THE DESIGN OF REXX

MIKE COWLISHAW
IBM

The Design of REXX

Mike Cowlishaw

IBM UK Laboratories, Winchester, UK

Introduction

REXX is a flexible personal language that was designed with particular attention to feedback from users. The electronic environment used for its development has evolved a tool that seems to be effective and easy to use, yet is sufficiently general and powerful to fulfil the needs of many professional applications. As a result REXX is very widely used in IBM, and has been implemented for a variety of operating systems and machines.

The philosophy of the REXX language reflects the environment in which it was developed. A strong emphasis on readability and usability means that the language itself provides a programming environment that encourages high productivity while reducing the occurrence of errors.

REXX is useful for many applications, including command and macro programming, prototyping, and personal programming. It is a suitable language for teaching the principles of programming, since it includes powerful control constructs and modern data manipulation. It lets the student concentrate on the algorithms being developed rather than on language mechanics.

The REXX programming language has been designed with just one objective. It has been designed to make programming easier than it was before, in the belief that the best way to encourage high quality programs is to make writing them as simple and as enjoyable as possible. Each part of the language has been devised with this in mind; providing a programming language that is by nature comfortable to use is more important than designing for easy implementation.

The first section of this paper introduces the REXX language, and the other two sections describe the concepts and design environment that shaped the language.

Summary of the Language

REXX is a language that is superficially similar to earlier languages. However, most aspects of the language differ from previous designs in ways that make REXX more suited to general users. It was possible to make these improvements because REXX was designed as an entirely new language, without the requirement that it be compatible with any earlier design.

The structure of a REXX program is extremely simple. This sample program, TOAST, is complete, documented, and executable as it stands.

TOAST

```
/* This wishes you the best of health. */
say 'Cheers!'
```

TOAST consists of two lines: the first is a comment that describes the purpose of the program, and the second is an instance of the SAY instruction. SAY simply displays the result of the expression following it – in this case a literal string.

Of course, REXX can do more than just display a character string. Although the language is composed of a small number of instructions and options, it is powerful. Where a function is not built-in it can be added by using one of the defined mechanisms for external interfaces.

The rest of this section introduces most of the features of REXX.

REXX provides a conventional selection of *control constructs*. These 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. No GOTO instruction is included, but a SIGNAL instruction is provided for abnormal 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 9 arithmetic operators (including integer division, remainder, and power operators), 3 concatenation operators, 12 comparative operators, and 4 logical operators. All the operators act upon strings of characters, which may be of any length (typically limited only by the amount of storage available).

This sample program shows both expressions and a conditional instruction:

GREET

```
/* A short program to greet you.                          */
/* 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'!'
```

The expression on the last SAY (display) instruction concatenates the string **'Hello'** to the variable ANSWER with a blank in between them (the blank is here a valid operator, meaning "concatenate with blank"). The string **'!'** is then directly concatenated to the result built up so far. These simple and unobtrusive concatenation operators make it very easy to build up strings and commands, and may be freely mixed with arithmetic operations.

In REXX, any string or symbol may be a *number*. Numbers are all "real" 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 (you may calculate with one thousand significant digits, for example). 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 will 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 arrays (many-dimensional) that have the property of being indexed by arbitrary character strings. These are in effect content-addressable data structures, which can be used for building lists and trees. Groups of variables (arrays) with a common stem to their name can be set, reset, or manipulated by references to that stem alone.

This example is a routine that removes all duplicate words from a string of words:

JUSTONE

```
/* This routine removes duplicate words from a string, and  */
/* illustrates the use of a compound variable (HADWORD)     */
/* which 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 while we have data */
    /* split WORDLIST into the first word and the remainder */
    parse var wordlist word wordlist
    if hadword.word then iterate /* loop if had word before */
    hadword.word=1     /* record that we have had this word */
    outlist=outlist word   /*  add this word to output list */
    end
  return outlist                /* finally return the result */
```

This example also shows some of the built-in *string parsing* available with the PARSE instruction. This provides a fast and simple way of decomposing strings of characters 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 is the *command* (which is similar to a *message* in the Smalltalk-80[1] system), in which a clause that consists of just an expression is evaluated. 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*. These may be internal to the program, built-in (part of the language), or external. Within an internal routine, variables may be shared with the caller, or protected by the PROCEDURE instruction (that is, be made local to the routine). If protected, selected variables or groups of variables belonging to the caller may be exposed to the routine for read or 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 this language definition.

The INTERPRET instruction (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 following SHOWME program – an almost trivial "instant calculator":

---

[1] See, for example: Xerox Learning Research Group, **The Smalltalk-80 system,** *Byte* **6**, No. 8, pp36-47 (August 1981).

SHOWME

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

This program first sets REXX arithmetic to work to 20 digits. It then assigns the first argument string (perhaps typed by a user) to the variable INPUT. The final instruction evaluates the expression following the keyword INTERPRET to build a SAY instruction which is then executed. If you were to call this program with the argument "**22/7**" then the instruction "**Say 22/7**" would be built and executed. This would therefore display the result

```
3.1428571428571428571
```

Input and *output* functions in REXX are defined only for simple character-based operations. Included in the language are the concepts of named character streams (whose actual source or destination are determined externally). These streams may be accessed on a character basis or on a line-by-line basis. One input stream is linked with the concept of an *external data queue* that provides for limited communication with external programs.

The language defines an extensive *tracing* (debugging) mechanism, though it is recognised that some implementations may be unable to support the whole package. The tracing options allow various subsets of instructions to be traced (Commands, Labels, All, and so on), and also control the tracing of various levels of expression evaluation results (intermediate calculation results, or just the final results). Furthermore, for a suitable implementation, the language describes an *interactive tracing* environment, in which the execution of the program may be halted selectively. Once execution has paused, you may then type in any REXX instructions (to display or 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 minimize these effects a number of concepts were chosen and used as guidelines for the design of the REXX language. The following list includes the major concepts that were consciously followed during the design of REXX.

Readability

If there is one concept that has dominated the evolution of REXX syntax, it is *readability* (used here in the sense of perceived legibility). Readability in this sense is a rather subjective quality, but the general principle followed in REXX is that the tokens which form a program can be written much as one might write them in European languages (English, French, 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 lexical familiarity that also increases readability. Good readability leads to enhanced understandability, thus yielding fewer errors both while writing a program and while reading it for debug or maintenance. Important factors here are:

1. There is deliberate support throughout the language for upper and lower case letters, both for processing data and for the program itself.

2. The essentially free format of the language (and the way blanks are treated around tokens and so on) lets you lay out the program in the style that you feel is the most readable.

3. 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 proves less frustrating than the syntax of languages such as Pascal.

4. Modern concepts of structured programming are available in REXX, and can undoubtedly lead to programs that are easier to read than they might otherwise be. The structured programming constructs also make REXX a good language for teaching the concepts of structured programming.

5. Loose binding between lines and program source ensure that even though programs are affected by line ends, they are not irrevocably so. You may spread a clause over several lines or put it on just one line. Clause separators are optional (except where more than one clause is put on a line), again letting you adjust the language to your own preferred style.

Natural data typing

"Strong typing", in which the values that a variable may take are tightly constrained, has become a fashionable attribute for languages over the last ten years. I believe that the greatest advantage of strong typing is for the interfaces between program modules, where errors may be difficult to catch. 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 data depends entirely on its usage. All values are defined in the form of the symbolic notation (strings of characters) that a user would normally write to represent that 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*. The results of all operations have a defined symbolic representation, so you can always inspect values (for example, the intermediate results of an expression evaluation). Numeric computations and all other operations are precisely defined, and will therefore act consistently and predictably for every correct implementation.

This language definition does not exclude the future addition of a data typing mechanism for those applications that require it, though 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

The values that REXX manipulates are (from the user's point of view, at least) in the form of strings of characters. It is extremely desirable to be able to manage this data as naturally as you would manipulate words in other environments, such as on a page or in a text 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 novel *blank operator* that concatenates two data strings together with a blank in between. Furthermore, if two syntactically distinct terms (such as a string and a variable name) are abutted, then the data strings are concatenated directly. These operators make it especially easy to build up complex character strings, and may at any time be combined with the other operators available.

For example, the SAY instruction consists of the keyword SAY followed by any expression. In this instance of the instruction, if the variable **N** has the value **'6'** then

```
say n*100/50'%' ARE REJECTS
```

would display the string

```
12% ARE REJECTS
```

Concatenation has a lower priority than the arithmetic operators. The order of evaluation of the expression is therefore first the multiplication, then 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, where the physical position of an instruction 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 an instruction is only affected by the instructions that have already been executed (rather than those that precede or follow it in the program source).

REXX scoping is purely dynamic. This implies that it may be efficiently interpreted because only minimal look-ahead is needed. It also implies that a compiler is harder to implement, so the semantics includes restrictions that considerably ease the task of the compiler writer. Most importantly, though, it implies that a person reading the program need only be aware of the program *above* the point which is

being studied. Not only does this aid comprehension, but it also makes programming and maintenance easier when only a computer display terminal is being used.

The GOTO instruction is a necessary casualty of dynamic scoping. In a truly dynamic 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 an "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 since it cannot be used to transfer control within a loop (for which alternative instructions are provided).

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 above.

Implicit declarations do take place during execution, but the only true declarations in the REXX language are the markers (*labels*) that identify 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. 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 cases this leads to added expense in implementation (and in 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 only allow 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 for languages (such as Pascal and its derivatives) that have large fundamental syntactic units. For these languages, a small error can often have a major and unexpected effect on the parser.

Dealing with reality

A computer 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 this 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.

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 this feature?". If a feature, accidentally misused, gives apparently unpredictable results, then it has a high astonishment factor and is therefore undesirable.

Another important attribute of a reliable software tool is *consistency.* A consistent language is by definition predictable and is often elegant. The danger here is to assume that because a rule is consistent and easily described, it is therefore simple to understand. Unfortunately, some of the most elegant rules can lead to effects that are completely alien to the intuition and expectations of a user; who, after all, is human.

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

Originally, REXX allowed almost all options on instructions to be variable (and even the names of functions were variable), but many users fell into the pitfalls that were the side-effects of this powerful generality. For example, the TRACE instruction allows its options to be abbreviated to a single letter (as it needs to be typed often during debugging sessions). Users therefore often used the instruction "TRACE I", but when "I" had been used as a variable (perhaps as a loop counter) then this instruction could become "TRACE 10" – a correct but unexpected action. The TRACE instruction was therefore changed to treat the symbol as a constant (and the language became more complex as a consequence) to protect users against such happenings. A VALUE option on TRACE allows variability for the experienced user. There is a fine line to tread between concise (terse) syntax and usability.

Be adaptable

Wherever possible the language allows for extension of instructions and other language constructs. For example, there is a large set of characters available for future extensions, since 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 few are reserved within the local context of a single clause).

A language needs to be adaptable because *it certainly will be used for applications not foreseen by the designer.* Although proven effective as a command programming and personal language, REXX may (indeed, probably will) prove inadequate in certain future applications. Room for expansion and change is included to make the language more adaptable.

Keep the language small

Every suggested addition to the language was considered only if it would be of use to a significant number of users. My intention has been to keep the language as small as possible, so that users can rapidly grasp most of the language. This means that:

- The language appears less formidable to the new user.
- Documentation is smaller and simpler.
- The experienced user can be aware of all the abilities of the language, and so has the whole tool at his disposal to achieve results.
- There are few exceptions, special cases, or 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, it is recommended that it should be of such a magnitude that few (if any) users will be affected.

Where implementation limits are necessary, the language encourages the implementer to use familiar and memorable values for the limits. For example 250 is preferred to 255, 500 to 512, and so on. There is no longer any excuse for forcing the artifacts of the binary system onto a population that uses only the decimal system. Only a tiny minority of future programmers will need to deal with base-two-derived number systems.

History and Design Principles

The REXX language (originally called "REX") borrows from many earlier languages; PL/I, Algol, and even APL have had their influences, as have several unpublished languages that I developed during the 1970's. REXX itself was designed as a personal project in about four thousand hours during the years 1979 through 1982, at the IBM UK Laboratories near Winchester (England) and at the IBM T. J. Watson Research Center in New York (USA). As might be expected REXX has an international flavour, with roots in both the European and North American programming cultures.

There are several implementations of the REXX language available from IBM, for both large and small machines. My own System/370 implementation has become a part of the Virtual Machine/System Product, as the System Product Interpreter for the Conversational Monitor System (CMS), and is also part of the TSO/E product. This implementation of the language is described in the Reference Manuals for these products. A different IBM implementation, written in C, provides a subset of the language as part of the IBM PC/VM Bond product, running on various models of the IBM Personal Computer. In 1989, the IBM VM REXX Compiler for CMS was announced, and also

REXX for OS/2. The AS/400 version – completing the four SAA implementations – was added in 1990.

There are now many other implementations of REXX and, in 1991, the process of ANSI standardization was started.

The design process for REXX began in a conventional manner. The REXX language was first designed and documented; this initial informal specification was then circulated to a number of appropriate reviewers. The revised initial description then became the basis for the first specification and implementation.

From then on, other less common design principles were followed, strongly influenced by the development environment. The most significant was the intense use of a communications network, but all three items in this list have had a considerable influence on the evolution of REXX.

Communications

Once an initial implementation was complete, the most important factor in the development of REXX began to take effect. IBM has an internal network, known as VNET, that now links over 3100 main-frame computers in 58 countries. REXX rapidly spread throughout this network, so from the start many hundreds of people were using the language. All the users, from temporary staff to professional programmers, were able to provide immediate feedback to the designer on their preferences, needs, and suggestions for changes. (At times it seemed as though most of them did – at peak periods I was replying to an average of 350 pieces of electronic mail each day.)

An informal language committee soon appeared spontaneously, communicating entirely electronically, and the language discussions grew to be hundreds of thousands of lines.

On occasions it became clear as time passed that incompatible changes to the language were needed. Here the network was both a hindrance and a help. It was a hindrance as its size meant that REXX was enjoying very wide usage and hence many people had a heavy investment in existing programs. It was a help because it was possible to communicate directly with the users to explain why the change was necessary, and to provide aids to help and persuade people to change to the new version of the language. The decision to make an incompatible change was never taken lightly, but because changes could be made relatively easily the language was able to evolve much further than would have been the case if only upwards compatible extensions were considered.

Documentation before implementation

Every major section of the REXX language was documented (and circulated for review) before implementation. The documentation was not in the form of a functional specification, but was instead complete reference documentation that in due course became part of this language definition. At the same time (before implementation) sample programs were written to explore the usability of any proposed new feature. This approach resulted in the following benefits:

- The majority of usability problems were discovered before they became embedded in the language and before any implementation included them.
- Writing the documentation was found to be the most effective way of spotting inconsistencies, ambiguities, or incompleteness in a design. (But the documentation must itself be complete, to "final draft" standard.)
- I deliberately did not consider the implementation details until the documentation was complete. This minimized the implementation's influence upon the language.
- Reference documentation written after implementation is likely to be inaccurate or incomplete, since at that stage the author will know the implementation too well to write an objective description.

The language user is usually right

User feedback was fundamental to the process of evolution of the REXX language. Although users can be unwise in their suggestions, even those suggestions which appeared to be shallow were considered carefully since they often acted as pointers to deficiencies in the language or documentation. The language has often been tuned to meet user expectations; some of the desirable quirks of the language are a direct result of this necessary tuning. Much would have remained unimproved if users had had to go though a formal suggestions procedure, rather than just sending a piece of electronic mail directly to me. All of this mail was reviewed some time after the initial correspondence in an effort to perceive trends and generalities that might not have been apparent on a day-to-day basis.

Many (if not most) of the good ideas embodied in the language came directly or indirectly from suggestions made by users. It is impossible to overestimate the value of the direct feedback from users that was available while REXX was being designed.

Conclusions

A vital part of the environment provided to programmers is the programming language itself. Most of our programming languages have, for various historical reasons, been designed for the benefit of the target machines and compilers rather than for the benefit of people. As a result they are more demanding of the programmer than they need be, and this often leads to errors.

REXX is an attempt to redress this balance; it is designed specifically to provide a comfortable programming environment. If the user – the programmer – finds it easy to program, then fewer mistakes and errors are made.