Froglet: a source-level stepper for Lisp

Stuart Watt
Human Cognition Research Laboratory
The Open University, Milton Keynes, MK7 6AA
Email: `S.N.K.Watt@open.ac.uk`

Abstract

Froglet is a source-level stepper for Common Lisp. Unlike previous steppers, which used pretty-printed reconstructions of definitions to show the progress of execution, Froglet uses the text from which the definition was read. This means that forms can be shown under evaluation in the right context, complete with comments and related functions. It also provides views onto the stack and onto the lexical environment of the form currently being evaluated.

This paper describes the background to Froglet and how it is related to steppers for conventional languages, gives an overview of its implementation and interface, and shows that it could form a key addition to Lisp environments.

Keywords: Lisp, program development environments, steppers, debugging

1 Introduction

Ten years ago, Lisp environments were the best program development environments available, and were widely promoted on claims that they provided a speed of development unparalleled by other languages and environments. Today, the situation is quite different. Other languages, particularly C and C++ now have environments—often built by former Lisp suppliers—which rival those of Lisp. As Gabriel (Gabriel, 1990) says "almost no progress in Lisp environments has been made in the last 10 years". This is particularly true in debugging: "In the recent past conventional language debuggers have surpassed those offered by Lisp" (Richardson, 1990).

The principal change was the advent of source-level steppers for conventional languages. Debugging these languages used to be a complicated and messy matter, often involving an assembly-language representation of the compiled program. Only when steppers and debuggers became powerful enough to work at a source-level and show execution in terms of the defining text rather than any reconstructed representation did the final payoff of high-level languages come about—true abstraction from the details of the underlying hardware platform.

Lisp, of course, had supported abstraction from the underlying machine for many years, but at a cost. It abstracted using an internal representation—the S-expression—as a higher-level representation of a program. When everybody else was debugging machine-code, Lisp programmers were debugging S-expressions and had a strong advantage. Now everybody else is debugging at the program text level, and Lisp programmers are stuck at the S-expression level.

It is this extra representation that costs. By having two representations, Lisp programmers must constantly relate their original program text to the S-expression intermediate. By contrast, conventional language programmers only have one representation to deal with. Novices, of course, have different requirements from experts in this; these intermediate representations can be very useful in the learning process.

The Froglet stepper is an attempt to overcome these problems. It is a tracing and stepping system intended for expert use which is genuinely grounded in the source text and not in any intermediate such as S-expressions. It does this by augmenting a number of key elements of the Lisp environment, notably the reader and the evaluator, to preserve the mapping between the program and its source. Building on the traditional foundations of Lisp debugging tools, it enhances them to overcome their limitations and produce a system that illustrates the approach and which can be used in real debugging situations.

2 Background

Steppers for Lisp have an extensive pedigree, including, for example ANTICS (Dionne and Mackworth, 1978), HEVAL (Mikelsons, 1983), and ZSTEP (Lieberman, 1984), although virtually every Lisp environment has some kind of stepper. With all these tools available, it remains an important question why they never caught on. Most programmers today still rely on inserting `print` and `break` forms into their programs to see what's going on, and `trace` is probably still the most widely used debugging tool. Steppers are among the least-used tools in Lisp, where in languages like C they are the key debugging tools.

The principal problem with traditional Lisp steppers is that they *reconstruct* a representation of the source text, usually using a pretty-printer (HEVAL maintains its display in a structure editor, which has a similar effect.) This representation is different from the original text; the layout is subtly different and is set in a different context. Although the definition originally appears in a file with the programmer's comments and layout, and next to related functions it is presented in the stepping process in a much sparser context.

Harlequin's LISPWORKS environment (Richardson, 1990) does have some source code management. It associates definitions with their corresponding source file, and will attempt to find a definition's text by searching through that file for the definition, then parsing that definition to find the required element within it and highlighting it. However, searching frequently doesn't work, both because of duplicate calls and because the location of a definition in a file is defined by where the form was read and not by where the enclosing top-level form was compiled.

Many steppers for conventional languages, such as Apple's SADE, can in principle be used to step through a program written in any language, or any mixture of languages. They work by associating ranges in a file with ranges in program instructions. When a program is being run, the stepper can use these tables to find the corresponding character range in the file, and then open the file and highlight the character range. This approach could, in theory, be used for Lisp, although there are a number of special problems imposed by its different syntactic and execution models.

First, Lisp has a reader which converts a lexical representation into an internal representation, *and discards the mapping between the two.* The definitions in a file are converted to definitions in memory and from then on there is no reference to the file representation.

Secondly, Common Lisp provides a number of hooks into program execution, but the scope of these hooks is limited; there are some things which just cannot be implemented using the existing hooks, and which require more complete access to the evaluator to work.

Froglet is as complete a stepper as can be built within these limits. It implements most of the features of a typical source-level stepper for another language, such as C or C++. It has a source-level view, but it also provides views onto the lexical environment, the stack, and the execution history in a single integrated framework. Curiously enough, this approach means that unlike HEVAL, ANTICS and ZSTEP, Froglet distances itself from the notion of evaluation of S-expressions being the core of Lisp. Instead the approach is much more like a conventional language. S-expressions are effectively hidden so the execution goes directly from forms that appear in the file to their results.

3 Implementation

A source-level stepper naturally breaks down into three systems. First, there is an augmented parser which keeps around enough lexical information to be able to map back from the execution level to the lexical level. Second, there is an augmented execution system which provides access to normally hidden information, such as variable bindings, and which can influence the execution itself by setting breakpoints and generally tracking the program execution. Third, there is an interface which presents the state of the program execution mapping back to the lexical level. Froglet contains all three components.

The augmented parser is a patched version of the Common Lisp standard readtable. New character macro functions are written, principally for the parenthesis "`(`", the quote "`'`", and the function "`#'`". All call the reader recursively to read the next forms in the stream in the same way as standard Common Lisp, but before returning their value they store a reference to it, and associate it with a range of character offsets in the input stream. Some other reader features, such as the "`#n=`"and "`#n#`" labelling forms, also need to be treated specially for the form tables to be correct.

Symbols and other atoms also need to be treated specially. Unlike lists, two references to the same symbol can't be distinguished by `eq`. A typical symbol like `let` may appear several times in the same definition, so the patched reader also provides a reference to the enclosing form. This, being a list, allows `let` in one list to be distinguished from `let` in another. A count is also used so that multiple references to the same symbol in the same enclosing list can be distinguished.

The patched reader generates a form table which associates each enclosing form with all its subforms, and for each subform, a count and list of ranges in the file which read to that subform. A form table is kept for each definition that the stepper is aware of, and for any definition where this form table isn't available, the stepper will skip the corresponding definition completely. Of course, if the definition is edited, the form table will be out of date. This will be corrected when the form is next evaluated, but in the interim the editor can adjust the ranges as text is added to and deleted from the definition. For our prototype implementation this ensures that the text and the form table are kept in step as far as reasonably possible.

The second part of the stepper is the modified evaluation system. This was implemented using the standard Common Lisp `*evalhook*`, and the interpreter was used to step through the code. In principle, a compiler could also provide the stepping facility, by associating ranges in compiled code with ranges in the source file. (This is the approach usually taken by source-level steppers for other languages; it has the advantage of being both fast and non-intrusive, but it would require significant changes to the compiler.)

Using `*evalhook*` has its limitations; the state of the execution is only partially visible to the hook function. Consider evaluating the following form.

```
(let ((x n)
      (y n))
  (+ x y))
```

The hook function will first be given the whole form, and then, on the recursive calls, it will be called twice with the variable `n`, and then once finally to evaluate `(+ x y)`. The execution semantics of the `let` form are all that allow the two evaluations of `n` to be distinguished. The interpreter itself will be aware that the first `n` belongs to the first `let` binding, and the second `n` to the second `let` binding, but this information is never available to the hook function. Because of this, source-level stepping forms like this cannot be done accurately with only an evaluation hook.

Macros cause a special problem; their expansion can mix code from the macro definition with code from the macro call quite freely. Currently, macros can't be traced, but code in a macro call is traced to follow the execution of the expanded code. In figure 1, for instance, the selected form is being evaluated in a `case` macro body, following the correct order of evaluation of `case` macro calls. Higher-order functions such as `apply` and `funcall` can cause similar problems. When calling a function constructed dynamically it may not be steppable. Such special cases are the exception ather than the rule; virtually all code can be stepped correctly.

With a fuller integration with the interpreter or compiler more features become possible. A frequent problem in debugging is to step too far; this is one area where "post-mortem" steppers have an advantage because they can go back to an earlier state directly. Many Prolog tracers, such as PTP (Eisenstadt, 1984), work like this, but in Lisp the debugging process is different; programmers frequently change definitions in the middle of execution and this can't be supported by post-mortem steppers. A more appropriate solution for Lisp would be to permit an 'Undo' in the interface, which rewinds the program back to a snapshot of the state kept before the last stepping command was executed. Scheme's `call-with-current-continuation` could be used to store the previous state, but Common Lisp, unfortunately, does not have anything equivalently powerful, and, without reimplementing the evaluator, access to the entire computation state is not possible and a true 'Undo' cannot be implemented.

The third component of the stepper is the interface. At the core of this is a simple text editor, but an editor which can reflect changes in highlighted text made under program control. As the program execution proceeds, the text corresponding to the expression just evaluated (or just about to be evaluated) is highlighted in the editor.

Since the display is derived from the original textual representation of the code, and not from any pretty-printed reinterpretation, the definitions are seen in the right context and complete with any appropriate comments. It is this that makes Froglet different from other Lisp steppers.

Figure 1 shows an evaluation of a function `lastpos` in the stepper. `lastpos` is a recursive function implemented as part of a lexical analyser generator. It takes an analyser expression labelled with positions and returns a list of all the possible last positions in the expression. Figure 1 shows the evaluation with a stack view—the inner-most call is shown on the top of the stack. The pattern of execution of this function is particularly clear in this case: the function generally recurses down expressions composed with the `seq` combiner.

The stack view is not sufficient for most debugging purposes, though, so two other views are available, and these can be used either instead or with the stack view. The first of these, the bindings view, shown in figure 2, shows the current expression being evaluated (or its return value, when leaving the evaluation), but set in the lexical binding context presented as if created by a `let` form. This provides a watch on all bound variables. The `let` form is constructed to show the current evaluation in a textual representation immediately intelligible to Lisp programmers.

The second alternative view, trace view, shown in figure 3, roughly parallels the Lisp tracer, and shows functions being called and returning values.

4 Using the stepper

At first the stepper behaves just like a traditional text editor. Files can be opened and their contents viewed and changed, and typical text editing functions like searching are all available. The most obvious difference is that to the left-hand-side of each definition there appears a vertical marker bar, which behaves as 'handle' allowing direct manipulation of the definition as an object. These can be seen clearly in figure 1. Clicking on the handle selects the whole definition, and when a definition has been selected, a property-sheet can be opened on it. A typical definition property sheet is shown in figure 4.

The property sheet allows the stepper to be set for a particular definition. In figure 4 this is shown by the check button for the definition `lastpos` being selected. Once this change to the definition has been applied, any calls to the function will call up the stepper window and start stepping. This property sheet also allows the standard Lisp tracer and breakpoint system to be used. Setting the Lisp tracer button traces calls in the normal way, and setting the breakpoint button calls `break` when the function is called.

From this point, when the user (or a program) calls the function `lastpos` the stepper window will come to the front of the display; the user can then step through

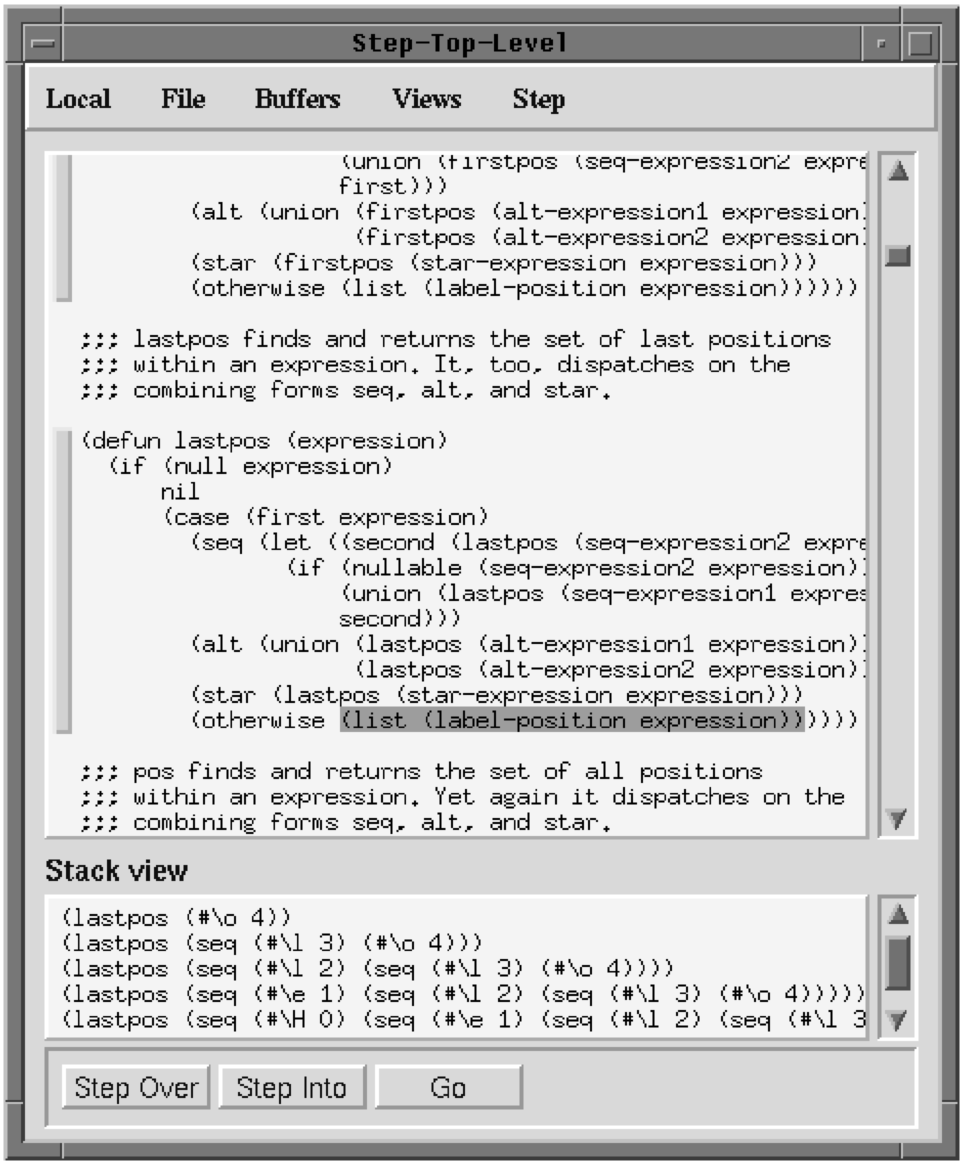

Figure 1: The Lisp stepper with a stack view

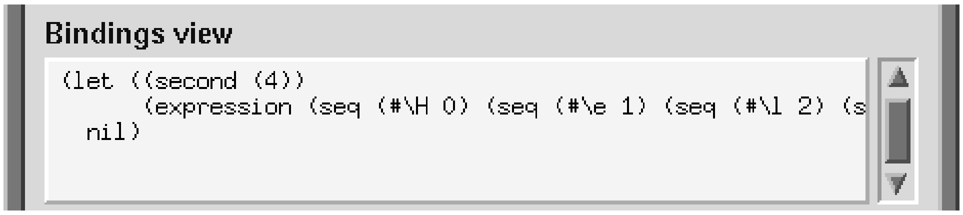

Figure 2: The bindings view

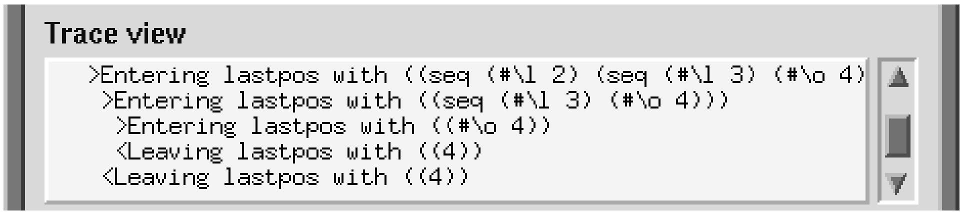

Figure 3: The trace view

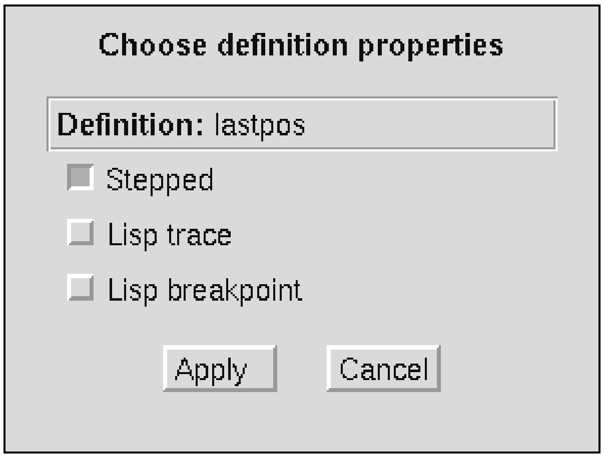

Figure 4: A sample definition property sheet

the definition using the ‘Step into’, ‘Step over’, and ‘Go’ buttons. ‘Step over’ and ‘Go’ can both animate the execution; that is, the forms being evaluated are rapidly shown being highlighted in turn to give an overall impression of the pattern of the program execution.

The stepper is not intended to replace the existing Lisp debugging tools, and in particular the tried and tested trace and breakpoint systems. Rather, it is intended to complement them with the features common in development environments for other languages, and to provide a coherent development environment which integrates these existing approaches.

5 Future work

This program has proved that source-level stepping is possible in Lisp, but there are many areas where it remains to be improved. Most importantly, an ‘Undo’ command is needed, so that when the user over-steps there is a way to get back and have another go.

Secondly, there are many techniques people commonly use in debugging which the stepper can be enhanced to support more fully. In particularly, most programmers resort to using `print` in programs to get access to the contents of variables and data structures. Unfortunately, this is intrusive and all the added `print` calls need to be removed. A better solution would be to enhance the editor so it could add ‘print points’ into the code while keeping the textual representation unchanged. From the user’s point of view, it should be possible to select a form in a definition and set it as a print point, so that every time it evaluated the form and the value are displayed, without ever changing the text.

For some functions, particularly recursive ones like the factorial function, the stack view will often show a large number of frames following a pattern; so many that a high-level understanding is buried under too much information. To overcome this, the stack view needs to be augmented with a pattern detection system to compress the kinds of repeat which are typical of simple or mutual recursion.

Customisation is another topic for future work. Different programmers work in different ways, and should be able to add their own enhancements to the system. In part this is reflected in the choice of different views. Once the principles of mapping between text and the internal program representation have been proved many features can be added to the stepper within this framework.

6 Conclusions

Froglet is a working stepper; it was designed to show that it is possible to design and build a source-level stepper for Common Lisp and it has shown that it works and can be an effective debugging tool.

Its development has raised a number of issues relating to Common Lisp itself. Some features either couldn't be implemented, or were much more complex than they needed to be; sometimes this is because a vendor's access to the details of an implementation needs to be brought to bear on the problem, and sometimes it is because tools like `*evalhook*` were just being stretched beyond their breaking point. Nevertheless, this initial prototype has proved highly successful, and is convincing enough to encourage future work on integrating Froglet into implementations as a debugging tool to compete with those of the more advanced environments of conventional programming languages.

Perhaps some of this shouldn't be regarded as new; Froglet is an attempt to borrow from debugging concepts and techniques developed outside the Lisp community into the next generation of environments, to produce an integrated whole that offers the best of both worlds. Lisp as a language is good enough to provide a significant advantage in program development, but only so long as it isn't hampered by environments that don't at least keep up with those of conventional languages. Froglet would be a useful addition to any Lisp environment that wishes to redress the balance.

Acknowledgements

I would like to thank John Domingue for his encouragement, Paul Mulholland for the discussions that let to the 'Undo' feature—even though I couldn't implement it—and Marc Eisenstadt. And especially to Henry Lieberman for his support.

References

Dionne, M. S. and Mackworth, A. K. (1978). Antics: A system for animating lisp programs. *Computer Graphics and Image Processing*, 7(1).

Eisenstadt, M. (1984). A powerful prolog trace package. In *Proceedings of the Sixth European Conference on Artificial Intelligence*.

Gabriel, R. P. (1990). Lisp: Good news; bad news; how to win big. In *Proceedings of the First European Conference on the Practical Application of Lisp*, Cambridge, UK.

Lieberman, H. (1984). Steps towards better debugging tools for lisp. In *Proceedings of the ACM Symposium on Functional Programming*.

Mikelsons, M. (1983). Interactive program execution in LispEdit. *ACM Software Engineering Notes*, 8(4).

Richardson, C. (1990). Lispworks: A common lisp programming environment for unix workstations. In *Proceedings of the First European Conference on the Practical Application of Lisp*, pages 127–134, Cambridge, UK.