tween complex human behaviors and computer programs—consistent reports of correlations coefficients exceeding 0.90 allow us to be comfortable in characterizing the results as "amazingly accurate."

Reply to T. P. Baker by A. Fitzsimmons and T. Love

Baker's confusion is resolved by the important distinction between the effort to *invoke* a function and the effort to *perform* the function. The effort required to invoke a single-input, single-output function is independent of the function. Whereas it takes more effort to compute the square root of X than the absolute value of X, the effort to write the statements

$$Y = \text{SQRT}\ (X)$$
$$X = \text{ABS}\ (X)$$

is (roughly) the same. The E-measure attempts to quantify the effort of writing a program at a given level of abstraction, not the work to perform all the operations at lower levels.

Baker's confusion about language level (λ) is resolved by remembering that λ measures the *mean* level of a particular language (across many programs written in that language); it thus provides a measure of the *average* ease (or difficulty) of programming some task in a given language. It is true that for a small subset of the programs in a given language, the language level will deviate significantly from the mean. The large standard deviations associated with λ-values in our Table 5 reveal this clearly.

We regret that Baker felt our review to be "uncritical." Criticizing a new theory is far simpler than testing it empirically. We undertook our own testing precisely because we were skeptical. Our conclusion, reached very carefully, is that this theory cannot be dismissed easily. We call on others to perform more controlled experiments and either corroborate or refute our findings. That is what science is all about.

ANN FITZSIMMONS
TOM LOVE
General Electric Co.
401 N. Washington St.
Rockville, Md. 20850

Structured Editing with a LISP

[*Editor's Note:* Richard Stallman writes to clarify a point about MACLISP, one of the LISP text editors mentioned by Erik Sandewall in "Programming in the Interactive Environment: the LISP Experience," COMPUTING SURVEYS **10,** 1 (March 1978), 35–71. Erik Sandewall replies.—PJD]

I disagree with Dr. Sandewall's conclusions regarding the relative desirability of text editing versus list-structure editing interactive systems. I want to correct a misimpression that his article may inadvertently have given: that everyone agrees that MACLISPS file-based text editing approach is inferior, and the new LISP Machine of MIT's Artificial Intelligence Laboratory will be an opportunity to switch. We at MIT actually believe that editing text in files is better for the user, and our LISP machine editor works that way. The advantages cited by Dr. Sandewall for close coupling of the editor to the rest of the LISP system are not lost by the text-based approach. In fact, the LISP machine editor, which is the best editor we know, is fully compatible with our latest PDP-10 editor, which is the best list-structure editor we know. Now that the MIT PDP-10 editor EMACS is being exported, some INTERLISP users are using it.

Our current PDP-10 system organization was just in its exploratory stages when Dr. Sandewall was here. Since that stage (as described on p. 47 of his paper), the system has changed completely. Here is how we now use it: a user has a LISP job and an editor job, which communicate. The editor operates on text files. The list structure (or compiled code) is kept in the LISP environment. When the user has a change to make, he gives LISP a command to switch to the editor. In the editor, he can ask to find and change particular functions (the editor knows which file contains each function). When he says he is done, the text files are updated on disk, and the changed functions are sent to LISP to be read in and redefined. Only the text files are kept permanently. Information passes only from the text files to the LISP job, so that the user's choice of formatting is never overriden by a LISP

pretty-printer. Reading just the changed functions is very fast.

The two real issues are whether to edit text or list structure, and whether to edit the program in the same environment in which it is tested. On the MIT LISP machine, editing is done on text, in the same LISP environment as the program is tested, but not on the list structure.

Dr. Sandewall considers also whether to save programs as text files or LISP environments. It is worth asking which option should be used primarily in a system that provides both. The article lists some reasons why text files are "necessary even on a residential system," or, from a neutral point of view, some advantages text files have over saved environments. These include robustness, house-cleaning, and ease of loading several programs together. Text files are also essential for operating on programs with tools not written in LISP, or with tools written in LISP but not part of the standard system. A saved environment has only one advantage: if it contains precisely what you want, it is faster to load. MACLISP users generally save an environment only for a tool or system which is to be loaded frequently. These environments usually contain compiled code, the uncompiled code being stored in text files.

The advantage of editing and testing the program in the same environment is the close coupling cited by Dr. Sandewall. The disadvantage is that the erroneous program being tested can alter its definition, or simply mess up the whole LISP environment. The only known remedy is to save the text on disk often.

Here are the advantages of editing text rather than list structure:

1) The user can specify any style of indentation and the system will never override it. The editor supplies standard indentation as a default.
2) Comments are easily stored and formatted as the user likes them.
3) The user can create unbalanced parentheses while editing a function. This causes no trouble as long as the function is not redefined from the text at such times. The user can also move, delete, or copy blocks of syntactically unbalanced text. In a list-structure editor, these operations are impossible or require peculiar and unintuitive commands.
4) The editor can provide commands to move over balanced objects or delete them. The commands work by parsing the expressions (forward or backward).
5) A text editor can support extended syntax. Extensibility, as Dr. Sandewall points out, is one of the strong points of LISP. In MACLISP and INTERLISP, the syntactic "macro character" escapes at parsing time to a user-supplied function, allowing arbitrary syntax extensions. For example, " " " is normally a macro character: 'FOO is equivalent to (QUOTE FOO). Extensions destroy the one-to-one relationship between internal and printed forms. With a text editor, the user automatically edits the representation he chose to type in. A structure editor cannot come close to this without being told fully about each new extension. Text editors also need to be told about syntax extensions if expression-parsing commands are to work on them, but the instructions are simple (e.g., "treat commas like single-quotes").
6) A text editor can be used for languages other than LISP (including English and alternate LISP-syntaxes) with no change. The LISP-specific commands amount to only a small fraction of the whole editor.
7) With a structure editor, temporary semantic bugs can be dangerous. In editing a function which is a vital part of the system or the editor, one cannot introduce a bug one moment and fix it the next without risking a crash. But in editing text, changes take no effect until the user gives the command.
8) The editing commands most natural for use on a display terminal are those whose meaning is obvious in terms of the displayed text. A data structure of text is natural for them, but implementing them in a structure editor would be very difficult. There are few screen-oriented structure editors.

The commands which our editors provide for LISP programs include moving over and deleting *s*-expressions, moving to the beginning or end of the current function definition, automatic indentation of new lines or old ones, automatic indentation of new or old comments, and finding quickly (without searching) the definition of named function. Further information is available in the MIT Artificial Intelligence Laboratory memo, "An Introduction to EMACS."

In closing, I note that LISP 1.6 was an improvement of an early MACLISP, but the current export version of MACLISP has superseded it. It is not true that our PDP-10 editor is a "variant of the standard DEC text-editor, TECO." In fact, DEC's editor is a variant of an early and quite primitive version of MIT's editor, TECO, which has since become a language for writing the editors.

RICHARD STALLMAN
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Mass. 02139

Author's Reply

Two current LISP systems, MACLISP and INTERLISP, represent different approaches to editing and maintenance of programs; the relative merits of these approaches has been debated intensively for a long time. In the paper, I tried to summarize the pros and cons of both approaches, although clearly there was not space to review all the arguments of each side. Mr. Stallman's letter states the case of MACLISP on this issue and is more explicit than my paper, although many of Mr. Stallman's observations had already been made in the paper. It seems that we agree on the major issues, and that any differences of opinion consist of weighing pros and cons differently. The following remarks to some of Mr. Stallman's points are therefore relatively marginal.

The description of the current system organization for MACLISP at MIT is in fact in the paper (p. 47, left column), although it is remarked that this facility "does not seem to be in widespread use yet." This was based on information given by R. Greenblatt at MIT in August 1977. Evidently the same facility is now in widespread use.

I do not believe that "advantages text files have over saved environments" represents a "neutral" view. Text files are needed in all LISP systems for certain purposes (house-cleaning, etc.), but that is in itself no reason why they should also be used as a basis for editing.

Should a program be edited in the environment where it is tested, or in another environment? I am skeptical of Mr. Stallman's argument that the program may destroy itself, or may destroy its environment so badly that saving is impossible. This would seem to be a real danger only in low-level systems work, and then should not influence the design of the whole system too much. After all, the system is built for the real users, not for the systems hackers. Also, when this danger is present, it can always be overcome in a residential environment by a simple safety measure: after you have typed in a substantial amount of text, and before you start testing, save the program in a text file.

However, a permanent problem when discussing these issues is that several different design decisions are usually intertwined and affect each consideration. In this case, the robustness of the programming system itself is significant. It is sometimes argued (at least by the INTERLISP faction) that the MACLISP system does less dynamic checking, of course in order to gain efficiency. Similarly, the UNDO facility in INTERLISP provides additional robustness and is useful when debugging systems which tend to destroy themselves. Less checking and lack of an UNDO in MACLISP may account for Mr. Stallman's different experience in this respect.

Mr. Stallman cites eight reasons for editing text rather than list structure, numbered 1 through 8. Of these, 2, 4, 6, and 8 are also in the paper, and they indeed represent advantages for the text editing approach. In particular, we agree about the intrinsic difficulty to perform screen editing (using cursor movements) in a structure-editing environment, although it has in fact been done (see reference [13] in my paper).

Allowing a variable style of indentation (1) is probably the kind of facility which is appreciated by those who have it, and not missed by those who have never had