX)           :at (27 . 33))
NIL           => (:from NIL              :at NIL)
(IF (BAZ) ..) => (:from (COND ((BAZ) ..)):at NIL)
(BAZ)         => (:from (BAZ)           :at (21 . 26)))
(FOO)         => (:from (FOO)            :at (7 . 12)))
(PROGN (QUUX))=> (:from (PROGN (QUUX))   :at NIL)
```

As can be seen, numerous new forms appeared in the expansion, but `MNESIC-MACROEXPAND-ALL` succeeded in keeping the source information information for all of the relevant ones.

3.3.2 Annotating interesting forms and putting it all together. A final piece of the puzzle is needed. A function named `FORMS-OF-INTEREST` is to be given the fully macroexpanded tree and along with the source-tracking information for that tree. Its task is to traverse the tree while looking for each of the 25 Common Lisp special compound forms[10], considering the evaluation rules of each. Unknown forms are assumed to be function calls, whose evaluation rules are equally well known. For each sub-expression in a position of execution, the source location is looked up and the form is collected, so that it can later be reported to SLY/Emacs's sticker system for annotation. Though its listing is too large to include here, its implementation is straightforward but for one detail described in section 4.1.

[7] This expansion is SBCL's.

[8] Furthermore, `AGNOSTIC-LIZARD` contains useful provisions to shield user code from certain nonconformities in the macroexpansions of certain built-in macros, such as `DEFUN`.

[9] This is Allegro CL's expansion. SBCL's is much simpler, and thus not so good for illustrative purposes.

[10] In reality, a few macros like `COND` and `DEFUN` are left unexpanded by `AGNOSTIC-LIZARD` so they are analysed separately as well

As we are nearing the end of our journey, we can now start putting all the pieces together. The following snippet is the final form of our Common Lisp function. Its results can be handed to SLY/Emacs for instrumentation through stickers as described in section 3.1. After compilation, the instrumented code is now steppable.

```
(defun stepper-sticker-locations (string)
  (with-input-from-string (stream string)
    (let* ((form-positions (make-hash-table))
           (form-tree
             (read-tracking-source
              stream nil nil nil
              (lambda (form start end)
                (setf (gethash form form-positions)
                      (cons start end))))))
      (multiple-value-bind (expanded-tree
                            expansion-positions)
          (mnesic-macroexpand-all form-tree
                                  form-positions)
        (forms-of-interest
         expanded-tree expansion-positions)))))
```

4 RESULTS AND FURTHER WORK

We have released the result of our work on the GitHub platform[11].

From an end user's perspective, to put the new SLY/Emacs stepper to work means pressing the key chord `C-c C-s P` (*control-c, control-s, capital P*) while the cursor is on a top-level form. This causes the interesting sub-forms of that top-level form to be automatically decorated with sticker *overlays*, which by default uses different shades of the color gray. As described in 3.1, a posterior compilation of that same top-level form shall *arm* the stickers and convert the overlays' color to shades of blue. From this point on, the stickers are executed as soon as the user arranges for the instrumented code to be run as usual.

Note that the default behavior of the sticker system doesn't equate the execution of an instrumented form to a break point, i.e. the invocation of the Lisp debugger. This is by design. As was explained in section 3.1, the default behavior is to have sticker executions merely record the return values (or non-local exits) for later replay. This which can be achieved with the key chord `C-c C-s C-r` or via `M-x sly-stickers-replay`. Alternatively, the key chord `C-c C-s S` (or `M-x sly-stickers-fetch`) can be used to fetch the most recent recordings for each sticker and visually decorate the source code, indicating (1) stickers that have been executed; (2) those that haven't yet, and (3) those that have exited non-locally.

Finally, to enable the classic stepper functionality, the user must explicitly select "breaking stickers" by affecting the value of the `SLYNK-STICKERS:*BREAK-ON-STICKERS*` variable.

We shall see, as we discuss its limitations, that the resulting SLY/Emacs stepper tool is still in its infancy. It can nevertheless be said to work reasonably well for a majority of normal circumstances, succeeding in instrumenting forms effectively and efficiently, while providing satisfactory methods of navigation among stickers.

[11] See `https://github.com/joaotavora/sly-slepper.git`.

4.1 Limitations concerning atoms

In section 3.3.2, we sidestepped a notorious difficulty with atomic forms, i.e. one-symbol symbolic expressions. This class of difficulties is already alluded to in [9, I-4.8]. The problem with atoms can be observed with the simplest of forms:

```
(lambda (x) x)
```

In this example, we note that the atom `X` has two different *manifestations* in the encompassing form. Naturally one wants only the latter to be annotated, and not the first. However, that is hard to determine reliably since both are represented by the very same object. This is in stark contrast to compound forms represented by different `CONS` cells.

We can enhance the form/position pairings table used above to record the fact that there is more than one manifestation of an atom, but that's not enough to know in `MNESIC-MACROEXPAND-ALL` which of those is a in a position of execution. The reason is that the `AGNOSTIC-LIZARD` macroexpander will only traverse sub-forms of forms actually returned by a macro's expansion. In this example, our hook is only called on the `(lambda (x) x)` and `x` forms, *not* on the `(x)` form. The latter form is merely an argument to the macro itself where we have no power of intervention, and thus there is no easy way to invalidate the first manifestation.

However, since we *do* know that `x` is manifested at `(9 . 10)` and `(12 . 13)` it is possible to devise heuristics to trace back to the knowledge gathered when first reading the form and traverse the atom's parent forms, given only the atom. A very simple heuristic can proceed like this: if the atom exists inside a compound form that does not occur in the final expansion, then that atom isn't interesting, otherwise, it is. This appears to solve the above situation but fails miserably in the presence of the `LOOP` macro since this macro has all the variable definition "unprotected" by parenthesis.

Hard-wiring exceptions to `LOOP` and other macros could ameliorate the situation, but overall this strategy feels murky and insufficient. On the other hand, if more aggressive strategies of atomic annotation are attempted, the SLY/Emacs sticker system has already shown to be reliable in the sense that if it needs to fail (because of an incorrect form being annotated), it will mostly do so early. Thus the potential to mislead users to wrong debugging conclusions is minimized.

A different approach to solve this problem could revisit `Eclector`'s "concrete syntax trees" or a variation thereof and use a portable, programmable macroexpander that also understands these types of trees, where different manifestations of the same atomic form are represented by different objects.

For now, the proposed SLY/Emacs's stepper works around this limitation by behaving conservatively and only annotating atoms in positions that are guaranteed to be safe, such as inside function call forms. This makes for the majority of situations in practice. Furthermore, users can always manually add stickers to other atom manifestations they are interested in and know to also be safe.

4.2 Interface limitations

As seen in section 3.1, SLY/Emacs's sticker system has no notion of a stack: all the armed stickers are enumerated serially and thus hierarchically equivalent. Therefore, the common "step in/step out/finish"

functionality of common steppers is unavailable as such. Once stepping has been initiated, it is currently not possible to "step over" arbitrarily large sections of uninteresting code, nor is it yet possible to designate a sticker to continue to. The nearest thing available is the possibility to ignore a particular sticker number. Furthermore, there is as of yet no notion of a stepping "session": once armed, stickers take effect immediately and stay armed (even if the corresponding source code is deleted) until the definition they pertain to is compiled again without stickers.

These features don't seem hard to realize. E.g., to enable more sophisticated navigation behavior behavior the `RECORD` macro could use a special variable to be made aware of recursive invocations to itself or to the currently executing stack frame. Thus we could keep track of stickers being executed *inside* each other, i.e. within the dynamic scope of a previously active sticker annotation.

4.3 Portability

The system as been described as "portable" or "near-portable". Indeed, if a completely new conforming implementation of Common Lisp were to spring into being, support for our stepper system would have to deal with three potential sources of non-portability: (1) support of the `AGNOSTIC-LIZARD` code-walking program, (2) differing representations of constituent characters in readtables (for the source-tracking reader described in section 3.2), and (3) support for "gray streams". We defer the discussion of (1) to [10], noting that that system is built with portability as its foremost requisite. The "shielding" it offers is useful e.g. in dealing with SBCL's implementation of `CL:DEFMETHOD`, which itself produces a complicated transformation of its body. In (2), we note that the adjustments to the read-table were needed only for SBCL's implementation, which doesn't use a macro-character function for constituents. If the hypothetical new implementation *also* did so, it is plausible that the current code would support it. Otherwise, it would behave as the remaining implementations, also requiring no extra work. Lastly, for (3), we think it reasonable to expect support for "gray streams" in new implementations, since it is a widely adopted extension, and required by SLY/Emacs to begin with.

5 DISCUSSION

Common Lisp users are concerned about features that facilitate day-to-day development. `TRACE`, REPLs, `PRINT` forms, the interactive debuggers, profilers and stickers are all ways to solve certain debugging problems: some are more suitable to some situations than others. Therefore, it's important to note that program stepping is just another tool in the toolbox, not a panacea.

By the same token, if one does implement such a tool, it should be done in a manner that is of actual, practical use. This was the reasoning behind the SLY/Emacs stepper. We think it especially fortunate that the annotation-based approach and the manner of source-code correlation in SLY/Emacs's stickers don't make use of direct references to source files or file positions. As was shown in 3.1, we merely keep an enumerated list of stickers identifiers synchronized between Common Lisp and SLY/Emacs, a mechanism that is simple but effective. It adequately resists some modifications to the source of the instrumented form or its whereabouts, such as moving it around in the source file, or even adding white-space and comments inside it. This detail is crucial in making stickers and stepping usable for day-to-day programming, since the user isn't dragged away from his editor or forced to a special confinement while stepping. We note that these advantages of annotation-based steppers were already hinted at back in 1989 by the proponents of evaluation-based steppers[6, l.41, l.42].

By contrast, a hypothetical evaluation-based visual stepper (such an as enhanced version of [4]) would find it difficult to maintain this advantage since the source-code correlation is achieved at evaluation-time and is harder to mutate effectively afterwards. Perhaps this fact can explain why the non-portable visual steppers of the Allegro CL and LispWorks implementations don't work directly with the source-editing facilities present in these IDEs. A possible solution would be to represent source-code correlation in terms of sub-expression paths in the form tree instead of character positions, but it would still be hard to resist deletions and insertions at top level. A more heavy-handed solution would lock the source file read-only for the duration of the stepping session. However, users are normally adverse to such confinements.

We also think it fortunate that stepping is implemented as an increment to the existing stickers functionality. As in [9], we consider it an advantage of the annotation-based systems that users are allowed to instrument only the definitions they are interested in. Indeed, it is often the case that steppers become tedious to operate because they step on *too much*. Yet, in our system, users can manually adjust the automatically placed stickers, removing the ones they are not interested in, or adding others.

It may also be noted that the usual stepping paradigm where the program is stopped at each point is only one of the possibilities afforded by stickers. As we explained in sections 3.1 and 4, two further ones constitute innovative means of debugging: the *post-mortem* replay of sticker recordings and the visual decoration of source code with colors indicating the state of the most recent execution of each sticker.

In formulating the development of the SLY/Emacs stepper, we have also described (1) a simple, portable technique for constructing a source-tracking reader in terms of a compliant reader implementation and (2) a reliable technique to carry over source-tracking information to macroexpansion. It is conceivable that other debugging tools could be constructed from either of these elements.

The SLY/Emacs stepper described in this essay is an effective example of a portable, visual stepping facility for the Common Lisp ecosystem. To the best of our knowledge, it is indeed the *only* system combining these characteristics. As such, there is little to compare it against. The closest match could well lie outside of Common Lisp, in the aforementioned `edebug.el`[7] stepper, an annotation-based approach that is equally well integrated with the source code editor. That stepper has a more developed interface, but requires custom declarations for stepping into macro expansions, something the SLY/Emacs stepper handles automatically. Its current limitations notwithstanding, we believe that the underpinnings of the SLY/Emacs stepper – *stickers*, a simple source-tracking reader, and mnesic macroexpansion – are solid. We envision enhancements to its interface, perhaps by incorporating ideas of non-portable visual steppers, or steppers for other programming languages.

6 ACKNOWLEDGEMENTS

We would like to thank Luís Oliveira and Michael Kirkland for early comments and advice when discussing the stepper feature, as well as Andrew Lawson for the encouragement to produce this work. We also offer thanks to the reviewers who provided very insightful comments.

REFERENCES

[1] Free Software Foundation 1984. *Emacs GNU Emacs: An extensible, customizable, free/libre text editor — and more.* Free Software Foundation. https://www.gnu.org/software/emacs/

[2] LispWorks 2011. *LispWorks IDE User Guide: The Stepper.* LispWorks. http://www.lispworks.com/documentation/lw61/IDE-W/html/ide-w-496.htm

[3] Pascal Bourguignon. 2007. *Implements the Common Lisp Reader.* https://hub.darcs.net/hu.dwim/hu.dwim.reader/browse/source/reader.lisp

[4] Pascal Bourguignon. 2012. *Implements a Common Lisp stepper.* https://gitlab.com/com-informatimago/com-informatimago/-/blob/master/common-lisp/lisp/stepper.lisp

[5] Helmut Eller, Luke Gorrie, Eric Marsden, et al. 2003. *SLIME: The Superior Lisp Interaction Mode for Emacs.* Common-Lisp.net. https://common-lisp.net/project/slime/

[6] Ivo Haulsen and Angela Sodan. 1989. UnicStep-a visual stepper for COMMON LISP: portability and language aspects. *ACM Sigplan Lisp Pointers* III (07 1989). https://doi.org/10.1145/121999.122003

[7] Daniel LaLiberte. 1988. *Edebug: a source-level debugger for Emacs Lisp.* FSF. https://github.com/emacs-mirror/emacs/blob/master/lisp/emacs-lisp/edebug.el

[8] Jan Moringen and Robert Strandh. 2018. *A Portable, Source-tracking Reader for Common Lisp.* LaBRI, University of Bordeaux. https://github.com/s-expressionists/Eclector/blob/master/papers/to-submit-1/eclector.tex

[9] Glen Randolph Parker. 1987. Annotation-Based Program Stepping. *SIGPLAN Lisp Pointers* 1, 4 (Oct. 1987), 3–11. https://doi.org/10.1145/1317216.1317217

[10] Michael Raskin. 2017. Writing a best-effort portable code walker in Common Lisp., In Proceedings of 10th European Lisp Simposium. *10th European Lisp Simposium* I, 1, 11. https://doi.org/10.5281/zenodo.3254669

[11] João Távora et al. 2014. *SLY: Sylvester the Cat's Common Lisp IDE.* GitHub. https://github.com/joaotavora/sly.git