Editing Data Structures

CHRISTOPHER W. FRASER and A. A. LOPEZ
The University of Arizona

Text is not the only data that needs editing. For example, interactive debuggers edit data structures internal to running programs. This paper describes **eds**, a generalized editor that allows users to edit arbitrary data structures. Examples show **eds** maintaining simple databases, editing LISP S-expressions, debugging SNOBOL4 programs, and creating and modifying data structures for a computer graphics system.

Key Words and Phrases: data structures, debuggers, text editing
CR Categories: 3.73, 4.42, 4.49

1. INTRODUCTION

Editing means examining and modifying data. Though most editing programs edit text files, other data structures need editing too [15]. For example, interactive debuggers edit data structures internal to running programs [3], LISP editors edit LISP's arbitrarily nested lists [11, 14], and miscellaneous ad hoc utilities edit such databases as calendars, gradebooks, and accounting files. This paper describes **eds**, a program that edits arbitrary combinations of vectors, binary trees, linked lists, records, character strings, symbol tables, and other data structures. **eds** has been used in program debugging, database manipulation, and data reformatting. **eds** is written in SNOBOL4 and edits SNOBOL4 data structures. Most programming languages can support some form of **eds**, and similar techniques can edit more permanent "external" data structures such as file system directories [4]. Section 2 presents **eds** through simple examples, Section 3 describes some applications, and Section 4 discusses implementation.

2. THE EDITOR

The command syntax of **eds** is borrowed from the line editor **ed** [7, 8]. Commands begin with *indices* that select components of the data structure, just as line numbers select lines of text for a conventional text editor. Indices are terminated with a period, and two (or more) indices may be concatenated to select a component of a component. In other respects, the syntax of indices varies according to the type of data structure. For example, integers index vectors, field

Authors' addresses: C. W. Fraser, Department of Computer Science, The University of Arizona, Tucson, AZ 85721; A. A. Lopez, Computer Services Center, University of Minnesota, Morris, MN 56267.

names *and* integers index records, and strings index symbol tables. Thus if **eds** is editing, say, a class gradebook stored as a vector of records each with a **name, number**, and **score** field, then the index list

1.name.

might address the **name** field of the first record,[1] the index list

1.

addresses the entire first record, and the null index addresses the entire vector. The reserved index "**$**" addresses the last element of any data structure for which a last element is defined (e.g., vectors, but not symbol tables). For example,

$.name.

addresses the **name** field of the last record of the gradebook.

The index list, if any, is followed by a one-character command and optional argument:

a *expr*	inserts an item containing *expr* after the indexed item
c *expr*	changes the addressed item to contain *expr*
d	deletes the addressed item
p	prints the addressed item

The interpretation of these commands, like the interpretation of indices, depends on the data structure. For example, items cannot be inserted in or deleted from static-length records, items are not inserted "after" anything in unordered symbol tables, and items are added to the beginning of linked lists by being inserted at the otherwise illegal index **0**. Similarly, different data structures are printed differently. For example,

1.name.p

simply prints the **name** field of the first gradebook record, whereas

1.p

prints the entire record, separating fields with colons:

J. Adams:123456:50

and

p

[1] For readers curious about details, this command is appropriate for a vector of records but not for a vector of pointers to records, where the index

1.

would select the first pointer, the index

1..

would select the record to which it points (the second index is null because pointers have only one component to select), and the index

1..name.

would select that record's **name** field. SNOBOL4 relies less on explicit pointers than does, say, PASCAL, so this detail would be more important in a PASCAL-based **eds**.

prints (a template for) the entire vector of records:

ARRAY(30)

Structure-dependent routines process single indices and the commands described above. A single structure-independent driver assigns duties to these routines and interprets the data-independent features described below.

2.1 Sequencing

Two numeric indices separated by a comma specify a range of indices to which the command is to be applied. For example,

1,$.name.p

prints all names in the gradebook database, and

$.2,3.p

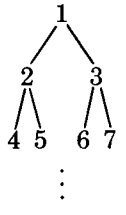
prints fields 2 and 3 of the last record in the gradebook.

2.2 Relative Indexing

Conventional index arithmetic is provided, and the reserved index "@" abbreviates the last index used so that complex indices need not be retyped. For example,

@−1,@+1.p

prints the last item addressed surrounded by its immediate neighbors and then sets "@" to index the last item printed. With the binary tree editor, which numbers its nodes thus:



multiplication and division are also useful:

@/2.p	prints the last node's ancestor
@*2.p	prints its left descendant
@*2+1.p	prints its right descendant
@±1.p	prints its sibling

"@" may hold multicomponent indices. For example, after the command

1.2.3.p

"@" is **1.2.3** and

@.4.p

is then equivalent to

1.2.3.4.p

Appending "↑" to an index deletes its last component. For example, if "@" is **1.2.3**, then "@↑" is **1.2** and "@↑↑" is **1**. Thus "@" and "↑" may be used to move up and down a tree in small steps.

An empty command line is equivalent to the command

@+1.p

so the user can step through a sequential data structure by striking carriage returns.

The "=" command evaluates and then echoes its index. For example,

@.=

prints the index of the last node indexed, and

$.=

prints the length of a vector, record, or linked list.

2.3 Content-Sensitive Indexing

The index

?pattern?

is equivalent to the index of the next node (starting from "@") that, if printed, contains the string *pattern*. For example,

?Buchanan?.p

finds and prints Buchanan's gradebook record. Context-sensitive indices may appear wherever simpler indices may be used. For example,

?Fillmore?+1,?Hamilton?−1.p

prints gradebook records starting with the one after Fillmore's and ending with the one before Hamilton's. **eds** currently searches only structures with numeric indices, because the indices of other structures cannot be enumerated in a structure-independent fashion. An earlier implementation used structure-dependent primitives like CLU's "iterators" [9] to search arbitrary structures.

2.4 Changing the Focus

Like text editors that allow users to "open" a new file, **eds** allows users to "open" a new structure, with the **e** command. Thus, someone probing an interrupted SNOBOL4 program can open any variable to look for bugs. For example,

e x

starts editing variable **x**. If subsequent examination implicates variable **y[i]**, a

e y[i]

will shift the focus there. The focus is not the same as "@". The focus corresponds to a line editor's "open file," and "@" corresponds to a line editor's "current line." The argument to the **e** command must be a SNOBOL4 variable name.

The focus is initialized to the name of an anonymous variable so that the user may begin creating a structure without an initial **e** command. For example, the initial sequence

```
c ARRAY(30)
1.c student("J. Adams")
2.c student("C. Arthur")
  ⋮
```

changes the initial focus to a new heterogeneous vector returned by the SNOBOL4 built-in function ARRAY and then creates a new gradebook in this array.

2.5 Accessing Files

The **r** and **w** commands access the file system:

r *file* inserts lines from *file* after the indexed node
w *file* prints the indexed node(s) into *file*

For example,

```
c list( )
0.r gradebook
1.d
1,$.w gradebook
```

starts editing a new (empty) linked list, reads file **gradebook** into it, deletes the first line, and writes it back out.

Theoretically, the **r** and **w** commands allow **eds** to be used as a conventional line editor, though an editor tailored to the manipulation of text files is likely to be more facile. Note that, while most line editors offer separate command sublanguages for inter- and intra-line editing, **eds** offers both styles with a common, if somewhat verbose, command language:

```
3.c "xyz"
```

replaces the entire third line, and

```
3.4.c "xyz"
```

replaces only its fourth character (with a three-character sequence).

3. APPLICATIONS

In addition to editing such self-contained databases as the gradebook shown above, **eds** can operate on data created by or for other software systems. The following sections show **eds** editing LISP S-expressions, debugging SNOBOL4 programs, and creating and modifying data structures for a computer graphics system.

3.1 A LISP Editor

When loading **eds** into a SNOBOL4 system, the user may supply additional routines to extend its capabilities. For example, by supplying **eds** with a function **sexpr** that accepts LISP S-expressions like

```
(cond ((gt x 0) 1) (t −1))
```

and returns their standard representation as nested linked lists, the user creates a LISP editor. The command

```
c sexpr("(cond ((gt x 0) 1) (t −1))")
```

creates and starts editing the S-expression above;

```
1.a sexpr("((lt x 0) −1)")
```

changes it to the S-expression

(cond ((lt x 0) −1) ((gt x 0) 1) (t −1))

and

$.2.c 0

changes that to

(cond ((lt x 0) −1) ((gt x 0) 1) (t 0))

Thus, a version of **eds** written in LISP—which might even be able to use LISP as its command language—could serve as an S-expression editor. Of course, a special-purpose S-expression editor [14] can offer more list-specific features than **eds** because it need not concern itself with other data structures.

3.2 A SNOBOL4 Debugger

Addition of a facility for trapping errors and, perhaps, programmer-defined events turns **eds** into an interactive debugger for finding errors in complex data structures. If the user loads **eds** with the program to be debugged and arranges for **eds** to get control when the SNOBOL4 interpreter finds an error, **eds** may be used to look for bugs in data structures. Consider an object code optimizer written in SNOBOL4 that represents object programs as doubly-linked lists of instructions, with extras: Labels are inserted between some instructions, extra links represent control flow, and dead variable lists are attached to each node. To examine the fifth instruction, the user types

5.p

whereas the user of a conventional SNOBOL4 debugger [5] types something like

value(next(next(next(next(top)))))

The **eds** user strikes a carriage return to see the next instruction where the user of a conventional SNOBOL4 debugger types something like

value(next(next(next(next(next(top))))))[2]

To insert or delete a node with a conventional debugger requires explicit pointer manipulation, which, if bungled, could compromise the integrity of the object program; with **eds**, the **a** and **d** commands are shorter, and, like language-specific editors that will not create syntactically-incorrect programs [13], **eds**' list editor will not create incorrectly linked lists (e.g., with dangling pointers). Finally, **eds** offers some operations offered by few conventional debuggers:

?x4:?+1.p

prints the instruction after label **x4**, and

1,$.w bugs

[2] The user might simulate "@" by assigning a long expression to a variable, but this device may interfere with the variables of the program being debugged.

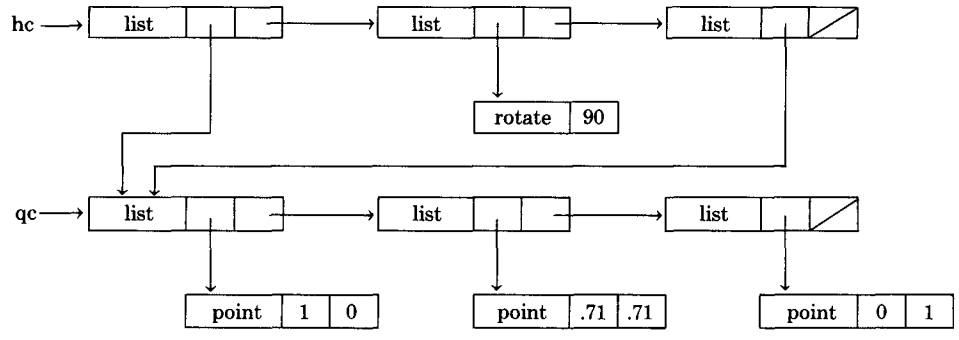


Figure 1

records the current state of the object program in a file for subsequent examination. Integrating **eds**' facilities into conventional debuggers benefits both.

3.3 A Graphics Editor

eds also accommodates heterogeneous structures. Consider a simple computer graphics system that allows users to create and edit simple hierarchical pictures. A picture is represented as an arbitrarily nested list of (1) points to be connected with lines and (2) commands to scale, rotate, or translate subsequent and subordinate points. For example,

```
e qc
c list(point(1, 0))
$.a point(.71, .71)
$.a point(0, 1)
```

creates a two-stroke approximation to the unit quarter-circle in the first quadrant and assigns it to the variable **qc**. Similarly,

```
e hc
c list(qc)
$.a rotate(90)
$.a qc
```

creates a semicircle out of two quarter-circles, one rotated into the fourth quadrant. The resulting structure, including type codes, is shown in Figure 1.

In practice, this structure would be only part of a much larger picture. For example, these pictures might be used to define logic gates: an AND gate (D) is a semicircle and a vertical stroke, a NAND gate (D○) is an AND gate and a shifted, reduced circle, which is, in turn, two semicircles, one reflected about the y-axis, etc. Though such structures quickly grow complex, **eds** can edit them without much difficulty if the user has included sufficient entry points (e.g., **qc, hc**). For example,

```
e qc
2.c point(.87, .5)
2.a point(.5, .87)
```

changes all quarter-circles, including those that comprise **hc**, to use a three-stroke approximation, and

```
e hc
2.c scale(1, −1)
```

replaces the **rotate** command in each semicircle with a **scale** command that is more efficient at producing fourth-quadrant quarter-circles from first-quadrant quarter-circles; it does so without changing the constituent quarter-circles. Again, a special-purpose graphics editor [10] may be more appropriate than **eds**, but *some* structure editor is needed to correct the errors that occur when building such complex structures.

4. IMPLEMENTATION

4.1 A SNOBOL Implementation

eds is written in the SPITBOL dialect [2] of SNOBOL4. It is a single data-independent driver—about 100 lines of code—that assigns duties to several small structure-dependent slave editors, typically 5–25 lines each. (In languages such as ALPHARD [12], CLU [9], and SIMULA [1], these slaves might form part of the datatype definition, providing a simple editor for all values of that type.) The driver interprets the data-independent features of **eds** (e.g., index arithmetic, "@"-replacement). The slaves process isolated indices and simple commands. The rest of this section describes this organization in more detail.

The list editor *EdList* outlined in Figure 2 is a typical slave. It accepts a command and a pointer to a list represented as a record with two fields: a value and a pointer to another such record. It contains the code to insert an item, delete an item, change and print lists as a whole, compute the length of the list, and find the *n*th value in a list.

As Figure 2 shows, the organization of lists induces an asymmetry into the list editor that is invisible to the user but that must be addressed by the implementer. An item in a list can be changed or printed by changing or printing some isolated field, so the list editor implements these commands by computing a pointer to the indexed field and then passing this pointer to a slave that will finish the command. On the other hand, inserting or deleting an item in a list requires pointer manipulations that only the list editor understands, so the list editor interprets both the index and command character for these commands. In general, the slaves interpret that which only they can interpret and ask the routine *Next* to invoke a slave to interpret the rest.

Next (also in Figure 2) simplifies the next component (up to the next period) of the command's index (if there is one) and calls *Dispatch* (also in Figure 2) to select the appropriate slave to continue the interpretation of the command. It interprets "=" commands itself, it replaces "$" indices by asking a slave (using an **l** command) for the length of the structure, it searches structures by printing them into a temporary variable and comparing the result with a pattern, it evaluates arithmetic expressions, and it expands sequences (the comma operator) by calling *Dispatch* with a sequence of single-index commands.

The remainder of **eds** is straightforward. A main routine repeatedly reads commands and passes them to *Next* through a routine that substitutes the last

```
procedure Next(cmd, ptr)
  if cmd = "=" then
      print the value of "@"
  else if cmd does not begin with an index then
      call Dispatch to interpret the command
  else
      let x be the first component of cmd's index
      if x contains a "$" then
          call Dispatch with an l command to learn the length of this structure and replace each
          "$" in x with the result
      if x contains a ?pattern? then
          use p commands with output redirected into a temporary to enumerate the elements of
          the current structure starting with the component of "@" that corresponds to x, and
          stopping when a string returned matches pattern. Replace ?pattern? in x with the index
          of the element that matched pattern
      if x contains any arithmetic operators then
          replace any expressions in x with their values
      if x is two numbers, n1 and n2, separated by a comma then
          for i := n1 to n2 do
              call Dispatch with i and the rest of cmd
      else
          call Dispatch with x and the rest of cmd
      append x to the new value of "@"

procedure Dispatch(cmd, ptr)
  case type of the value at which ptr points of
      Array:   EdArray(cmd, ptr)
      List:    EdList(cmd, ptr)
      Record:  EdRecord(cmd, ptr)
      String:  EdString(cmd, ptr)
      Table:   EdTable(cmd, ptr)
      Tree:    EdTree(cmd, ptr)

procedure EdList(cmd, ptr)
  if cmd matches number.a expr then
      link a new node holding expr into the list after node number
  else if cmd matches number.d then
      relink node number − 1 to point at node number + 1
  else if cmd matches c expr then
      replace the entire list with expr
  else if cmd matches l then
      compute and return the length of the list
  else if cmd matches p then
      print the template "List"
  else if cmd starts with a number n then
      call Next with the remainder of cmd and a pointer to the value field of the nth node in the list
  else error
```

Fig. 2. An overview of the implementation of **eds**.

index seen for "@" and that replaces **r** and **w** commands with equivalent, simpler commands: **r** commands are simulated with a series of

@+1.a[3]

[3] An item inserted after item n becomes item $n + 1$, so the **a** command's "+1" is needed to insert the file's line $m + 1$ after line m.

commands, and **w** commands are simulated by redirecting i/o to a file and then executing an equivalent **p** command.

eds is normally invoked as a stand-alone program and used to create, edit, read, and write structures. It may be loaded with additional routines (e.g., **sexpr** of Section 3.1) that further simplify editing. It may also be loaded with another program and used as a debugger by explicitly calling its command interpreter from the program to be debugged or by arranging for **eds** to intercept errors [5].

4.2 A Compiled Implementation

Because SNOBOL4 is more convenient than usual, the implementation of **eds** in a more conventional language like PASCAL deserves consideration. Compilation complicates expression evaluation but does not preclude it. **eds** constructs and evaluates expressions on the fly in only two places: when evaluating numeric index expressions and when evaluating the expressions on the end of **a**, **c**, and **e** commands. The first case is easily interpreted in PASCAL, and the second can be approximated by an interpreter that allows the user to give simple expressions involving, say, only addition and subtraction of scalar variables and numeric and address constants. One can, of course, interpret a larger class of expressions if one is willing to code a larger interpreter.

Block structure and static typing complicate many programming environments. Block structure may make it expedient to restrict symbol table access to just global and immediately local variables, and static typing may force users to identify types explicitly. However, such problems are not peculiar to **eds**; all debuggers for such languages suffer from them. **eds** should also have some way to exploit PASCAL's dynamic memory allocation, but, at least for structures like lists and trees, garbage collection can be avoided because **eds** knows when a node is being deleted and can free it explicitly.

The resulting debugger may even apply to several languages, because languages that are similar enough to share a subroutine library should be similar enough to share data formats and, hence, an editor that understands those formats. Indeed, the presence of a structure-editing utility, like the presence of a subroutine library, may encourage some regularity of data formats among similar languages.

5. FUTURE WORK

eds is an experimental editor. It has not been exhaustively tested or documented, it lacks some important features (e.g., backward searching, global replacement), and many users would prefer a more concise command language.

One of the most challenging extensions involves display-based [6] structure editing. A predecessor of **eds** [4] was display-based but was restricted to structures that could be presented as lists of strings. This restriction is not easily removed. There are so many ways to display complex structures—trees, for example, may be displayed with boxes and arrows, or indentation, or parentheses—that the editor must handle both variable structures (like **eds**) *and* variable display formats. Current work on **eds** is addressing this problem with an editor parameterized by a translation grammar. Syntax rules define the structure, and semantic actions define the display format.

Programs like **eds** blur the distinction between editing and programming. Since **eds** is more than a conventional editor but less than a conventional programming

language, it is natural to wonder if there are other useful processors to be discovered in this gray area and if some of them are suited to editing *and* programming. If such processors can be found, they might make editors more powerful and programming environments easier to code and easier to learn.

ACKNOWLEDGMENT

The referees made many useful suggestions.

REFERENCES

1. BIRTWISTLE, G.M., DAHL, O.-J., MYHRHAUG, B., AND NYGAARD, K. *SIMULA BEGIN.* Petrocelli, 1973.
2. DEWAR, R.B.K., AND MCCANN, A.P. MACRO SPITBOL—A SNOBOL4 compiler. *Softw. Pract. Exper. 7,* 1 (Jan. 1977), 93–113.
3. DIGITAL EQUIPMENT CORP. DDT—Dynamic Debugging Technique. Publ. DEC-10-UDDTA-A-D, 3d ed., Maynard, Mass., 1974.
4. FRASER, C.W. A generalized text editor. *Commun. ACM 23,* 3 (March 1980), 154–158.
5. HANSON, D.R. Event associations in SNOBOL4 for program debugging. *Softw. Pract. Exper. 8,* 2 (March 1978), 115–129.
6. IRONS, E.T., AND DJORUP, F.M. A CRT editing system. *Commun. ACM 15,* 1 (Jan. 1972), 16–20.
7. KERNIGHAN, B.W. A tutorial introduction to the UNIX text editor. Tech. Rep., Bell Labs., Murray Hill, N.J., 1978.
8. KERNIGHAN, B.W., AND PLAUGER, P.J. *Software Tools.* Addison-Wesley, Reading, Mass., 1976.
9. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20,* 8 (Aug. 1977), 564–576.
10. NEWMAN, W.M., AND SPROULL, R.F. *Principles of Interactive Computer Graphics,* 2d ed. McGraw-Hill, New York, 1979.
11. SANDEWALL, E. Programming in an interactive environment: The LISP experience. *Comput. Surv.* (ACM) *10,* 1 (March 1978), 35–71.
12. SHAW, M., WULF, W.A., AND LONDON, R.L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM 20,* 8 (Aug. 1977), 553–564.
13. TEITELBAUM, T. The Cornell Program Synthesizer: A syntax-directed programming environment. *SIGPLAN Notices* (ACM) *14,* 10 (Oct. 1979), 75.
14. TEITELMAN, W. Interlisp Reference Manual. Xerox PARC, Palo Alto, Calif., Oct. 1978.
15. VAN DAM, A., AND RICE, D.E. On-line text editing: A survey. *Comput. Surv.* (ACM) *3,* 3 (Sept. 1971), 93–114.

Received January 1980; revised June and September 1980; accepted November 1980