High-Level Language Facilities for Low-Level Services†

Christopher W. Fraser
Department of Computer Science, The University of Arizona
Tucson, Arizona 85721

David R. Hanson‡
Department of Electrical Engineering and Computer Science, Princeton University,
Princeton, New Jersey 08544

Abstract

EZ is a language-based programming environment that offers the services provided separately by programming languages and operating systems in traditional environments. These services are provided as facilities of a high-level string processing language with a 'persistent' memory in which values exist indefinitely or until changed. In *EZ*, strings and associative tables provide traditional file and directory services. This paper concentrates on the use of *EZ* procedures and their activations, which, like other values, have indefinite lifetimes. In *EZ*, the low-level aspects of procedure execution, such as activation record creation, references to local variables, and access to state information, are accessible via high-level language constructs. As a result, traditionally distinct services can be provided by a single service in the *EZ* environment. Furthermore, such services can be written in *EZ* itself. An editor/debugger that illustrates the details of this approach is described.

1. Introduction

EZ [6-8] is a software system that integrates the traditionally distinct facilities of programming languages and operating systems into a single system. This integration is achieved by providing these facilities in a high-level string processing language. The result of this integration is a system in which all interaction is performed using the language facilities of *EZ*. Examples of interaction that are traditionally performed by the operating system or its utilities include editing, debugging, and filing. Traditional 'systems' programming is also done in *EZ*.

EZ facilitates this degree of integration through several unconventional features. *EZ* uses late binding times, execution-time scope rules, a 'persistent' memory model in which objects have infinite lifetimes, and a type system that integrates or 'fuses' [6] conventionally distinct types into single types. In addition, *EZ* provides high-level data types such as strings, associative tables, and procedure activations. Objects that are manipulated by utilities in traditional operating systems are manipulated by operators and control structures in *EZ*. Previous papers [6, 8] have concentrated on the use of *EZ* strings and tables as a file system; this paper concentrates on the use of *EZ* procedure activations.

The version of *EZ* described here is written in C and runs under UNIX on a VAX-11/780, but it is intended to use *EZ* as the complete environment on personal computers. The following sections briefly describe the language facilities of *EZ*, its procedure mechanism, and an editor/debugger written in *EZ*, which displays the merits of its facilities.

2. Data and Control

As a programming language, *EZ* is a high-level string-processing language derived from SNOBOL4, SL5 [11], and Icon [10]. It has most of the basic attributes of those languages, such as concise, expressive constructs, run-time flexibility, untyped variables, heterogeneous structures, and automatic type conversion. Strings are treated as scalars, and there are numerous 'mid-level' string operations similar to those in Icon, but pattern matching operations are not provided. Many of *EZ*'s features are similar to

†This work was supported by the National Science Foundation under Grants MCS-8102298, MCS-8302398, and DCR-8320257.

‡Permanent address: Department of Computer Science, The University of Arizona, Tucson, Arizona 85721.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1984 ACM 0-89791-147-4/85/001/0217 $00.75

those in traditional languages and the string-processing features are similar to those in Icon. The reference manual [7] contains a complete description of *EZ*'s syntax, semantics, built-in values, and usage.

EZ has a 'persistent' memory much like an APL workspace in which values exist until changed. Unlike systems that access workspaces explicitly [2], this memory model is implicit in *EZ* and is a part of the language, much as in the Gem system [13]. This is one characteristic that distinguishes *EZ* from various LISP environments, which access traditional operating system services through functions or other explicit mechanisms [16]. The implementation assumes responsibility for managing primary and secondary memory, including garbage collection.[1]

EZ supports four basic types of values: numerics, strings, procedures, and tables. Numerics include integers and reals, which serve their conventional purposes. Strings are sequences of characters and have arbitrary length. Tables are heterogeneous one-dimensional arrays that can be indexed by and can contain arbitrary values. Procedures are described below.

Since values persist until changed, assignment of a string to a variable provides the same facility as creating a 'file' in a traditional operating system. Substring operations provide 'random access' facilities; s[i:j] specifies the substring between character positions i and j. As in Icon, character positions are numbered from the left starting at 1 and refer to positions between characters. Positions may also be specified relative to the right end of a string starting at 0 and continuing with negative values toward the left. For example, the positions in the string HAT are

```
 H A T
↑ ↑ ↑ ↑
1 2 3 4
-3 -2 -1 0
```

Note that the position after the last character may be specified. Substrings may also be specified by starting position and length, e.g., s[i|l] specifies a substring of length l starting at position i.

Tables provide an associative array facility similar to that provided by tables in SNOBOL4 and Icon and by arrays in **awk** [1]. Tables are created automatically as necessitated by subscripting. Tables, like other values, persist until changed and thus subsume directories in traditional operating systems. For example,

```
paper["title"] = "High-Level Language...
paper["authors"] = cwf || "\n" || drh
```

creates a table representing this paper and establishes it as the value of paper. (|| is string concatenation.) The notation *e.id* is equivalent to *e["id"]*, permitting *EZ* tables to subsume records in traditional programming languages; the example above is equivalent to

```
paper.title = "High-Level Language...
paper.authors = cwf || "\n" || drh
```

Table indices and values can be of arbitrary types. For example,

```
paper[1].heading = "Introduction"
paper[1].body = "EZ is a software...
paper[1].top = paper
```

creates a table for Section 1 of this paper, and establishes it as the value associated with index 1 in paper. In addition, the value associated with top in the table for Section 1 is the table for the entire paper. Arbitrary cyclical structures, such as this example, are permitted and are, in fact, frequently used in *EZ*. Tables are as large as necessary to accommodate their contents. Entries are removed by the built-in function remove. For example,

```
remove(paper[1], "top")
```

removes the value associated with top in the table for Section 1 constructed above. Tables can be used to construct a hierarchical 'file system', such as that provided by UNIX.

Expressions usually compute values, but, as in Icon, some expressions may fail to yield values. The absence of values is used to drive control structures. For example, the relational operators return their right operand only if the relation is satisfied. The absence of values terminates **for** and **while** loops and determines the flow of **if** statements. For some operators, such as assignment, the absence of a value inhibits the execution of the operation.

EZ strings, tables, operators, and control structures provide the facilities of files, directories, and file system utilities found in operating systems. For example, listing the contents of a table corresponds to listing the names in a directory. Whereas in most operating systems, this service is provided by a 'list directory' program, it can be provided in *EZ* with a simple loop. For example,

```
for (i in paper)
    s = s || "," || i
```

sequences through the table given by paper, repeat-

[1] In the current version, UNIX provides environmental support such as terminal i/o and disk services, but in a personal computer environment, the entire computer and its peripherals are to be devoted to supporting the *EZ* environment.

edly assigns the indices of the table to i, and concatenates the indices onto the end of s.

Automatic conversions between data types obviate the need for most 'conversion' utilities found in traditional languages and operating systems. Numeric operators convert their operands to integers or reals as necessary. Similarly, operands of string operators are converted to strings as necessary. For some operators, the operation performed depends on the type of the operands. For example, the relational operators perform lexical comparison if both operands are strings and numeric comparison (with the appropriate conversions) if either operand is numeric.

Conversions between tables and strings are also provided. Tables are converted to strings by concatenating their elements, and strings are converted to tables by constructing a table with the string associated with the index 1. Thus, for example, simply typing the name of a table displays its contents.

3. Procedures and Activations

EZ procedures are data objects that contain executable code. A procedure 'declaration' amounts to an assignment of the procedure 'constant' to the identifier. For example, after the execution of

```
procedure ls(t) local i, s
   s = ""
   for (i in t)
      s = s || "," || i
   return (s[2:0])
end
```

the value of ls is a procedure that returns a list of the indices in a table.

Conversions between procedures and strings are performed automatically. Procedures are converted to strings by returning their source code. Simply typing the name of a procedure, such as ls, therefore, displays its code. Strings are converted to procedures by compiling them. Thus,

```
for (i in work)
   work[i]()
```

executes the values in the table work, compiling those values that are not procedures. In a sense, compilation is simply an optimization in *EZ*.

Scope rules, which dictate the interpretation of free identifiers, depend on the contents of tables interrogated by the compiler. Unlike most other systems, these 'symbol tables' are *EZ* tables, which can be manipulated at the source-language level. As exemplified in ls, above, identifiers may be declared local and their use is restricted to the associated procedure in the traditional manner. An interpretation of free identifiers is sought by searching the table that is the current value of the variable root for an index value lexically equal to the identifier. Thus, the assignment

```
message = "I'll return soon"
```

is equivalent to

```
root["message"] = "I'll return soon"
```

If the identifier is not found in root, the compiler searches the chain of tables given by root[".."], root[".."][".."], and so on until the identifier is found or a table without a ".." entry or whose ".." entry is not a table is encountered. If this search fails to locate the identifier, it is entered in root.

This interpretation of identifiers is under complete control of the user. By changing the value of root and altering the path given by the ".." entries, rules such as the inheritance rules in Smalltalk [9], the 'search lists' in UNIX, and the information-hiding aspects of modules and own variables can be obtained. For example,

```
root = ["previous":root]
```

uses a table constructor in which index-value pairs are given as *index:value* to assign a new table to root that associates the previous value of root with the entry previous. The absence of a ".." entry forces free identifiers to be associated with the new table. The subsequent input

```
s = 0
previous.random = procedure (n)
   s = (s*12621 + 21131)%10000
   return (s*n/10000 + 1)
end
```

defines a procedure that generates a sequence of pseudo-random numbers in the range 1 to n using the linear congruence method. The procedure is made accessible by placing it in the original table, but the table containing the state variable s is inaccessible except for the references within random. Other variations, such as saving the table containing s so that it can be changed, are also possible. The assignment

```
root = previous
```

resets root to its original value.

Procedures may be invoked as in traditional languages, e.g. ls(root). In addition, conversion from a procedure to a table, provided by the built-in function table, yields a table that is an activation record for the procedure. This table contains entries for each of the parameters and locals declared in the

procedure and entries describing the current state of the activation. Tables created in this manner can be used as coroutines and exist until they are inaccessible.

For example,

```
procedure decode(cmd, keymap)
   local c, s, t

   s = ""
   for (t = keymap; c = cmd[1|1]; t = t[c]) {
      s = s || c
      cmd = cmd[2:0]
      if (type(t[c]) == "procedure") {
         t[c](s)
         return (1)
         }
      else if (type(t[c]) ~= "table")
         return
      }
end
```

defines a procedure to traverse a set of nested *EZ* tables using the characters in cmd as indices until an entry containing a procedure is found. The expression

```
d = table(decode)
```

assigns to d a table representing an activation record for decode. d contains entries for "cmd", "keymap", "s", "c", and "t", each uninitalized. 'Invoking' d begins execution of decode. Arguments can be initialized by position, e.g.,

```
d(nextcmd, z19map)
```

or by explicit assignment, e.g.,

```
d.cmd = nextcmd
d.keymap = z19map
d()
```

Execution continues until decode returns. Upon return, the entries in d are the values of the parameters and locals upon return. For example, if

```
d("abcd", z19map)
```

led to the integer 24 instead of a table or procedure in the tree beginning with z19map,

```
if (d(nextcmd, z19map))
   ...
else
   error = "unbound sequence <" ||
      d.s || d.cmd ||
      "> yields " || d.t[d.c]
```

assigns

```
unbound sequence <abcd> yields 24
```

to error.

In addition to entries for the parameters and locals, the value associated with the index "Procedure" is the procedure itself, and the value associated with the index "Resumption" is the resumption point or 'location counter' for the activation. Resumption points correspond to executable expressions or statements in the procedure body. Resumption points are numbered sequentially following the lexical order of the statements and expressions. For example, the resumption points for decode, given as superscripts, are as follows.

```
procedure decode(cmd, keymap)
   local c, s, t

   1
   2s = ""
   3for (4t = keymap; 5c = cmd[1|1]; 6t = t[c]) {
      7s = s || c
      8cmd = cmd[2:0]
      9if (10type(t[c]) == "procedure") {
         11t[c](s)
         12return 13(1)
         }
      else 14if (15type(t[c]) ~= "table")
         16return 17
      }
   18
end
```

Resumption points on statements other than expressions are used for debugging purposes, as described below.

Activations can be resumed at any point by changing the value of the "Resumption" entry. Assigning an integer to "Resumption" causes execution to be resumed at the corresponding resumption point upon the next invocation. For example,

```
d.Resumption = 1
d("abcd")
```

restarts the activation of decode with a new value for the first argument.

Subscripting a *procedure* with an integer yields the source code for the corresponding resumption point. For example, decode[7] returns the string

```
s = s || c
```

Substringing a procedure yields the source code for the resumption point whose code most closely surrounds the substring specified; i.e., a normal substring specification is 'widened' to the boundaries of the nearest resumption point. For example,

```
decode[find("type", decode)!1]
```

returns the string

```
type(t[c]) == "procedure"
```

find(s1,s2) is a built-in procedure that returns the leftmost position in s2 where s1 occurs as a substring.

The source code for resumption points can be changed at any time, and subsequent resumptions of activations refer to the new source code. For example,

```
decode[11] = "if (~t[c](s)) return"
```

changes decode to

```
procedure decode(cmd, keymap)
   local c, s, t
   ...
      if (type(t[c]) == "procedure") {
         if (~t[c](s)) return
         return (1)
         }
   ...
end
```

and renumbers the resumption points accordingly. Subsequent resumption of d, for example, uses the new code. Such changes are, of course, typically accomplished with an editor, as described below.

Activations are just tables and persist until changed; entries can be added, removed, or changed as desired. For example, the Procedure entry need not correspond to the procedure from which the activation was created; it can be changed to any procedure. Missing local variables and parameters are created as necessary during execution.

Treating activations and tables as a single type induces a different programming style that emphasizes activations over procedures. It is typical, for example, for tables to contain *both* data and activation information. For example, keymap in decode could be used as the activation instead of the argument:

```
z19map.keymap = z19map
z19map.Procedure = decode
z19map.Resumption = 1
```

includes the necessary entries to use z19map as the activation instead of an argument to an activation. Subsequently, expressions such as

```
z19map("abcd")
```

accomplish command decoding. Including a formatting procedure as the Procedure entry in paper, described above, is another example.

As these examples suggest, activations can be constructed from scratch directly in *EZ*. For instance,

```
random = [
   "s" : 0,
   "Procedure" : procedure (n) local s
      s = (s*12621 + 21131)%10000
      return (s*n/10000 + 1)
   end,
   "Resumption" : 1
   ]
```

builds an activation for the random number generator described above. Each resumption of the generator, e.g., random(100), returns the next random number in the sequence. Note that s is hidden as a local variable in this version of random; in the previous version, s was hidden in another table using the scope rules.

4. An *EZ* Editor/Debugger

Just as *EZ*'s built-in operations generalize traditional operations, *EZ*'s editor generalizes traditional editing functions. In particular, since activations are just *EZ* tables, the editor is automatically a debugger as well.

The editor looks like a conventional screen editor [15] based on the Irons model [12]. It displays a screen of text and updates it after each command so that what you see is what you get. Hitting cursor keys moves a cursor around the screen, and hitting printable characters replaces the character at the cursor with the input character. A few special function or control keys insert and delete text, move the focus, etc. The editor's command dispatcher served as an example in the previous section. Ultimately, menus and a mouse should replace the control and cursor keys required by the development system.

Where conventional editors edit only text files, the *EZ* screen editor edits all *EZ* values. *EZ*'s type integration and automatic conversions allow the editor to accomplish this goal by managing only two types, strings and tables. Both types share one user interface. A data-independent front end manages the display and translates the user's commands into calls on a small set of primitive editing routines that actually manipulate the data. When invoked, the front end examines its argument and selects the string or table interface accordingly. These routines print, change, insert, and delete one line in a string or one entry in a table. This technique adapts to edit a wide range of types [5], and it also allows tailoring of the editor to sub-abstractions (e.g., relational databases

represented as tables of tables).

The editor treats numbers, strings, and procedures much as a conventional editor treats text files, performing conversions to strings as necessary. At the end of the editing session, it converts the edited value back to the type of the original value.

The editor treats tables by allowing the user to edit the table's keys as though they were text. It displays an image of the keys, and it changes them to reflect the user's changes to the screen. For example, when editing a table for an activation of random, the editor displays its keys

```
Procedure
Resumption
n
s
```

Now, for example, changing the last line will change the *name* of random's seed variable s.

An 'enter' command recursively invokes the editor on the value associated with the key on the line holding the cursor. For example, invoking the enter command on the line holding s above edits its associated value, namely the *value* of the seed. That is, the value of s is displayed, and it may be changed by simply overstriking it. Because tables subsume conventional file system directories, the enter command allows users to walk 'directories'; because tables subsume activations, the enter command subsumes traditional debugger commands to examine variables as well.

Breakpoints are also implemented as special cases of editing. For example, invoking the enter command on the line holding Procedure above edits its associated value, namely the source code, which can now be edited as a string. For example, inserting

```
edit(random)
```

before random's assignment to s changes random to

```
procedure (n) local s
   edit(random)
   s = (s*12621 + 21131)%10000
   return (s*n/10000 + 1)
end
```

The edited code calls the editor/debugger on each re-entry and passes the activation of random to edit for examination and modification. Thus the insertion of this line effectively sets a breakpoint. The breakpoint command merely abbreviates editor commands for such an insertion. Breakpoints are deleted by deleting the inserted text.

Activations can be executed 'incrementally' by executing the code at a single resumption point. Incremental execution is implemented by editing the activation record. The user selects a portion of the code fragment to be executed, say from position i to position j, which is widened to resumption point boundaries. The substring is extracted, the code is executed, and the resulting value is displayed. To use the proper environment (e.g., the proper variables), the incremental execution command reuses the activation that is being edited, but with a new procedure that consists of just the code for the single resumption point from the original procedure. This is accomplished by a procedure containing the following *EZ* code.

```
savP = x.Procedure
savR = x.Resumption
x.Procedure = x.Procedure[i:j]
x.Resumption = 1
x()
```

x is the editor's single parameter, which holds the datum being edited, here the activation record being scrutinized. After saving the current procedure and resumption point, x's procedure is changed to just the code for the resumption point. This new procedure is executed by resetting the resumption point to the beginning and invoking the activation. After executing the code for the single resumption point, the activation returns, and the editor restores the original procedure and resumption point:

```
x.Procedure = savP
x.Resumption = savR
```

By positioning the cursor, the user can execute arbitrary portions of a procedure in an arbitrary order and see the effects. Since debugging is simply editing in *EZ*, errors can be detected and corrected, and correct execution resumed without resorting to the 'debug-edit-compile-debug' cycle of traditional programming environments.

Incremental execution is similar to the 'single-stepping' mode provided by traditional debuggers. For example, in many cases, using

```
x.Resumption += savR
```

in place of the last line of code above advances the resumption point of the original procedure past the code just executed. The code for setting breakpoints and for incremental execution is specific to activation records, but they are the only parts of the editor with this property.

Because tables subsume conventional file system directories, the editor subsumes traditional file system commands to list directories and to remove, create, and rename files. Because tables subsume activations, the editor subsumes traditional

debuggers as well. It also subsumes 'sub-editors' within such utilities. For example, under conventional systems, setting a breakpoint and changing a variable in the debuggee require quite different commands, but under *EZ*, they are both done with the single generalized editor. The user learns to navigate structures as much as to operate commands. Where, for example, the UNIX manual describes many more commands than structures, the emerging *EZ* manual describes fewer commands and more structures. Since structures are described declaratively where commands are described procedurally, this may yield a simpler system.

The editor is to become *EZ*'s primary user interface. The current fetch-execute loop may be used only to bootstrap up the editor. The use of an editor as the main user interface has been proposed before [4, 13, 17], but it has yet to be fully exploited [15].

5. Discussion

EZ's design seeks to provide high-level facilities by simplifying and generalizing traditionally low-level facilities and encapsulating them in language constructs. The resulting facilities are simple and flexible; using tables as activations is an example. *EZ*'s 'open' approach and ability to modify activations at the source-language level are in contrast to the 'closed' approaches of previous coroutine facilities [14], in which activations are manipulated with a few specific constructs and their internals are inaccessible.

One of the design goals of *EZ* is to provide the services of a modern operating system, such as those provided by the UNIX system calls (viz. §2 of Ref. 19). Asynchronous processes are an important omission from the list of such services. Current work is directed toward using activations as processes, permitting them to be resumed asynchronously. Likewise, it is intended to use activations to respond to interrupts and other asynchronous events.

The full potential of resumption points needs further exploration and perhaps refinement. For example, setting resumption points using knowledge of the source code is somewhat primitive. Higher level functions for 'scanning' the source code, much as the string scanning functions scan strings, is a possible alternative. This kind of capability also suggests the use of other high-level operations, such as pattern matching and database functions, on objects such as activations and procedures. Manipulating resumption points to handle exceptions and error conditions is also a possibility.

Efficient implementation of the *EZ* procedure mechanism is another area for further work. The current implementation is straightforward; little attempt is made to execute procedure activations efficiently and to avoid unnecessary conversions. Using lazy evaluation and multiple representations for objects, such as is done in recent implementations of Smalltalk [3, 18], are areas of current implementation work.

The central challenge in *EZ* is finding the natural linguistic encapsulation of system services. The encapsulation of primary and secondary memory as strings and tables is more obvious than similar encapsulations for interactive devices, such as keyboards and displays, or higher-level representations such as windows and processes. Treating procedure activations as data, accessible as data, provides a possible encapsulation for such services.

References

1. A. V. Aho, B. W. Kernighan and P. J. Weinberger, Awk—A Pattern Scanning and Processing Language, *Software—Practice & Experience 9*, 4 (Apr. 1979), 267-279.
2. M. Atkinson, K. Chisholm, P. Cockshott and R. Marshall, Algorithms for a Persistent Heap, *Software—Practice & Experience 13*, 3 (Mar. 1983), 259-271.
3. L. P. Deutsch, Efficient Implementation of the Smalltalk-80 System, *Conf. Rec. 11th ACM Symp. on Prin. of Programming Languages*, Salt Lake City, UT, Jan. 1984, 297-302.
4. J. R. Ellis, N. Mishkin, M. van Leunen and S. R. Wood, Tools: An Environment for Timeshared Computing and Programming, *Software—Practice & Experience 13*, 10 (Oct. 1983), 873-892.
5. C. W. Fraser, A Generalized Text Editor, *Comm. ACM 23*, 3 (Mar. 1980), 154-158.
6. C. W. Fraser and D. R. Hanson, A High-Level Programming and Command Language, *Proc. of the SIGPLAN '83 Symp. on Programming Language Issues in Software Systems*, San Francisco, CA, June 1983, 212-219.
7. C. W. Fraser and D. R. Hanson, The *EZ* Reference Manual, Tech. Rep. 84-1, Dept. of Computer Science, The Univ. of Arizona, Tucson, AZ, Jan. 1984.
8. C. W. Fraser and D. R. Hanson, Integrating Operating Systems and Languages, Tech. Rep. 84-2, Dept. of Computer Science, The Univ. of Arizona, Tucson, AZ, Jan. 1984.

9. A. Goldberg, D. Robson and D. H. H. Ingalls, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.

10. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1983.

11. D. R. Hanson and R. E. Griswold, The SL5 Procedure Mechanism, *Comm. ACM 21*, 5 (May 1978), 392-400.

12. E. T. Irons and F. M. Djorup, A CRT Editing System, *Comm. ACM 15*, 1 (Jan. 1972), 16-20.

13. E. T. Irons, Software for a Graphics Editing Machine, *Proc. of the Fifth Texas Conf. on Computing Systems*, Austin, TX, Oct. 1976, 13-19.

14. C. D. Marlin, *Coroutines: A Programming Methodology, A Language Design, and An Implementation*, Springer Verlag, Berlin, 1980.

15. N. Meyrowitz and A. van Dam, Interactive Editing Systems: Part II, *Computing Surveys 14*, 3 (Sep. 1982), 353-415.

16. E. Sandewall, Programming in the Interactive Environment: The Lisp Experience, *Computing Surveys 10*, 1 (Mar. 1978), 35-71.

17. J. Scofield, Editing as a Paradigm for User Interaction—A Thesis Proposal, Tech. Rep. 81-11-01, Dept. of Computer Science, Univ. of Washington, Seattle, WA, Nov. 1981.

18. N. Suzuki and M. Terada, Creating Efficient Systems for Object-Oriented Languages, *Conf. Rec. 11th ACM Symp. on Prin. of Programming Languages*, Salt Lake City, UT, Jan. 1984, 290-296.

19. *UNIX Programmer's Manual, Volume 1*, Computer Science Div., Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, CA, Seventh Edition, Virtual VAX-11 Version, June 1981.