A High-Level Programming and Command Language†

Christopher W. Fraser
David R. Hanson

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Abstract

Unifying programming and command languages is a promising idea that has yet to be thoroughly exploited. Most attempts at such unification have used Lisp or traditional languages, such as Pascal. This paper describes the command and programming language *EZ*, which attempts to unify command and programming languages by using high-level string-processing concepts, such as those in SNOBOL4 and Icon. *EZ* has particularly simple data abstractions that attempt to bridge the gap between the data abstractions of command languages and those of programming languages. This is accomplished by *type fusion*, which pushes the differences between some classes of types, such as strings and text files, out of the language and into the implementation. The language, its use, and its implementation are described.

1. Introduction

Traditional programming languages and command languages are fundamentally different. Their differences are induced by the differences in their intended application areas.

At a more fundamental level, programming and command languages differ most radically in the data abstractions they must support and in their principal binding times. Typical Pascal-like programming languages support data types that are close to those provided by most computers, such as integers, reals, characters, and arrays. Command languages, on the other hand, tend to support 'high-level' data types that bear little resemblance to the data types provided by most architectures. Examples are strings, files, directories, and programs.

Conventional programming languages tend toward early binding times. They insist on compile-time definition of many aspects of programs, such as the size and type of variables, primarily for efficiency reasons. Command languages tend toward late binding times, not only because efficiency is less important, but because the kinds of operations performed demand more flexibility. Indeed, compilation is delayed until execution in most command languages. Varying-length strings and lists, automatic type conversion, procedures with a variable number of arguments, and dynamic procedure definition are examples of command language features that require late binding times.

The control abstractions of programming and command languages do have many superficial similarities, and the structured control constructs of programming languages appear in some command languages, such as the UNIX shell [4, 17] and the CMS EXEC [1]. Since control abstractions and data abstractions interact closely, command languages tend to offer more flexible control structures. Examples are loops that iterate over lists of items and 'generative' or pattern-matching constructs, such as file name expansion.

Unifying programming and command languages is a promising idea that has been examined from a number of viewpoints [16, 19, 21, 22]. Unifying programming and command languages would make learning multiple languages unnecessary for some users, remove the distinction between 'real programs' and 'command programs', and simplify the use and implementation of programming environments.

When traditional languages, such as Pascal, have been used as command languages [3, 5], they have

†This work was supported by the National Science Foundation under Grants MCS-7802545 and MCS-8102298.

seldom offered more than procedure calls in the traditional languages. Other aspects of the programming language, such as data types, appear in restricted forms or are omitted. This shortcoming can be attributed to the differences mentioned above; traditional languages lack the flexibility and data abstractions necessary for command languages.

Attempts at using high-level languages as command languages have been more encouraging. Lisp and programming environments built around Lisp are the prime examples [6, 22, 24]. Much of this success is because Lisp is flexible and has many of the concepts and facilities needed in a command language [19].

This paper describes the command and programming language *EZ*, which attempts to unify command and programming languages by using the concepts of high-level string-processing languages, such as SNOBOL4 and Icon [12]. Instead of embedding a command language in a programming language, or vice versa, however, *EZ* was designed as a unified language *from the onset.* For the most part, the control abstractions in *EZ* are traditional; some aspects of Icon's goal-directed expression evaluation [11] and coroutines [13] will be included in future versions. The data abstractions in *EZ* are particularly simple and are an attempt to bridge the gap between the data abstractions of command languages and those of programming languages. This is accomplished by *type fusion,* which moves the differences between some classes of types out of the language and into the implementation. For example, strings and text files are identical at the source-language level; the use of secondary storage is hidden within the implementation. The remainder of this paper gives an overview of *EZ* and its use, a description of its implementation, and a glimpse of its future development.

2. Language Overview

As a programming language, *EZ* fits somewhere between high-level string processing languages like SNOBOL4 and low-level Algol-like languages like C [18]. It shares many of the basic attributes of languages like SNOBOL4 and Icon, such as concise, expressive constructs, run-time flexibility, untyped variables, heterogeneous structures, and automatic type conversion. It also treats strings as scalar types and has numerous 'mid-level' string operations similar to those in Icon. It does not, however, have pattern matching or any form of backtracking.

Syntactically, *EZ* is similar to C with a few additional control constructs. *EZ* programs are sequences of statements and procedure declarations. Statements outside procedures are executed immediately, and procedure declarations assign procedures to identifiers. Identifiers are either global or local. Statements include the usual structured control flow constructs and expressions. *EZ* programs exist in a 'workspace' environment much like APL. Values, global variables, and procedures exist until changed.

EZ supports fewer data types than SNOBOL4 and Icon because it treats several classes of types as a single type. Types of variables vary during execution and are the types of the values assigned to them. Values are converted to the appropriate types automatically as necessary for most operations, e.g. operands of arithmetic operations are converted to numeric types if possible. The following table lists the *EZ* types and their traditional counterparts.

EZ type	*traditional types*
numeric	integer, real
string	string, text file
procedure	procedure, program
table	array, record, associative table, directory

Integers and real numbers serve their conventional purposes. In arithmetic operations, integer arithmetic is performed if both operands are integers, otherwise, the operands are converted to reals and real arithmetic is performed.

Strings are sequences of characters. In string operations, operands are converted to strings, if possible. Text files and strings are linguistically equivalent. Files are manipulated as strings and may be as large as desired. The implementation handles the allocation and use of secondary storage, as in Poplar [21]. Substrings, and hence subfiles, can be manipulated and arbitrarily changed.

Tables are heterogeneous one-dimensional arrays that can be indexed by, and can reference, arbitrary values much like SNOBOL4 tables and **awk** arrays [2]. Tables are as large is necessary to accommodate their contents. Since tables can contain arbitrary types, such as files, they are a generalization of directories in traditional systems.

EZ procedures serve their conventional purpose and are declared as follows.

```
procedure name ( ... )
   local identifiers
   statement...
end
```

local identifiers are known only within the procedure in which they are declared. Undeclared identifiers are global. Procedures are also data values; 'executing' a procedure declaration assigns the procedure to *name*, as if the assignment

```
name = procedure ( ... )
   local identifiers
   statement...
end
```

were made. Programs are also treated as procedures; invoking a program is linguistically equivalent to calling a procedure.

Most expressions compute values as usual, but some do not yield a value. The relational operators, for example, return their right operand if the relation is true, and do not return a value if their relation is false. Other operators, such as assignment, ignore the absence of values. For example,

```
max = max < a
```

changes max only if a is greater than max. Likewise, variables do not have a value initially, and the lack of a value in a context that requires one, such as for addition, yields a run-time error. This facility is a simplification of the sequence of values returned by generators in Icon [11].

Control structures are driven by the presence or absence of values. For example, in

```
if (a > max) max = a
```

max is changed only if the comparison yields a value. *EZ* includes the conventional while and repeat loops and the C for loop.

2.1 Primitive Operations

EZ includes the conventional arithmetic, comparison, and logical operations, although some have unconventional semantics. For example, the type of a comparison operation depends on the types of the operands. If both operands are strings, lexical comparison is performed; if either operand is numeric, arithmetic comparison is performed, automatically converting the other operand as necessary. This strategy permits strings to be compared with numerics without requiring explicit conversions. Explicit conversions can be used to force numeric comparison between strings.

String concatenation and substring selection are included in addition to the string comparison operations. Concatenation, denoted by the binary operator ||, creates a new string by appending the second operand to a copy of the first operand. The operands are automatically converted to strings as necessary. Substrings, and hence subfiles, can be manipulated and arbitrarily changed. Substrings are specified by *s*[*i*:*j*], where *i* and *j* are character positions in *s* starting at 1 from the left. *s* is automatically converted to a string, if necessary. This facility subsumes random access mechanisms in traditional systems. This results from the fusion of strings and text files into a single type.

As in Icon, substring numbering refers to positions between characters, so that, for example, the positions in the string "HAT" are

```
 H A T
↑ ↑ ↑ ↑
1 2 3 4
```

Note that the position after the last character may be specified. Positions may also be specified relative to the right end of a string starting at 0 and continuing with negative values toward the left:

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

For this string, the positions 4 and 0 are equivalent, positions 3 and −1 are equivalent, etc. The substring operation does not return a value if the substring specification refers to a non-existent portion of the string. Assignment to a substring changes the substring to the new value and assigns the new string to the subscripted variable. The value assigned to the substring is automatically converted to a string as necessary. For example,

```
s = "The file contains 72 characters"
s[19:21] = 64*64
```

assigns

```
"The file contains 4096 characters"
```

to s, and is equivalent to

```
s = s[1:19] || 64*64 || s[21:0]
```

The positions in a substring specification can be given in either order; s[21:19] specifies the same substring as s[19:21]. In addition, substrings can be specified by starting position and length; for example, s[19!2] and s[21!-2] are equivalent to the specification above.

Tables are created automatically when a variable is subscripted, e.g.

```
count["procedure"] = 1
```

assigns 1 to the table element at index "procedure". If the value of count is not a table, one is created and assigned to count. Tables and table entries are created only on assignment; referencing a non-existent entry does not create the entry.

The control structure

```
for (id in expr) statement
```

sequences through the table returned by *expr*. It repeatedly assigns the index value of a table element to *id* and executes *statement* until all of the elements have been processed. For example,

```
sum = 0
for (i in t)
   sum = sum + t[i]
```

sums the elements of table t.

2.2 Built-in Values

Numerous built-in procedures and values are provided as the initial values of global variables. The built-in procedures include string analysis functions similar to those in Icon, type conversion and interrogation functions, and debugging functions. Examples include upto, many, and size. upto(s1, s2, i, j) returns the leftmost position in s2[i:j] where a character in s1 occurs. If none of the characters in s1 occur, no value is returned. i and j are optional and, if omitted, default to 1 and 0, respectively. many(s1, s2, i, j) operates similarly, but returns the position of the first character in s2[i:j] that does not appear in s1. size(x) returns the number of elements in a table or the number of characters in a string. For other types, x is converted to a string, and if the conversion fails, no value is returned.

Although explicit type conversion is rarely needed, built-in functions are provided so that run-time errors that occur when implicit conversions fail can be avoided. For example, numeric(x) returns the result of converting x to a real or an integer. If the conversion is not possible, no value is returned. The functions string, integer, and real are similar, and type(x) returns the type of x as a string (e.g. "string"), or "void" if x has no value.

External input and output are performed by the built-in procedures read and write. read(f) reads the next line from the external file f or from the standard input if f is omitted. write(f, ...) writes its arguments to the external file f or to the standard output if f is omitted.

The initial values of global variables ascii, lcase, and ucase are the ASCII character set, lower-case letters, and upper-case letters, respectively. The initial values of input, output, and errout are the standard input, the standard output, and the error output, respectively.

The value of the built-in variable root is an *EZ* table that *is* the global symbol table, which contains the built-in procedures and values described above. The compiler determines the meaning of free identifiers by searching this table, making insertions as necessary. For example, the assignment

```
section = "Introduction"
```

is equivalent to

```
root["section"] = "Introduction"
```

Changing root changes meanings of identifiers for subsequent compilation. Assuming t is a table containing a restricted set of built-in values,

```
root = t
```

would restrict subsequent programs to those values. Actually, the compiler searches a chain of tables beginning with root and continuing with root[".."] and so on until the identifier is found or a table with no ".." entry or whose ".." entry is not a table is encountered. If the identifier is not found by this search, it is inserted in root. As illustrated below, the user *and* applications programs may construct and modify these chains as desired.

3. Applications

Many operations performed in traditional systems by command interpreters and utility programs are trivial in *EZ*. For example,

```
size(old)
```

prints the length of a file, and

```
if (new ~== old)
   write("files are different\n")
```

determines if two files hold the same text. Traditional string-processing utilities are easily written in *EZ*. For example,

```
procedure wc(s)
   local nl, nw, i, wchrs

   wchrs = ascii[upto(" ", ascii)+1:-1]
   nl = nw = 0
   while (i = upto(wchrs || "\n", s))
      if (s[i!1] == "\n") {
         nl = nl + 1
         s = s[i+1:0]
         }
      else {
         nw = nw + 1
         s = s[many(wchrs, s, i):0]
         }
   return (nl || " " || nw)
end
```

counts the number of words and lines in a string (or file). For example,

```
write(size(old), " ", wc(old), "\n")
```

prints the number of characters, words, and lines in old. wchrs is a string containing the printable characters.

Tables provide a general directory structure. The indices are the 'names' and the values are the 'files', but both indices and values may be of arbitrary types. So, for example, 'help' documents for a set of procedures might be organized as a table indexed by the procedures themselves instead of their names, which might be ambiguous. Thus

```
procedure mail(...)
   ...
end
doc[mail] = "mail — receive mail"
```

defines and documents mail.

Many operations on directories that require utility programs in other environments are simple in *EZ*. For example,

```
size(paper)
```

prints the number of entries in the directory paper, and

```
for (i in paper)
   write(i, "\n")
```

lists the 'names' of the 'files' in it. Deleting a file is accomplished by the built-in function remove, which removes an element and its index from a table:

```
remove(paper, "oldabstract")
```

Since table entries can refer to values of arbitrary types, entries referring to other tables permit the construction of hierarchical directory structures. Actually, arbitrary graphs are permitted. Maintaining a focus of attention or 'current directory' amounts to maintaining a global variable whose value *is* the focus. For example,

```
cd = paper
```

moves the focus to 'directory' paper, and

```
cd["title"] = "EZ Programming"
```

creates a title 'file' in it. Additional entries can be made to facilitate moving about a directory structure much as in UNIX. For example,

```
cd["sections"][".."] = cd
```

causes ".." in directory sections to refer to its ancestor paper, and *creates* sections if necessary. Henceforth,

```
cd = cd[".."]
```

moves the focus up one level.

The generality of *EZ* types permits flexibility in directory manipulation. Consider the procedure chdir, which changes root to a different directory after establishing a path back to root.

```
procedure chdir(dir)
   root[dir]["last"] = root
   root = root[dir]
end
```

chdir maintains a graph of tables whose interconnections through the "last" entries represent the recent history of the focus. This mechanism represents dynamic relationships where the UNIX directory mechanism represents static ones. The latter relationships, other relationships, or combinations thereof could be represented using similar conventions and procedures. For example, if the table structure is a tree and if ".." is used in place of "last", the compiler resolves free identifiers by searching the 'path' from root to the root of the tree. This usage offers a facility similar to Smalltalk's nested classes [9]. Completely different schemes, such as maintaining the static structure, dynamic history, and contents in separate tables, are also possible.

Type fusion allows procedures to apply in multiple contexts. Consider the emerging *EZ* line editor. It initially scans a string (text file) and converts it to a table (directory) with one entry per 'line' of the file because it is cheaper to insert and delete entries in a table than it is to insert and delete characters in a string. Immediately, all procedures and utilities written to operate on files are available to edit individual lines in a file. For example, a character transliteration procedure can be applied selectively to specified lines of a file instead of to all lines indiscriminately.

Conversely, line editor procedures are available to edit structures other than text files [7]. Consider the editor's procedure to delete line n from a global table tbl, which slides up the remaining lines to fill the gap:

```
procedure Delete(n)
   if (integer(n))
      for (; n < size(tbl); n = n + 1)
         tbl[n] = tbl[n+1]
   remove(tbl, n)
end
```

Delete was written to remove a line from a file, but it can also be used remove a subdirectory from a directory. For example, as suggested above, a document might be structured as a tree, with the root containing a directory of several sections, with each section containing several paragraphs and figures, etc. The editor's procedure to delete lines of text from a file can also delete sections from a paper, paragraphs from a section, etc. If they were numbered, it also slides up their successors, which is as useful when deleting sections or paragraphs as when deleting lines.† Delete subsumes a text editor's line deletion command and a directory system's file deletion command.

The editor's insertion procedure is also useful in multiple contexts. It inserts a value v just before position n, sliding down the remaining entries if n specifies a spot in a numbered sequence:

```
procedure Insert(n, v)
   if (integer(n))
      for (i = size(tbl); i >= n; i = i - 1)
         tbl[i+1] = tbl[i]
   tbl[n] = v
end
```

Insert and Delete combine to offer many common text- and directory-editing functions.

```
Insert(3, read())
```

inserts new text, and

```
Insert(3, tbl[1])
```

makes a copy of line 1 before line 3. The sequence

```
Insert("new", tbl["old"])
Delete("old")
```

renames a file in a directory, and the analogous operation on lines in a file

```
Insert(3, tbl[1])
Delete(1)
```

'renames' line 1 to line 2, that is, moves line 1 to line 2.

These procedures implement the editor and are called by a general command interpreter. Different structures can be edited by binding different editing procedures to the same command interpreter. This is accomplished by compiling the command interpreter with different symbol tables. This generality and type fusion combine to allow otherwise separate functions to share a common implementation. Text editors have been extended to offer directory editing before [7, 23], but the new functions require new code. *EZ* represents both files and directories as tables and thus subsumes both editors with a single table editor. With fewer types, commands have wider applicability.

†Besides allowing structured documents, this representation also allows a clean separation of text from formatter parameters. For example, the text could be represented as a numbered table of strings with unnumbered table entries recording the number of lines of text, the indentation, etc.

4. Implementation

The current implementation of *EZ* is written in C and runs under UNIX. Many of the implementation techniques used are similar to those used in SNOBOL4 and Icon. The 'compilation' part of *EZ* is implemented using YACC [15] and associated traditional techniques. *EZ* programs are compiled into an interpretive code and executed accordingly.

Type fusion, the longevity of values, the applicative nature of strings (and files), and the potential for large values (e.g. tables and files), complicate the implementation of the execution-time environment. At the lowest level, storage is divided into fixed-size pages that reside on secondary storage. All values are represented by small, fixed-size *descriptors* that include the necessary data or point to pages organized according to type. All types of data can be paged to secondary storage. A large software cache is used to reduce paging activity. Non-resident strings, tables, and procedures correspond to the traditional concepts of text files, directories, and object or executable files.

Strings and text files are implemented as linked lists of substrings bounded by the page size. The distribution of strings is used to determine the page size; the intent is for most strings to fit on one page. In the current implementation, the page size is 128 bytes.

The exact representation of tables is adapted to their contents during execution [14]. Tables with small integer indices are organized much like UNIX files: the descriptor points to a page that points to other pages containing table elements or further pointers [25]. When a table with small integer indices is assigned an non-integer index or a large integer index, it is reorganized as a hash table. Both of these organizations efficiently accommodate tables with non-contiguous indices.

Other values are implemented in similar fashion. For example, compiled code is stored in linked lists of pages, and, like all other values, is paged in on demand during execution. Since all pages, including those containing root, ultimately reside on secondary storage, this approach facilitates saving the state of the system.

Secondary storage is allocated in units of pages and is accomplished by simply extending the external file by the requested amount. Secondary storage is reclaimed by an off-line process using a traditional garbage collection algorithm.

The straightforward implementation used in the initial version of *EZ* suffers from a few efficiency problems. The applicative nature of strings (and hence files), which has well established benefits, causes the worst of these problems. In the current implementation, after executing

```
a = "... a long string ..."
b = a
```

a and b logically refer to two different strings. Internally, however, they refer to the same string. Thus, executing

```
b[i:j] = "... another string ..."
```

which refers to the substring between positions i and j, causes a copy of the original string to be made in order to preserve the value of a. An alternative strategy, currently under investigation, is to use lazy evaluation to reduce the amount of copying necessary.

5. Conclusion

The unification of a high-level programming language and a command language is accomplished in *EZ* by fusing normally distinct linguistic mechanisms. Type fusion and the implementation of root as an *EZ* table exemplify this general principle. It is difficult to place examples of use into either the programming or command language category. Typical use, in which users maintain tables of procedures, blurs the distinction beyond the point of recognition. Such usage appears particularly promising for stand-alone systems on small computers. The current version of *EZ* has been implemented with this application in mind.

Initial experience with *EZ* has suggested additional fusion. Treating strings and files as a single type has proven very convenient. It is useful, in some circumstances, to include tables in this fusion. For example, the editor described above converts a file to a table. When editing is finished, the editor converts the table back to a file. In the *EZ* environment, these conversions appear artificial. Fusing tables with strings and files would simplify this kind of processing as well as directory processing. For example, scanning a directory could be accomplished by scanning all of the files in the directory, and many operations defined on files would generalize to tables. Similarly, automatic conversion from strings to procedures via compilation would blur the somewhat artificial distinction between source and object code. This facility would permit

```
while (read()())
    ;
```

to serve as a simple command interpreter much as in Lisp. Current research is focused on these kinds of fusion.

There are other control abstractions that might be useful in *EZ*. Three under active consideration are coroutines, history mechanisms, and goal-directed evaluation. Coroutines are the main control abstraction in a companion research effort in command languages [8], which seeks to decompose system utilities into their fundamental components, and have proven useful in other programming languages [13, 26]. Adding coroutines to *EZ* would make it an ideal language in which to write command language interpreters. History mechanisms and Icon's goal-directed expression evaluation offer retentive control facilities that add another dimension of programmability to *EZ*, especially in program development and maintenance applications.

The generality of *EZ* complicates its evaluation. For example, the use of root and its associated search strategy yield unusual 'scope rules' whose benefits and implications are yet to be completely understood. More use and experimentation with *EZ* is needed to determine its range of applications and future development.

Despite the embryonic state of the *EZ* environment, it is clear that using a high-level language as both a programming and command language shows great potential [20]. *EZ* also represents an approach to providing the flexibility and dynamic mechanisms so often advocated as essential to programming environments [10].

Acknowledgements. D. S. Chimenti implemented major components of the initial version of *EZ*, and R. L. Campbell and C. J. Lerner made improvements.

References

1. *IBM Virtual Machine Facility/370: EXEC User's Guide*, IBM Corp., Order No. GC20-1812.
2. 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.
3. D. Beech, Command Language Directions, *Proc. IFIP TC 2.7 Working Conf. on Command Languages*, Lund, Sweden, 1979.
4. S. R. Bourne, The UNIX Shell, *Bell System Tech. J. 57*, 6 (July 1978), 1971-1990.

5. P. Brinch Hansen, The Solo Operating System: Job Interface, *Software—Practice & Experience 6*, 2 (Apr. 1976), 151-164.

6. J. R. Ellis, A Lisp Shell, *SIGPLAN Notices 15*, 5 (May 1980), 24-34.

7. C. W. Fraser, A Generalized Text Editor, *Comm. ACM 23*, 3 (Mar. 1980), 154-158.

8. C. W. Fraser, A Software System and Command Language Based on Connecting Coroutines, Tech. Rep. 80-17, Dept. of Computer Science, The Univ. of Arizona, Tucson, AZ, June 1980.

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

10. J. W. Goodwin, Why Programming Environments Need Dynamic Data Types, *IEEE Trans. on Software Eng. SE-7*, 5 (Sep. 1981), 451-457.

11. R. E. Griswold, D. R. Hanson and J. T. Korb, Generators in Icon, *ACM Trans. Prog. Lang. and Systems 3*, 2 (Apr. 1981), 144-161.

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

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

14. D. R. Hanson, Data Structures in SL5, *J. Computer Lang. 3*, 3 (Oct. 1978), 181-192.

15. S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

16. A. K. Jones, The Narrowing Gap Between Language Systems and Operating Systems, *Proc. IFIPS 77*, Montreal, Canada, 1977, 869-873.

17. W. N. Joy, An Introduction to the C Shell, Tech. Rep., Computer Science Div., Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA, Nov. 1980.

18. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

19. J. Levine, Why a Lisp-Based Command Language?, *SIGPLAN Notices 15*, 5 (May 1980), 49-53.

20. J. R. Mashey, Using a Command Language as a High-Level Programming Language, *Proc. 2nd Int. Conf. on Software Eng.*, San Francisco, CA, Oct. 1976, 169-176.

21. J. H. Morris, E. Schmidt and P. Wadler, Experience with a String Processing Applicative Language, *Conf. Rec. 7th ACM Symp. on Prin. of Programming Languages*, Las Vegas, NV, Jan. 1980, 32-46.

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

23. R. M. Stallman, EMACS, The Extensible, Customizable Self-Documenting Display Editor, *Proc. ACM Symp. on Text Manipulation*, Portland, OR, June 1981, 147-156.

24. W. Teitelbaum, Interlisp Reference Manual, Tech. Rep., Xerox PARC, Palo Alto, CA, Dec. 1975.

25. K. Thompson, UNIX Implementation, *Bell System Tech. J. 57*, 6 (July 1978), 1931-1946.

26. N. Wirth, *Programming in Modula-2*, Springer Verlag, New York, NY, 1981.