Received January 1978; revised October 1979; accepted December 1979

References

1. Avery, K.R., and Avery, C.A. Design and development of an interactive statistical system (SIPS). Proc. Comptr. Sci. and Statistics: 8th Ann. Symp. on the Interface, Health Sciences Comptg. Facility, UCLA, Los Angeles, Calif., 1975, pp. 49–55.
2. Box, G.E.P. Science and statistics. *J. of the Amer. Statistical Assn. 71* (Dec. 1976), 791–799.
3. Brode, J. Generalizing the function call to statistical routines—An application from the DATATRAN language. Proc. Comptr. Sci. and Statistics: 10th Ann. Symp. on the Interface, Nat. Bureau of Standards, Gaithersburg, Md., 1977, pp. 357–361.
4. Brode, J., Stamen, J., and Wallace, R. The DATATRAN language. Proc. Amer. Statistical Assn., Statistical Comptg. Section, 1976, pp. 126–129.
5. Buchness, R., and Engleman, L. MiniBMD: A minicomputer statistical system. Proc. Comptr. Sci. and Statistics: 10th Ann. Symp. on the Interface, Nat. Bureau of Standards, Gaithersburg, Md., 1977, pp. 9–13.
6. Fox, D.J. Some considerations in designing an interactive data analysis system. Proc. Comptr. Sci. and Statistics: 8th Ann. Symp. on the Interface, Health Sciences Comptg. Facility, UCLA, Los Angeles, Calif., 1975, pp. 61–65.
7. Fox, D.J., and Guire, K.E. *Documentation for MIDAS.* Revised 2nd edition, Statistical Res. Lab., Univ. of Michigan, Ann Arbor, Mich., Aug. 1974.
8. Francis, I., Heiberger, R.M., and Velleman, P. Criteria in the evaluation of statistical program packages. *Amer. Statistician 29* (Feb. 1975), 52–55.
9. Guthrie, D., Avery, C., and Avery, K. Statistical Interactive Programming System (SIPS), User's Reference Manual. Oregon State Univ., Corvallis, Oregon, 1974.
10. Isaacs, G.L. Interdialect translatability of the BASIC programming language. ACT Tech. Bull. No. 11, The Amer. College Testing Program, Iowa City, Iowa, 1972.
11. Kennedy, T.C.S. The design of interactive procedures for man-machine communication. *Internat. J. Man-Machine Studies 6* (1974), 309–334.
12. Klensin, J.C., and Dawson, R. CS: The consistent system. In *A Comparative Review of Statistical Software,* Ivor Francis, Ed., The Internat. Assn. for Statistical Comptg., The Netherlands, 1979, pp. 151–161.
13. Ling, R.F., and Roberts, H.V. User's Manual for IDA. The Scientific Press, Palo Alto, Calif., 1980.
14. Ling, R.F. Constraints in the design and implementation of interactive statistical systems for minicomputers. Proc. Comptr. Sci. and Statistics: 10th Ann. Symp. on the Interface, Nat. Bureau of Standards, Gaithersburg, Md., 1977, pp. 26–34.
15. Ling, R.F., and Roberts, H.V. IDA and user interface. Proc. Comptr. Sci. and Statistics: 8th Ann. Symp. on the Interface, Health Sciences Comptg. Facility, UCLA, Los Angeles, Calif., 1975, pp. 91–94.
16. Plattsmier, R.A. Criteria for evaluation of interactive statistical programs and packages. Proc. Comptr. Sci. and Statistics: 10th Ann. Symp. on the Interface, Nat. Bureau of Standards, Gaithersburg, Md., 1977, pp. 384–388.
17. Ryan, T.A., Joiner, B.L., and Ryan, B.F. *Minitab Handbook.* Duxbury Press, North Scituate, Mass., 1976.
18. Ryan, T.A., Joiner, B.L., and Ryan, B.F. Minitab II. In *A Comparative Review of Statistical Software,* Ivor Francis, Ed., The Internat. Assn. for Statistical Comptg., The Netherlands, 1979, 185–196.
19. SPEAKEASY-3 Reference Manual Level Lambda IBM OS/VS Version. Compiled by S. Cohen and S.C. Pieper, Argonne Nat. Lab., Argonne, Ill., 1976.
20. Sterling, T. D. Guidelines for humanizing computer information systems: A report from Stanley House. *Comm. ACM 17,* 11 (Nov. 1974), 609–613.
21. Velleman, P., and Welsch, R.E. Some evaluation criteria for interactive statistical program packages. Proc. Amer. Statistical Assn., Statistical Comptg. Section, 1975, pp. 10–12.
22. Whitten, D.E., and deMaine, P.A.D. A machine and configuration independent Fortran: Portable Fortran (PFortran). *Trans. on Software Eng. SE-1* (March 1975), 111–124.

Artificial Intelligence/ Language Processing

C.A. Montgomery Editor

A Generalized Text Editor

Christopher W. Fraser
The University of Arizona

Text is not the only data that needs editing; for example, file deletion utilities edit directories. If all "editors" used the same command language, they would be easier to learn, remember, and code. This paper describes a generalized editor that edits text, directories, binary core images, and certain operating system data with a single user interface.

Key Words and Phrases: editor, text, command language, CRT

CR Categories: 3.7, 4.3

1. Introduction

Editing means examining and modifying data. Though most editing programs edit text, other types of data need editing too: Utilities that delete and rename files edit directories and interactive debuggers edit binary core images. Typically, each utility has its own command language and command scanner. However, each of these utilities is just an "editor" and, with careful design, might share the system text editor's command language and scanner. Duplicating command scanners wastes programming effort. Duplicating command languages frustrates users, especially naive users: More than one typist has abandoned computing over poor user interfaces. As computers proliferate, more naive users must be accommodated; hence the growth of interest in improving command languages [3, 10].

Author's present address: C.W. Fraser, Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

This paper describes the generalized text editor *s*. *s* offers commands like those of a conventional text editor, but, by changing a few lines of code, it can edit nontextual data with the same user interface. Instances of *s* edit text, directories, binary data, and the internal state of a computer. They affect nontext like a conventional editor affects text. Where a conventional editor presents text from a file, *s* presents the textual representation of nontext (utilities routinely present as text such nontext as directories and core images). Where a conventional editor changes a file to reflect changes requested by the user, *s* changes the underlying nontext to reflect similar user changes to its textual representation. As such, *s* offers a "canned" user interface for editor-like utilities. It has been retrofitted to existing utilities, it has simplified the implementation of new ones, and it promotes local standardization of user interfaces.

Section 2 shows how *s* is used to edit text, file system directories, machine states, and binary data. Section 3 describes its implementation.

2. Example Editors

2.1 A Text Editor

s is a generalized CRT-based editor based on the Yale editor *e* [4]. It is best introduced by considering the instance of *s* that edits text. It displays a two-dimensional window on a text file and allows the user to change it by moving a cursor about the screen with the terminal's cursor control keys and typing new characters over the old ones. It changes the data being edited to reflect the user's changes to the screen: The screen always displays the current version of the edited data. Special function keys—dedicated keys on some terminals, control characters on others—invoke special functions. For example, *+pages* displays the next page (screen) of the file. All special functions accept an optional argument:

escape 2 *+pages*

advances the screen's window on the file by two pages. *−pages* moves back; *+lines* and *−lines* move by so many lines, not pages; *+search* and *−search* move to the next occurrence of text matching their argument, a pattern given as a regular expression. All special functions remember and reuse their arguments:

escape abc *+search +search*

finds the first and then the second occurrence of "abc". *delete* deletes the line on which cursor resides (from the cursor on); if its argument is a number n, it deletes n lines; if its argument is a cursor movement (i.e., only cursor control keys were struck between *escape* and *delete*), it deletes the characters and lines between the initial and final cursor positions and so may be used to delete part of a line and to join lines. *delete* remembers what it deletes; a subsequent *put* inserts the last-deleted data at the current cursor position and so may be used to move data; if given an argument, *put* puts it instead and so may be used to insert new text. *pick* remembers what *delete* does but deletes nothing; it may be used to duplicate text. *insert* inserts as much (blank) space as *delete* deletes. *start* starts editing another file, named by its argument. *stop* terminates an editing session. This is *s*'s complete user interface; all instances of *s* use it, including those that edit nontextual data.

2.2 A Machine-State Editor

Some instructors at the University of Arizona introduce assembly language programming by having the students program a simplified, pedagogical machine. This machine has an accumulator, a program counter, 20 words of memory, and an instruction format simple enough to program absolutely: a one-digit opcode and a two-digit address (e.g., the opcode for *load* is 6, so instruction 620 loads the last memory cell into the accumulator). Until recently, students programmed in batch: They prepared cards that gave an initial machine state and received a printed trace of the resulting states. Using the generalized text editor as a canned front end, an interactive simulator was coded in an hour. The editor displays the state of the machine, which, initially, is

```
ac = 0
pc = 0
m[1] = 0
m[2] = 0
  ...
m[20] = 0
```

The student creates a program by editing this. For example, overstriking can yield the state

```
ac = 0
pc = 1
m[1] = 620
m[2] = 0
  ...
m[20] = 17
```

which points the pc at a *load* ("620") followed by a *halt* ("0"). m[20] holds the value to be loaded. The student presses *start* to execute the program. The editor executes until it encounters the *halt* in m[2] and then displays the new machine state

```
ac = 17
pc = 3
m[1] = 620
m[2] = 0
  ...
m[20] = 17
```

The accumulator has been loaded with 17 and the program counter points after the *halt*. The student may continue by editing this machine state or quit by pressing *stop*. This simulator was much easier to code than the batch version, and if the student knows the text editor, the simulator may be used without special training.

2.3 A Directory Editor

The directory editor presents a file system directory just as the operating system's (in this case, UNIX[1]) standard directory listing utility does. For the reader new to UNIX, a short directory might appear as

```
-rwxrwxrwx 1 cwf    1738 Feb 11 18:55 abc
drwxr-xr-x 1 cwf     624 Jan  3  6:45 def
```

The first character tells whether the file is itself a directory: file *abc* is not, file *def* is. The next nine give the read, write, and execute permissions for the file's owner, his project group, and all other users, respectively: Anyone can do anything to *abc*, but only the owner can change *def*. The last five fields give the number of file system pointers to the file ("1"), its owner ("cwf"), its length ("1738"), its creation date ("Feb 11 18:55"), and its name ("abc").

The directory may now be changed by editing this text. Deleting the first line deletes file *abc*. Deleting the "b" from the same line changes its name to "ac". Overstriking the "cwf" changes its owner. Overstriking the permission flags changes them. Overstriking the length field does nothing: Changes are made only if they make sense and the user's access privileges permit them. *start* displays another directory or directory fragment according to UNIX convention. For example, the conventional UNIX commands

```
ls
ls *.c
```

respectively list the user's entire directory (null argument) and just those files whose names end in ".c" ("*.c" argument). Analogously, the directory editor's commands

```
escape start
escape *.c start
```

edit what the *ls* commands above displayed. All in all, the directory editor subsumes five different UNIX commands, which account for 16 percent of all command accesses.

2.4 A Binary Disk Editor

Binary patches to a disk are sometimes necessary. For example, when a physical disk block goes bad, it is usually removed from the file it represents and added to a file of "dead" blocks that will never be referenced. Under UNIX, a system expert usually does this by hand, with a utility that can present and change arbitrary, uninterpreted words on the disk. Because the job is so delicate, it would help if the file descriptors, called "inodes" [8], were displayed in a more natural form. This is where *s* is useful.

An inode is a fixed-length record of 18 fields: Some are flags, best shown in octal; others are disk addresses, best shown in decimal; still others are best suppressed. Using the generalized editor, an inode editor has been built that presents a device's inodes, one per line with each field in its natural format. Overstriking, the expert removes the address of the bad block from the user's inode and then adds that address to the "dead" inode.

Though it plays a special role, the inode list resembles many conventional databases: It is a list of records. So the inode editor is easily adapted to other similar databases: The user need only change the structure declaration and the routines that convert between the internal and display formats.

3. Organization

Changing a file name differs from changing an inode, even though a generalized editor makes the processes look the same. Different data types require slightly different editors: With so many types of data to edit, construction of editors must be made easy. This suggests separating the editor into a data-independent, front-end display manager that calls on a set of editing primitives that actually edit data. Implementing an editor for a new data type means implementing a new set of primitive subroutines. The next three sections describe issues that bear on the choice of a user interface, of a set of editing primitives, and of an implementation strategy.

3.1 The User Interface

To the user, a generalized text editor looks like a conventional text editor. Many user interfaces are possible for a text editor, so many are possible for a generalized editor; however, a generalized text editor modeled after a line editor inherits certain problems. Why? The user of a line editor indicates target text by giving a line number or a pattern that the target must match. If none is given, the editor may assume a default line number (perhaps the last line addressed), but the user is often forced to give an explicit address. For example, to delete the next line containing "editor" using UNIX's text editor *ed* [5], the user would type

```
/editor/d
```

A directory editor modeled after such a line editor would use the same command to delete a file named "editor", instead of using the standard UNIX file deletion utility *rm*:

```
rm editor
```

The generalized editor excuses users from learning the syntax of this command (and the others that edit directories) but offers the user little more than a certain elegance of having all editing operations use the same syntax: Both commands require typing the name plus three characters of overhead. To delete the same line with a screen editor, the user merely holds down a cursor control key until the cursor is positioned on the line containing "editor" (many CRTs have repeating cursor

[1] UNIX is a trademark of Bell Laboratories.

control keys) and then presses a *delete* key. The number of characters of overhead is about the same—one cursor control key, a pause until the cursor is in position, and *delete*—but the name need not be typed. Even more typing is saved if the user needs to check the directory before or after editing. With a typical operating system, or with a generalized line editor, listing the directory requires another command; with a generalized screen editor, which always displays a screen full of the current version of the data being edited, the directory is already visible (at least partly) and so need not be listed explicitly. It was to save such typing that a screen editor interface—rather than a line editor interface—was built on *s*'s editing primitives.

3.2 The Editing Primitives

s's editing primitives are a set of subroutines that edit a collection of vectors of lines or, more precisely, arbitrary nodes represented as lines of text. A screen editor may be constructed for any data that may be viewed in this way; some data structures do not oblige, but as the examples of Section 2 indicate, many do. The primitive subroutines edit lines, not characters, for good reason: To change a file's name from "abc" to "def", the user will first overstrike the "a", then the "b", then the "c", giving the trace

abc
dbc
dec
def

Clearly, this requests one rename operation, not three, so the front-end screen editor handles intra-line editing internally and sends requests to change lines (of text, directories, etc.) only when the user moves the cursor to another line or executes a special function.

The primitives are:

fetch(int: n; string: l)	fetch node n into l
change(int: n; string: l)	change node n to l
insert(int: n; string: l)	insert l after node n
delete(int: n)	delete node n
search(int: n; string: l)	search from node n on for s
open(string: f)	edit structure f
close( )	stop edit of last-opened structure
init( )	initialize primitives
stop( )	terminate primitives

The screen editor front end translates screen editor commands into calls on these primitives. For example, it translates overstrikes into calls on *fetch* (to retrieve what is there) and *change* (to record the overstrike). It translates the screen editor *delete* command into calls on the *delete* primitive (one for each line deleted) and on *change* (in case just part of a line is deleted).

This set of primitives is more than enough to implement *s*'s commands. The *search* primitive is not strictly necessary—in the directory editor, it is implemented with calls on the *fetch* primitive and a string matching routine—but performance is sometimes improved by allowing for the use of another algorithm (see the next section). The change primitive is strictly necessary: perhaps not for text where delete/insert does as well, but certainly for directories where a change (i.e., a rename) is not a deletion followed by an insertion. For some editors, some primitives have no effect; for example, one cannot insert or delete a word in memory, so the machine-state editor does not implement *insert* or *delete*.

3.3 Implementing the Editing Primitives

The most natural implementation of the editing primitives is as a closed set of subroutines that actually edit data. For example, the machine-state editor maintains variables that simulate the machine's accumulator, program counter, and memory: *init* initializes them, *fetch* retrieves and formats their values, *change* changes them, and *open* simulates the machine-language program that starts at *m*[*pc*]. Similarly, the inode editor actually fetches and changes inodes (if the user has the proper privileges), and one version of the text editor borrowed its primitives from a previously existing line editor.

When a generalized editor is being interfaced to an existing utility, another strategy is available: The primitives may command the existing utility to do their editing for them. For example, the first set of primitives for the text editor created a slave line editor process and commanded it to fetch and change lines from text files [2] (here the *search* primitive tells the line editor to do the searching, avoiding repeated fetches). Similarly, the directory editor's primitives create a slave command interpreter and send it conventional, user-level commands to delete and rename files. Perhaps surprisingly, if the primitives avoid asking the slave for data available locally, the slave-process editors run almost as fast as their single-process counterparts. Furthermore, slave-process editors are often the easier to implement, especially when it is hard to borrow code from the slave for a single-process version; in fact, to ensure file system integrity, an operating system may prevent a directory editor from deleting files itself, forcing a slave-process design. On the other hand, slave processes do complicate error detection. Suppose the directory editor is asked to delete a protected file. The *delete* primitive's slave may print a (possibly cryptic) error message, but it may not have been designed to notify its parent process of the error. Since the front end cannot know whether anything was deleted, it must refresh the screen from scratch: It cannot rely on local data already on the screen.

A variant of the slave-process scheme that is occasionally useful accumulates slave commands in a file instead of sending them directly to a slave process. For example, a directory editor might operate on a textual copy of a directory and accumulate a file of commands that—when executed later—will change the actual directory as requested. Delaying the actual editing precludes error checking entirely, but it does result in a directory editor that exploits existing utilities without using more

than one process at a time (a restriction imposed by many operating systems), and that allows the user to back out of a bungled editing session without changing the directory.

The increasing availability of programmable terminals presents a final variant of the slave-process scheme. The front end (including the primitives) can run on some intelligent terminals, presenting commands to, and interpreting responses from, a slave running on a larger host computer. The host need not support multiple processes per user because multiple processors realize the multiprocessing. Experience with such an arrangement may suggest changes to *s*; for example, the front end might retain several screenfuls locally—and even prefetch data when the connection to the host is free—to reduce communication delays. Other documents expand on this design [2].

4. Implementation

s extends a previously written screen editor that was a front end to a line editor, and only a line editor [2]. It is a 300-line program written in the language C [6] and runs on a PDP-11/70 under UNIX at the University of Arizona. Available editing primitives edit text files, directories, inodes, the state of a pedagogical machine, the date and time, and other data; they are 42 to 94 lines long. It took one person two weeks to construct the first directory editor; it took one to eight hours to construct each of the others. For its improvements to the various command languages, the generalized editor adds about 8,000 bytes in memory requirements and a CPU overhead that is imperceptible but otherwise unquantifiable—it depends so on the editing session. This particular implementation optimized development time; processor utilization could be improved was there a need to do so.

5. Discussion

The generalized text editor primitives embody a simple theory of editing that offers, at least for such simple data structures as lists (of text lines, directory entries, words of memory), a set of operations that yield a useful editor. More complex data structures may require a few additional primitives; for example, Lisp editors, which edit nested lists represented as binary trees, have *left*, *right*, and *up* positioning commands [9]. Less complex data structures may permit automatic construction of editors; for simple databases like the inode list, it is easy to imagine a single-process generalized editor parametrized by a description of the data structure it edits. The logical extrapolation of this is the inclusion of editing primitives in data structure definitions, Simula-style [1].

Acknowledgment. R. Griswold suggested several improvements to an early version of this paper.

Received July 1978; revised July 1979; accepted December 1979

References

1. Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., and Nygaard, K. *SIMULA BEGIN.* Petrocelli, New York, 1973.
2. Fraser, C.W. A compact, portable CRT-based editor. *Software—Practice and Experience 9*, 2 (Feb. 1979), 121–125.
3. Ingalls, D.H.H. The Smalltalk-76 programming system: Design and implementation. Conf. Rec. 5th Ann. ACM Symp. on the Principles of Programming Languages, Tucson, Arizona, 1978, pp. 9–16.
4. Irons, E.T. and Djorup, F.M. A CRT editing system. *Comm. ACM 15*, 1 (Jan. 1972), 16–20.
5. Kernighan, B.W. A tutorial introduction to the *ed* text editor. Tech. rep., Bell Laboratories, Murray Hill, N.J.
6. Kernighan, B.W. and Ritchie, D.M. *The C programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.
7. MacLeod, I.A. Design and implementation of a display-oriented text editor. *Software—Practice and Experience 7*, 6 (Nov. 1977), 771–778.
8. Ritchie, D.M. and Thompson, K. The UNIX time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.
9. Sandewall, E. Programming in the interactive environment: The LISP experience. *Comptg. Surv. 10*, 1 (March 1978), 35–71.
10. Unger, C. *Command Languages.* North-Holland, Amsterdam, 1975.