UnicStep - a Visual Stepper for COMMON LISP

Portability and Language Aspects

Ivo Haulsen and Angela Sodan

GMD-FIRST at Tech. Univ. Berlin

Hardenbergplatz 2

D 1000 Berlin 12

Fed. Rep. of Germany

arpanet: gmdtub@ivo.uucp / gmdtub@angela.uucp

csnet: ivo%gmdtub.uucp@ira.uka.de

Abstract: This article presents a visual stepper for Common Lisp with simple backup feature. The stepper is based on Lieberman's approach for visual stepping. The stepper operates at the source level of programs and uses the GNU Emacs editor for source code processing. Problems encountered on implementing this stepper are described: read-syntax, macroexpansion, and source-localization. Especially, language features are discussed which affected our goal to make this tool portable across Common Lisp systems.

Introduction

For finding bugs in a program two main strategies are applied: inspecting the state of a program after an error has happened and watching the execution of a program. Whereas in conventional languages like C these strategies are combined in a single tool called debugger, in Lisp the involved functionalities traditionally are separated in different tools. The tool called debugger usually only has the functionality of statically inspecting the state of the data and of the control stack after an error has happened. For dynamically watching the execution of a program the tools tracer and stepper are supplied. The tracer provides the chance to print information or to interact with the user at function entry or exit without changing the body of the function. This is normally realized by creating an encapsulating function and can be applied to interpreted as well as to compiled code. The stepper allows to evaluate a Lisp program form by form by stopping before the evaluation of a form, displaying results, and taking new instructions from the user. Stepping usually can be applied only in interpretative mode.

Commercially available steppers present the internal representation which originally (i.e. in LISP 1.5) has been equivalent to the external representation besides to comments and formatting. In Common Lisp, however, the internal representation differs from the appearance in the source text in respect to reader macros which play a fairly important role. Additionally, in some implementations the internal representation includes special optimizations for the interpreter (e.g. memoization of macro expansions). However, the user should not be faced with the interia of the internal representation. He should be able to operate on source level because this reflects the way he has written down the program and he is thinking about it.

Additionally, the steppers normally present the current form and the evaluated results and thus are hard to control: They print out too much garbage and they do not show the context of an evaluation. And as general problem of all debuggers Lisp steppers can only work on a low level or a high level of execution (eval a form at once or in detail). It is not possible to go back in the process of evaluation.

One effort to solve these problems was the visual stepper proposed by Henry Lieberman [1]. His prototye "Zstep" had the following conceptional features:

- The context of the form to be stepped is shown. This context is the definition of the function being executed.
- The form to be stepped is highlighted.
- A substitution model being based on the functional background of Lisp is applied: a form is substituted by its result within the context.
- It is possible to back up from errors and normal evaluations to examine the intermediate steps of the evaluation (side-effects, however, are not undone).
- Instead of the internal representation the source code is used for presenting the context and the form.

This stepper is still based on the traditional interpretative approach of Lisp. For source code handling the Zmacs editor has been used which is integrated in the Symbolics Lisp system.

We picked up this idea of a visual stepper and realized the above features with some improvements like stopping also before a function is invoked or offering several ways for macro handling. Our goal was to make the stepper portable across Common Lisp systems. For source code handling we used the GNU Emacs editor (a wide-spread public domain Emacs version for the UNIX operating system) [9] which is not integrated to Common Lisp systems but runs as a separate process. We will now present our stepper in more detail, describe the problems we had with implementing the tool, especially in respect to portability, and discuss the interpretative approach for stepping in contrary to the so-called annotation-based approach.

UnicStep

Our stepper [7] was built for the Common Lisp System UnicLisp [8], which has been developed at the GMD. UnicLisp runs on the MC68020 machine UNIMA under UNIX System V. We have ported the stepper to Lucid Lisp using a SUN 3 workstation.

A visual stepper needs to communicate with an editor for presenting the source code to highlight parts of it and to substitute the evaluated results. The editor has to be able to handle Lisp syntax. We used the GNU Emacs with LISP running as a subprocess. The GNU Emacs provides an interface between Emacs and Lisp which allows communication in both directions. A higher-level protocol which is function-oriented has been built on top of this simple character communication interface. On Emacs side the filter feature has been used for realizing the interface extension. We did not create new communication ports for the Emacs like the Allegro socket links [6].

Like Zstep UnicStep shows two windows at the screen. The source code of a function or macro stays unchanged in the code window, whereas in the result window the forms are substituted by their results.

At the command level the main differences to conventional steppers are:

- You can back up to the previously evaluated form to examine deeper levels of evaluation.
- You can back up after an error has occured (it is not necessary to start the stepping again).
- You can evaluate silently (i.e. without stopping) until a specified function call is reached or a condition is satisfied.
- You can finish the current function without intermediate steps (i.e. without stopping).
- You can examine the invocation hierarchy by switching to windows of the callers (one context of a form is the current function, another the position of the function within the invocation path).
- You can choose how macros are to be handled: step through the user supplied original arguments of the macro call (e.g. the body of a DO-Macro), step through the full expansion of the macro, or step through the expander function.
- You can reload a compiled function or macro by interpreted code and step through.

Problems of the Implementation

The Identification of the Current Form within the Source Text

Two problems arise when mapping back to the source, i.e. when highlighting the current form in the source text or substituting result values: The first one is to find the position of the current form within the internal representation of the enclosing lisp function (i.e. within the lambda-list on which the evaluator works). Lists can be found by comparison with EQ provided that the evaluator works on the forms of the defining lambda expression and not on copies of them and that subexpressions being EQUAL are not collapsed to a single list by the reader or the loader. (We believe that these restrictions should be specified by the Common Lisp standard.) However, because different appearances of symbols are mapped to a single unique identifier by nature they only can be found by remembering the enclosing list form and their position within that list.

The second problem is to find the corresponding location within the source text. A simple textual searching is no solution because of the reader macros and because a form can occur more than one time within a function. So, we look both at the source text and the internal representation as trees. Then, the task is to match these trees. Path specifications are used to describe positions within each tree.

The matching of the trees is complicated by the fact that reader and loader have processed the function definition on entering it to the Lisp system. Thereby the original structure may have been changed and even in a way which is implementation dependent. This happens before the stepper begins to work and so the stepper has to find out what transformations have been done by reader and loader, i.e. the stepper has to emulate the behaviour of reader and loader.

Emulation of the Reader

To emulate the behaviour of the Common Lisp reader, to some extent we use the GNU Emacs. This editor can handle LISP syntax, but uses another algorithm to parse the text than the Common Lisp reader. So we have to apply several tricks to get the right affects. In addition, the expansion of reader macros is implementation dependent (e.g. for BACKQUOTE). Thus, the stepper has to be modified in this respect if it is being ported.

If we could use an editor, which understands exactly the syntax of Common Lisp and works on the appropriate syntax table of the LISP system, these problems would not appear. Especially the latter aspect, however, could only be fulfilled by an Emacs editor being written and integrated within the Common Lisp system.

Even then there would remain two further problems: First, if the parser uses the syntax table and applies the reader-macros, this may produce side-effects. Second, there is no specified binding of syntax tables to files so that it is impossible to decide which syntax table is to be used when parsing a function. Defining reader macros with side-effects is allowed but strange. All code processing tools like compiler or cross reference analyzer rely on the assumption that the user obtains from defining such reader macros so that the working environment is not damaged by reading in the files to be processed. (Perhaps the language definition can explicitly specify that it is an error to define reader macros with side-effects.) The second problem is a general one, too. A main advantage of program development with LISP is that functions can be changed and reloaded selectively. So on sending a function definition from the editor to the LISP system the right syntax table has to be chosen (as well as the right package). Common Lisp would need a standardization of both package and syntax table specification. For example, in SYMBOLICS Common Lisp every file has a header containing packagename and syntax table. In addition package and readtable creation should be centralized.

Emulation of the Loader (Executing the Defining Forms)

On loading a function the DEFUN macro binds the defining lambda-expression to a symbol. However, the header (containing name, arguments, documentation string, and declarations) of the original form is not identical to the header of the lambda-expression finally stored. Due to the Common Lisp specification a BLOCK form is to be inserted enclosing the body of the function. It is not specified whatelse can be done. By processing the header to find the body, some implementations modify the forms processed, e.g. they combine several declarations encountered to a single one. Additionally, the body of the lambda-expression might be changed for optimization. Such transformations cause problems for establishing the relation between source text and

internal representation. If macros are expanded and thereby the original lambda expression has been destroyed then there is no longer any chance to do the mapping. Macro expansion at load time, however, is explicitly allowed by Common Lisp.

For macro definitions there exist similar problems. However, the relation between the DEFMACRO form and the expander function is more implementation dependent and therefore even harder to establish.

For stepping through the macro call instead of through the expander or the full expansion, the user supplied arguments have to be identified within the expansion of the macro. This is only possible for lists (which can be compared with EQ) and for symbols within argument lists. Thereby, the possibility to step macros on the level of the macro call is based on the vague assumption, that the expander function retains most of the shape of the macro arguments and that no unnecessary copying takes place. The latter can partly be regarded as a restriction on DEFMACRO, otherwise this is a matter of writing expanders. System macros like DO, however, can be defined to follow these restrictions and usually do. To avoid restrictions on stepping through macro calls templates would have to be defined. This could be done for at least system defined macros.

Further Aspects of the Reader and Loader Problems

A solution to the problems with DEFUN/DEFMACRO and with reader macros would be to specify exactly what transformations are done. Implementation-dependencies in transformations not only make source mapping hard but in a sense also disrupt the original characteristic of LISP to handle code as data and to even write selfmodifying code because it is unpredictable to what internal lists the edited text results. However, handling code as data seems only to be applied by tools like the compiler or for creating functions dynamically during execution. And for those applications only the internal representation is interesting.

There are, however, even implementations which insert some special internal forms/functions to mark expressions created by predefined reader macros like BACKQUOTE so that expressions can be re-transformed and printed in the way they have been typed in by the user. This re-installs the equivalence between internal and external representation and seems to allow to use a structure editor for writing programs instead of switching to a source file editor. However, this would raise new problems in respect to packages and read-tables for applying the changes. On the other hand, source file editing is essential because it guarantees to have saved all changes in a consistent way, centralizes the task of program creation, and is necessary for handling dependencies in processing order or processing time (as e.g. imposed by EVAL-WHEN). So source file handling would not become obsolete, but steppers would not have to map to the source (except to the loss of the original formatting and comments).

Backup and Error

A simple backup facility as in Zstep is provided by remembering the last evaluated form and the attached state of the stepper. This enables to re-evaluate a form, but side effects can not be undone automatically. So the user has to be conscious of potential side-effects if he backups. The stepper establishes a condition handler such that errors happening during execution of a form pass control back to the stepper (e.g. for allowing to re-evaluate the form with more detailed stepping).

Source Localization

For showing the source text and for re-loading the interpretative version of a compiled function the definition has to be localized within the source files. There are two different approaches for such a source localization: the TAG table mechanism of the GNU Emacs can be used, or the related source files can be remembered within the LISP system when files are loaded. The first alternative can only be applied to a static collection of files. Since tag table creation and modification is slow, GNU tag tables are not feasable for interactive program development with frequent changes. In addition, this approach is not applicable to functions or macros which are not created by DEFUN or DEFMACRO (like functions created by DEFSETF) because there do not exist textual search patterns.

Thus source memoization inside the LISP system is more appropriate, but cannot reflect changes at the level of file handling done via the operating system (like moving, merging, and so on). Nethertheless we use the latter

alternative and, for escape, i.e if the file of a function/macro cannot be determined, the user is asked to specify the file containing the function/macro in question.

However, it is impossible to realize such a source memoization in an implementation independent way. Each DEFUN or DEFMACRO would have to be extended to add the appropriate information. At least for compiled code, however, at present the only solution is to modify the LISP system (i.e. the compiler) so that within the header of every compiled file the corresponding source file is named and that the compiled defining forms allow the necessary bookkeeping when loaded. It seems not to be realistic, to demand for source memoization being included as a Common Lisp feature. However, because in Common Lisp the equivalence between external and internal representation is being lost, the reference back to the source is not only of interest for steppers, but also e.g. for support in selective modification (i.e. editing) of functions. So the possibility to memoize corresponding source files is generally important for implementing portable program development environments. A solution might be to include at least information about the corresponding source file in the header of each compiled file and to add something like a DEFINE-HOOK to Common Lisp. The function bound to the hook would be called on every change of a symbols's definition. By using this hook, environments could add a source code memoization facility without having to dig into the implementation of the underlying Common Lisp system.

As far as the details are concerned, there remain some problems in respect to read-time conditionalization with features, generators (see below), or lexical closures created by FUNCTION. There might not exist a perfect solution but source memoization can concentrate on the most common and most important cases.

Generators

Common Lisp includes some generators: e.g. DEFSTRUCT which produces functions for slot-access and construction. The current version of UnicStep does not handle such created functions at source code level. Instead the internal representation of the generated functions is pretty-printed. This is not a significant restriction for e.g. a constructor function. In case of methods, however, which are created by DEFMETHOD and which may contain much important code and may be used intensively in object-oriented programming style it is hard to deal with the internal representation.

The problem for source mapping is primaryly to find the source position. There is no directly corresponding function definition in the source text. As with DEFMETHOD, usually the interesting part of the function code is contained in the generator call. For finding the appropriate generator call, the source memoization would have to be extended. The fact that generators may even store the created functions in some special way instead of introducing them with defun or defmacro may cause problems in determining the name of the created function and attaching the positional information to the function. When the call is localized, the generator call would have to be read in again, and then stepping for the generated function could operate similar to stepping through the arguments of a macro call.

Another Approach for Mapping to the Source

Instead of emulating reader and loader another approach is imaginable. If the definition would be read again and if there would be a READER-HOOK which is invoked on every end of parsing a LISP form, the problem could be solved in a more convenient way. The function to be bound to the reader-hook should have three arguments: the sub-expression-list being created, the start position and the end position on the file-stream or the string. (This would require a character counter within READ.)

After having read the definition and having saved all information about the forms within the definition and their positions in the source text, the defining form would be re-executed. The transformations done by that defining form, e.g. a DEFUN, can now be found out by searching the forms within the resulting lambda-list (applying a comparison with EQ).

This proceeding provides more flexibility than the emulation of reader and loader, and eliminates the need for specifying the transformations allowed by DEFUN or DEFMACRO as well as the need for having an integrated Emacs editor which can work on the current syntax table. However, the problems of side-effects in reader

macros and of determining the syntax table to the used are still the same. As stated above, these problems can be solved by other measures.

Experiments have shown that the reader-hook appoach does work and that the possibility to call the hook function does not slow down normal reading perceptibly.

Summary of Requirements for Portable Stepping

The Common Lisp features evalhook and applyhook provide a mechanism which allows to interact with the interpreter and to modify thereby the evaluation process without changing the interpreter itself. As stated above, this, however, is not enough to build a test tool like our stepper in a portable way. Here is a summary of the already suggested modifications and additions. Most points are also of impact for other source processing or program development tools:

- Information about syntax table and package should be incorporated in the header of sources files. The creation of packages and syntax tables should be centralized.
- A define-hook should be incorporated.
- Side effects in reader macros should be disallowed.
- Copying or collapsing expressions should be explicitly disallowed for reader, loader and evaluator.
- The transformations done by reader and loader should be specified or a reader-hook should be added.

However, we detected some more problems in respect to portability and realization which result to the following requirements:

- Lexical environments should become first class data object.

The stepper needs access to the lexical environment of the interpreter, e.g. for searching for local function or macro definitions. At present the environment is passed to the evalhook function as an argument but no implementation-independent accessor functions are provided. Henry Lieberman and Christopher Fry [3] proposed a standardization of the EVAL function with some additions to Common Lisp. The most important aspect is that lexical environments should become first class data objects. This would be the appropriate solution for the stepper problem.

- An accessor functions should be provided which delivers the name of a function. (This extension, however, is already sugggested by the clean-up committee of the ANSI Common LISP standardization.)

The stepper needs to know the original name of a function to refer back to the source. However, this name may be different from the one which is being used to call the function because the definition might have been shuffled around (by use of SETF SYMBOL-FUNCTION).

- Pretty-printing should be able to show the same representation as printing does except to formatting.

Pretty-printing is the designated escape representation for the situations where the stepper is unable to map back to the source. However, some implementations choose to transform internal expressions resulting from e.g. BACKQUOTE back to the originating reader macros when pretty-printing. These, however, sometimes are just the expressions which can not be handled by the stepper within the source representation. So the escape handling fails. Generally, we believe that it should optionally be possible to show the internal representation with reader macros being expanded even in case of pretty-printing.

These last points seem also to be of quite general interest. Because most of the suggestions are of more general importance they should be taken into consideration even if support for stepping seems not to be essential. On the other hand, the implementation of the stepper succeeded without having this support. However, for porting the stepper some additional effort would have to be spent.

Evaluator-Based contra Annotation-Based Approach

UnicStep is based on the interprative approach. The stepper interacts with the interpreter and, therefore, can only be used in interpretative mode. However, other approaches than this traditionally Lisp-like one have been made. I.e. G. R. Parker [2] developed a stepper for Lisp following a so-called annotation-based approach. This approach prepares the code to be stepped by inserting stepper commands in the source code and works still with the compiled version of the code. Like Zstep and UnicStep, Parker's stepper presents source code context by using an editor. Today, many users prefer to compile their programs at once instead of testing them interpretatively first. Moreover, some Lisp systems running on small machines have eliminated the interpreter totally and rely only on compilation. So the annotation-based approach seems to be attractive. Additionally, it is interesting if the problems discussed above can be avoided in this approach. We will now discuss the advantages and disadvantages of the interpreter-based in contrary to the annotation-based approach. Thereby, we will reply to Parker's arguments in favour of the annotation-based approach:

- Ability to step compiled code: For annotating a compiled function it is necessary to read the modified source of the function, and to compile it again. On the other hand, it would not cause more overhead to reload the interpretative version of a compiled function. So, if the user prefers to compile at once, it would still be possible to switch to the interpretative version selectively. (This works unless the user relies on compiled mode so strictly that he puts macros only in the compiler environment or produces other strange dependencies). If, however, the annotation-based approach does not influence the normal code generation of the compiler significantly and therefore would allow to use the stepper for testing the code generation of the compiler, the ability to step compiled code would have additional positive aspects for system development.
- Efficiency: In most cases it is no efficiency problem to interpret the code of just a few functions. Efficiency may become critical if the whole program or greater parts of it are interpreted. However, this problem can be reduced by executing the bulk of the program in compiled mode, and switching to interpretative mode only when the tracer stops at the entry to the concerned functions.
- No evaluator needed: At least, access to the lexical environment (which seems to be quite important for a stepper) and execution of the macro expander are no longer available in compiled code. If, however, theses functionalities would be added to the annotation-based stepper this would result in the re-implementation of an evaluator. If not, less information can be obtained than by using the interpreter-based approach. Despite from this information aspect, for annotation we need a code walker, which has to know about the semantics of the language to a certain extent (e.g. which forms are special forms). If a code walker is not already supplied by the Lisp system (e.g. as part of the compiler) the costs for the implementation will be nearly as high as for an interpreter.
- Language and environment independence: An annotation-based stepper consists of two parts: one for wrapping the code and thus preparing the function for stepping, and another one which performs the real stepping and is invoked by the annotated function. The last one might be language independent, but the code for wrapping is highly dependent from the language and its semantics. However, it is an advantage of the annotation-based approach, that there are two separate parts with a clear interface.
- Convenient selectivity: Selectivity means, that a user can specify exactly which code sections are to be stepped. This, however, can be accomplished with the interpreter-based approach equally well and is only a matter of the interface handling routine which has to create the appropriate path specification from pointing to a form. On the contrary, stepping all the code of a function or deciding dynamically which parts of the code should be stepped is equally important, because the stepper often is used just for that cases where the location of an error is unknown. If in that respect the annotation-based approach does provide the same flexibility as the interpreter-based approach it comes close to be an evaluator.

Besides to these aspects, the interpreter-based approach is more comfortable and flexible because the user does not have to specify in advance what to step and where to stop. Additionally, it seems easier to provide the functionality of backup and re-evaluation.

In respect to the implementation problems discussed above, the annotation-based approach might be slightly advantageous. For source mapping the reader has to be emulated likewise. There, however, is no necessity for

emulating the loader. Source memoization is useful as support for users but the ability to find the source files automatically is not so essential as in the interpretative approach because the user has to specify anyway what is to be stepped. Especially, the memoization problems in respect to generators or closures created by FUNCTION might be at least reduced. Some minor problems like environment access or copying/collapsing of sublists, however, do not exist is the annotation-based approach.

Experiences with UnicStep

By writing our stepper we got some experience in the aspects of 1) portability, 2) efficiency and practicability of using an external Emacs editor, 3) efficiency of visually stepping in general, and 4) practicability of the visual stepping approach.

We succeded in porting our stepper to Lucid Lisp with only a small loss of functionality although we had no knowledge about the interia of this LISP system. However, we had to dig into the implementation at some points. The main disadvantage of the Lucid version of our stepper is that source file memoization only works for interpreted code.

Programming the GNU Emacs editor for the appropriate source processing was not a very difficult task. All the problems discussed above are not caused by the fact that the editor runs as a separate process. The efficiency of the communication between Emacs editor and LISP system differs according to the underlying UNIX version. The original stepper version on UNIX System V suffers from slow communication. Investigations have shown that this is due to the missing SELECT call in System V. The GNU Emacs simulates the SELECT call by sleeping for a time and then looking again for input. This results to accepting input only in intervals of one second. On a SUN with SUN/OS the total runtime is about five times faster than on the UNIMA because of this fact., whereby the speed of the machines can be compared.

The efficiency of the real visual stepping has proven to be satisfactory. However, due to the slow communication in System V, the interaction often is hampered is this environment. On the SUN in most situations there are no noticable delays in response. The only time-critical situations are:

- The invocation of a function, because the corresponding source file has to be loaded into the editor and searched for the definition.
- Evaluation without stopping until a condition is met or a specific function/macro is called. In the background the evalhook function is called on every evalution. This causes a significant portion of additional runtime for long computations. The problem can be solved for the worst case of looking for functions even in nested calls by tracing the functions at which stepping should be continued and switching stepping off in between.

The stepper has turned out to be a really useful tool for testing. In many situations the stepper has helped to find the location of an error in much shorter time than by using the conventional tools. Especially, the visual stepper was much more uselful than the normal stepper. Usually the simple backup feature suffices for providing the aspired comfort of stepping at a high level of detail. However, there are applications (like compilers working on large intermediate representations) where the restriction that side-effects can not be undone automatically is a significant limitation

Perfect solutions in source code presentation seem to be impossible or only avaible at high costs (see generators and source code memoization). With some cut-off and the application of some heuristics, however, most things still work reasonably well. For complicated situations escape mechanisms can be applied like asking the user to specify an unknown source file or switching to the internal presentation.

Future Work

The current version of UnciStep does not support programs written in CLOS (Common Lisp Object System). For stepping on the semantic level of CLOS instead of its implementation more work is necessary.

The command set of the stepper should be enhanced. Most important would be to allow the specification of breakpoints (either in the current functions or backword in a caller or forward in a potential callee) by pointing to it. Generally, within a window system environment the mouse should be used to take some control on the stepper.

The source memoization should be extended so that it is able to reflect at least some kinds of changes done to the source files at operating system level.

Very important for the acceptance of such a tool is to be integrated with other test tools like TRACE, DEBUG, and INSPECT. E.g. it should be possible to start stepping when a specific function is called by tracing that function. As another example, it might be useful to call INSPECT for more detailed investigation of objects referred to by the current function. When an error has occured and not enough information can be obtained from DEBUG it would be helpful if a function found on the stack could be re-called with stepping switched on.

It might even be possible to create an interactive tool for tightly coupled testing and editing: It is imaginable to write a source program text in the editor, examine the program with the stepper, change the text presented by the stepper directly (i.e. within the same window the stepper is working on), and re-evaluate the modified parts. If this way of program development should be applied from the beginning and the development style is top-down then return values could be specified by hand for still undefined functions.

This, however, would require that the stepper handles side-effects because it is very difficult for the user to keep track of side-effects when walking through the program. Undoing side-effects seems to be the most profitable enhancement but is evidently the most difficult problem.

The stepper could also be used as a tutorial for Common Lisp. By stepping some examples of LISP code the functionality of Common Lisp and the process of evaluation could be demonstrated. To support such an application a TRACE MODE could be installed which allows to step without interaction with the user. This automatic proceeding of stepping should run at a low speed and should always be interruptable by the user.

Literature

1. Henry Lieberman. Steps Toward Better Debugging Tools for LISP. Proceedings of the 1984 ACM Conference on LISP and Functional Porgramming, Austin, Texas, 1984
2. Glen R. Parker. Annotation-Based Program Stepping. LISP Pointers,Vol. 1, No. 4, Oct/Nov 1987
3. Henry Lieberman and Christopher Fry. Common Eval. LISP Pointers, Vol. 2, No. 1, Jul/Aug/Sep 1988
4. Guy L. Steele. Common Lisp: The language. Digital Press, Maynard, Mass.,1985
5. Thomas Gruber. XREF: A Case Study in Common Lisp Portability. LISP Pointers, Vol. 1. No. 1, 1987
6. Jon Foderaro and D. Kevin Layer. The Franz Inc Allegro CL/GNU Emacs Interface. LISP Pointers, Vol. 2, No. 1, Jul/Aug/Sep 1988
7. Ivo Haulsen. UnicStep. Diploma Thesis, 1988
8. Angela Sodan. UnicLisp - ein neues LISP-System fuer Anwendungen der Kuenstlichen Intelligenz. Internal paper of GMD FIRST, 1987
9. GNU Emacs Manual. Free Software Foundation Inc., Cambridge, Mass., 1988.