Automated programmering—The programmer's assistant

by WARREN TEITELMAN*

Bolt, Beranek, & Newman
Cambridge, Massachusetts

INTRODUCTION

This paper describes a research effort and programming system designed to facilitate the production of programs. Unlike automated *programming*, which focuses on developing systems that *write* programs, automated *programmering* involves developing systems which automate (or at least greatly facilitate) those tasks that a programmer performs other than writing programs: e.g., repairing syntactical errors to get programs to run in the first place, generating test cases, making tentative changes, retesting, undoing changes, reconfiguring, massive edits, et al., plus repairing and recovering from mistakes made during the above. When the system in which the programmer is operating is cooperative and helpful with respect to these activities, the programmer can devote more time and energy to the task of programming itself, i.e., to conceptualizing, designing and implementing. Consequently, he can be more ambitious, and more productive.

BBN-LISP

The system we will describe here is embedded in BBN-LISP. BBN-LISP, as a programming *language*, is an implementation of LISP, a language designed for list processing and symbolic manipulation.[1] BBN-LISP as a programming *system*, is the product of, and vehicle for, a research effort supported by ARPA for improving the programmer's environment.** The term "environment" is used to suggest such elusive and subjective considerations as ease and level of interaction, forgivingness of errors, human engineering, etc.

Much of BBN-LISP was designed specifically to enable construction of the type of system described in this paper. For example, BBN-LISP includes such features as complete compatibility of compiled and interpreted code, "visible" variable bindings and control information, programmable error recovery procedures, etc. Indeed, at this point the two systems, BBN-LISP and the programmer's assistant, have become so intertwined (and interdependent), that it is difficult, and somewhat artificial, to distinguish between them. We shall not attempt to do so in this paper, preferring instead to present them as one integrated system.

BBN-LISP contains many facilities for assisting the programmer in his non-programming activities. These include a sophisticated structure editor which can either be used interactively or as a subroutine; a debugging package for inserting conditional programmed interrupts around or inside of specified procedures; a "prettyprint" facility for producing structured symbolic output; a program analysis package which produces a tree structured representation of the flow of control between procedures, as well as a concordance listing indicating for each procedure the procedures that call it, the procedures that it calls, and the variables it references, sets, and binds; etc.

Most on-line programming systems contain similar features. However, the essential difference between the BBN-LISP system and other systems is embodied in the philosophy that the user addresses the system through an (active) intermediary agent, whose task it is to collect and save information about what the user and his programs are doing, and to utilize this information to assist the user and his programs. This intermediary is called the programmer's assistant (or p.a.).

THE PROGRAMMER'S ASSISTANT

For most interactions with the BBN LISP system, the programmer's assistant is an invisible interface between the user and LISP: the user types a request, for example, specifying a function to be applied to a set of arguments. The indicated operation is then per-

* The author is currently at Xerox Palo Alto Research Center 3180 Porter Drive, Palo Alto, California 94304.

** Earlier work in this area is reported in Reference 2.

formed, and a resulting value is printed. The system is then ready for the next request. However, in addition, in BBN-LISP, each input typed by the user, and the value of the corresponding operation, are automatically stored by the p.a. on a global data structure called the *history list.*

The history list contains information associated with each of the individual "events" that have occurred in the system, where an event corresponds to an individual type-in operation. Associated with each event is the input that initiated it, the value it yielded, plus other information such as side effects, messages printed by the system or by user programs, information about any errors that may have occurred during the execution of the event, etc. As each new event occurs, the existing events on the history list are aged, with the oldest event "forgotten".*

The user can reference an event on the history list by a pattern which is used for searching the history list, e.g., FLAG:←$ refers to the last event in which the variable FLAG was changed by the user; by its relative event number, e.g. -1 refers to the most recent event, -2 the event before that, etc., or by an absolute event number. For example, the user can retrieve an event in order to REDO a test case after making some program changes. Or, having typed a request that contains a slight error, the user may elect to FIX it, rather than retyping the request in its entirety. The USE command provides a convenient way of specifying simultaneous substitutions for lexical units and/or character strings, e.g., USE X FOR Y AND + FOR *. This permits after-the-fact parameterization of previous events.

The p.a. recognizes such requests as REDO, FIX, and USE as being directed to *it*, not the LISP interpreter, and executes them directly. For example, when given a REDO command, the p.a. retrieves the indicated event, obtains the input from that event, and treats it exactly as though the user had typed it in directly. Similarly, the USE command directs the p.a. to perform the indicated substitutions and process the result exactly as though it had been typed in.

The p.a. currently recognizes about 15 different commands (and includes a facility enabling the user to define additional ones). The p.a. also enables the user to treat several events as a single unit, (e.g. REDO 47 THRU 51), and to name an event or group of events, e.g.,NAME TEST -1 AND -2. All of these capabilities allow, and in fact encourage, the user to construct complex *console* operations out of simpler ones in much the same fashion as programs are constructed, i.e., simpler operations are checked out first, and then combined and rearranged into large ones. The important point to note is that the user does *not* have to prepare in advance for possible future (re-) usage of an event. He can operate straightforwardly as in other systems, yet the information saved by the p.a. enables him to implement his "after-thoughts."

* The storage used in its representation is then reusable.

UNDOING

Perhaps the most important after-thought operation made possible by the p.a. is that of *undoing* the side-effects of a particular event or events. In most systems, if the user suspects that a disaster might result from a particular operation, e.g., an untested program running wild and chewing up a complex data structure, he would prepare for this contingency by saving the state part of or all of his environment before attempting the operation. If anything went wrong, he would then back up and start over. However, saving/dumping operations are usually expensive and time-consuming, especially compared to a short computation, and are therefore not performed that frequently. In addition, there is always the case where disaster strikes as a result of a supposedly debugged or innocuous operation. For example, suppose the user types

```
FOR X IN ELTS REMOVE PROPERTY
'MORPH FROM X
```

which removes the property MORPH from every member of the list ELTS, and then realizes that he meant to remove this property from the members of the list ELEMENTS instead, and has thus destroyed some valuable information.

Such "accidents" happen all too often in typical console sessions, and result in the user's either having to spend a great deal of effort in reconstructing the inadvertently destroyed information, or alternatively in returning to the point of his last back-up, and then repeating all useful work performed in the interim. (Instead, using the p.a., the user can recover by simply typing UNDO, and then perform the correct operation by typing USE ELEMENTS FOR ELTS.)

The existence of UNDO frees the user from worrying about such oversights. He can be relaxed and confident in his console operations, yet still work rapidly. He can even experiment with various program and data configurations, without necessarily thinking through all the implications *in advance.* One might argue that this would promote sloppy working habits. However, the same argument can be, and has been, leveled against interactive systems in general. In fact, freeing the user from such details as having to anticipate all of the consequences of an (experimental) change usually re-

sults in his being able to pay more attention to the conceptual difficulties of the problem he is trying to solve.

Another advantage of undoing as it is implemented in the programmer's assistant is that it enables events to be undone *selectively*. Thus, in the above example, if the user had performed a number of useful modifications to his programs and data structures before noticing his mistake, he would not have to return to the environment extant when he originally typed FOR X IN ELTS REMOVE PROPERTY 'MORPH FROM X, in order to UNDO that event, i.e., he could UNDO this event without UNDOing the intervening events.* This means that even if we eliminated efficiency considerations and assumed the existence of a system where saving the entire state of the user's environment required insignificant resources and was automatically performed before every event, there would still be an advantage to having an undo capability such as the one described here.

Finally, since the operation of undoing an event itself produces side effects, it too is undoable. The user can often take advantage of this fact, and employ strategies that use UNDO for desired operation reversals, not simply as a means of recovery in case of trouble. For example, suppose the user wishes to interrogate a complex data structure in each of two states while successively modifying his programs. He can interrogate the data structure, change it, interrogate it again, then undo the changes, modify his programs, and then repeat the process using successive UNDOs to flip back and forth between the two states of the data structure.

IMPLEMENTATION OF UNDO**

The UNDO capability of the programmer's assistant is implemented by making each function that is to be undoable save on the history list enough information to enable reversal of its side effects. For example, when a list node is about to be changed, it and its original contents are saved; when a variable is reset, its binding (i.e., position on the stack) and its current value are saved. For each primitive operation that involves side effects, there are two separate functions, one which always saves this information, i.e., is always undoable, and one which does not.

Although the overhead for saving undo information is small, the user may elect to make a particular operation *not* be undoable if the cumulative effect of saving the undo information seriously degrades the overall performance of a program because the operation in question is repeated so often. The user, by his choice of function, specifies which operations are undoable. In some sense, the user's choice of function acts as a declaration about frequency of use versus need for undoing. For those cases where the user does not want certain functions undoable once his program becomes operational, but does wish to be able to undo while debugging, the p.a. provides a facility called TESTMODE. When in TESTMODE, the undoable version of each function is executed, regardless of whether the user's program specifically called that version or not.

Finally, all operations involving side effects that are *typed-in* by the user are automatically made undoable by the p.a. by substituting the corresponding undoable function name(s) in the expression before execution. This procedure is feasible because operations that are typed-in rarely involve iterations or lengthy computations *directly*, nor is efficiency usually important. However, as a precaution, if an event occurs during which more than a user-specified number of pieces of undo information are saved, the p.a. interrupts the operation to ask the user if he wants to continue having undo information saved.

AUTOMATIC ERROR CORRECTION—THE DWIM FACILITY

The previous discussion has described ways in which the programmer's assistant is *explicitly* invoked by the user. The programmer's assistant is also automatically invoked by the system when certain error conditions are encountered. A surprisingly large percentage of these errors, especially those occurring in type-in, are of the type that can be corrected without any knowledge about the purpose of the program or operation in question, e.g., misspellings, certain kinds of syntax errors, etc. The p.a. attempts to correct these errors, using as a guide both the context at the time of the error, and information gathered from monitoring the user's requests. This form of *implicit* assistance provided by the programmer's assistant is called the DWIM (*D*o-*W*hat-*I*-*M*ean) capability.

For example, suppose the user defines a function for computing N factoral by typing

```
DEFIN[((FACT (N) IF N=0 THEN 1 ELSE
    NN*(FACT N-1)*].
```

When this input is executed, an error occurs because DEFIN is not the name of a function. However, DWIM

* Of course, he could UNDO all of the intervening events as well, e.g., by typing UNDO THRU ELTS.

** See Reference 1, pp. 22.39–43, for a more complete description of undoing.

* In BBN-LISP] automatically supplies enough right parentheses to match back to the last [.

notes that DEFIN is very close to DEFINE, which is a likely candidate in this context. Since the error occurred in type-in, DWIM proceeds on this assumption, types =DEFINE to inform the user of its action, makes the correction and carries out the request. Similarly if the user then types FATC (3) to test out his function, DWIM would correct FATC to FACT.

When the function FACT is called, the evaluation of NN in NN*(FACT N−1) causes an error. Here, DWIM is able to guess that NN probably means N by using the contextual information that N is the name of the argument to the function FACT in which the error occurred. Since this correction involves a user *program*, DWIM proceeds more cautiously than for corrections to user type-in: it informs the user of the correction it is about to make by typing NN(IN FACT)→N ? and then waits for approval. If the user types Y (for YES), or simply does not respond within a (user) specified time interval (for example, if the user has started the computation and left the room), DWIM makes the correction and continues the computation, exactly as though the function had originally been correct, i.e., no information is lost as a result of the error.

If the user types N (for NO), the situation is the same as when DWIM is not able to make a correction (that it is reasonably confident of). In this case, an error occurs, following which the system goes into a suspended state called a "break" from which the user can repair the problem himself and continue the computation. Note that in neither case is any information or partial results lost.

DWIM also fixes other mistakes besides misspellings, e.g., typing eight for "(" or nine for ")" (because of failure to hit the shift key). For example, if the user had defined FACT as

```
(IF N=0 THEN 1 ELSE NN*8FACT N−1),
```

DWIM would have been able to infer the correct definition.

DWIM is also used to correct other types of conditions not considered errors, but nevertheless obviously not what the user meant. For example, if the user calls the editor on a function that is not defined, rather than generating an error, the editor invokes the spelling corrector to try to find what function the user meant, giving DWIM as possible candidates a list of user defined functions. Similarly, the spelling corrector is called to correct misspelled edit commands, p.a. commands, names of files, etc. The spelling corrector can also be called by user programs.

As mentioned above, DWIM also uses information gathered by monitoring user requests. This is accomplished by having the p.a., for each user request, "notice" the functions and variables being used, and add them to appropriate spelling lists, which are then used for comparison with (potentially) misspelled units. This is how DWIM "knew" that FACT was the name of a function, and was therefore able to correct FATC to FACT.

As a result of knowing the names of user functions and variables (as well as the names of the most frequently used system functions and variables), DWIM seldom fails to correct a spelling error the user feels it should have. And, since DWIM knows about common typing errors, e.g., transpositions, doubled characters, shift mistakes, etc.,* DWIM almost never mistakenly corrects an error. However, if DWIM *did* make a mistake, the user could simply interrupt or abort the computation, UNDO the correction (all DWIM corrections are undoable), and repair the problem himself. Since an error had occurred, the user would have had to intervene anyway, so that DWIM's unsuccessful attempt at correction did not result in extra work for him.

TABLE I—Statistics on Usage

Sessions	exec inputs	edit commands	undo saves	p.a. commands	spelling corrections
1.	1422	1089	3418	87	17
2.	454	791	782	44	28
3.	360	650	680	33	28
4.	1233	3149	2430	184	64
5.	302	24	558	8	0
6.	109	55	677	6	1
7.	1371	2178	2138	95	32
8.	400	311	1441	19	57
9.	294	604	653	7	30
10.	102	44	1044	1	4
11.	378	52	1818	2	2

STATISTICS OF USE

While monitoring user requests, the programmer's assistant keeps statistics about utilization of its various capabilities. Table I contains 5 statistics from 11 different sessions, where each corresponds to several

* The spelling corrector also can be instructed as to specific user misspelling habits. For example, a fast typist is more apt to make transposition errors than a hunt-and-peck typist, so that DWIM is more conservative about transposition errors with the latter. See Reference 1, pp. 17.20–22 for complete description of spelling corrections.

TABLE II—Further Statistics

exec inputs	3445
undo saves	10394
changes undone	468
calls to editor	387
edit commands	3027
edit undo saves	1669
edit changes undone	178
p.a. commands	360
spelling corrections	74
calls to spelling corrector	1108*
# of words compared	5636**
time in spelling corrector (in seconds)	80.2
CPU time (hr:min:sec)	1:49:59
console time	21:36:48
time in editor	5:23:53

* An "error" may result in several calls to the spelling corrector, e.g., the word might be a misspelling of a break command, of a p.a. command, or of a function name, each of which entails a separate call.

** This number is the actual number of words considered as possible respellings. Note that for each call to the spelling corrector, on the average only five words were considered, although the spelling lists are typically 20 to 50 words long. This number is so low because frequently misspelled words are moved to the front of the spelling list, and because words are not considered that are "obviously" too long or too short, e.g., neither AND nor PRETTYPRINT would be considered as possible respellings of DEFIN.

individual sessions at the console, following each of which the user saved the state of his environment, and then resumed at the next console session. These sessions are from eight different users at several ARPA sites. It is important to note that with one exception (the author) the users did not know that statistics on their session would be seen by anyone, or, in most cases, that the p.a. gathered such statistics at all.

The five statistics reported here are the number of:

1. requests to executive, i.e., in LISP terms, inputs to evalquote or to a break;
2. requests to editor, i.e., number of editing commands typed in by user;
3. units of undo information saved by the p.a., e.g., changing a list node (in LISP terms, a single *rplaca* or *rplacd*) corresponds to one unit of undo information;
4. p.a. commands, e.g., REDO, USE, UNDO, etc.;
5. spelling corrections.

After these statistics were gathered, more extensive measurements were added to the p.a. These are shown for an extended session with one user (the author) in Table II below.

CONCLUSION

We see the current form of the programmer's assistant as a first step in a sequence of progressively more intelligent, and therefore more helpful, intermediary agents. By attacking the problem of representing the intent behind a user request, and incorporating such information in the p.a., we hope to enable the user to be less specific, and the p.a. to draw inferences and take more initiative.

However, even in its present relatively simplistic form, in addition to making life a lot more pleasant for users, the p.a. has had a suprising synergistic effect on user productivity that seems to be related to the *overhead that is involved when people have to switch tasks or levels*. For example, when a user types a request which contains a misspelling, having to retype it is a minor annoyance (depending, of course, on the amount of typing required and the user's typing skill). However, if the user has mentally *already performed that task*, and is thinking ahead several steps to what he wants to do next, then having to go back and retype the operation represents a disruption of his thought processes, in addition to being a clerical annoyance. The disruption is even more severe when the user must also repair the damage caused by a faulty operation (instead of being able to simply UNDO it).

The p.a. acts to minimize these distractions and diversions, and thereby, as Bobrow puts it, "... greatly facilitates construction of complex programs because it allows the user to remain thinking about his program operation at a relatively high level without having to descend into manipulation of details."[3] We feel that similar capabilities should be built into low level debugging packages such as DDT, the executive language of time sharing systems, etc., as well as other "high-level" programming languages, for they provide the user with a significant *mental mechanical advantage* in attacking problems.

REFERENCES

1 W TEITELMAN D G BOBROW A K HARTLEY D L MURPHY
BBN-LISP TENEX reference manual
BBN Report July 1971

2 W TEITELMAN
Toward a programming laboratory
Proceedings of First International Joint Conference on Artificial Intelligence
Washington May 1969

3 D G BOBROW
Requirements for advanced programming systems for list processing (to be published July 1972 CACM)