The design of the REXX language

by M. F. Cowlishaw

One way of classifying computer languages is by two classes: languages needing skilled programmers, and personal languages used by an expanding population of general users. REstructured eXtended eXecutor (REXX) is a flexible personal language designed with particular attention to feedback from its users. It has proved to be effective and easy to use, yet it is sufficiently general and powerful to fulfil the needs of many demanding professional applications. REXX is system and hardware independent, so that it has been possible to integrate it experimentally into several operating systems. Here REXX is used for such purposes as command and macro programming, prototyping, education, and personal programming. This paper introduces REXX and describes the basic design principles that were followed in developing it.

Computer languages may be classified in many ways. One way, for example, is to divide them into two usability classes: those for data processing professionals and those for the rest of the users. Most languages currently available (such as FORTRAN, COBOL, and C) have been designed as tools for professionals and require a significant amount of training before they can be used effectively. A few languages (notably BASIC and LOGO) have been designed with more general users in mind. As a result, these languages have found wide application in the field of personal computers. BASIC especially is widely used, but it was originally designed for simpler applications. The popularity of BASIC continues, and there have been many attempts to improve its structure and syntax. This has resulted in many different dialects of the language.

REstructured eXtended eXecutor (REXX) is a new language designed for the general user yet suitable for many professional applications. REXX borrows significantly from earlier languages, but it differs in one fundamental respect. Instead of being designed (consciously or otherwise) to be easy to compile or easy to interpret, it is designed (with the help of feedback from hundreds of users) to be easy to use.

Three major factors affect the usability of a language. First, the basic concepts of a language affect its syntax, grammar, and consistency. Second, the history and development of a language determine its function, usability, and completeness. Third, but quite independently, the implementation of a language affects its acceptability, portability, and distribution. This paper introduces REXX and then discusses basic concepts and developmental history as applied to the design of the REXX language.

There are several experimental implementations of the REXX language within IBM for both large and small machines. One of these, by the author, has become a part of the Virtual Machine/System Product (VM/SP), as the System Product Interpreter for the Conversational Monitor System (CMS). The most complete published documentation of the language may be found in Reference 1.

What kind of language is REXX?

REXX is a new language that allows programs and algorithms to be written in a clear and structured way. Its primary design goal was that it should be genuinely easy to use both by computer professionals and by the more casual general users. A language that is designed to be easy to use must be adept at

manipulating the kinds of symbolic objects that people normally deal with: words, numbers, names, and so on. Most of the features in REXX are included to make this kind of symbolic manipulation easy. REXX is also designed to be highly system independent, but it has the capability of issuing both commands and conventional interlanguage calls to its host environment.

The REXX language structure covers several application areas that traditionally have been serviced by fundamentally different types of programming language.

Personal programming. REXX provides considerable function with powerful character and mathematical abilities in a simple framework. Short programs may be written with minimum overhead, yet facilities exist to allow the writing of robust large programs. The language is well suited to interpretation and is therefore rather suitable for the applications for which such languages as BASIC and LOGO are currently used.[2,3] REXX has proved to be an easy language to learn and to teach.

Command program interpreters are increasing in importance in modern operating systems.

Tailoring user commands. Command program interpreters are increasing in importance in modern operating systems. Nearly all operating systems include some form of EXEC, SHELL, or BAT languages.[4–7] In many cases such a language is so embedded into the operating system that it is unlikely to be of use outside its primary environment, as for example Mxec.[8] There is, however, a clear trend toward providing command programming languages that are both powerful and capable of more general usage.[9–12] REXX carries this principle further by being a language that is designed primarily for generality but also for suitability as a command programming language.

Within IBM, many REXX EXECs for the Conversational Monitor System (CMS) have been written. Many of these EXECs embody hundreds and even thousands of lines. Product models consisting of over 20 000 lines of REXX have been reported, and at least one IBM location now reports applications involving over one million lines of code written in REXX.[13]

Macros. Many applications are programmable by means of macros. In the data processing world, there is a different macro language for almost every type of application. There are macro languages for editors, assemblers, interactive systems, text processors, and, of course, for other languages. The work of Stephenson[14] and others has highlighted similarities between these applications and the need for a common language. Because REXX is essentially a character-manipulation language, it can provide the macro facility for all these applications.

Macro languages often have unusual qualities and syntax that restrict their use to skilled programmers. REXX has a more conventional syntax. It is also a flexible language. Thus, it allows the same jobs to be done in less time by less skilled personnel.

Prototyping. The current interpreter implementation of REXX can be highly interactive. Therefore, as might be expected, developing programs in REXX is very fast. This productivity advantage, together with the ease of interfacing REXX to system utilities for display and for data input and output, makes the language very suitable for modeling applications and products. It has also proved to be useful for setting up experimental systems for usability and human factors studies.

The design of REXX is such that the same language can be effectively and efficiently used for many different applications that would otherwise require the learning of several languages.

The REXX language

REXX is a language that is superficially similar to earlier languages. However, every aspect of REXX has been critically reviewed and usually differs from other languages in ways that make REXX more suited to general users. REXX was designed as an entirely new language, without the requirement to be compatible with any earlier language. This has allowed important improvements to be included. The following description is intended as an introduction to the language. Because many of the subtleties of REXX are

best appreciated with use, the reader is urged to use the language.

Language summary. The REXX language is composed of a rather small number of instructions and options, yet it is powerful. Where a desired function is not built in, it can be added easily by using one of the

All the operators act upon strings of characters of any length.

several mechanisms for external interfacing. The following summary introduces most of the features of REXX. Full details may be found in Reference 1.

REXX provides a conventional selection of *control constructs* that include IF-THEN-ELSE, SELECT-WHEN-OTHERWISE-END, and several varieties of DO-END for grouping and repetition. These constructs are similar to those of PL/I, but with several enhancements and simplifications. The DO looping construct can be used to step a variable TO some limit, FOR a specified number of iterations, and WHILE or UNTIL some condition is satisfied. DO FOREVER is also provided. Loop execution may be modified by LEAVE and ITERATE instructions that significantly reduce the complexity of many programs. A SIGNAL instruction is provided for abnormal outward transfer of control, such as error exits and computed branching.

REXX *expressions* are general in that any operator combinations may be used, provided of course that the data values are valid for those operations. There are nine arithmetic operators (including integer division, remainder, and exponentiation), three concatenation operators, eight comparative operators (including some that test for exact equality), and four logical operators. All the operators act upon strings of characters of any length, and the strings are typically limited only by the amount of virtual storage available.

Figure 1 shows a sample program, called HELLO, that illustrates both expressions and a conditional instruction. The expression on the last SAY (display) instruction concatenates the string 'Hello' to the variable ANSWER with a blank between them. The blank is here a valid operator that means *concatenate with blank*. The string "!" is then directly concatenated to the result built up so far. These simple concatenation operators make it very easy to build up strings and commands, and these operators may be freely mixed with arithmetic operations.

In REXX, any string or symbol may be a *number*. Numbers are all real numbers and may be specified in exponential notation if desired. An implementation may use appropriately efficient internal representations, of course. The arithmetic operations in REXX are completely defined, so that different implementations must always give the same results.

The NUMERIC instruction may be used to select the *arbitrary precision* of calculations, which, for example, may calculate with 1000 or more significant digits. The same instruction may also be used to set the *fuzz* to be used for comparisons, and the exponential notation (scientific or engineering) that REXX is to use to present results. The term *fuzz* refers to the number of significant digits of error permitted when making a numerical comparison.

Variables all hold strings of characters and cannot have aliases under any circumstances. The simple *compound variable* mechanism allows the use of multidimensional arrays that have the property of being indexed by arbitrary character strings. These are, in effect, content-addressable data structures and permit lists and trees to be built quite simply. Groups of variables (arrays) with a common stem to their names can be set, reset, or manipulated by references to that stem alone.

The example JUSTONE shown in Figure 2 is a routine that removes all duplicate words from a string of words. Figure 2 also shows some of the built-in *string parsing* available with the PARSE instruction. This instruction provides a fast and simple way of decomposing strings of characters (or data acquired from the user or external environment) using a primitive form of pattern matching. A string may be split into parts using various forms of patterns and then assigned to variables by words or as a whole.

A variety of internal and external *calling mechanisms* are defined. The most primitive calling mechanism is the *command*, which is similar to a *message* in the Smalltalk-80 system,[15] and in which an instruction that consists of just an expression is eval-

Figure 1 A sample program, called HELLO, illustrating expressions and a conditional instruction

```
/* A short program to greet a new user.              */
/* First display a prompt:                           */
say 'Please type your name and then press ENTER:'
parse pull answer      /* Get the reply into ANSWER */

/* If nothing was typed, then use a fixed greeting, */
/* otherwise echo the name politely.                 */
if answer='' then say 'Hello Stranger!'
            else say 'Hello' answer'!'
```

uated. The resulting string of characters is passed to the currently selected external environment, which might be an operating system, an editor, or any other functional object. The REXX programmer can also invoke *functions* and *subroutines* that may be internal to the program, built in (part of the language), or external to the program. Within an internal routine, variables may be shared with the caller or protected, that is, they may be local to the routine. If protected, selected variables or groups of variables belonging to the caller may be exposed to the routine for read/write access.

Certain types of *exception handling* are supported. A simple mechanism associated with the SIGNAL instruction allows the trapping of run-time errors, halt conditions (external interrupts), command errors (errors resulting from external commands), and the use of uninitialized variables. No method of return from an exception is provided in the current language definition.

The INTERPRET instruction, which is intended to be supported by interpreters only, allows any string of REXX instructions to be interpreted dynamically. It is useful for some kinds of interactive or interpretive environments, and can be used to build the almost trivial instant calculator program, called SAY, shown in Figure 3.

The language defines an extensive debugging or tracing facility, though it is recognized that some implementations may be unable to support the whole package. The tracing options allow various levels and subsets of instructions to be traced (commands, labels, all, and so on) and the display of various levels of expression evaluation results, either intermediate-calculation results or the final results. Furthermore, for a suitable implementation, the language describes an *interactive debug* option in which the execution of the program may be halted selectively. Once execution has paused, the user may then type in any REXX instruction string (to display/alter variables, and so on), step to the next pause, or re-execute the last clause traced.

Fundamental language concepts

Language design is always subtly affected by unconscious biases and by historical precedent. To mini-

Figure 2 The routine, called JUSTONE, removes all duplicate words from a string of words

```
/* This routine removes duplicate words from a string, and     */
/* illustrates the use of a compound variable (HADWORD) that   */
/* is indexed by arbitrary data (words).                       */
Justone:  procedure              /* make all variables private */
  parse arg wordlist                   /* get the list of words */
  hadword.=0                  /* show all possible words as new */
  outlist=''                       /* initialize the output list */
  do while wordlist¬=''    /* loop so long as we have some data */
    /* split WORDLIST into the first word and the remainder    */
    parse var wordlist word wordlist
    if hadword.word then iterate  /* loop again if already had */
    hadword.word=1       /* remember that we have had this word */
    outlist=outlist word   /* and add this word to output list */
    end
  return outlist                    /* finally return the result */
```

mize the effect of bias, a number of concepts have been chosen and used as guidelines for the design of the REXX language. Discussed here are the major concepts that were consciously followed during the design of REXX. Each topic merits a paper of its own, and many of these topics are well discussed in the literature. Unfortunately, these few paragraphs can be only summaries of fuller discussions and thoughts on the ideas.

Readability. If there is one concept that has dominated the evolution of REXX syntax it is *readability*, which is used here in the sense of perceived legibility. Readability in this sense seems to be a rather subjective quality, but the general principle followed in REXX is that the tokens that form a program can be written much as one might write them in English, French, German, and so forth. Although the semantics of REXX is of course more formal than that of a natural language, REXX is lexically similar to normal text.

The structure of the syntax means that the language readily adapts itself to a variety of programming styles and layouts. This helps satisfy user preferences and allows a familiarity of syntax that also increases

Figure 3 An instant calculator called SAY

```
/* Simple calculator, interprets input as a REXX expression */
numeric digits 20      /* Work to 20 significant digits     */
parse arg input        /* Get user's input into INPUT       */
interpret 'say' input  /* Build and execute SAY instruction */
```

readability. Good readability leads to enhanced understandability, thus yielding fewer errors during both the writing of a program and the reading for debug or maintenance. Important readability factors here are the following:

- There is deliberate support throughout the language for mixed upper- and lower-case letters, both for processing data and for the program itself.
- The essentially free format of the language and the way blanks around tokens are treated allow the user to lay out the program in the way he feels is most readable.
- Punctuation is required only when absolutely necessary to remove ambiguity (though it may often be added according to personal preference, so long as it is syntactically correct). This relatively tolerant syntax noticeably reduces frustration during use of the language, as compared with experience with such languages as Pascal.
- Modern concepts of structured programming are available in REXX and can lead to programs that are easier to read than they might otherwise be. Structured programming facilities also make REXX a good language for teaching the concepts of structured programming.
- Loose binding between lines and program source ensures that even though programs are affected by line ends, they are not irrevocably so. A user may spread a statement over several lines or put it on just one line. Statement separators are optional, except where more than one statement is placed on a line, again allowing the programmer to adjust the language to his style.

Natural data typing. *Strong typing,* in which the values a variable may take are tightly constrained, has become a fashionable attribute for languages over the last ten years. In this author's opinion, the greatest advantage of strong typing is for the interfaces between program modules. Errors within modules that would be detected by strong typing (and would not be detected from context) are much rarer and in the majority of cases do not justify the added program complexity.

REXX, therefore, treats types as naturally as possible. The meaning of a constant depends entirely on its usage. All data are defined in the form of the symbolic notation (strings of characters) that a user would normally write to represent the data. Since no internal or machine representation is exposed in the language, the need for many data types is reduced. There are, for example, no fundamentally different concepts of *integer* and *real.* There is just the single concept of *number.* Since all data have a defined symbolic representation, the programmer can always inspect values, such as, for example, the intermediate results of an expression evaluation. This means that numeric computations and all other operations can be precisely defined and therefore act consistently and predictably.

The current language definition does not exclude the future addition of a data-typing mechanism for those applications that require it, though at present there seems to be little call for this. The mechanism would be in the form of ASSERT-like instructions that assign data type checking to variables during execution flow. An optional restriction, similar to the existing trap for uninitialized variables, could be defined to provide enforced assertion for all variables.

Emphasis on symbolic manipulation. From the user's point of view, the data that REXX manipulates are in

the form of strings of characters. It is highly desirable for the user to be able to manage data as naturally as he would manipulate words on a page or in an editor. The language therefore has a rich set of character manipulation operators and functions.

Concatenation is treated specially in REXX. In addition to a conventional concatenate operator (| |), there is a new *blank operator* that concatenates two data strings together with a blank between. Furthermore, if two syntactically distinct terms, such as a string and a variable name, are abutted, the data strings are concatenated directly. These operators make it especially easy to build up complex data items and strings and may at any time be combined with the other operators available to the REXX programmer. To illustrate this point, consider the SAY instruction, which consists of the keyword SAY followed by any expression. In the following example of the instruction SAY, if the variable *N* has the value '6',

```
SAY N* 100/50'%' ARE REJECTS
```

displays the string

```
12% ARE REJECTS
```

Concatenation has a lower priority than arithmetic operators. The order of evaluation of the expression is therefore first the multiplication, followed by the division, then the direct concatenation, and finally the two concatenate-with-blank operations.

Dynamic scoping. Most languages, especially those designed to be compiled, rely on *static scoping*. That is, the physical position of a statement in the program source may alter its meaning. Languages that are interpreted or that have intelligent compilers generally have *dynamic scoping*. Here, the meaning of a statement is affected only by the statements that have already been executed, rather than those that precede it in the program source.

Purely dynamic scoping is a characteristic of the REXX language. Dynamic scoping implies that REXX may be efficiently interpreted because only minimal look-ahead is necessary. It also implies that a compiler is more difficult to implement. Therefore, the semantics includes restrictions that considerably ease the task of the compiler writer. Of greater importance is the fact that with dynamic scoping a person reading the program need only be aware of the program above the point at which he is studying. Not only does this aid comprehension, but it also makes programming and maintenance easier when only a display device is being used.

The GOTO statement is a necessary casualty of dynamic scoping. In a truly dynamically scoped language, a GOTO cannot be used as an error exit from a loop. If it were, the loop would never become inactive. Some interpreted languages detect control jumping outside the body of the loop and terminate the loop if this occurs. These languages are therefore relying on static scoping. REXX instead provides the abnormal transfer-of-control instruction SIGNAL that terminates all active control structures when it is executed. Note that it is not just a synonym for GOTO because it cannot be used to transfer control within a loop. Alternative instructions are provided for this purpose.

Implicit declarations take place during execution.

Nothing to declare. Consistent with the philosophy of simplicity, REXX provides no mechanism for declaring variables. Variables may of course be documented and initialized at the start of a program, and this covers the primary advantages of declarations. The other, data typing, is discussed earlier in this paper. Implicit declarations do take place during execution, but the only true declarations in the REXX language are the markers or labels identifying points in the program that may be used as the targets of signals or internal routine calls.

System independence. The REXX language is independent of both system and hardware. REXX programs, though, must be able to interact with their environment, and such interactions necessarily have system-dependent attributes. However, these system dependencies are clearly bounded, and the rest of the language has no such dependencies. In some instances, this leads to added expense in implementation and language usage, but the advantages are obvious and well worth the penalties.

As an example, string-of-characters comparison is normally independent of leading and trailing blanks.

The string " Yes " means the same as "Yes" in most applications. However, the influence of underlying hardware has subtly affected this kind of decision, so that many languages allow only trailing blanks but not leading blanks. By contrast, REXX permits both leading and trailing blanks during general comparisons.

Limited-span syntactic units. The fundamental unit of syntax in the REXX language is the *clause*, which is a piece of program text terminated by a semicolon, usually implied by the end of a line. The span of syntactic units is therefore small, usually one line or less. This means that the parser can rapidly detect errors in syntax, which in turn means that error messages can be both precise and concise.

It is difficult to provide good diagnostics in languages with large fundamental syntactic units, such as Pascal. A small error can often have a major and unexpected effect on the parser.

Dealing with reality. The REXX language is a tool for use by real people to do real work. Any tool must, above all, be reliable. In the case of a language, reliability means that it should do what the user expects. User expectations are generally based on prior experience, including the use of various programming and natural languages, and on the human ability to abstract and generalize concepts.

It is difficult to define exactly how to meet user expectations, but it helps to ask the question: Could there be a high *astonishment factor* associated with the new feature? If a feature is accidentally misapplied by the user and causes what appears to him to be an unpredictable result, that feature has a high astonishment factor and is therefore undesirable. If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature.

Another important attribute of a reliable software tool is *consistency*. A consistent language is by definition predictable, and it is often elegant. The danger here is to assume that because a rule is consistent and easily described, it is therefore simple for a user to understand. Unfortunately, some of the most elegant of rules can lead to effects that are completely alien to the intuition and expectations of a user. The user is a human being, not a computer.

Consistency applied for its own sake can easily lead to rules that are either too restrictive or too powerful for general use by human beings. Thus, during its design, I found that simple rules for REXX syntax often had to be rethought to make the language a more usable tool.

Originally, REXX allowed almost all options on instructions to be variable—even the names of functions were variable. Many users, however, stumbled into pitfalls that were side effects of this powerful generality. For example, the TRACE instruction allows its options to be abbreviated to a single letter, because it must be typed often during debugging sessions. Users therefore often used the instruction TRACE I. When I had been used as a variable, perhaps as a loop counter, the TRACE I instruction could become TRACE 10—a correct but unexpected action. Therefore, the TRACE instruction was changed to treat the symbol as a constant to protect users against such things happening. As a result, the language became more complex. A VALUE option on TRACE allows variability for the experienced user. Similarly, there is a fine line to tread between concise (terse) syntax and usability.

Adaptability. Wherever possible, the REXX language allows for the extension of instructions and other language constructs. For example, there is a large set of characters available for future extensions, because only a restricted set is allowed for the names of variables (symbols). Similarly, the rules for keyword recognition allow instructions to be added whenever required without compromising the integrity of existing programs that are written in the appropriate style. There are no globally reserved words, though a number of words are reserved within the local context of a single clause.

A language must be adaptable because it certainly will be used for applications not foreseen by the designer. Although it has proved to be effective as a command programming and personal language, REXX may prove to be inadequate in unforeseeable future applications. Thus room for expansion and change is included to make the language more adaptable.

Keep the language small. Every suggested addition to the language has been considered on the basis of its likely number of users. My intention was to keep the language as small as possible, so that users can rapidly grasp most of the language. This self-imposed guideline has had a number of beneficial results, among which are the following:

- The language appears less formidable to a new user.
- Documentation is smaller and simpler.

- The experienced user can be aware of all the facilities of the language, and so has the whole tool at his disposal to achieve a goal.
- There are few exceptions, special cases, and rarely used embellishments.
- The language is easier to implement.

No defined size or shape limits. The language does not define limits on the size or shape of any of its tokens or data, although there may be implementation restrictions. It does, however, define the minimum requirements that must be satisfied by an implementation. Wherever an implementation restriction has to be applied, the language rules recommend that it be of such a magnitude that few if any users are affected by the restriction.

Where implementation limits are necessary, the language encourages the implementer to use familiar and memorable values for the limits. For example, 250 is preferable to 255, 500 is preferable to 512, and so on. It is unnecessary to force artifacts of the binary system onto a population that uses only the decimal system. Only a tiny minority of future programmers will deal with binary representations of quantities.

Language design concepts

The REXX language was designed over the four-year period from 1979 through 1982, at the IBM United Kingdom Laboratories Limited at Hursley, England, and at the IBM Thomas J. Watson Research Center at Yorktown Heights, New York. The process was first to design and document a basic REXX language. This initial informal specification was then circulated for review and critique. On the basis of advice received, I revised the initial informal description, which became the basis for a specification and implementation. REXX was first implemented under the Conversational Monitor System (CMS), which supported the concept of interpreted programs that could be directly invoked by users.

The most important factor in the development of REXX began to take effect when the first interpreter was distributed over the IBM communication network known as VNET. (This network links over 1400 mainframe computers in forty countries.) From the beginning, many hundreds of people were using the language. All these users, from temporary staff to professional programmers, were able to provide immediate feedback to the designer on their preferences, needs, and suggestions for change. An informal language committee then appeared spontaneously and communicated among themselves and with the designer entirely electronically. The discussions of the committee grew to be hundreds of thousands of lines, and these and the similar quantity of mail from the users were all kept for later review.

As time passed, it became clear that changes in the language were necessary. Using the network, the designer could interactively explain and discuss the changes that were required, some of which were incompatible with the then-current version of the language. The decision to make an incompatible change was never taken lightly, but—because changes could be made relatively easily and explained to users in detail—the language was able to evolve much further than would have been the case if upward compatibility only were considered. Several other important concepts guided the process of enhancing the language.

Documentation before implementation. Each major section of the REXX language was documented and circulated for review before its implementation. These sections were in the form of complete reference documentation that in due course became part of the language reference manual. At the same time, and before implementation, sample programs were written to explore the usability of each proposed new feature.

The benefits of this approach were marked:

- The majority of usability problems were discovered before they became embedded in the language or before any implementation of the language included them.
- The writing of documentation was found to be the most effective way of spotting inconsistencies, ambiguities, or incompleteness in a design.
- The designer did not consider implementation details until the documentation was complete, so as to minimize the implementation's influence upon the language.
- Reference documentation written after implementation is much more likely to be inaccurate or incomplete than that written before implementation. After the documentation has been written, the author is likely to know the implementation too well to write an objective description.

User feedback. User feedback was fundamental to the process of evolution of the REXX language. Although users can often be incorrect in their suggestions, even those suggestions that appeared to be

shallow were considered carefully because they were often pointers to deficiencies in the language or documentation. As a result of the effective communications network, many details of the language and documentation could be revised and circulated efficiently. Many if not most of the good ideas embodied in the language came directly from users. It is impossible to overestimate the value of the direct feedback from users during the development of REXX.

Concluding remarks

REXX is designed to be a practical and powerful language, intended to provide maximum effect for the minimum of effort on the part of the programmer. Close attention to the details of syntax and semantics has resulted in many differences from earlier languages, as well as many similarities. The crucial concept, however, is that the language has been designed for the user, not the implementer. This emphasis is particularly visible in the areas of readability, natural data typing, and representation of data. In addition to being easy to learn and to use, the language contains sufficiently powerful constructs that it satisfies the needs of many professional applications. In addition to its use as a personal language, a variety of major programming tasks have been accomplished using REXX, including product prototypes, macro libraries, and command programming.

The REXX language has benefited especially from wide usage and feedback during its development. The advantages of user experience and feedback have far outweighed the problems caused by occasional incompatibilities. The value to language design of a worldwide telecommunications network connecting language users cannot be overestimated.

Acknowledgments

Many hundreds of people have contributed to the development of the REXX language in one way or another, so it is impossible to thank them all. However, I am especially indebted to several individuals who have spent much time discussing REXX and have made major contributions: P. G. Capek, C. W. Christensen and the REXX Language Committee, S. Davies (who also designed and implemented most of the built-in functions), M. Hack, S. Nash, C. J. Stephenson, and C. H. Thompson. I also thank the management of the IBM United Kingdom Limited Scientific Centre and the IBM United Kingdom Laboratories Limited for the opportunity to write this paper.

Cited references

1. *Virtual Machine/System Product: System Product Interpreter Reference,* IBM Reference Manual SC24-5239, IBM Corporation; available through IBM branch offices.
2. T. E. Kurtz, "BASIC," *ACM SIGPLAN Notices* **13,** No. 8, 103–118 (August 1978); *ACM SIGPLAN History of Programming Languages Conference,* Los Angeles, CA, June 1–3, 1978.
3. B. Harvey, "Why Logo?", *Byte* **7,** No. 8, 163–193 (August 1982).
4. *Virtual Machine/System Product: EXEC 2 Reference,* IBM Reference Manual SC24-5219, IBM Corporation; available through IBM branch offices.
5. S. R. Bourne, "The UNIX shell," *Bell System Technical Journal* **57,** No. 6, 1971–1990 (July 1978).
6. *Wylbur Command Procedures,* Computer Center, National Institutes of Health, Bethesda, MD 20205 (December 1980).
7. *IBM Personal Computer Disk Operating System, Version 2,* IBM Corporation; available through authorized IBM personal computer dealers.
8. W. L. Ash, "Mxec: Parallel processing with an advanced macro facility," *Communications of the ACM* **24,** No. 8, 502–509 (August 1981).
9. J. R. Mashey, "Using a command language as a high-level programming language," *IEEE, Proceedings of the Second International Conference on Software Engineering,* San Francisco, CA (October 13–15, 1976), pp. 169–176.
10. A. W. Colijn, "Experiments with the Kronos control language," *Software—Practice and Experience* **6,** No. 1, 133–136 (1976).
11. J. Levine, "Why a Lisp-based command language?", *ACM SIGPLAN Notices* **15,** No. 5, 49–53 (May 1980).
12. A. W. Colijn, "A note on the Multics command language," *Software—Practice and Experience* **11,** No. 7, 741–744 (July 1981).
13. E. C. Haeckel, IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, CA 95150, personal communication (1984).
14. C. J. Stephenson, "On the structure and control of commands," *ACM SIGOPS* **7,** No. 4, 22–26, 127–136 (1973).
15. Xerox Learning Research Group, "The Smalltalk-80 system," *Byte* **6,** No. 8, 36–47 (August 1981).

Reprint Order No. G321-5228.

Mike Cowlishaw *IBM United Kingdom Limited Scientific Centre, Athelstan House, St. Clement Street, Winchester, Hants, SO23 9DR, England.* Mr. Cowlishaw joined the IBM United Kingdom Laboratories Limited at Hursley in 1974, after receiving a B.Sc. in electronic engineering from the University of Birmingham. From then until 1980, he worked on the design of the hardware and software of multimicroprocessor test tools. He spent any spare time exploring the concept of the human-machine interface. This included the implementation of the Structured Editing Tool (STET), which is an editor that gives a tree-like structure to programs or documentation. Other results of these spare-time explorations are several compilers and assemblers and the first version of the REXX language. In 1980, Mr. Cowlishaw joined the IBM Thomas J. Watson Research Center to work on a text display system with real-time formatting and on specifications for new facilities for interactive operating systems. He returned in 1981 to the Hursley Laboratory, where he completed work on the REXX language. In 1982, Mr. Cowlishaw joined the IBM United Kingdom Scientific Centre in Winchester to do research on image systems. His current research is concerned with color perception and the modeling of brain mechanisms.