On the Usefulness of Syntax Directed Editors

Bernard Lang
INRIA
B.P. 105, 78153, Le Chesnay, France

Abstract

The intent of this position statement is to relate our experience with the actual use of a syntax directed editor (Mentor [5]) in different contexts: teaching, real software production, software maintenance and associated tasks, language design and development, development of programming tools and new programming structures.

1. Introduction

Originally the subject of this presentation was motivated by discussions to be found in the technical literature [1,2,3], and by the open questioning in less technical publications (as well as in private discussions) of the real usefulness of the syntax directed approach, its practicality and its high cost for allegedly low returns [4]. While the author does not dispute many of the points raised in the technical litterature, he believes they emphasize too much the role of syntax directed tools for program creation, and thus lead to a biased assessment of their real usefulness. This is currently resulting in the marketing of sophisticated syntax directed editors, restricted to program creation, whose cost effectiveness may indeed be questionable.

The ideas expressed here reflect only the point of view of this author, and are intended as a basis for discussion on the role of syntax directed editors. The personal judgements on various aspects of syntax directed environments are not meant as an assessment of the usefulness of research in those areas, but only as a tentative evaluation of the relative importance of these aspects for users, given the current state of the art and the author's own experience.

This experience is based almost solely on the use of the Mentor system [5]. Though comparison of several systems would be desirable, the youth of this technology makes it rather difficult to acquire extensive practical experience with more than one of them.

2. Functionality and Programmability

Our experience with syntax directed environments is based on the design, use and distribution (academic and industrial) of the Mentor system. Mentor is a syntax directed document manipulation system which we started developing at INRIA in 1974. The only previous experiment in this area had been Hansen's Emily system (ignoring Lisp environments that do not quite address the same problems). Mentor was written in Pascal and originally could manipulate only Pascal programs. In 1977 we began using Mentor for its own development. This bootstrap was an essential step in the evolution of the system, since it gave us first hand practical experience on the effectiveness of the approach for medium scale software production (about 50 000 lines of Pascal). Quickly Mentor became the main support tool for both program creation and maintenance. Indeed reference source programs were kept in tree form even on file storage (to prevent the loss of information on comments that is unavoidable when unparsing). Thus we had to use Mentor for all editing and maintenance activities, and we were barred from traditional text editors such as emacs.

The system was used in two ways:

- as an editor to enter or modify programs interactively, but in a syntax directed way,
- as a program analysis and transformation system.

This second use was possible because the command language of Mentor (for this implementation in Pascal) is a programming language called Mentol. This language has variables, control structures and recursive procedures, and it is designed with a very strong bias towards the manipulation of abstract syntax trees. Mentol was used to program a variety of program analysis, manipulation and transformation tools, that may be called interactively by the users of Mentor, or may be executed in batch mode.

The basic system provided elementary tree navigation and modification commands, quite independent from the actual semantics of the edited language (Pascal at first). The first use of Mentol is to program an environment of small functions that encapsulate some knowledge of the semantics of the edited language, and thus can better assist the programmer in his task. Examples of such tools are:

- search for the declaration of a procedure or of a variable,
- automatic declaration of labels,
- elimination of useless code,
- automatic commenting of loop or procedure ends, etc.

Heavier uses of Mentol include complex program analysis (e.g. alias analysis, software metrics) and mechanized program tranformations for optimization (e.g. recursion removal) or for porting programs on machines with different dialects. Mentor itself was ported mechanically on many very different dialects of Pascal, sometimes requiring complex syntactic or even semantic transformations (programmed in Mentol) because of the differences between these dialects. It has been developed successively on CII 10070 (Sigma7), CII IRIS-80, CII-HB DPS-8 under Multics, Vax with Unix (TM) and then VMS. It has been ported to the DEC-10, and is currently developed and maintained (by an industrial distributor) on many types of computers (VAX VMS or Unix, SUN, APOLLO, SM-90, RIDGE, HP-9000).

Actually the system is open and extensible, and new tools may be programmed in Pascal (by accessing directly the Pascal functions implementing the Mentol primitives) and called interactively as if programmed in Mentol.

There are several ways of using such a system (ignoring other features that will be discussed later).

a) simple use of elementary semantics-free editing commands: tree cut-and-paste, pattern matching and search, ... These commands act identically for syntactically similar structure such as, say, *assignment, sum* or *while* which are all tree binary operators. For such uses, it is not clear whether the little help provided by the structured view and manipulation of program warrants the conceptual and implementation costs. Also the author does not feel that the fast response of incremental tools usually integrated at this level, either syntactic (e.g. pretty-printing, syntax checking) or even semantic (e.g. type checking), brings an essential productivity improvement over an off-line use of the same tools.

b) use of pre-packaged environments of semantics dependent functions (supplied by another party) such as those described above. Such environments are in our case usually programmed in Mentol, but may also be programmed in Pascal or produced by any other means, though this is not relevant for the user. These language dependent tools have proved to be a substantial improvement over direct use of the raw syntactic system (i.e. semantics-free commands). Experience seems to show that, for a given edited language, each user makes a heavy usage of a small numbers of the available tools, but that these are not quite the same for all users. *Programmability of the system is essential* in order to select and tailor the environment to fit the standards, methods, style and abilities of teams and users.

c) use of the Mentol language itself for programming special purpose editing and maintenance tools, either for personal use or for use by other members of a project. This requires the user to be fairly expert and to fully understand the system.

It is to be stressed that easy programmability of a syntax directed editor is a substantial qualitative improvement for maintenance activities, allowing the *reliable* mechanization of repetitive and error-prone analysis and/or modifications of the maintained program. An example is the systematic transformation in the edited program of expression using a given set of primitive functions into equivalent expression using a different set of primitives. An other example, evoked above, is the transport of programs between dialects.

We feel so strongly about the importance of syntax directed editing for maintenance, that we have undertaken the definition of a non-trivial syntax for Lisp in order to be able to maintain our Lisp programs with Mentor (cf. section 4).

3. The User Interface

The user interface of the older versions of Mentor was poor, rather similar to that of a line oriented editor with a glass or paper teletype. The user had a tree cursor, and could type in commands that applied to this cursor: cursor motion, tree modification, tree visualization, i/o commands, etc. From the very beginning the standard mode of input was through a parser rather than by syntax directed menus.

In 1981 we developed a full screen interface (analogous to that of Emacs) and in 1984 we started using a bit-map and mouse interface with direct pointing in the trees. Input of new programs by syntax directed menus was introduced only at this time, more as a help to beginners than for experienced users.

The originally weak interface was imposed both by the technology of the available terminals (the system was quickly distributed and had to use standard technology) and by the low bandwith of the computer connections (1200 baud at best in most cases) preventing fast full screen refresh.

At some point in this history, glass-teletype Mentor was competing with full-screen Emacs. It appeared that neither of the two systems could really replace the other. Emacs was most convenient for typing in new programs but could not compete with Mentor for the maintenance activities described in the previous section. Problems related to the correct placement of comments in the abstract tree representation prevented switching between Emacs and Mentor via the parser and pretty-printer.

Thus Emacs was commonly used to enter in a file a new sizable (more than 5 or 10 lines) fragment of program, which was then parsed into Mentor, and was henceforth edited within it. Syntactically incorrect fragments were refused by the Mentor parser and had to be first corrected with Emacs. A major improvement was the implementation of a link between Mentor and Emacs allowing a user to call Emacs from Mentor to input a new program fragment or to edit in a non-structured way an existing fragment (i.e. a subtree) unparsed for the purpose. This was especially useful for editing expressions, long literals, and textual parts (comments). It permitted the use of Mentor for languages with important textual components, such as the language Rapport for the production of technical reports.

The menu system was rarely used (if at all) by experts. Started as an experiment (all other syntax directed editors had menus) it turned out to be a good device (together with its associated help system) to introduce novice users to Mentor, or to assist an experienced Mentor user when learning a new language.

The conclusion of our experience is that for maintenance activities the advantages of syntax directed tools outweigh the friendliness of full screen editing. This conclusion is further supported by the fact that the better man-machine interface could have been developed somewhat earlier. However, as experts users of the system, we felt a greater need for better functionality than for a nicer user interface. What forced our hand was that the system became too difficult to learn for novice users (especially without local assistance), though those who made the effort to learn it

where henceforth quite happy to use it. The better user interface was to a large extent a way to make our tool more acceptable by assisting the user, by bridging the conceptual gap of the syntax directed paradigm, and by simply looking nicer. The use of a pointing device (mouse) to move in trees is particularly effective in bringing the novice user gently to a tree-structured perception of textual objects.

We must however note that, even with a good user interface, the syntax directed paradigm is too complex and bothersome for inputting programs or for performing simple editing tasks. As might be expected, it is particularly inconvenient for editing text/program fragments that are non-structured (strings, comments) or poorly structured (expressions).

4. Language Independence

In 1981 Mentor became language independent. New languages can be specified to Mentor by means of the specification language Metal. In addition, different languages may be mixed in the same document through the commenting mechanism. Metal specifications are currently limited to syntactic aspects, while specification and generation of semantic tools (with a meta-language called Typol) is still at the prototype stage.

Language independence is essential for the adaptability of the environment to different dialects or to the evolution of a language. It is also a factor of uniformity between environments for differents languages.

Language independence allowed us to define environments for a large number of languages. But the important and unexpected effect was that a good part of our user community started using Mentor as a tool to assist the design of new languages. This is probably an important step towards the effective support of application specific languages, defined within a general purpose one, by easy generation of the corresponding environments.

However, even with a good language specification language, we believe that the definition of abstract syntaxes and finely tuned pretty-printers are non-trivial tasks that require much care and experience, especially if the language has been originally defined in terms of a concrete syntax, as is too often the case. Typically, independently of Mentor, the abstract syntax of Ada took several weeks and was revised several times.

Languages with a poor syntax (Lisp, Fortran) or with non-syntactic features (Lisp, C) such as string macros and arbitrary file inclusion fit with difficulty the syntax directed paradigm. The case of Lisp is interesting because though this language has a well defined syntax with parenthesis (ignoring the problem of macro-characters), this syntax is too trivial to be more useful than the structuring of a text as a string of characters, and it does not reflect the semantics of the language. Lisp does have a better structured syntax, but it is hidden under the parenthesis. Experiments to generate a Mentor environment for Lisp on the basis of its hidden syntax have not been too successful so far, both because there is a bad fit between concrete and abstract representation, and because Mentor cannot easily be used within an interactive Lisp environment.

We have only begun to explore the potential of mixing languages with different syntaxes, but we expect it to have impact on the organization of environment tools, and probably also on the structure of (programming) languages themselves.

5. Teaching and Learning

Mentor has been used to teach Pascal in several universities since 1977, apparently with some success. Younger people adapt very easily to syntax directed editing quite independently of their prior training. This has been confirmed by our own experience with young engineers from companies that collaborated with our research group. However, some older professionals have had difficulty fully adapting to the syntax directed approach.

Menu oriented user interfaces are of considerable help in learning the system, or learning a new language, and tend to be abandonned as the user gains experience. This has been observed consistently with students, who were however using alphanumeric terminals without a mouse.

6. Conclusion

The use of a syntax directed environment for trivial tasks is expensive. It is not clear that the advantages gained are sufficient to outweigh the cost of this technology, both in terms of the complexity of processors and computing power required, and in terms of the added difficulty for users learning more complex tools. There is however some benefit for novice users since syntax directed environments may be considered a good medium to help learn a new language and its structure. At this level, the quality of the man-machine interface plays an essential role.

For advanced users, a syntax directed environment is irreplaceable for program maintenance, especially when used to the full capacity of this technology, with the ability to use all tools and program new ones in terms of the syntax directed paradigm, i.e. on the basis of the abstract syntax of the manipulated languages. However, this implies playing the game without reservations and essentially accepting abstract trees as the standard representation of documents in place of text. Of course this may (and must) be mitigated by occasionally allowing the editing in textual form of small fragments of documents.

The author believes that the effective power and programmability of tools is more important than incrementality (which does not mean that incrementality is not worthwhile), especially when one considers the cost of incrementality within the current state of the art. In particular, high quality user interfaces are essential to lure and/or assist new users, but are probably less important than functionality for experienced people.

Finally, and unexpectedly, syntax directed environment generators are a privileged tool for language designers. In the long run, we expect that the new organizational facilities offered by these environments will foster the development of new linguistic structures in programming languages.

Acknowledgements: Though the points of view expressed here are the author's, the Mentor system and much of the experience that came with it is the result of the work of a great many people, developers and/or users of the system.

References:

[1] R.C. Waters, *Program editors should not abandon text oriented commands,* SIGPLAN Notices, vol. 17, n. 7, July 1982.

[2] U. Shani, *Should program editors not abandon text oriented commands?,* SIGPLAN Notices, vol. 18, n. 1, January 1983.

[3] R.J. Zavodnic, and M.D. Middleton, *YALE: The Design of Yet Another Language-based Editor,* SIGPLAN Notices, vol 21, n. 6, June 1986.

[4] B. Daniel-Jausine, Review of a meeting on software engineering, *O1 Hebdo,* n. 883, p. 61, December 1985.

[5] Donzeau-Gouge, V., Kahn, G., Lang, B., Mélèse, B., and Morcos, E., *Outline of a tool for document manipulation,* Proc. of IFIP '83 conf., Paris, R.E.A. Mason (ed.), North Holland, September 1983.

Space limitations prevent the inclusion of other references in this position paper. If need be, the reader will find expository papers and references in:

Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, P.B. Henderson Edit., Pittsburgh, April 1984.