C. N. Alberga
A. L. Brown
G. B. Leeman, Jr.
M. Mikelsons
M. N. Wegman

A Program Development Tool

In this paper we describe how we have combined a number of tools (most of which are tailored to a particular programming language) into a single system to aid in the reading, writing, and running of programs. We discuss the efficacy and the structure of two such systems, one of which has been used to build several large application programs. We report some of the experience we have gained in evolving these systems. We first describe the system components which users have found most important; some of the facilities described here are new in the literature. Second, we attempt to show how these tools form a synergistic union, and we illustrate this point with a number of examples. Third, we illustrate the use of various system commands in the development of a simple program. Fourth, we discuss the implementation of the system components and indicate how some of them have been generalized.

1. Introduction

High level languages are often used to improve programmer productivity and program quality. The actual writing of programs is a small portion of the entire task. Yet the tools which support the rest of the programming effort are relatively primitive and unrelated to one another. By allowing the various tools that the programmer deals with to know about the language (and about each other), the gains in using a high level language can be amplified. For example, a programmer should never have to look at the assembly language listing to debug a program not written in assembly language. This paper discusses the design goals that led to such a system and tries to justify the authors' belief that it has productivity advantages.

A major drawback with current methodology is that the programmer is forced to deal with several differing environments to accomplish his task, as illustrated in Figs. 1 and 2. For example, the source code is frequently created and modified with the help of an editor which merely manipulates strings of text. The editor may or may not provide some formatting, which usually amounts to the indentation of lines to emphasize various programming language structures. The programmer then typically leaves the editing environment and compiles the program. If syntax errors are encountered, he must return to the editor to make the desired changes. At some point the compilation succeeds, and the program is then loaded. This step may uncover additional errors. If so, the programmer must return to the editing environment, then the command environment, to repeat the preceding steps. Finally, the program runs successfully but often gives unsatisfactory results. There may be a debugging environment, but frequently its use requires another compilation.

We attempt in this paper to demonstrate that it is both simpler and more productive for the programmer to have a single, unified interface through which to accomplish his task. With advanced formatting techniques he can envision his program in the conceptual levels of detail in which he creates it. He can uncover syntax errors immediately. He can make changes to the source code and see the effects at once. Finally, he can monitor the execution of the program in varying amounts of detail. With the goal of making programs easier to write we have investigated the combining of a number of powerful tools so that the combination would be

easy to use [1]. The tools we have integrated are sensitive to the source language of the programs being written.

The collection of tools has been so highly integrated that they may be thought of as a single tool. This tool has five major components: a display system for programs and data (much like a pretty printer [2] except that it is designed for a CRT), an editor (for modifying program text), an interpreter (which allows dynamic program analysis), a compiler, and a file system. We call such a tool a Program Development Environment (PDE).

All of the tools we discuss share a common representation of a program, the program's parse tree, as illustrated in Fig. 3. This is the first point at which we differ from tool collections like The Programmer's Workbench [3]. The parse tree embodies more information about the program than the text string for that program. Thus, many tasks are much easier for our components. Another thing that is shared by all components is the display routine. All information about a program is printed through the same interface. A common read loop (a minor tool) is also shared.

A piece of the parse tree is assumed to be the user's focus of attention, corresponding to the bold subtree in Fig. 3. The display algorithm brightens the material in this subtree, and attempts to display the subtree itself and its surrounding context in more detail than other pieces of the program. Other pieces are replaced with ellipses.

Before going into more detail, we describe one sequence of interactions to illustrate our concept of a focus of attention. When editing, a programmer can, for example, move the entire focus to a different place in the program, without counting how many lines need to be moved. When executing, the focus is the current "location counter" indication. If the program is being executed and the programmer wishes to see only the execution of some sub-part which will not be executed until much later, he can use ordinary edit commands to move the focus of attention from the current location to the location he wishes to see in more detail. He may then execute a command which means "execute until this new focus is the location counter contents." Thus, commands normally associated with the editing of a program are valuable in a quite different context. Effects like this pervade our system.

The graceful incorporation of a tool such as an interpreter necessarily introduces language dependence into the system. However, the language-sensitive parts of our system can be isolated. Our first implementation of a PDE was LISPEDIT, an environment for the LISP programmer. LISPEDIT has been in use for several years; it has been used to maintain itself and to build several substantial application programs.

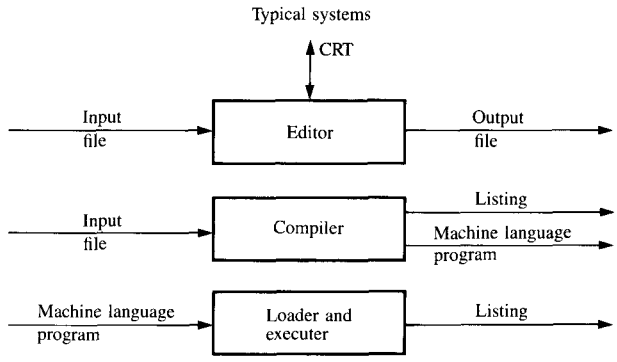

Figure 1 A typical environment provides tools like the ones above, which take an input file and produce one or more output files.

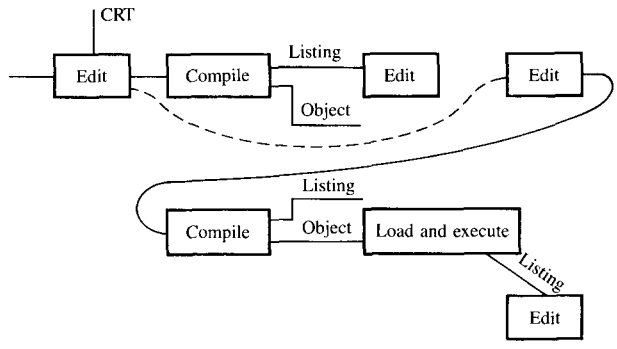

Figure 2 The programmer works by stringing together these uses of tools.

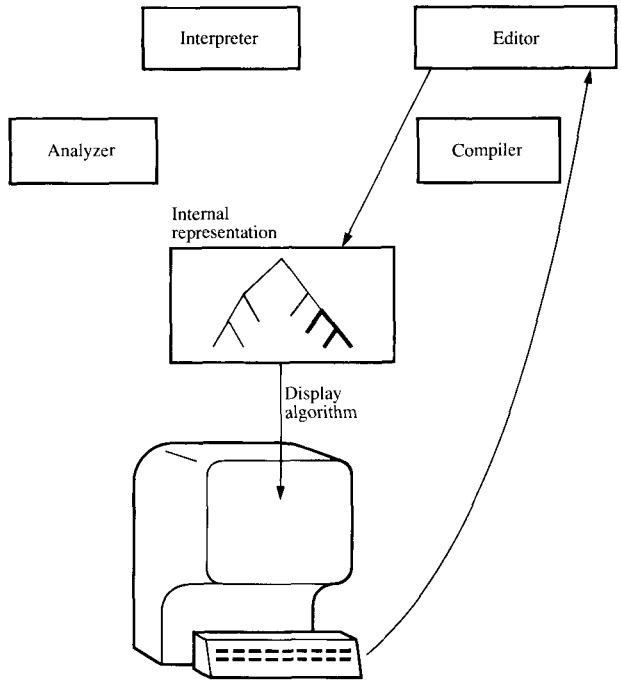

Figure 3 System configuration, showing the processing of an edit command.

In LISPEDIT, the display rules for LISP are built into the screen handler, and interactive execution is performed by a special interpreter. The second implementation is PL8EDIT, an environment for the PL.8 programmer. (PL.8 is a language derived from PL/I.) In PL8EDIT, many language dependencies are stored in tables, and during execution the editor interacts directly with special tables generated by the compiler.

By way of contrast, consider Interlisp [4], very likely the most highly developed program development environment widely available for a high level language. In Section 2.2 of [4], an example is presented of the development of a factorial program. In this scenario the programmer encounters three distinct user interfaces in the course of developing his program: the basic read-eval-print loop (for program creation), a debugger (which catches program errors), and an editor (for modifying and saving the program). Interlisp is a traditionally structured program development environment wherein the tools are independent entities able to provide different kinds of information in their individual contexts. Consequently, an Interlisp programmer is said to be *in* the break package (debugging system) or *in* the editor or *in* masterscope (analysis subsystem). We believe that having different interfaces for different tools focuses the user's attention on the tool rather than the program being developed.

To sharpen this last point we remark that the PDE user brings the tool to the program rather than the other way around. The program (actually a collection of programs) is an object about which a wide variety of information is *always* available as part of its *status*: the program text, the program status in long term (file) memory, the values of variables that are currently (in the dynamic sense) defined, the point of execution that has been reached within the program, the call stack, a variety of static flow information about the program, etc.

There are a number of other PDE's (see [5] for a list of them and also [6–11]), but none to our knowledge has 1) as high a level of integration of their components and 2) been used to build large systems. We discuss several programming tools, some of which are not new when considered in isolation; it is when their interaction is considered that they become interesting.

The remainder of this paper has five sections. Section 2 discusses the individual components from a user's point of view, demonstrating the display and some of the commands available. Section 3 shows how a simple program might be built and the way in which parts of the system interact with each other. Section 4 gives some examples of how the user might benefit from the integration of the system facilities. Section 5 discusses the implementation of the system and some of the important components. Section 6 presents some data gathered which provide insight into the relative utility of a number of commands.

2. System components

We observed in the preceding section that our system conceptually is comprised of five components. We do not discuss the compiler further, but we describe the four remaining components in detail. By overlaying this structure on the system for pedagogical reasons, we do not mean to imply that the components are not interrelated. In fact, the ability to use code from other components simplifies many tasks.

- *The display component*

One of the goals of our system is to keep the user from feeling that he is constantly changing environments while developing one program. While changing the text displayed is obviously necessary, it is important not to change the overall appearance of the display. The display format we describe allows us to display all the information a user needs, without giving that user the feeling that the environment has changed. (See Fig. 4.)

There are two critical objectives that we think a display must meet. It must show that information in which the user is interested, and it must show such information in an easy to use form. We presume a display with at least the capabilities of an IBM 3277, which can brighten fields and refresh the screen very rapidly.

We discuss two ways in which the programmer's interests may be determined. First, in most editors there is a notion of a focus of interest. In line editors this can be a particular line in the display, often brightened. In full screen editors it may be a cursor, such as an underscore or a blinking box. The focus acts as a reference point for commands, including (but hardly confined to) those which change or replace the data represented by the focus, those which insert new data in the vicinity of the focus, and those which shift the focus to other data. In short, the system assumes that the user is interested in that piece of data which he has last acted upon (via the editor's command language), and it reflects that historical fact via some form of highlighting on the screen. The PDE display routine shows the focus and parts of the data related to the focus. We attempt to determine automatically what related parts of the program the user is interested in viewing. Many other systems only show the user some fixed number of lines above and below the focus.

Second, a user frequently has occasion to direct his attention to multiple foci. For example, the user might request to be shown all instances of the variable x. The display for [2, 12] can brighten all such instances and show

their interrelationships. All the points in multiple foci are placed in an object called a *tag list*. Just as the single focus acts as a reference point for commands, tag lists can act as ways of passing information to commands. For example, one might want execution to continue until a variable was used. One could locate all instances of the variable, putting those points in a tag list, and then resume execution until a point in the tag list was reached. (We do not currently support execution to a point in a tag list.)

On the one hand, it must be possible to display a large program or a large data structure. However, the display must be small enough to be comprehensible and to fit on a CRT device. Since the syntactic structure of the program is available to the display program, by assigning priorities to the parts of the program, those parts with higher priorities can be displayed and others replaced with ellipses. If short representations of the syntactic units are available, they may be used instead of ellipses.

If we are interested in one specific syntactic structure in a particular program, say a loop, we give high priority to the subsections of that loop, the surrounding statements, and the enclosing control structure, which shows how the loop is reached in the program. If the loop is too large to show on one screen or the part of the screen allocated to the description of the loop itself, then some of the leaves of the parse tree for the loop, such as the "then" and "else" clauses of an "if . . . then . . . else" statement may be elided.

By modifying the assignment of priorities we can focus simultaneously on several subsections of a program and the syntactic structure which unites them. This facility is a natural replacement for the voluminous cross reference listings produced by compilers. The display can instantly highlight all occurrences of a variable, assignment statement, or procedure call. There are more complex uses: For example, if B must always be done after A in a program, one use of the display system would be to display all instances of A and B to ensure that the proper order has been preserved.

The display is recomputed dynamically. With every terminal interaction, the data or program being displayed is traversed and the structure again pretty printed. Thus, the programmer need not worry about the indentation of a line being entered, and if a group of expressions that is three syntactic levels of nesting down is moved to a location four levels down, the programmer need make no adjustment of margins. Because the dynamic redisplay is done automatically and correctly, errors caused by reading indentation rather than parentheses (or semicolons in other languages) do not occur. The format of the program on the screen is a reliable guide to the structure of the program.

The ability to distinguish various regions of the displayed text is useful. The example of the matching A and B actions above provides an opportunity for making such a distinction. Several kinds of display hardware assistance (generically called "highlighting") can be imagined: multiple fonts, underscoring, reverse video, and brightening. Presently we use brightening to distinguish the current focus of attention.

When a standard text editor cannot fit an entire entity (usually a "file") on a single screen, the user is normally presented with a fixed-size rectangular window, centered on the current cursor. Movement through the item being edited is accomplished by sliding this fixed window over a static field representing the item. In many cases, the data shown by such a display consist of a number of unrelated parts of the item, while parts which are closely related to the data designated by the cursor are outside the displayed area. Our dynamic display, through the use of a number of heuristics, is much more likely to display the right items. To the best of our knowledge, our algorithm is a new and general approach to the problem of displaying information.

Both Mentor [9] and Interlisp [4] can display a program (or program fragment) to a fixed, user-specified level of detail. To achieve the first display with the Cornell Synthesizer [11] would require one command for each object that was elided. In LISPEDIT and in PL8EDIT the depth of detail is automatically adjusted to produce a syntactically coherent picture of the program while attempting to make optimal use of the available display area. In our approach, the user points to the interesting parts of the program by choosing a focus; in the other systems, the user must explicitly suppress uninteresting sections of the program.

The read loop

In addition to the display component that we have already discussed, the system has a minor component called a command-read loop that supports positioning and updating commands. The read loop recognizes commands, invokes the appropriate routines, and calls the display routine to show the results. It is important that the command execution routines not call the display routine directly. Doing so would preclude one command from calling another to perform some simple function. The read loop also performs service functions, such as remembering the last ten commands. Paraphrasing traditional LISP parlance, we might characterize the command-read loop as a READ-EVAL-DISPLAY loop.

The editor component

We have borrowed several features from typical text and structural editors. There are commands, in LISPEDIT and PL8EDIT, which move the focus to parents, children, and siblings in the tree. It is also possible to locate a string and

```
PL8EDIT: SORTADS PL1   A2 Kw=U Id=L Unit=TOKEN Alt=0 Err=0 Size=219
>> sortads: PROC(ad_array, from, to);
            DCL 1 ad_array(3000),
                    2 aa_ad_# FIXED BIN, . . .
                  (swap_ad_#, swap_count, i, j) FIXED BIN,
                  up FIXED BIN,
                  (from, to) FIXED BIN;
           i=from;
           j=to;
           up=1;
           swap_ad_#=aa_ad_#(j);
           swap_count=aa_count(j);
           DO UNTIL (i>j);
                IF up THEN IF aa_ad_#(i)>swap_ad_# THEN . . . ELSE . . .
                IF up THEN i=i+1; ELSE j=j-1;
              END;
           aa_ad_#(j)=swap_ad_#;
           aa_count(j)=swap_count;
           CALL sortads(ad_array, from, j);
           CALL sortads(ad_array, i, to);
          END sortads; <<
```

(a)

```
PL8EDIT: SORTADS PL1   A2 Kw=U Id=L Unit=TOKEN   Alt=0 Err=0 Size=219
>>sortads: PROC(ad_array, from, to);
             DCL 1 ad_array(3000), 2 aa_ad_# FIXED BIN, . . . . . ;
             i=from;
             j=to;
             up=1;
             swap_ad_#=aa_ad_#(j);
             swap_count=aa_count(j);
             DO UNTIL(i>j);
                      IF up
                         THEN IF aa_ad_#(i)>swap_ad_#
                                 THEN DO; aa_ad_#(j)=aa_ad_#(i); . . .
                                          END;
                                 ELSE IF swap_ad_#>aa_ad_#(j) THEN DO; . . . END;
                      IF up THEN i=i+1; ELSE j=j-1;
              END;
             aa_ad_#(j)=swap_ad_#;
             aa_count(j)=swap_count;
             CALL sortads(ad_array, from, j);
             CALL sortads(ad_array, i, to);
            END sortads; <<
```

(b)

Figure 4 (a) Example of display of a program from top level—some details are elided. (b) View of program when user is interested in a loop and its surrounding text.

have the focus move to the first complete subexpression of that string. In PL8EDIT it is possible to point to a token on the screen and have the token become the focus. In LISP-EDIT changes to the structure must leave a valid structure, and there are commands which facilitate doing this. PL8EDIT has no such restriction. An incremental parser is

```
PL8EDIT: SORTADS PL1   A2   Kw=U Id=L Unit=TOKEN   Alt=0 Err=0 Size=219
. . . DCL 1 ad_array(3000), 2 aa_ad_# FIXED BIN, 2 aa_count FIXED BIN,
        (swap_ad_#, swap_count, i ,j) FIXED BIN,
        up FIXED BIN, . . .;
    i =from;
    j=to; . .
    DO UNTIL( i >j);
        IF up
            THEN IF aa_ad_#(i)>swap_ad_#
                    THEN DO; aa_ad_#(j)=aa_ad_#( i );
                             aa_count(j)=aa_count( i );
                             up=0;
                             END;
                    ELSE IF swap_ad_#>aa_ad_#(j)
                             THEN DO; aa_ad_# ( i )=aa_ad_#(j);
                                      aa_count( i )=aa_count(j) ; . . .
                                  END;
        IF up THEN i = i + 1;ELSEj=j - 1;
      END;
    aa_ad_#(j)=swap_ad_#; . . .
    CALL sortads(ad_array, i ,to);
                                  (c)
```

Figure 4 (c) Display of program when user wants to see all places where variable i is used.

used to allow syntactically invalid changes to be made temporarily [13]. The errors are always the newly changed material. As soon as the structure becomes valid, all the errors are removed; sometimes a few errors can be removed while others remain. It is possible to insert a *do* and have the *do* be in error. If the matching *end* is inserted, both errors go away. Now, if the *end* is deleted, the missing *end* will be the error, despite the fact that the program is textually the same as when the extra *do* was flagged as the error.

We have kept the number of primitive updating functions small. This is necessitated by the fact that changes to the program text result in the incremental modification of certain "look-aside" data structures that describe various aspects of the program being edited. Included among such structures are data that describe how the program is to be pretty printed and data that describe the findings of static program analysis. In order to keep the look-aside data structures synchronized with the program as it is developed, the parts of the system concerned with maintaining those structures must be informed of changes. By keeping the number of primitive functions small we can isolate in a few places interfaces with the look-aside updating functions. Hence, instead of providing a primitive function to change one expression somewhere in a program to another, this is done in terms of *locate* and *replace*, which are primitive.

• *The dynamic analysis package*

One of the most important things to know about a program is how it runs on examples of input data. We can run a program on one or many different instances of data and use the display routine to display results of the execution. One of the more interesting tools we have in this line is a hierarchical, single-stepping interpreter (the command *heval* stands for *h*ierarchical *eval*uator). This interpreter allows single stepping much like console debugging. However, since the language has a fair amount of structure, a step might be the evaluation of an expression or might be to start the evaluation of a subexpression of that expression. We allow the size of a step to be changed interactively. Before an expression is to be evaluated, the user has the option of simply evaluating that expression or evaluating each of the subexpressions. In either case, the value of the expression evaluated is shown.

The hierarchical evaluator is a recursive descent interpreter. It uses the fact that in LISP/370 one can create states and evaluate inside of them. It has a number of important user features, some of which are illustrated in the example in Section 3. We discuss the commands *run*, *step*, *come*, *runfast*, *trap*, *check*, and *value*. The command *trap* takes as arguments procedure names. It changes those procedures so that when they are invoked they are hierarchically evaluated (*heval*ed). When a procedure is being *heval*ed, all normal

editing commands are legal, although changes to the program may be impossible to execute directly because of macro expansion. The focus ordinarily identifies the object to be evaluated next.

If *run* is typed, the focus is evaluated, and the next object to be evaluated is displayed. The value of the object that was just evaluated is displayed in the message area. (Values put in the message area can always be edited recursively, if they are too big to fit. A partially elided form is displayed then.)

If *step* is typed, evaluation of the object is started, but is halted when the first displayable part of that object (e.g., the conditional portion of a *cond*, part of a LISP *if* statement) is about to be executed. Evaluation then proceeds to the second displayable part reached in execution. In the case of a function call, when all the arguments have been evaluated, the message line shows that that function is about to be applied to those arguments.

The *step* and *run* commands allow the user to traverse the execution tree. This is easy to grasp in a system like ours where the entire focus (to be executed by the *run* command) is highlighted. In a system like the Synthesizer [11], which denotes the current execution pointer by a single point, the user must infer the scope of the run command. The Synthesizer does not allow execution of subexpressions of arithmetic expressions. Subexpression evaluation is particularly useful to the user who has made a mistake in understanding the precedence rules of the language. But a user who does not understand the precedence rules will not be able to understand the scope of the *run* command from a display which identifies a subexpression from a single point.

The *value* command causes any LISP expression following it to be evaluated in the context of the procedure being evaluated. This allows the user to, for example, change the value of a variable by evaluating an assignment to that variable. One can also edit, recursively, the value of that variable and change its value before returning to the edit session in which the evaluation is taking place.

Typing *check* causes an expression to be evaluated each time the system displays the program. If a procedure is called which has been *trap*ped, while *heval*ing another procedure, the editor switches its attention to the *trap*ped procedure until it has finished its execution. This can, of course, happen to the new program, causing many levels of stack.

Typing *runfast* causes running and also temporarily causes procedures which had been *trap*ped merely to be executed, not *eval*ed. This allows careful examination of certain calls to some procedures, without forcing it in all cases.

The user of the command *come* positions himself at the point where the code in which he is interested starts; by his then saying *come*, execution continues with no intermediate displays until the focal expression (i.e., the expression that has just been focused on) is about to be executed. For more details, see [14].

3. An example of program creation and debugging

In this section we attempt to acquaint readers with the Program Development Environment by showing some instances in which our system is useful. We do this by tracing the building and debugging of a program. So that the reader may easily follow this process, the program built is a simple one. Unfortunately, this blunts many of the points we are trying to make. We wish to show that our system makes it easier to understand programs. But, in the interests of brevity, the program we are forced to consider may be easily understood without any of the tools we are building.

As the example progresses, the programmer changes his principal activity. First the basic program is entered. A few changes are made, and then the program is debugged. This is realistic only for small programs. In a larger program there are many iterations of this process. A programmer does not often spend large amounts of time doing primarily one kind of activity. The fact that the interface presented by LISPEDIT does not change substantially becomes important in the context of many iterations of the programming process.

In the following we first show what the user would type as a command line, followed by a summary of the information that would be displayed on the screen thereafter. The CRT we ordinarily use has twenty-two lines of text. Here, we ordinarily assume something more like six.

Command: *GH Quicksort*
Display:
()

Explanation: We want to create a quicksort program to sort elements in descending order. So, we get the value of quicksort from the file system. It is initially NIL, and that is what is displayed.

When applied to a list α with initial element y, the quicksort program Q creates two lists μ and ν whose elements are those in the tail of α which are $>$y and $\leq$y, respectively. The value $Q(\alpha)$ is then obtained by appending together the three lists $Q(\mu)$, (y), and $Q(\nu)$.

Command: *i*
Display:
()

Explanation: The display indicates that we are in input mode and are about to replace the current focus by changing one of the headings on the screen.

Command: *(lambda (sortme)*
Display:
```
(lambda
  (sortme))
```

Explanation: **(sortme)** is now the focus and anything which is input will be inserted after it. The focus is the last completed expression; balancing closing parentheses are added by the read loop.

Command: *(prog ()*
Display:
```
(lambda
  (sortme)
  (prog
    ( ) )
```

Explanation: *sortme* is the list which is passed in, and we will deal with it in a program—prog—which at the moment has no local variables.

Command: *(cond ((0or1 sortme)(return sortme)))*
Display:
```
(lambda
  (sortme)
  (prog
    ()
    (cond ((0or1 sortme) (return sortme))) ))
```

Explanation: *0or1* is a user-defined predicate which returns true if sortme is a list of 0 or 1 elements. In that case the list is already sorted.

Command: *(setq y (car sortme))*
Display:
```
(lambda
  (sortme)
  (prog
    ()
    (cond ((0or1 sortme) (return sortme)))
    (setq y (car sortme)) ))
```

Explanation: We will be comparing elements to y, which is the first element in the list. If we had a color terminal we might display a section of a program which is wrong in red (which is mnemonic for stop) as was done in [15]. Here y would be red because it is undeclared.

Command: *$declare y*
Display:
```
(lambda
  (sortme)
  (prog
    (y)
    (cond ((0or1 sortme) (return sortme)))
    (setq y (car sortme)) ))
```

Explanation: *declare* is a command. It you are in input mode, a $ preceding a command forces that command to be executed.

Command: *(for x in (cdr sortme) do (cond*
Display:
```
(lambda (sortme)
  (prog (y)
    (cond ((0or1 sortme) (return sortme)))
    (setq y (car sortme))
    (for x in (cdr sortme) do
      (cond))))
```

Explanation: *for* is a macro which executes the statement after the *do* repeatedly with x assuming all the values in (cdr sortme).

Command: *((greaterp x y) (setq bucket1 (cons x bucket1)*
Display:
```
(lambda (sortme)
  (prog (y)
    ...
    (for x in (cdr sortme) do
      (cond (  (greaterp x y)
               (setq bucket1 (cons x bucket1)) )))))
```

Explanation: Note the ellipsis. Remember that we are assuming a six-line screen and must elide something. The setq to y and the test for empty or null list are probably the best choices. Since bucket1 is undeclared, it is shown in red. This does not require us to do anything before we proceed; however, unless we wish to use it as a free variable (which may be declared to eliminate the redness), it is a mistake and must be fixed eventually. We now pretend that the current focus is a large expression and that we wish to avoid retyping it.

Command: *$copy*
Display: the same as above.

Explanation: *copy*, which is executed as a command, stores a copy of the focus so that it may be inserted elsewhere.

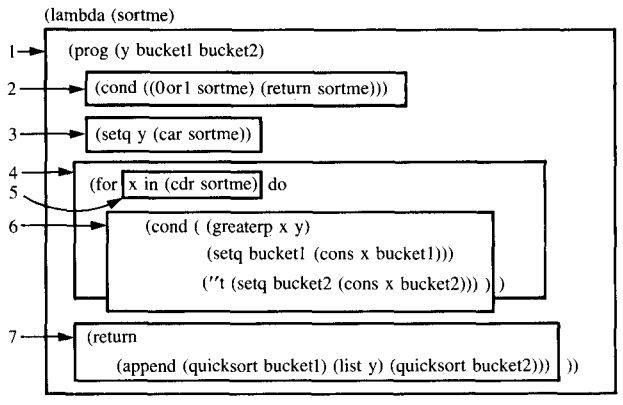

Figure 5 Final program.

Command: *)(" t*
Display:

```
(prog (y)
  ...
  (for x in (cdr sortme) do
    (cond (  (greaterp x y)
             (setq bucket1 (cons x bucket1)))
          ("t)))))
```

Command: *$to*
Display:

```
(prog (y)
  ...
  (for x in (cdr sortme) do
    (cond (  (greaterp x y)
             (setq bucket1 (cons x bucket1)))
          (  "t (setq bucket1 (cons x bucket1))  )))))
```

Explanation: *to* inserts the value that was stored by *copy*. The command *to* may be reused later to insert the same thing.

Command: *cng bucket1 bucket2 **
Display:

```
(prog (y)
  ...
  (for x in (cdr sortme) do
    (cond (  (greaterp x y)
             (setq bucket1 (cons x bucket1)))
          (  "t (setq bucket2 (cons x bucket2))  )))))
```

Explanation: The *change* command works in the scope of the current focus. The * that is the third argument to *cng* causes bucket1 to be changed in all those instances.

We eventually finish the program, and the final version appears in Fig. 5. We have drawn in boxes which indicate various values of the focus in what follows.

We wish to run this program. We can do so simply by typing the LISP expression which invokes the sort routine. Alternatively, we can run it, examining the program as it is being executed. In order to observe the running program, we type in the command *heval*, invoking the hierarchical interpreter. The *heval* command allows the user to step through parts of his program and display intermediate results on the screen.

Command: *heval "(1 2 3 4)*
Display: the (elided) program with Box 1 as the focus.

Explanation: This command starts the evaluation of the quicksort program with sortme bound to the list (1 2 3 4).

Command: *step*
Display: Box 2 is the focus.

Explanation: *step* causes the evaluation of each of the subexpressions of the focus to be shown.

Command: *run*
Display: Box 3 is the focus.

Explanation: *run* causes the evaluation of the focus, without showing the evaluation of the subparts.

Command: *run*
Display: Box 4 is the focus, LAST VALUE = 1

Explanation: The value of y is now 1, and this is the value of the last focus displayed and evaluated.

Command: *display data bucket1 bucket2*
Display: Box 4 is the focus on the top half of the screen, bucket1 = (), and bucket2 = () on the bottom of the screen.

Explanation: This allows us to observe the value of certain variables as execution continues. The implementation sets up another edit session on the bottom of the screen. Thus, if the data are too large we can use edit commands to look at specific portions of the data.

Command: *step*
Display: Box 5 is the focus, bucket1 = (), bucket2 = ()

Command: *run*
Display: Box 6 is the focus, LAST VALUE = 2, bucket1 = (), bucket2 = ()

Explanation: The value of x is now 2, and this is the value of the last focus displayed and evaluated.

Command: *run*
Display: Box 5 is the focus, LAST VALUE = (2), bucket1 = (2), bucket2 = ()

We may become tired of running around in the loop. Using edit commands we position the focus at Box 7.

Command: *come*
Display: **Box 7 is the focus, LAST VALUE = 3, bucket1 = (4 3 2), bucket 2 = ()**

Explanation: The program has been executed until Box 7 is about to be executed. The value of the last focus displayed was 3; x on that iteration of the loop was 3. **Bucket2** has become a larger list in subsequent executions of the loop.

Command: *run*
Display: **top level display of quicksort, value = (4 3 2 1)**

Explanation: We have finished the execution of the quicksort, and it has returned the right value. We have now exited *heval*.

In LISP/370 all variables bound in a *prog* are initialized to nil. Were they not, the first use of **bucket1** and **bucket2** would have been highlighted with red, because they were not initialized. If we were confused about the function which **bucket1** was performing, we would have all instances of **bucket1** highlighted, with other statements being elided. We could do this highlighting to a procedure call which used or stored into **bucket1** even if the procedure call did not have **bucket1** mentioned in its argument, or only used it by calling another routine which used it.

This concludes our example. It is unfortunate that for obvious pedagogic reasons we cannot illustrate the utility of our system in developing large programs. However, we point out that LISPEDIT has been used in its own development, as well as other unrelated large programs.

4. Examples of integration of facilities

In this section we describe some of the ways users have utilized the fact that all of the tools we provide interact well with one another.

One example is the way the command *come* is used. Execution is begun in *heval*. At some point the user wishes to skip seeing at a detailed level anything but the execution of a certain piece of code. Using ordinary positioning commands the focus is moved to that section, the command *come* is executed, and execution continues until that point is reached, after which the user sees the execution in detail again. This example points out the need to integrate the positioning and execute commands.

Another two examples are illustrated by the command *e =*. When an expression is typed by the user, it is evaluated and the value displayed on one line in the message area. That line starts with **value =**. If the value does not fit on one line, parts of the value are elided (using the same elision strategy as in the display). The value is also put in the global variable =. That value may be examined by editing that global variable (just as any other variable may be edited). While editing a global variable, all of the positioning commands which work while editing programs still work. This saves a lot of relearning. Commands also put values in the global variable =. If the command *run* is given, in *heval*, the brightened expression is evaluated, and its value is placed in =. The focus moves to the next location. The message area holds the message **Last value =** and one line of the value. Thus, it must be possible to integrate the ability to edit objects and data in the same editor. It is also important to allow the evaluation of expressions inside of the editor.

The command *trap* is also useful. This flags a certain procedure to be hierarchically evaluated. It is important that it work both when the user is editing and when evaluating a potentially calling procedure. Since only the source may be hierarchically evaluated, there is a problem when *trap* is applied to a compiled routine. If the compiled routine stored the location in the file system where the source would be found, the problem could be alleviated. We do not currently do this. Thus, it is important to integrate the file system, which keeps track of where the source for a program is, with the debugging system.

When we first implemented *trap*, we made a mistake and copied the object before it was evaluated. Users, knowing that the system was highly integrated, would find an error while stepping through the program, and they would fix the error. They would then stop the monitoring of the program; since the original program was restored, so was their bug. If compiled code is to be accessible to the *trap* command, updating must be stopped and the formerly compiled program run interpretively, or the program recompiled. The latter is better. This would be particularly true in a system for a different language, where when the change made was to a declaration, that change might cause other programs to be recompiled. It is possible that the best situation would utilize an incremental compiler, which would help the user to avoid the decision of whether to compile or interpret his code.

Often one wants to be able to edit one of a collection of functions without affecting the others. When one does this the date stamp should not change for the others, even if the other routines need to be recompiled. Occasionally, however, one wants to edit the whole collection (for example, to find all instances of a free variable). Thus, it is desirable to integrate the file system with the editing system.

5. Implementation and other system facilities

In this section we describe some of the facilities which we have found useful in implementing the system. These facili-

ties are also available to the user as part of the system. We also go into somewhat more detail on the algorithms used and the precise features offered to the user.

• *The pretty print/display algorithm*

There are two major reasons why conventional pretty printing algorithms fail in our display situation. 1) It is necessary to examine the ancestor(s) of a node in order to know how to pretty print that node. For example, a number might be printed differently if it were a label or a quantity to be added; in the former case it would be placed at the beginning of a line, in the latter it might be in the middle. Standard pretty print algorithms usually start at the root node. But we have the notion of a focus or foci of interest, which, if sufficiently localized, can cause the loss of surrounding portions of the display, including the root itself. So we cannot start printing at the root; we can start some analysis at the root. 2) Standard algorithms assume an infinitely long sheet of paper, rather than our limited CRT display, and so need not squeeze as much on a line; we must elide sections intelligently.

In our system the pretty print algorithm and the display algorithm are two separate but interacting components. The pretty print algorithm converts the structure of the parse tree into a structure of nested *boxes* that reflects the esthetic intentions inherent in pretty printing. The display algorithm translates these boxes into a two-dimensional image that fits on the output device. We now consider the box which would be produced by the pretty print algorithm for a *prog*. There is a particular type of box, called a *vertical box*, which should if possible be displayed with all of its component boxes lined up vertically. All the statements in the *prog* are grouped in a vertical box. Other boxes are used to allow enough indentation of this vertical box to enable labels to be shown to the left of the labeled statements. The bound variables of the *prog* are shown in a box adjacent to *prog*. The purpose of the box structure is to map the parse tree into a structure that indicates the preferred two-dimensional layout of program text, to specify the relative priorities (or importance) of phrase components, and to define a minimal elided form for each statement.

The display algorithm carries out the instructions in the box structure under the constraint of one or more foci of attention and within the limitations of the available display area. In the single focus case, display generation begins at the box that defines the current focus. The first step in display generation is to initialize a priority queue with the current focus. The next step is to attempt to show the minimal elided form of the first box in the queue. If the attempt fails, that particular box remains elided and we go to the next box in the priority queue. If the attempt succeeds, we add to the priority queue any immediate neighbors of the current box that have not yet been visited. The above steps are then repeated for the next entry in the queue. The process terminates when the display is full or the queue is empty.

The PL8EDIT display algorithm assigns decreasing priorities to new entries in the queue by dividing the current priority by a constant that depends on the relative position of the new entry. The constant for a box that contains the current box is slightly larger than the constant for a component box. The constant for adjacent boxes is close to one. This scheme produces a display that favors expansion of the focus over expansion of the context surrounding the focus and preserves detail in statements close to the focus.

If several foci are present, PL8EDIT places all the foci on the priority queue during initialization and biases the priority allocation scheme to favor branches in the structure that connect foci. The result of this strategy is to generate a display that shows the relative positions of the foci in the text of the program, with connecting text shown mostly in elided form.

• *Incremental update*

If all the data structures associated with a program were to be re-created each time the user made a change to the program, the PDE would be unusable. In the two critical areas, parsing and display generation, we have devised efficient algorithms that update the relevant data structures incrementally. Consequently, the computing cost of an update operation is normally not proportional to the size of the program.

The programs that decide on the display format must traverse a program top-down in order to make decisions in the appropriate context. When changes are made to the program in PL8EDIT, we mark the data structures near the changes as invalid or "spoiled," and then propagate that information to the root. The box-generating programs then begin at the root, as if trying to re-create the entire structure. Any substructure that has not been spoiled can be re-used without further descent and processing, and only the modified parts of the program need be examined.

6. Statistics

Much of what we have learned in the process of implementing this system is intuitive and cannot easily be communicated to someone who has not used the system. In this section we try to convey some of this information by describing some of the usage statistics that we have gathered.

We would also like to enter a plea with all other builders of similar systems to gather similar statistics so that comparisons might be made. We have found the statistics useful in

our development work. They 1) show us what components are slow and often used, 2) tell us which portions are most important to improve since they are heavily used, 3) help us find simple sets of commands to show new users, and 4) tell us when an experienced user is unaware of a command as evidenced by the fact that he does not use it.

The statistics were gathered (when the system was not in a period of change, for example, when we switched from nonshared to shared pages for code) over a period of nine months. Three to four users were the primary participants in this experiment, and 91 402 interactions with the host computer were recorded.

A heavy user seems to have about 600 interactions with the host computer a day. These interactions consist of about ten characters. When in input mode, the number of characters per typed line approximately doubles. The system maintains seven predefined control keys; these accounted for about one quarter of the interactions (these single key commands were not counted in the ten-character figure above). One interesting number, which our data are not presently precise enough to determine, is what percentage of a user's time he is being held back by his typing speed. This would suggest how much attention should be paid to having commands which allowed abbreviated typing. For example, it has been suggested that one can type a program more rapidly in Teitelbaum's system [11] because with a command like ".DW" one can insert "do while end;".

We now describe our grouping of commands (see Table 1). In each group we have gathered similar commands, and we briefly describe the most important members of the group.

Group 1 contains the selection commands that move the focus through the program tree. *Son* x y z changes the current focus to be the zth son of the yth son of the xth son of the current focus. *Next* positions the focus on the largest s-expression to the right of the current focus in the entire expression being edited. *Right* finds the sibling to the right. A number command positions to the right, and if it cannot go further to the right, goes down. Some of these commands take as arguments either a number or *, and * goes as far as possible.

We have several searching commands that select the next focus on the basis of a pattern or predicate. These are grouped separately from the tree motion commands, since their effect is dependent on the atoms of the tree, rather than its shape.

Group 2 consists of commands that modify the text of the program. The most commonly used update commands are *replace*, *change*, *delete*, and *insert*.

Table 1 Usage patterns.

1. Motion commands		31%
1.1 *Son, Next, Up, Left, Right,* and *Number*	26%	
1.2 Locate commands	5%	
2. Update commands		20%
2.1 *Replace, Change, Delete,* and *Insert*	11%	
2.2 Input mode	4%	
2.3 *Declare, Fof*	3%	
2.4 *Copy, Move, To*	2%	
3. Debugging commands		20%
3.1 *Run, Step, Come,* and *Value*	20%	
4. Miscellaneous commands		29%
4.1 Expression evaluation	8%	
4.2 File system commands	7%	
4.3 Editing objects	4%	
4.4 Review commands	3%	
4.5 User defined commands	2%	
4.6 Other	5%	

Input mode was described in the programming example in Section 3. It is the mode commonly used to create new programs.

There are several language-dependent commands, such as *declare*, which is described in the example, and *fof*, which takes a list of arguments and replaces the focus by a list which is the list of arguments with the current focus appended. The command *fof* is very useful in an expression language.

The commands *copy*, *move*, *to* were described in the example in Section 3.

Group 3 contains *run*, *step*, *come*, and *value*. These commands control the execution of the hierarchical evaluator.

Group 4 contains all the other interactions, including the evaluation of LISP expressions.

File system commands include reading in functions so they may be interpreted or edited, as well as loading compiled modules and compiling source code.

There are commands which enable the user to edit different objects and to change edit sessions.

There are various commands to aid reviewing previous commands. The most common of these allows the user to modify the previous command and re-execute it.

User-defined commands include continuations of locate commands.

There remain several miscellaneous commands. The most common of these is the command to pass a command to the underlying operating system.

Since our editor does not accept full screen input, the only way to do updating is by moving the focus and making an update command. We were pleased to find that our average of 1.55 motion commands for each update command was that low. We belive that the ratio would be considerably higher in a text editor. Because many motion commands may be regarded as only a means to an end, a low number indicates that relatively few superfluous commands need be executed. The reasons for such a low number are probably twofold: 1) Our formatting algorithm accurately displays what the user wants to see. Thus, few motion commands are necessary in order to read the program; 2) it is relatively easy for the user to get to where he wants to make a change.

Note that 23% of the commands (groups 2.3 and 3) would be impossible in a less integrated system; this fact provides convincing evidence of the value of combining such tools into one system.

7. Conclusions

It is our belief that as better hardware becomes available, systems like ours will become easier to build. For example, the MENTOR system [9] had to be teletype-compatible and as a result does not interact as well as it might with a CRT. Another example comes from limited address space. In the absence of a large address space it becomes natural to have a tool which is invoked and brings in only code thought to be relevant to the kinds of interactions encountered when using that tool. With a large address space, these problems can be handled by hardware paging rather than software overlaying. The Cornell Program Synthesizer [11] has done a wonderful job of providing a unified environment, much like ours, in 56K bytes. Nevertheless, there is a noticeable delay when invoking a command involved with either execution or editing, when the last command was in the other set of commands. As a result of the larger address spaces and faster machines, we believe that systems like ours will become increasingly common and popular.

We have tried to demonstrate the advantage to a programmer of having a unified set of programming tools. We believe that the system is superior to a typical combination including a text editor, compiler, operating system, etc. The capabilities of the commands and the display component have enabled users to create and modify programs easily. Furthermore, the testing aids such as *heval* have speeded and simplified the debugging process. It is clear that the notion of establishing a single environment for program development via such a system is a significant step toward increasing programmer productivity. Finally, we believe that these techniques are language independent and will apply to most modern structured languages [11, 16, 17]. We hope to be able to create a set of tools which can be given a table and then work for the language described by that table. As was the case with compiler-compilers, this will probably not work perfectly at first and will require a small amount of special casing for each particular language.

Acknowledgments

We would like to thank Vincent Kruskal and Paul Kosinski, who participated in the initial discussions which became the basis for LISPEDIT. We would also like to thank our user community, which consists primarily of C. Leonard Berman, J. Lawrence Carter, and Alan Cobham, for a number of useful suggestions, several of which have been incorporated into system commands.

References and note

1. An earlier version of this paper appeared in the *Conference Record* of the Eighth Annual ACM Symposium on the Principles of Programming Languages.
2. M. Mikelsons, "Prettyprinting in an Interactive Environment," *Proceedings, ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, OR, June 1981, pp. 108–116.
3. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," *Bell Syst. Tech. J.* **57,** No. 6, Part 2, (July–August 1978).
4. W. Teitelman, *Interlisp Reference Manual*, Xerox Corp., Xerox Palo Alto Research Center, CA, December 1978.
5. A. Lederman, "An Abstracted Bibliography on Programming Environments," personal communication, June 1980.
6. B. Austermuhl, W. Henhapl, H. Kron, and R. Lutze, "On a Programming Environment and its Generation," *Pu1R2/79*, University at Darmstadt, W. Germany.
7. M. Brown and S. Wood, "A Display-Oriented Program Editor," *Technical Report*, Yale University, New Haven, CT.
8. T. Cheatham, J. Townley, and G. Holloway, "A System for Program Refinement," *TR-05-79*, Aiken Computation Laboratory, Harvard University, Cambridge, MA, August 1979.
9. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A Structure Oriented Program Editor: A First Step Toward Computer Assisted Programming," *Proceedings, International Computing Symposium,* North-Holland Publishing Company, Amsterdam, 1975, pp. 113–120.
10. J. R. Reiser, "A Debugger for Sail," *Memo AIM-270*, Stanford Artificial Intelligence Laboratory, Stanford University, Palo Alto, CA, October 1975.
11. R. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax Directed Programming Environment," *Commun. ACM* **24,** 563–573 (1981).
12. M. Mikelsons and W. Wegman, "PDE1L: the PL1L Program Development Environment—Principles of Operation," *Research Report RC-8513*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, November 1980.
13. M. Wegman and C. N. Alberga, "Parsing for a Structured Editor (Part II)," *Research Report RC-9197*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1982.
14. M. Mikelsons, "Interactive Program Execution in LISPEDIT," presented at the ACM SIGPLAN/SIGSOFT Symposium on High Level Debugging, Asilomar, CA, March 1982.
15. V. Kruskal, "An Editor for Parametric Programs," *Research Report RC-6070*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, June 1976.
16. M. N. Wegman, "Parsing for a Structural Editor," *Proceedings, 21st Symposium on the Foundations of Computer Science*, October 1980, pp. 320–327.
17. C. Ghezzi and D. Mandrioli, "Incremental Parsing," *ACM Programming Lang. Syst.* **1,** 58–70 (July 1979).

Received August 5, 1983; revised September 20, 1983

Cyril N. Alberga *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Alberga joined IBM's Advanced Systems Development Division in 1960 in the American Airlines Sabre project. He came to the Research Division in 1963 in the Computer Aided Instruction project. He worked on the control program and the compiler for the original coursewriter system. Mr. Alberga has worked on a number of projects, including FS, the Business Definition System, LISP370, and program development environments. He is currently involved in LISP system and compiler development. Mr. Alberga received a B.S. in mathematics from the University of Wisconsin in 1960 and an M.S.E. in computer and information sciences from the University of Pennsylvania in 1965, under an IBM Resident Study Program. His interests include on-line dictionaries and spelling correction.

Allen L. Brown, Jr. *Xerox Corporation, 3450 Hillview Avenue, Palo Alto, California 94304.* Dr. Brown was at the IBM Research Center from 1975 until 1979, during which time his principal contributions were in the areas of knowledge-based systems and program development environments. Since 1979, he has been a member of the Systems Development Department of Xerox's Office Systems Division, where he has done work in programming languages, database systems, performance analysis, and digital image processing. Most recently he has been involved in the design and development of an integrated electronic publishing system. Dr. Brown was graduated from the Massachusetts Institute of Technology in 1967 with a B.S. in mathematics and chemical engineering and in 1975 with a Ph.D. in artificial intelligence.

George B. Leeman, Jr. *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Leeman joined IBM Research in 1972 as part of the microprogram certification project. His current interests are in the areas of office applications research and the theory of univalent functions. He received the B.A. magna cum laude from Yale University, New Haven, Connecticut, in 1968 and the M.S. and Ph.D. from the University of Michigan, Ann Arbor, in 1969 and 1972; all three degrees were in pure mathematics. Dr. Leeman has held a number of different positions at IBM Research, including technical assistant to the Director of Research from 1976 to 1977. He is currently manager of the advanced office prototypes project.

Martin Mikelsons *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Mikelsons has been a Research staff member at the Thomas J. Watson Research Center since 1966. His activities there have ranged from real-time systems and laboratory automation to artificial intelligence. He is presently working on syntax-directed display and editing of programs. He graduated from Princeton University with an A.B. in mathematics in 1964. He received an M.S. in systems and information science from Syracuse University, New York, in 1972.

Mark N. Wegman *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Wegman has worked in IBM since 1975 on programming environments and algorithms. The algorithmic work includes algorithms for hashing, data compression, unification, data flow analysis, cryptography, and incremental parsing. He has managed the Program Development Environment project since 1980. Dr. Wegman has a B.A. in mathematics and philosophy from New York University, received in 1971, and a Ph.D. in computer science from the University of California at Berkeley, received in 1980.