SLAC-379
CONF-9105170
UC-405
(M)

PROCEEDINGS OF THE REXX SYMPOSIUM FOR DEVELOPERS AND USERS

May 8–9, 1991
Asilomar, Pacific Grove, California

Sponsored by
STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94309

Cathie Dager
Symposium Organizer

Dave Gomberg
Site Organizer

Bebo White
Program Advisor

Prepared for the Department of Energy
under Contract number DE-AC03-76SF00515

Printed in the United States of America. Available from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal road, springfield, Virginia 22161. Price: Printed Copy A11, Microfiche A01.

PROCEEDINGS OF THE REXX SYMPOSIUM
FOR DEVELOPERS AND USERS

TABLE OF CONTENTS

SUMMARY

Glancing through these Proceedings of the 2nd REXX Symposium, one cannot help but be impressed by the progress and accomplishments since the meeting at SLAC last year. The participants of the Asilomar meeting (and many more who could not attend) show a dedication to the development of the REXX language and to the contributions which they believe it can make to the computing community at large.

In the past year we have seen:

* the beginning of a REXX standardization effort;

* the emergence of REXX on a wider variety of computing platforms and supporting multiple operating systems;

* an increasing awareness and interest in REXX as demonstrated by activity in the Usenet group "comp.lang.rexx" and the Listserv list "REXX-L";

* beginning interest in the formation of an international REXX Users Group.

This year's symposium had something for everybody—from the humor of Linda Green's talk on "REXXoids" to the demonstrations of the roles that REXX can play in "cutting edge" programming technologies such as Object-Oriented Programming (Simon Nash) and Expert Systems (Marc Irwin). The panel discussion by REXX implementers showed the diversity with which common problems were solved on the various platforms represented. With UNIX expected to play such a major role in systems of the '90s, the UNIX panel discussion concentrated on the inroads REXX is making in that arena. Both panel discussions invoked considerable audience participation.

The Symposium was also delighted to welcome back Mike Cowlishaw. His insights and expertise always contribute greatly.

We all owe a debt of gratitude to those persons who contributed so much to the technical content of this year's symposium. Their work is presented in these proceedings for closer examination and reference.

Work has already begun on next year's Symposium. All persons willing to help in any capacity should make themselves known. The success of the Asilomar Symposium will be a tough act to follow, but the dedication of the REXX users, developers and implementers will make it happen.

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.

REXX ORACLE INTERFACE

GEORGE CRANE
SLAC

SLAC DB ENVIRONMENT

Currently support three mainframe database systems.

SPIRES	Stanford university product, hierarchical, acquired 1970, library, inventory etc.
NOMAD2	Must Software, acquired 1985, accounting type applications
ORACLE	Acquired 1989, all platforms, variety of applications.

INTERFACE TOOLS

* All 3 DB products allow 3GL type interfaces via FORTRAN, PL/1, etc.

    - In reality (at least in the SLAC environment) they are used very little.

* SPIRES additionally allows SUBCOM entry commands, very useful.

    - Issue any SPIRES command with Address SPIRES '....'

    - Data can be exchanged via stack, files, or Xedit in both directions.

* Pro*Rexx for Oracle.

USER VIEW OF THE WORLD

* Not everyone wants to learn a DB procedural language.

* Existing Rexx exec's can be enhanced with DB capabilities.

* Comfort level is higher within Rexx

* Development faster than with 3GL

* Large production systems can more easily migrate.

PRO*REXX IMPLEMENTATION

* Command driven interface between Rexx and Oracle

* Runs under VM/SP, VM/HPO, VM/XA

* No 3GL language, No pre-compilers, No compilers, No linking , much less confusing.

* Communicates with Rexx applications through Rexx variables.

* Has built-in "control" RDBMS variables structures which are always available after a database connect.

* Certain SQL statements are optimized to return data in Rexx arrays such as SELECT and FETCH.

PRO*REXX IMPLEMENTATION

* Implementation almost identical to RX/SQL for SQL/DS.

* Static and Dynamic SQL statements - allows building of SQL statement at runtime.

* Supports concurrent Oracle connections & access to remote data bases via SQL*Net.

* No data conversions are performed by Pro*Rexx, data is passed in CHAR format to and from Oracle.

* All necessary RDBMS entities are "in line", what you see is what you execute.

EXAMPLE

```
/* Dynamic command to retrieve data */

RXS CONNECT userid/password

RXS PREP stmt SELECT * FROM table

RXS OPEN stmt

DO

        RXS FETCH stmt INTO OUTPUT.

END

RXS CLOSE stmt

RXS COMMIT WORK RELEASE
```

HOW SLAC IS USING PRO*REXX

* As an interface tool between SPIRES, NOMAD and ORACLE

  - Refresh Oracle tables using a SPIRES table.

  - Update Oracle tables from data collected via non-Oracle screen tools such as Xmenu, GDDM and Xedit macros when database editors such as SQL*Forms are not practical.

  - As user exit routine via SQL*Forms to issue SQL commands normally prohibited in Oracle forms.

* To update Oracle tables on multiple platforms.

ADVANTAGES

* Multiple product functionality.

* Eliminates need for multiple components.

* Maintenance, documentation, debugging.

* Work in familiar territory.

ISSUES IN THE SPECIFICATION OF REXX

CHARLES DANEY
QUERCUS SYSTEMS

Issues in the specification of REXX

Charles Daney
Quercus Systems
P. O. Box 2157
Saratoga, CA 95070
(408) 867-REXX

REXX is, for the most part, clearly and thoroughly specified in *The REXX Language*. This presentation deals with a few areas which have been found to be less completely specified.

File I/O facilities

- File "opening"

  Many operating systems require an explicit "open" operation before performing file I/O functions. This operation is important for specifying file sharing and processing options. REXX permits an open to be done in the STREAM() function, but does not specify any standard syntax for widely applicable options.

- File read/write pointers

  REXX specifies that independent read and write pointers shall be maintained for I/O positioning. But the specification isn't sufficiently clear and emphatic, with the result that significant implementations, such as IBM's OS/2 REXX, do not maintain independent pointers. This can cause serious and difficult to detect problems when applications are ported to different platforms.

- File "closing"

  Although LINEOUT() and CHAROUT() can be used to close a file, their use is not intuitive for input files. In addition, the specific syntax used makes it impossible to close the default input stream.

- Ambiguities in LINES() and CHARS()

  These functions permit an application to determine whether any data remains to be read in an input stream. It is recognized that a given file system may have difficulty implementing both precisely. But there is no way for an application to determine when the results may be inexact. The behavior of these functions on "transient" streams needs to be clarified to distinguish whether a value of 0 means no data currently available, or whether an "end of file" indication has been received. (This is an issue, for instance, with "pipes".) A related problem is to specify when CHARIN() may return with fewer than the requested number of characters (as opposed to when it should wait).

- Lack of STREAM() standardization

  The STREAM() function was recently added to the language. While it does provide a means of performing certain I/O operations like opening, closing, seeking, and obtaining file information, it uses an un-REXX-like command syntax. In addition, because the exact command syntax isn't specified, STREAM() is useless for writing portable applications.

- Lack of file maintenance functions

  There should be standard functions for common file maintenance operations, such as "create", "delete", "rename", and the like. Most systems where REXX is currently implemented also support the hierarchical file directory concept, and REXX needs analogous standard maintenance functions for directories as well.

- Implementation capability determination

  While it is clear that implementations of REXX in different environments cannot be expected to support fully equivalent file I/O capabilities, there is no means for an application to determine in a portable way what is supported. The capabilities which are likely to be of interest include line or character orientation of files, read or write protection, end-of-file detection, and ability to perform random access.

- I/O error handling with NOTREADY

  The recently-added NOTREADY condition behaves differently from all other conditions (except HALT), in that it can occur multiple times within a single clause. It is not specified whether the condition will be raised independently for each occurrence, nor in what order in relation to other possible conditions.

- Independence of file operations in REXX programs

  Because there is no explicit open operation in REXX, it is unclear how operations on the same file by separate REXX programs should be handled. If a file is "opened" in one program, it is also "open" in another program called from the first? This affects the handling of file status information such as the read/write pointers. Some environments (e. g. OS/2 and Unix) allow child processes to "inherit" open files but also permit independent access to the same files. Which way should REXX go? And at what point can a file be assumed to be closed automatically if an explicit close isn't done?

Miscellaneous language issues

- External data queue

  The concept of an external data queue is firmly embedded in the REXX definition, yet system support for it varies greatly across operating systems. It is not very clear how the queue should interact with a keyboard input stream,

particularly for programs run by REXX. Important queue facilities, such as the ability to have independently named queues or separate "buffers" within a queue are outside the scope of the language definition, making it impossible for applications to use such features and remain portable. The problems are similar to I/O portability problems.

- Recommended function return values

  There are two incompatible ways of representing "success" and "failure" in the values returned by functions which are called primarily for their side effects. 1 to represent success and 0 to represent failure makes sense by analogy with the representation of "true" and "false". Yet 0 for success and 1 (or non-zero) makes sense by analogy with command return codes. REXX should have a recommended representation in order to avoid widespread confusion.

- Internal numeric precision of built-in functions

  REXX specifies that, with the exception of "mathematical" functions, NUMERIC DIGITS 9 is assumed for internal operations regardless of what prevails otherwise in the program. This is untenable for I/O functions like CHARIN()/CHAROUT() already, and may be so for other functions in the future (e. g. very long character strings).

- Byte ordering of D2C(), C2D(), etc.

  REXX takes no position on byte ordering issues, despite the differences existing between various CPU types. This causes problems primarily when REXX programs must access data from other sources at a byte level. The lack of a REXX standard for this means that a REXX program, even if it knows what type of CPU it is running on, cannot tell what byte order is in use. This problem is more acute for REXX than for other languages, because all REXX data is untyped and nominally treated as strings of characters. Most CPU types lay out character strings in ascending address order, regardless of how they represent numbers. But REXX has no way to tell the difference.

- Confusing terminology for condition handling

  There are two separate events in the handling of one condition which are not clearly distinguished. The first is the occurrence of the circumstances which define the condition (e. g. an I/O error for NOTREADY), and the second is the invocation of a handler (either default or user-defined). These two events can be separate, at least for NOTREADY and HALT. Consequently, it is not possible to speak precisely about how REXX actually processes such conditions. The term "trapped" is sometimes used to describe either event, and in addition the situation that a user-defined handler has been enabled for the condition. "Enabled" itself is used ambiguously to mean the event can occur, or that a condition handler has been defined.

- Labels allowed within IF, SELECT, and DO instructions

There is no clear prohibition of labels in inappropriate contexts, and most implementations seem to allow them. Yet there seems to be little legitimate use for labels within IF, SELECT, and DO instructions, as actual use would almost always lead to errors. Perhaps they should be prohibited. A similar question is whether duplicate labels within a program should cause a syntax error.

API capability guidelines

Although capabilities of the API (application programming interface) are normally outside of the scope of a language definition, the unique position of REXX as a "glue" language between applications and the operating system raises the API to a level of importance it would not otherwise have. There is substantial informal agreement on a minimal set of capabilities that need to be present, such as interfaces for the variable pool, command invocation, and function packages. (Though even in these cases, there is much more vagueness than precision.) But various other interfaces appear desirable, yet haven't even begun to be defined well.

- Service exits

A service exit allows the REXX language processor to make a call to a handler defined by an application in order to process commonly used services such as keyboard and screen I/O. At least one implementation that does provide service exits (IBM's OS/2 REXX) seems to provide exits for keyboard/screen I/O at some times but not others, defeating the purpose of allowing an application to control the keyboard and screen.

- Exits for raising conditions

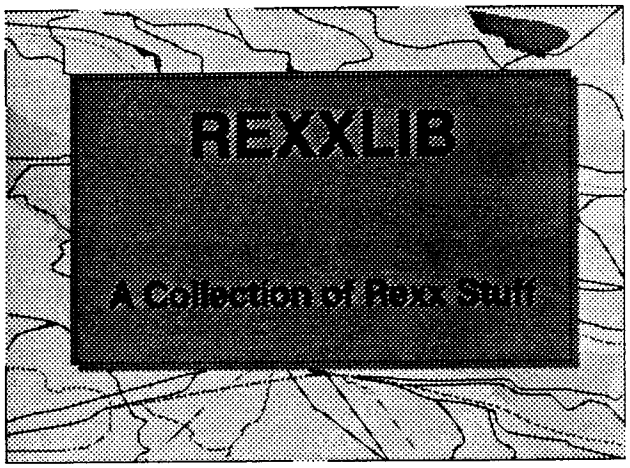
When an application is called from REXX for either command or function handling, it should be able to raise REXX conditions which will be recognized upon return to REXX, the HALT condition most especially. It would be nice if applications could raise appropriate conditions not already defined by REXX. This would require that REXX allow the enablement of "non-standard" condition handlers. Clarification of condition handling terminology to cover such cases is also needed.

- Specification of search order for external routines

Each REXX implementation currently defines a search order for external routines, and the order necessarily is implementation dependent. There is, however, no recognized API for an application to change the search order, other than (perhaps) allowing for the execution of programs already loaded into memory.

REXXLIB

BOB FLORES
CIA



Where Is It?

- BITNET / INTERNET
  - REXXLIB@PSUVM
- BLUE EDGE BBS
  - (301) 526 - 7243
  - Upload to REXXLIB

Precepts

- No official support
- Source code only
- Public domain only
- Code must be readable
- Documentation!

Documentation

- Name & Purpose
- Syntax
- Compiling Instructions
- Platform(s)
- Author & Address
- Dependencies

Dependencies

- List of Files
- Who Needs What
- What Needs Whom
- Globals

PLATFORM-SPECIFIC STANDARDS FOR REXX

ERIC GIGUERE
UNIVERSITY OF WATERLOO

Platform-Specific Standards for REXX: Issues for Developers and Implementors

Eric Giguère

Computer Systems Group
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Internet: giguere@csg.uwaterloo.ca
BIX: giguere

Introduction

Standards are important to the growth and acceptance of a programming language. However, standards cannot — and should not — specify everything about language implementations. Many details — type sizes, calling conventions, file manipulation methods — will change with the machine architecture and operating system, and sometimes at the whim of the implementation team. This is why standards documents are littered with the phrase "implementation-defined behaviour".

REXX is no different from any other language in this respect. For example, REXX has the capability to send an arbitrary command to a host environment and to receive a return code in reply. The *hows* of registering the host environment, sending the command, and receiving the reply are all implementation-defined. Even the command strings and reply codes have different meanings on different systems.

REXX is different from other languages, however, in that it explicitly provides the means for *integration* and *expandability*. A typical use for REXX is to tie together two applications from different vendors. The REXX implementation should make this integration as painless and seamless as possible, for both the application developers and the end users. And on a given machine/OS pair it shouldn't matter which REXX implementation a user uses — each version of REXX should be "plug compatible" with the other. Similarly, different applications should support similar REXX interfaces and command sets.

This paper discusses the idea of *platform-specific* standards as they apply both to REXX implementors and to developers of REXX-compatible products. The emphasis is on co-operation and the adaptation of existing standards to meet the needs of the REXX community.

Many of the examples in this paper will center around ARexx, an implementation of REXX for the Commodore Amiga. A brief description of the Amiga and ARexx can be found in the appendix. OS/2 REXX and CMS REXX are also mentioned.

1. The Host Interface

There are four basic problems in using or implementing the REXX host command interface.

Finding hosts: The use and syntax of the `ADDRESS` instruction is well-documented, but not what the host address actually means, or how REXX locates the host.

On the Amiga, host addresses correspond to named message ports. When an ARexx-compatible application starts execution it opens a message port to which commands may be sent. This *ARexx port* is a public resource that other applications, including ARexx, can search for by name.[1]

OS/2 and CMS require applications to register *subcommand handlers* with the REXX system. Each handler is a library/program entry point.

Naming hosts: When multiple copies of the same application are running, which host does the user actually want to talk to, and how do they specify it?

Naming rules are not enforced under ARexx. The system ports list is prioritized, but a friendly application should not duplicate or override any name already in the list. Commodore's developer guidelines advocate letting the user set whatever port name they desire. If the user doesn't assign a specific port name, then the application should use its own name, stripped of non-alphanumeric characters and converted to uppercase. Applications that support multiple projects/documents should append a *slot number* to the name, as in `EDIT.01`, `EDIT.02`, and so on (in other words, an ARexx port can be allocated for each document). The actual slot numbers are searched for and assigned dynamically. Slot numbers should also be used when two or more copies of an application are executing simultaneously.

Under OS/2 REXX a user can append the handler's *library name* to the host address, as in `ADDRESS 'Edit.Qedit'`, to differentiate between applications using the same host address.

Sending commands: Some form of inter-application communication is necessary for sending host commands and receiving the return codes in reply. On systems without inter-process communication the interface is obviously much harder to implement.

ARexx simply sends the command as a message to the appropriate port. The message structure is fixed and includes fields for strings, results, and action codes. When the application receives a message at its port, it retrieves the message and parses the command string. The ARexx program that sent the message is suspended until a reply arrives (each ARexx program is a separate task).

OS/2 and CMS call the handler function that was registered with the REXX system.

Starting REXX macros: How does an application access and start the REXX interpreter when it wishes to invoke a REXX macro program?

An Amiga application merely sends a message (using the same message structure described above) to the ARexx *resident process*. The resident process spawns the ARexx program as a new task. The application has the option of waiting for the program to finish execution or of continuing on with its work. The application must always be ready to process incoming ARexx command messages as well.

OS/2 and CMS applications call a system function to start a REXX program.

[1] Note that unlike OS/2 REXX, there is no formal registration to be made. ARexx will search for and locate the appropriate message port each time it has a message to send.

Note that all macros that are started by an application should have the application's port/handler set as their default host address. This minimizes the naming problems discussed above.

2. Command Standards

REXX performs minimal processing for commands: it merely evaluates an expression and sends the resulting string to a host application for processing. It's the application's responsibility to parse this string and act on its contents. This is as it should be: it is not REXX's mandate to assign meaning to these strings.

It would be nice, however, if REXX-aware applications used similar command sets.[2] A simple text editor macro such as:

```
/* Load a file, search for a string */

'open' arg(1)
'search' arg(2)
```

should be simple to adapt for another text editor. This makes it possible for developers to include fairly complete sets of REXX macros with their applications without having to explicitly support every programming tool on the market. Command standards make application integration much simpler.

As an example of what can happen without a set of guidelines, consider the text editors available for the Amiga. These days all Amiga text editors (and most other applications) are ARexx-aware. Unfortunately, their ARexx command sets are completely incompatible. While there are many interesting macros available for various purposes (automatic tracking of compiler errors, for example), the macros have to be rewritten from scratch for each text editor. Telecommunication utilities are in the same boat.

To address this situation, Commodore has just released a set of ARexx command guidelines to developers as part of the *Amiga User Interface Style Guide*. The guidelines list suggested commands (names and options) for common operations. Hopefully new applications will include these commands in their command set and ease the confusion that prevails right now.

3. Returning Command Results

It isn't enough to send commands to an application — a REXX program must be able to receive and process data from those commands as well, if only to know whether the command failed or succeeded.

"Vanilla" REXX only defines the concept of a *return code* when it comes to command results. When a command has been executed, the special variable `RC` will hold a numeric value which the program can then use as basis for further action:

```
/* Run a program */
address command
'run rxtools:rxtools'
```

[2] Commands to the underlying operating system being an obvious exception.

```
if( rc ~= 0 )then do
    say "Could not start RxTools."
    exit 1
end
```

The value in RC is (of course) implementation-defined. A general convention is that non-zero values indicate warning or error conditions.

ARexx implements an extension that allows applications to return *result strings* as well as return codes. Result strings must be requested before sending a command by using the OPTIONS RESULTS instruction. After a command has been sent, the special variable RESULT may have been set to a value if no error occurred (RC is 0):

```
/* Ask user for a string */
options results
address 'rexx_ced'
'getstring "Please enter search pattern:"'
if( result ~= "" & result ~= "RESULT" )then do
   ..... /* do stuff */
end
```

Note that RC doesn't actually need to be checked since if an error occurred the value of RESULT will be DROPped automatically.

The OPTIONS RESULTS extension is a useful one because it allows commands to act like function calls. Another way of passing back data is to use RVI/VPI, as discussed below.

Using a clipboard is also an effective way of passing information between applications. A 'PASTE' command could be sent to an application to place its data into the clipboard. A set of functions (either built-in or part of a function package) could then be used from within a REXX program to manipulate this data. The paste operation could even be performed manually by the user for applications that aren't REXX-aware.

4. Function Packages

A simple but effective way of extending the REXX language is to allow developers to create their own *function packages*. REXX programs can then call the functions in a package as if they were built-in functions. Unlike external functions, the functions in these packages are probably not written in REXX.

An obvious use for function packages is to extend REXX to include user interface support. At least two such packages (one freeware, one commercial) already exist on the Amiga. Another ARexx function package provides transcendental mathematics functions.[3] ARexx even allows applications to act as function packages (*function hosts*) as well as accepting command messages.

The user can run into trouble using function packages, however, through *namespace pollution*. Sooner

[3]For serious mathematics an ARexx-aware application program such as Maple is recommended instead.

or later one function package is going to use a name already used by another function package. Which function will get called? Can function packages override built-in functions?

5. RVI/VPI

The last important issue has to do with the sharing of data between REXX and application programs. Under ARexx this capability is known as the *REXX Variables Interface* (RVI), while OS/2 and CMS refer to it as the *Variable Pool Interface* (VPI).

RVI can only be used by REXX-aware applications. RVI allows these applications to set and examine the value of REXX variables. These alterations can only happen when a REXX program has sent a host command and is waiting for a reply. During the processing of the command the host application can use RVI to modify the REXX program's symbol tables.

RVI is a simple way to pass back complex information to a REXX program. For example, on CMS the XEDIT `'EXTRACT'` command can be used to copy state information into a stem variable in the calling program.

When processing a command, a host application must be careful to ensure that the command came from a REXX program and not some other application before using the RVI routines. Some method must be built into the host addressing to differentiate between REXX and non-REXX callers.

And of course, user documentation is very important. If an application can change a variable, the user should be made aware of the fact. Preferably, the user will have control over which variables are changed. Commodore's command standards, for example, include `'VAR'` and `'STEM'` options to allow the user to specify variables for command results.

Conclusions

This paper doesn't pretend to present any startling conclusions, but only some observations and some simple advice for any implementor: check out current REXX implementations, especially the ones on the platform you're developing for. If possible, offer a similar set of capabilities and interfaces, at least as an option. Consider the interfaces other macro languages — BASIC, for example — offer, especially if REXX is not the predominant macro language for your system. You can't expect every application to be REXX-aware, so it certainly helps if they can still use REXX even on the most rudimentary level.

A. ARexx: A Sample Platform

The Commodore Amiga is a microcomputer based on the Motorola 680x0 architecture. The base operating system, *Exec*, is a message-based, preemptive multitasking system. File I/O and command shells are provided by *AmigaDOS*, while graphics and user interface support are provided by *Intuition*.

ARexx is an implementation of REXX 3.5 with some Amiga-specific extensions. The ARexx interpreter is stored as a *shared library* and is about 33K in size. A *resident process* of about 3K runs as a

background task. The resident process is the master ARexx control program: it launches new ARexx programs and keeps track of global resources.

To start an ARexx program, a message is sent to the resident process. It spawns a new process which invokes the interpreter. Each ARexx program runs as a separate task and performs its own resource tracking.

Messages are at the heart of ARexx program interaction. ARexx defines its own message protocol as an extension of the Exec message structure. (The Amiga's memory space is shared between all tasks. Message ports are really just linked lists.) An ARexx message is defined as follows:

```
struct RexxMsg
  {
   struct Message  rm_Node;      /* Exec message structure       */
   APTR            rm_TaskBlock; /* global structure (private)   */
   APTR            rm_LibBase;   /* library base (private)       */
   LONG            rm_Action;    /* command (action) code        */
   LONG            rm_Result1;   /* primary result (return code) */
   LONG            rm_Result2;   /* secondary result             */
   STRPTR          rm_Args[16];  /* argument block (ARG0-ARG15)  */

   /* Extension fields (not modified by ARexx) */

   struct MsgPort *rm_PassPort;  /* forwarding port              */
   STRPTR          rm_CommAddr;  /* host address (port name)     */
   STRPTR          rm_FileExt;   /* file extension               */
   LONG            rm_Stdin;     /* input stream (filehandle)    */
   LONG            rm_Stdout;    /* output stream (filehandle)   */
   LONG            rm_avail;     /* future expansion             */
  };
```

The message structure includes fields for setting various *action codes* (whether a message is a host command or function call) and *modifier flags* (is a result string required? how many arguments are being passed?), a return code, a result string, and the arguments for the command or function call. Arguments are always passed as strings.

To send a host command, an ARexx program allocates a `RexxMsg` structure, fills in the appropriate values, and sends the message to the host's port. The host will receive the message, parse the command string, execute the command, set a return code (and if requested, a result string) and reply to the message. The ARexx program remains blocked until the reply message arrives.

Function calls are also handled with messages. If a function call cannot be resolved by an internal or built-in function, ARexx will search a prioritized *library list* maintained by the resident process. Each entry in the library list is either a *function library* or a *function host*. A function library is a shared library with a public entry point, while a function host is an application program with a public message port. ARexx will query each library/host in turn (by calling the library's entry point directly or by sending a message to the host) until the desired function is found. If this fails, a search is made for an external function.

References

[Amiga 91] Commodore-Amiga, Inc. *Amiga Programmer's Guide to ARexx*, 1991 (forthcoming).

[Cowlishaw 90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming, 2nd edition*, Prentice-Hall, 1990.

[Giguère 91] Eric Giguère. "Rexx: Not Just a Wonder Dog", *Computer Language*, Vol. 8 No. 3, March 1991.

[Hawes 87] William S. Hawes. *ARexx User's Reference Manual*, Wishful Thinking Development Corporation, 1987.

[IBM 87] International Business Machines Corporation. *SAA Common Programming Interface/Procedures Language Reference*, SC26-4358-0, 1987.

[IBM 88a] International Business Machines Corporation. *VM/SP System Product Interpreter Reference*, SC24-5239-03, 1988.

[IBM 88b] International Business Machines Corporation. *VM/SP System Product Interpreter User's Guide*, SC24-5238-04, 1988.

[IBM 89a] International Business Machines Corporation. *OS/2 1.2 Procedures Language 2/REXX*, 1989.

[IBM 89b] International Business Machines Corporation. *OS/2 1.2 Procedures Language 2/REXX Programming Reference*, 1989.

[Watts 90] Keith Watts. "REXX Language I/O and Environment Challenges", *Proceedings of the 1990 REXX Symposium*, SLAC Report 368, 1990.

REXXOIDS

LINDA SUSKIND GREEN
IBM

REXXoids

Linda Suskind Green
SAA Procedures Language Interface Owner

IBM
Endicott Programming Lab
G92/6C14
PO Box 6
Endicott, NY 13760

INTERNET: greenls@gdlvm7.vnet.ibm.com
Phone: 607-752-1172

May, 1991

Contents

Contents

- **REXXoids Summary**

Preface

This presentation is to be given at the second annual REXX Symposium held at Asilomar in California on May 8-9, 1991.

REXXoid is not an English word to be found in the dictionary. REXXoid is meant to be little bits of information I have collected about REXX.

If there were a Jeopardy category on REXX, this information would probably show up there. Most of this information will change as time passes. This presentation is meant to be a snapshot when I collected the information.

REXX History

REX becomes REXX

In the beginning, there was

REX (REformed eXecutor)

which eventually became

REXX (REstructured eXtended eXecutor)

REXX Firsts

- 1979 - Mike Cowlishaw (MFC) starts work on REX
- 1981 - First SHARE presentation on REX by Mike
- 1982 - First non-IBM location to get REX is SLAC
- 1983 - First REXX interpreter shipped by IBM for VM
- 1985 - First non-IBM implementation of REXX shipped
- 1985 - First REXX trade press book published
- 1987 - IBM Selects REXX as the SAA Procedures Language
- 1989 - First REXX compiler shipped by IBM for VM
- 1990 - SHARE REXX committee becomes a project
- 1990 - First SHARE presentation on Object Oriented REXX
- 1990 - First Annual REXX symposium held (organized by SLACs Cathie Dager)
- 1991 - First REXX ANSI committee meeting held

Jeopardy: REXX for $1000

Answer is:

5

Question is:

How many programming languages has MFC designed? Note that REXX is his latest!!!!

Jeopardy: REXX for $800

Answer is:

350

Question is:

What is the peak amount of REXX electronic mail MFC received per working day?

Jeopardy: REXX for $600

Answer is:

4000

Question is:

What is the approximate number of hours MFC spent on REXX before the first product shipped?

Jeopardy: REXX for $400

Answer is:

500,000

Question is:

What are the approximate number of REXX related electronic mail MFC has read since REXX started?

Jeopardy: REXX for $200

Answer is:

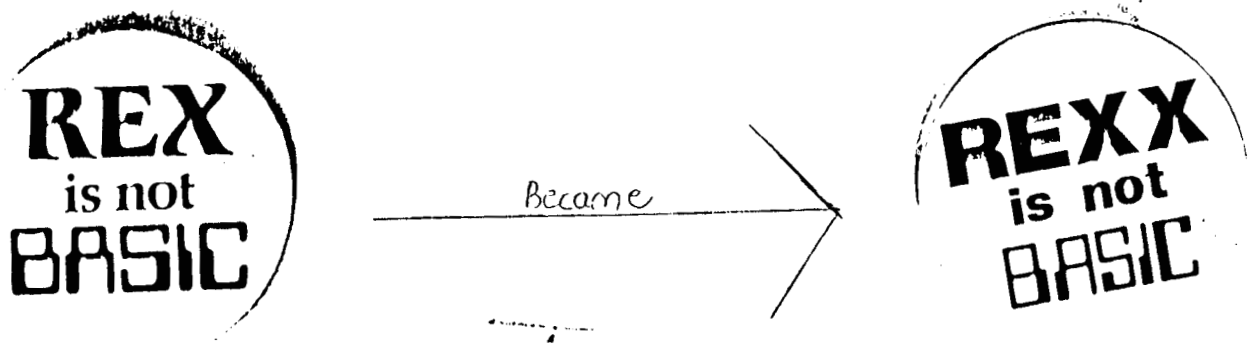
over 6,000,000

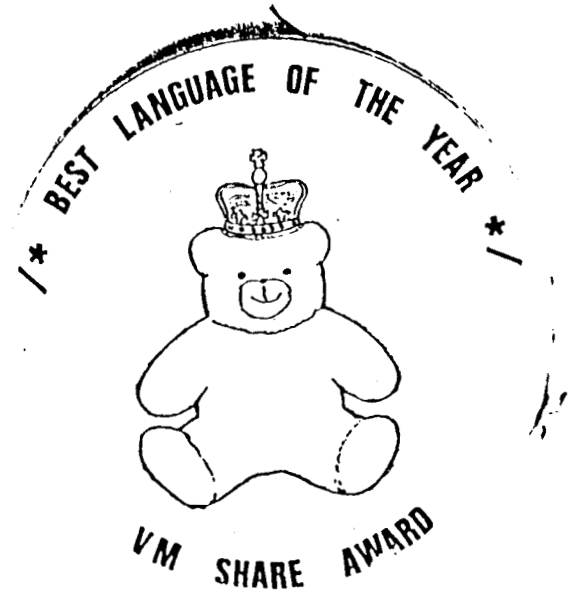
Question is:

What is the largest known total number of lines of REXX code used in any one company?

REXX Buttons

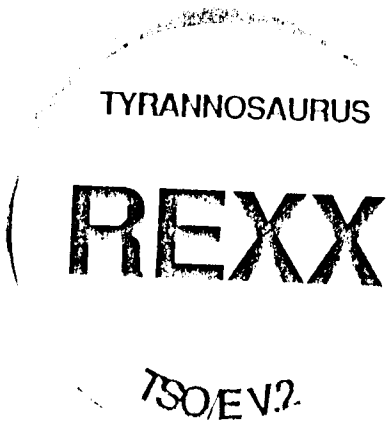
Customer Created:





REXX Buttons ...

Customer Created:



IBM Created:

SAA Procedures
Language/REXX

Text of the REXX Buttons

- General
  - REX is not BASIC
  - REXX is not BASIC
  - The beginning /* of the end
- VM
  - /* Best Language of the Year */ VM SHARE AWARD
  - VM/SP has REXX Appeal
  - RXSQL good medicine!
- SAA
  - SAA Procedures Language/REXX
- TSO/E
  - I practice safe REXX (TSO/E v2)
  - TSO/E is REXX rated!
  - Tyrannosaurus REXX TSO/E v2
  - TSO/E Puttin' on the REXX

REXX Excitements

REXX Excitement!

- ANSI committee started
- REXX Users start a yearly REXX Symposium in 1990
- SHARE elevated REXX to a Project
- Increasing number of books and articles on REXX
- Increasing number of REXX Implementations on different platforms by increasing number of companies

ANSI

REXX is one of 15 languages to be worked on as an ANSI standardized language. Others are:

- APL
- APT
- BASIC
- C
- C++
- COBOL
- DATABUS
- DIBOL
- FORTH
- FORTRAN
- LISP
- PASCAL
- PL/I
- PROLOG

Note that the languages listed are at different levels of standardization.

SHARE Interest in REXX

SHARE Meeting	Number of REXX Sessions
72 (3/89)	9
73 (8/89)	13
74 (3/90)	23
74.5 (5/90)	REXX Project approved
75 (8/90)	25
76 (3/91)	20

Note that the sessions are in the REXX Project, MVS Project, and CMS Project.

Publications

As of 12/90, REXX has been the subject of:

- **4 books (plus 4 in the works)**
- **40 User Group Presentations**
- **40 product manuals**
- **40 articles**

REXX Books

Published:

- The REXX Language, A Practical Approach to Programming by Mike Cowlishaw (1985, 1990)
- Modern Programming Using REXX by Bob O'Hara and Dave Gomberg (1985,1988)
- REXX in the TSO Environment by Gabriel F. Gargiulo (1990)
- Practical Usage of REXX by Anthony Rudd (1990)

Planned:

- REXX Handbook edited by Gabe Goldberg and Phil Smith
- Amiga Programmers Guide to AREXX by Eric Giguere
- 2 others

REXX is International

REXX books and manuals have been translated into many languages, including:

- **Chinese**
- **French**
- **German**
- **Japanese**
- **Portuguese**
- **Spanish**

REXX is International - Part 2

REXX presentations have been given in the following countries:

- Austria
- Australia
- Belgium
- Canada
- England
- France
- Germany
- Holland
- Japan
- Jersey
- Scotland
- Spain
- United States
- Wales

As of 1982, MFC had received mail from over 30 countries!

REXX Trade Press Article Titles

EXOTIC LANGUAGE OF THE MONTH CLUB

REXX: A beginner's alternative

All Hail REXX

or what you should know about IBM's soon to be ubiquitous procedural language

For Programmers

REXX — A Multifaceted Tool

WHY IS REXX SO POPULAR?

REXX: REXX the Wonder Language

REXX—Portrait of a New Procedures Language

Capture cross-system capabilities with REXX

REXX Trade Press Article Titles

ANALYSIS

Lotus Makes a Case for REXX To Foil Microsoft's BASIC Plan

Analysis \ *Powerful REXX vs. Popular BASIC*

Rexx in Charge

You're not really multitasking in OS/2 unless you're using Rexx

HELLO, REXX— GOODBYE, .BAT

REXX is a powerful new shell language that's as easy to use as BASIC. It may replace the DOS batch processor in the future.

REXX

A USER'S DREAM IN A SYSTEM PROGRAMMER'S WORLD

REXX Trade Press Article Titles

Utilize The Power Of REXX/CP/CMS

TEST DRIVE

AREXX

(The integrative language)

A language adopted by IBM comes to the Amiga – and has started to change many other applications.

REXX:

The system architecture application connection

REXX Language Level

REXX Level	Usage
3.20	CMS release 3
3.40	CMS release 4, 5, 5.5 MUSIC/SP version 2.2
3.45	CMS release 6 TSO/E version 2, release 1
3.46	CMS release 6 with SPE TSO/E ver 2, rel 1 with APAR CMS compiler SAA Procedures Language level 1
3.48	OS/400 rel 1.3
3.50	Cowlishaw 1985 book Portable REXX ver 1.05 (DOS) uniREXX (UNIX, AIX, DEC/VMS) Personal REXX version 2.0 (OS/2, DOS) AREXX (Amiga) TREXX (Tandem)
4.00	OS/2 release 1.3 Cowlishaw 1990 book SAA Procedures Language level 2 Personal REXX version 3.0 (OS/2, DOS) Portable REXX ver 1.10 (DOS)

Implementations

There are REXX implementations for:

- AIX
- Amiga
- DOS
- OS/2
- OS/400
- Tandem
- TSO
- UNIX
- VM
- VMS

from 7 different sources.

REXX Implementations by year First Available

Year	New Platform
1983	VM (IBM)
1985	PC-DOS (Mansfield)
1987	Amiga (W. S. Hawes)
1988	PC-DOS (Kilowatt) TSO (IBM)
1989	OS/2 (Mansfield) VM Compiler (IBM)
1990	Amiga (Commodore) UNIX (Workstation Group) AIX (Workstation Group) Tandem (Kilowatt) OS/2 (IBM) AS/400 (IBM)
1991	DEC/VMS (Workstation Group)
future	VM Compiler (Systems Center) 370 Compiler (IBM) Windows (Quercus)

REXX Curiosities

Name of a REXX Entity

What is the name of a REXX entity??? Is it:

- **Program**
- **Exec**
- **Macro**
- **Procedure**
- **Shell**
- **Script**

Name of a REXX Entity ...

Term definitions are:

Program: A sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it.

Exec procedure: In VM, a CMS function that allows users to create new commands by setting up frequently used sequences of CP commands, CMS commands, or both, together with conditional branching facilities, into special procedures to eliminate the repetitious rekeying of those command sequences.

Macro instruction: An instruction that when executed causes the execution of a predefined sequence of instructions in the same source language.

Procedure: A set of related control statements that cause one or more programs to be performed.

Name of a REXX Entity ...

shell: A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens and communicate them to the operating system.

script: In artificial intelligence, a data structure pertaining to a particular area of knowledge and consisting of slots which represent a set of events which can occur under a given situation.

Note: definitions come from the IBM "Dictionary of Computing".

Is REXX a....?

- Programming language
- Exec language
- Macro language
- Procedure language
- Command procedures language
- Extension language
- System Extension language
- Glue language
- Shell language
- Batch language
- Scripting language

Cowlishaw Book Cover

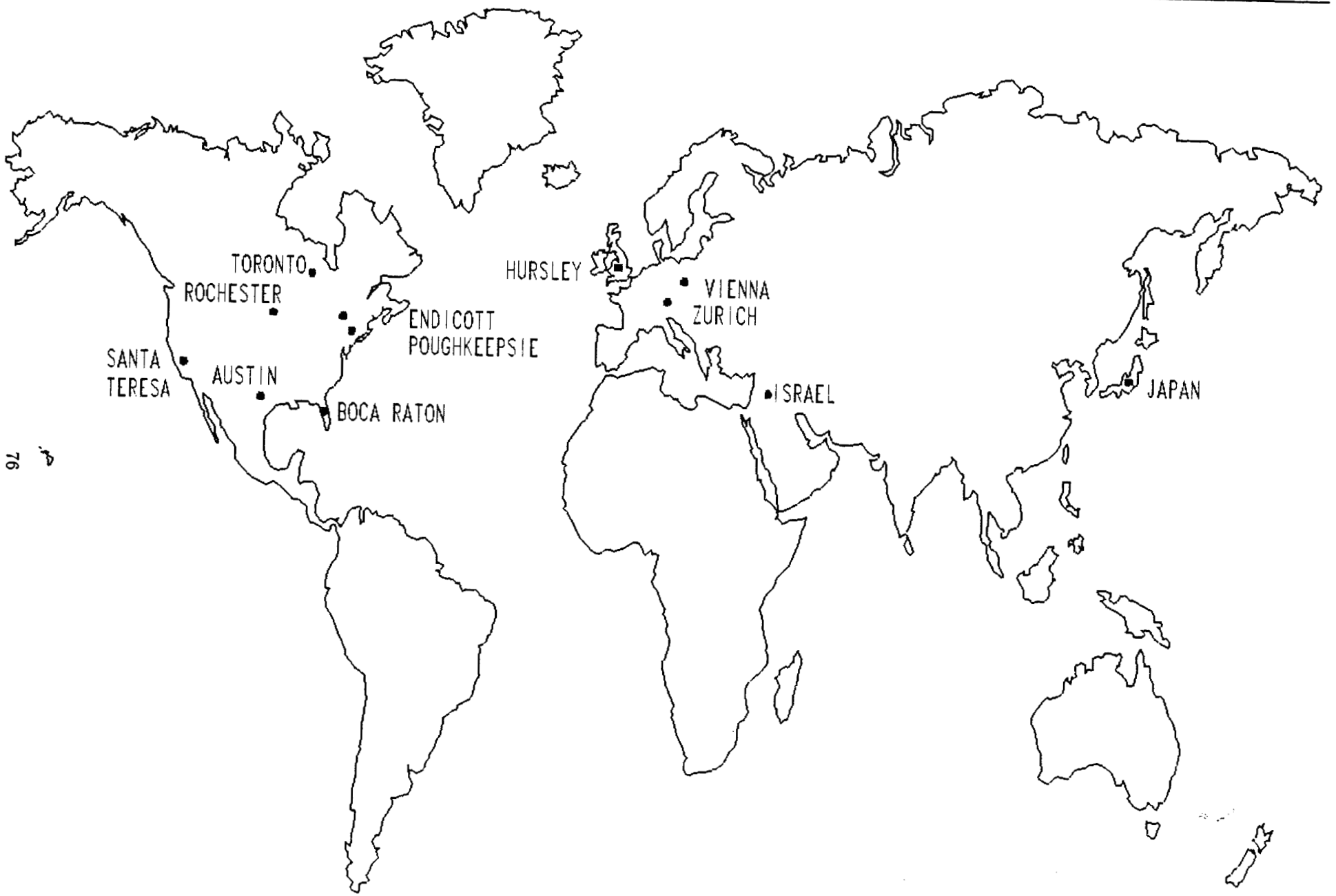
The 1990 edition of the Cowlishaw book has a new cover which includes the following changes:

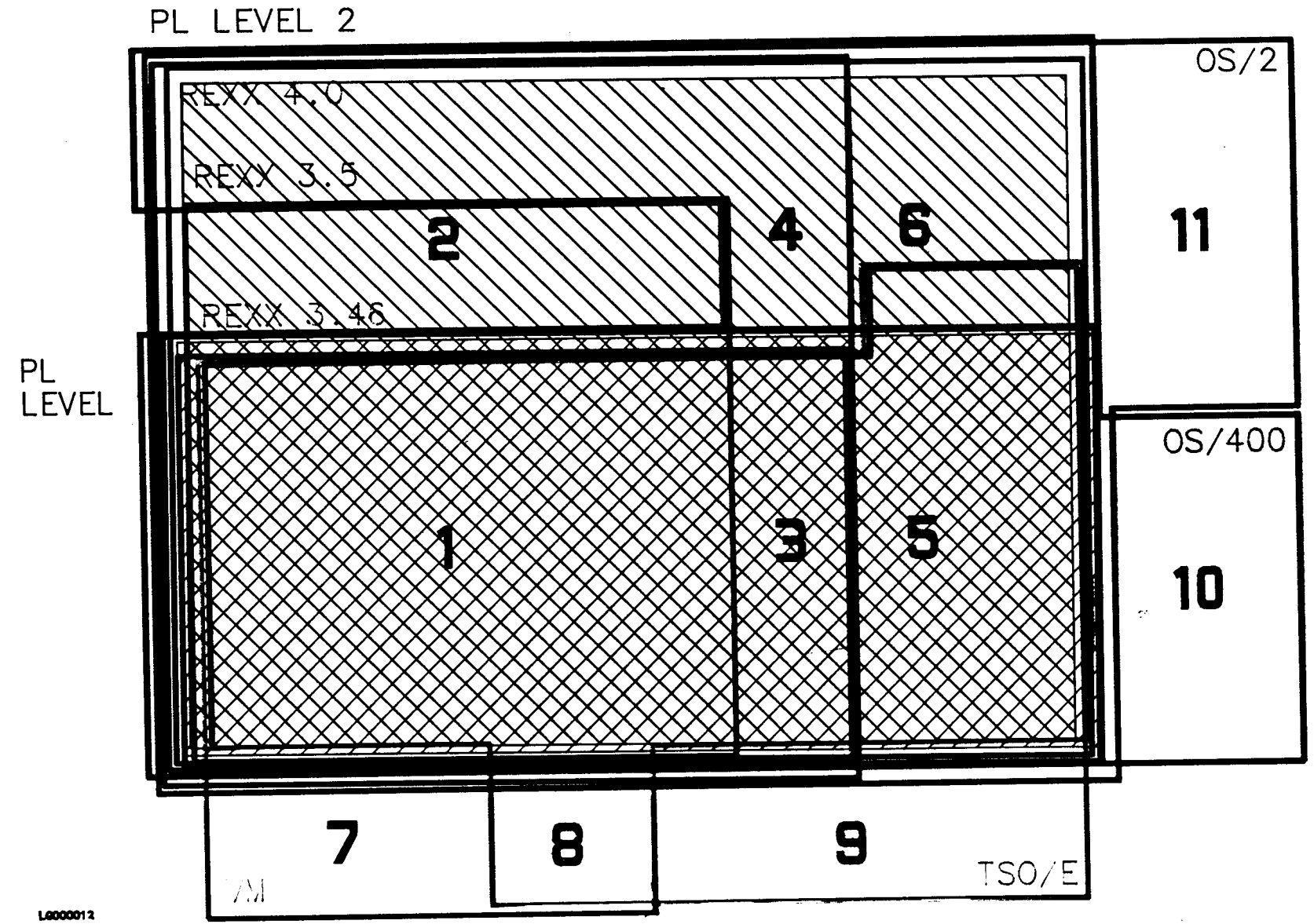
- King now matches the playing cards King of Spades meaning
  - King faces the opposite way
  - King holds the sword differently

King was chosen because REX is Latin for King!

IBM REXX Explanations

IBM PROCEDURES LANGUAGE/REXX: WORLDWIDE DEVELOPMENT




PL LEVEL 2
REXX 4.0
REXX 3.5
REXX 3.46
2
4
6
OS/2
11
PL
LEVEL
1
3
5
OS/400
10
7
8
9
TSO/E
77

Relationship of REXX/PL/IBM Implementations

Box	1	2	3	4	5	6	7	8	9	10	11
REXX language level 3.46	x		x								
REXX language level 3.5	x	x									
REXX language level 4.0	x	x	x	x							
SAA Procedures Language Level 1 (PL1)	x		x		x						
SAA Procedures Language Level 2 (PL2)	x	x	x	x	x	x					
IBM VM interpreter	x		x		x		x	x			
IBM TSO/E interpreter	x		x		x			x	x		
IBM OS/2 interpreter	x	x	x	x	x	x					x
IBM AS/400 interpreter	x		x	x	x	x				x	

Relationship of REXX/PL/IBM Implementations ...

	box	3.5	4.0	PL1	PL2	VM	TSO	OS2	400
Base REXX lang	1	x	x	x	x	x	x	x	x
Level 3.46 adds eg. CONDITION	3		x	x	x	x	x	x	x
Level 3.5 adds eg STREAM I/O	2	x	x		x			x	
SAA PL1 adds eg DBCS support	5			x	x	x	x	x	x
Level 4.0 adds eg binary data	4		x		x			x	x
SAA PL2 adds eg REXX exits	6				x			x	x
VM adds eg. DIAG TSO adds eg. DELSTACK 370 adds eg. UPPER	7 8 9					x x	 x x		
OS/2 adds eg. BEEP	11							x	
AS/400 adds eg. SETMSGRC	10								x

REXXoids Summary

REXXoids Summary

- REXX is an international language
- REXX is growing in numbers of
  - implementers
  - different platforms available
  - users
  - books/articles.
- REXX is in the process of being formally standardized.
- REXX usage is in the "eyes of the beholder"!

PIPELINES: HOW CMS GOT ITS PLUMBING FIXED

JOHN P. HARTMANN
IBM

How CMS Got Its Plumbing Fixed

John P. Hartmann IBM FSC, Nymøllevej 85, DK-2800 Lyngby, Denmark

Abstract

An overview of pipeline concepts is followed by a description of how these concepts were adapted to single-tasking CMS, and how the implementation evolved.

The data flow model of programming is well suited for many (but not all) programming problems. *CMS Pipelines* is a simple, robust, and efficient tool to use data flow techniques in VM/CMS.

Programs running in a pipeline read and write records on a symmetrical device-independent interface. A non-trivial problem is often solved by running a number of simple programs, each doing a little bit of the big problem; the pipeline combines programs, often to accomplish tasks that are not imagined when a particular program is written. Pipelines are entered from the terminal or issued as commands in REXX programs.

CMS Pipelines has features not found in most other systems supporting pipelines:

- Multistream pipelines support any number of concurrent streams through a program; a simple example is the master-file-update paradigm.
- A program can temporarily replace itself with a subroutine pipeline.
- A pipeline is run only when all stages of it are specified correctly; the syntax of built-in filters is checked before the pipeline is started.

Introduction

When programs run in a pipeline, the output from one program is automatically presented as input to the next program in the pipeline. Each program reads its input and writes its output through a device-independent interface without concern for other programs in the pipeline. Thus the standard output from a program can be read by the standard input of **any** program.

Why *CMS Pipelines*?

- Things get done that might not be done otherwise. The ease with which standard programs are bolted together means that users can perform ad-hoc processing of their data in ways that would not be as economical with traditional programming.
- Users can solve problems by functional programming. By selecting appropriate filters, users can apply functions to a stream of data and not worry about how to perform a particular function to all records in a file.
- Code is re-used each time a program is run in a pipeline; unlike traditional software engineering, code re-use with *CMS Pipelines* requires no modification or compilation.
- Pipeline programs are device-independent. This takes the drudgery out of writing programs. All pipeline programs can use new host interfaces as soon as a single device driver is written to support the interface in question.
- A complex task is often broken into simpler tasks, some or all of which are performed by built-in programs. What programs remain to be written, if any, are often significantly simpler than a program to perform the original task directly.
- A simple, efficient interface supports REXX programs in a pipeline, bringing device-independent I/O to REXX.
- *CMS Pipelines* supports most CP/CMS devices and interfaces, many of which are not available to REXX programs using standard CMS interfaces.

Sample Pipeline

Though CP has a command to display the number of users logged on to a system, there is no command to display the number of disconnected users. Figure 1 shows how to obtain this information with *CMS Pipelines*.

The first word is the CMS command to run a pipeline. The rest of the line is a *pipeline specification* defining which programs to select and run. There are five programs in this example; they are separated by a solid vertical bar (|).

cp issues the query command to CP and writes the response to the pipeline, with a line for each line of CP response. Four users are shown on each line.

split splits lines at the commas that separate the four users. The comma is discarded.

locate selects lines with the string `'- DSC'`. This selects all disconnected users.

count counts the number of lines in the input stream. This count is the number of disconnected users because there is a line for each user, and connected users have been discarded from the file.

console copies its input stream to the terminal of the virtual machine. It also copies the input lines to the output, but in this example the output from *console* is not connected.

How does one count the number of disconnected users with standard CMS commands? With difficulty, it would appear, and certainly not without writing a program.

```
pipe cp query names|split ,|locate /- DSC/|count lines|console
123
Ready;
```

Figure 1. Sample Pipeline

Hartmann

What is CMS Pipelines

CMS Pipelines Structure

The user sees three parts of *CMS Pipelines*:

Command Parser: Scans the argument string to the PIPE command to build a control block structure describing the pipeline to run. It ensures that the pipeline specification is well formed, that all programs exist, and that the syntax is correct for those programs where a syntax description is available to the parser.

Library of Built-in Programs: Contains device drivers, filters, and many utility functions that can be selected by the parser.

Dispatcher: Starts programs and passes control between programs to maintain an orderly flow of data through the pipeline. Programs call the pipeline dispatcher to read and write the pipeline. The dispatcher runs programs as *co-routines*; control passes from one program to another only when a program calls the dispatcher to transport data.

REXX Programs

Though many tasks can be performed with a combination of built-in programs, there are bound to be times when *CMS Pipelines* does not provide the primitive function needed for a particular task. A program must be written to perform the missing function when *pipethink* (chipping sub-problems off a big problem) does not come up with a useful solution.

However, the program to be written needs only to solve one particular little problem; most of the task should be performed with built-in programs.

Programs to process pipeline data can be written in REXX, PL/I, IBM C/370, Assembler, and other languages that use Assembler calling conventions. REXX is used exclusively in the examples in this paper.

A REXX program processing data in the pipeline is stored as a disk file; it has file type REXX to distinguish it from EXEC procedures. The REXX program can be EXECLOADed or installed in a shared segment just like all other REXX programs.

The default command environment for REXX pipeline programs processes *pipeline commands* to move data from the pipeline into the program's variable pool, and to write output lines into the pipeline.

As an example of a function that is not readily made with built-in programs, consider how to display the number of terminals that are in the state between displaying the VM logo and having a user logged on. Local terminals in this state are shown with the a user ID comprised of LOGO followed by the four-digit device address.

NOTLOG REXX

```
/* Select LOGOxxxx userIDs */
signal on novalue
signal on error
do forever
   'readto in'
   parse var in user+8 '-' device .
   If user = 'LOGO'device
      Then 'output' in
end
error: exit RC*(RC¬=12)
```

The program in the sample above reads input lines into the variable `in` which is parsed to obtain the user ID and the device address. The input line is copied to the output with the `output` command when the line represents a terminal in limbo. Note that the two commands are not symmetrical: the name of the variable to receive the next input line is a literal; the variable is set as a side effect of the command. The line to write is the argument string to the `output` command. The loop terminates when either of the two commands gives a non-zero return code. The return code from the filter is 0 at normal end-of-file or the return code from the pipeline.

```
pipe cp q n|split ,|strip|notlog|console
LOGOL097 - L097
Ready;
```

It becomes cumbersome to write long pipelines on the terminal, especially when fine-tuning a suite of filters: put them into a REXX EXEC instead. Commands on the terminal are in *landscape* format (a single line); when writing EXECs, it is more convenient to write pipelines in *portrait form* with one line per program. This is the sample above in portrait form:

```
/* Limbo sample */
'PIPE',
   'cp q n|',
   'split ,|',
   'strip|',
   'notlog|',
   'console'
exit RC
```

CMS Pipelines supplies a sample XEDIT macro to convert from landscape to portrait form.

Using *CMS Pipelines* in REXX EXECs

It is easy to augment REXX EXECs with pipelines: use the PIPE command with device drivers to read and write REXX variables.

Sort: The example below sorts the contents of the stemmed array `unsorted`. *stem* reads and writes a stemmed array. The variable `unsorted.0` has the number of variables in the array; the first variable is `unsorted.1`, and so on. The result is stored in the array `sorted`.

```
'PIPE stem unsorted.|sort|stem sorted.'
```

Discovering Stemmed Variables: The device driver *rexxvars* writes the source string and all exposed variables in a REXX program into the pipeline. It writes the name and value of a variable on separate lines. The first column is the record type (`n` for a variable name); the source string, name, or value begins in column 3. Given this, finding all variables with a common stem is a matter of *find*. To find the names of all variables that have the stem `array`:

```
'PIPE',
   'rexxvars|',      /* Read all variables */
   'find n ARRAY.|', /* Names of array     */
   'spec 3-* 1|',    /* Discard type prefix*/
   'buffer|',        /* Ensure no interf.  */
   'stem vars.'      /* Store in stemmed   */
```

rexxvars Reads the names and values of all exposed variables in the REXX program.

find selects name lines for variables with stem `array`. Discard the lines with the values of the stemmed variables and information about other variables.

spec moves the name (from column 3 onwards) to the beginning of the record.

buffer stores all lines in a buffer before writing any to the output. This ensures that the variables to be set with the result do not interfere with the variables being queried.

stem writes the names of all variables with the stem `array` into the stemmed array `vars` where they can be accessed with a numeric index.

Transporting Variables Between REXX Programs: The device drivers supporting REXX variables can manipulate REXX environments prior to the one issuing the PIPE command. To copy the stemmed array `parms` from the caller to the current REXX program:

```
'PIPE stem parms. 1|stem parms.'
```

The number after the first stem indicates that the EXECCOMM before the current one is to be read.

Find the Caller: The first line of output from *rexxvars* has the letter `s` in the first column and the source string (the string parsed with `Parse Source`) from column 3 onwards. When *rexxvars* is applied to the environment before the current one, the first line is the source string for the caller. This can be parsed to determine the caller of a REXX program. *var* sets the variable to the first line on the input stream.

```
'PIPE rexxvars 1|take 1|var source'
parse var source 3 . . c_fn c_ft c_fm .
parse source . . m_fn m_ft m_fm .
say m_fn m_ft 'called from' c_fn c_ft c_fm'.'
```

Multistream Pipelines

Imagination sets the limits for multistream pipelines; here we show two simple examples without attempting to explain how multistream pipelines work in general. Refer to the tutorial or reference manuals for further information.

A program reads and writes the pipeline through a *stream*. When the program has access to several streams, they are named the *primary stream*, the *secondary stream*, and so on. A stream has an input side and an output side. The input side reads from the left-hand neighbour (what is before the previous |); the output side writes to the right-hand neighbour (what is after the next |).

CMS Pipelines has many built-in *selection* programs to select subsets of the input file that satisfy some selection criterion. (*locate* has already been

used.) Selection stages discard records that are not selected when the program is in a straight pipeline. With multistream pipelines, selection stages direct rejected records to the *alternate output stream*, if defined.

Count Connected and Disconnected Users: As an example, the pipeline in Figure 3 displays the count of connected users and the count of disconnected users. Figure 2 shows the topology of the pipeline.

There are two pipelines in this example: the left-hand one is the *primary stream* for the programs in it. The wide boxes represent programs that use two data streams: the right-hand pipeline is the *secondary stream* for these programs. (Both of the *count* and *change* filters read and write their primary streams.)

Some trickery is needed to transform this two-dimensional picture into a parameter string which must of nature be one-dimensional. An *end-character* separates pipelines in a pipeline specification. It ends one pipeline and begins the next. There is no default end-character; it must be declared in each multistream pipeline.

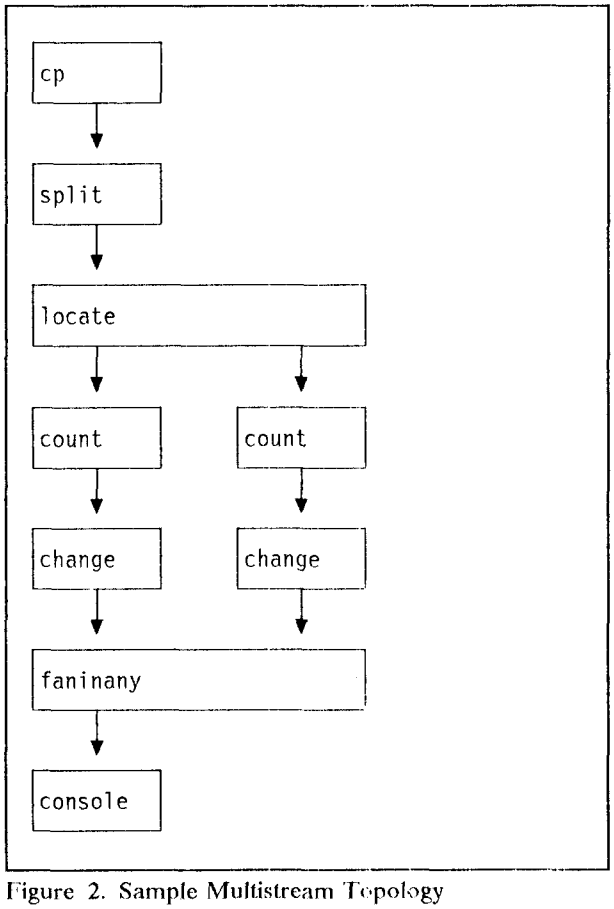


Figure 2. Sample Multistream Topology

```
/* Count logged and disconnected */
signal on novalue
address command
'PIPE (end \)',
   ' cp query names',
   '|split ,',
   '|l:locate /- DSC/',
   '|count lines',
   "|change //Disc'd: /",
   '|f:faninany',
   '|console',
   '\l:',
   '|count lines',
   '|change //Logged: /',
   '|f:'

exit RC
```
Figure 3. LND EXEC: Count Users

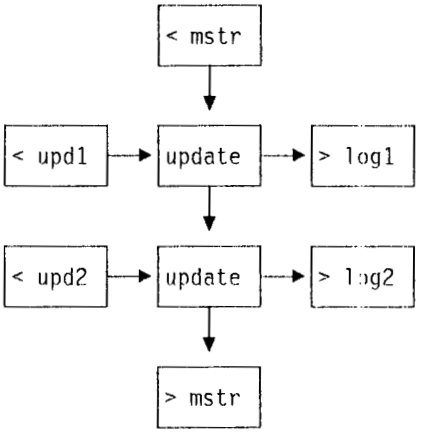
Parentheses at the beginning of the pipeline specification enclose *global options*. The end-character is one such. The backslant (\) is defined as the end-character in Figure 3.

`l:` and `f:` are *labels*. Both are used twice in this sample. The first time a label is used declares the *primary stream* for the particular invocation of the program written after the label. In the case of *locate*, it reads from the primary input stream (what is before it); it writes lines with the string to the primary output stream (what is after it). *locate* writes records without the required string to the *secondary output stream*. The secondary output stream is declared the next time the label is used (after the end-character in this sample). Whereas *locate* reads one input stream and writes to two output streams, *faninany* reads records from whichever input stream has one. *faninany* writes all records to the primary output stream. In this example it merges the lines with the count of selected and discarded records.

```
lnd
Disc'd: 130
Logged: 46
Ready;
```

Update: Many built-in programs support multi-stream pipelines. As an example, the *update* built-in program provides a subset of the function of the CMS UPDATE command. It reads the master file from the primary input stream and writes the updated file to the primary output stream. It reads the update from the *secondary input stream* and writes the update log to the *secondary output stream.*

update does not perform multilevel updates under the control of a control file. As a typical example of applied *pipethink*, *update* programs are cascaded (written one after the other) to implement multi-level updates. A controlling program reads the control file and auxiliary control file(s) to determine which updates to apply and their order.



Changing Pipeline Topology Dynamically

A pipeline program can issue pipeline commands to change the topology of its connections to other pipeline programs. The command CALLPIPE runs a *subroutine* pipeline; the program issuing the command resumes when the subroutine has completed. ADDPIPE adds a new pipeline to the set of running pipelines.

Subroutine Pipeline: Subroutine pipelines often hide the details of a task; they are the easiest way to create new pipeline filters.

In the previous examples, the sequence of *cp*, *split*, and *strip* was used over and over again. This example shows how to put these programs into a subroutine, USERS REXX, that can be called as a program.

A subroutine pipeline is likely to see more use than a cascade of filters in any one pipeline. Make sure it works in general, not just in the context where the cascade of filters comes from. In this case, the CP response is too long for the default buffer size when the system has between 400 and 500 users logged on. To ruggedise USERS REXX, a pipeline is added to query the number of users logged on, and allocate sufficient buffer space to hold the reply to the query. The second pipeline is the subroutine implementing the cascade of filters. (It also deletes lines listing virtual machines connected to the *CCS system service.)

USERS REXX

```
/* USERS REXX: Write a Line for each user */
signal on error

'callpipe',
   ' cp query users',
   '|strip',
   '|chop before 40',
   '|var users'

'callpipe',
   ' literal QUERY NAMES', /* Command    */
   '|cp' users*25+100, /* Issue CP       */
   '|nfind VSM_-', /* Ignore VTAMS       */
   '|split ,',    /* One line for each   */
   '|strip',      /* Strip leading blank */
   '|*:'          /* Pass on to next     */

error: exit RC
```

The argument to CALLPIPE is a pipeline specification like the argument to the PIPE command, with a difference. *: is a *connector* to show where to connect the input and output streams of the calling program. As used in this example, it specifies that the output stream of the subroutine pipeline is to be connected instead of the output stream of the calling program. The calling program's output stream is restored when the subroutine returns and the caller continues after the CALLPIPE command is complete.

Using USERS REXX, the combined function is performed by the command below. (It is late in the day, so the number of disconnected users has gone up since the last sample.)

```
pipe users|locate /- DSC/|count lines|console
131
Ready;
```

INCLPACK REXX

```
/* Include package files recursively */
signal on novalue

call dofile
exit

dofile: procedure
parse arg stack
do forever
   'readto in'
   If RC¬=0
      Then leave
   If left(in,7)¬==' &1 &2 '
      Then iterate  /* Comment */
   'output' in      /* Write line */
   parse var in . . fn ft fm .
   If ft¬='PACKAGE'
      Then iterate  /* Not a package */
   fid=fn'.'left(fm,1)
   If find(stack, fid)>0
      Then iterate  /* Recursion */
   'addpipe <' fn ft fm '|*.input:'
   If RC/=0
      Then exit RC
   call dofile stack fid
   'sever input'
end
If RC=12  /* EOF? */
   Then return
exit RC
```

Parallel Pipelines: The ADDPIPE pipeline command adds a pipeline specification to the current set of pipelines without suspending the program that issues the command. It can add programs, for instance, to process the input stream or divert the output stream temporarily.

As an example, INCLPACK REXX processes an input stream in the format used to describe files on the *CMS Pipelines* distribution tape (a PACKAGE file). Such a file has ' &1 &2 ' in columns 1 to 7; the file name, type, and mode are in the next 20 columns.

This program has a recursive procedure to process a file. The argument string to the procedure is the path of open package files. The loop body reads a line, checks if it identifies a file (otherwise it is assumed to be a comment that is discarded). The input line is copied to the output stream and inspected to determine if it represents a nested package file that has not already been processed in this path.

ADDPIPE puts the current input stream on a stack of dormant primary input streams for the stage and connects the primary input stream to < which reads the package file. The procedure `dofile` is called to process the package file. When done, the input stream (which is now at end-of-file) is severed. This re-instates the stream on top of the dormant stack to continue reading the file that referenced the one just done.

```
pipe < allpipe package|count lines|console
>11
>Ready;

pipe < allpipe package|inclpack|count lines|console
>126
>Ready;

pipe < allpipe package|inclpack|sort unique|count lines|console
>126
>Ready;
```

Figure 4. Processing a Package Recursively

```
pipe literal 60|dup *|delay|spec /lnd/ 1|subcom cms
```

Figure 5. Sample Event-driven Pipeline

Event-driven Pipelines

Most pipelines process lines as quickly as they are read from the host interface (for instance a tape or a CMS file). A few device drivers, however, wait for events and write a line to the pipeline when the event occurs:

- *delay* writes a line after an interval has elapsed or at a particular time-of-day.
- *immcmd* writes a line with the argument string when a particular immediate command is issued by the user at the terminal.
- *starmsg* connects to the message system service. It writes a line whenever CP presents a message or response to it.

These device drivers support pipelines in service machines to process user requests sent, for instance with SMSG, as well as authorised commands entered from the terminal when the virtual machine is connected, or sent with the SEND command from the secondary user.

The example of an event-driven pipeline in Figure 5 shows how to issue the LND command in Figure 3 on page 5 once a minute.

literal writes a literal 60 (the number of seconds to wait) into the pipeline.

dup makes an infinite number of copies of the line. (But only one at a time; this does not flood the pipeline.)

delay reads a line; the first word specifies when it must copy the line to the output. In this example it is the number of seconds to wait. The input line is copied to the output after the delay. Having written the line *delay* reads another input line and waits for 60 seconds once more. Thus, *delay* writes a line every 60 seconds.

spec is a program modelled on the COPYFILE option SPECS. As used here, it writes an output record with the literal string lnd for each input record (it does not reference fields in the input record). *spec* does not delay the record; in this pipeline it writes a record once every 60 seconds.

subcom passes input lines to the CMS subcommand environment which issues them with full command resolution. The response is written directly to the terminal by CMS.

How CMS Pipelines Works

CMS Pipelines is in two module files: PIPE MODULE is a small transient bootstrap module; the main pipeline module is PIPELINE MODULE. The main module can be disk resident or installed in a shared segment. A disk resident module is installed as a system nucleus extension; it is called from PIPE MODULE to install a PIPE user nucleus extension. The bootstrap module is not called by CMS once the main module is installed.

With this set-up, the pipeline code is protected, but CMS considers pipeline programs as user programs and recovers from an ABEND.

Filter Package: A filter package is a module file that contains filters with an entry point table defining its programs and optionally a message table for messages specific to programs in the filter package. The filter package also has a glue module that attaches it to the main pipeline module. A filter package is in a shared segment or NUCXLOADed. Once loaded, the filter package identifies itself to the pipeline module using an unpublished protocol; from then on programs in the filter package are considered an extension to the main pipeline module.

Four filter packages are installed automatically, if present, when the main pipeline module is initialised:

PIPPTFF Filters in this package replace built-in filters. This allows the replacement of some built-in programs without regenerating the main pipeline module. It also provides a convenient way to test fixes to built-in programs.

PIPSYSF System filter package. This is intended for programs to be available enterprise-wide.

PIPLOCF Local filter package. Filters available to all users in a particular system or installation.

PIPUSERF User filter package. A user can create a user filter package with private filters that are used often and thus should remain in storage.

A filter package can have any name. If a filter package is invoked as a CMS command, it installs itself as a nucleus extension (if not already one) and attaches its tables to the main pipeline module. Thus, to ensure that the contents of a filter package are available, one only has to issue the name of the package as a CMS command.

```
/* post processor */
address command
'PIPLSTPP' /* Ensure installed */
'PIPE < some listing|postproc ...'
```

Scanning a Pipeline Specification

The argument string to the PIPE command, as well as the CALLPIPE and ADDPIPE subcommands, is a pipeline specification that is processed by the parser. Having determined the over-all topology of the pipeline network, the parser resolves entry points and allocates working storage for programs that specify their requirements in a program descriptor. When the parser finds no errors in the pipeline specification, the control block structure is passed to the dispatcher for execution.

Resolve Entry Points: Entry points are resolved via *entry point tables*; each entry has the external name, flags, and a pointer.

Entry point tables are searched in this order:

1. The PIPPTFF filter package. This filter package is intended to hold replacements for built-in programs.
2. Built-in programs. These programs are in PIPELINE MODULE.
3. The PIPSYSF, PIPLOCF, and PIPUSERF function packages and other filter packages installed by the user or installation. The packages are searched in the order they are installed; by default, PIPSYSF is searched first.
4. Programs in the PIPPRV entry point table. This entry point table is intended for installation use to identify programs linked into the PIPELINE MODULE. The module shipped has no PIPPRV entry point table.

If an entry point is not resolved in any of these entry point tables, *CMS Pipelines* looks for a file with file type REXX (using EXECSTAT) and invokes the program as a REXX filter if one is found.

The entry point as resolved by look-up in an entry point table is not necessarily the first instruction of the program to run. The entry point table can specify that the entry point requires a high-level language runtime environment, or that the particular type of entry point be determined from inspection of storage at the address resolved so far.

When no high-level language is indicated, the entry can be an alternate format EXEC, an executable instruction, or a byte of binary zeros indicating an entry descriptor.

An alternate format EXEC is assumed to be a REXX filter. It is invoked with suitable parameter lists[1]. Other executable entry points are assumed to require CMS parameter lists (both extended and tokenised).

Entry Descriptor: An entry descriptor is defined by *CMS Pipelines* conventions. It has a byte of binary zero followed by three bytes of lowercase characters defining the type of descriptor:

cmd A pipeline command to be issued. The following fullword is the length of the command which follows. The command is usually CALLPIPE to invoke a subroutine pipeline to implement the function.

ept Another level of entry point table. The next word of the filter definition is looked up in the table that follows the descriptor.

---

[1] Due to the *CMS Pipelines* restriction that programs must not issue pipeline requests from commands (subroutines) that are called with CMSCALL macros, the runtime environment is called with a BALR; the runtime environment must be a nucleus extension or install itself as a nucleus extension when called (using CMSCALL) with a null parameter list.

lup A look-up routine (for instance *ldrtbls* to find an entry point in the CMS loader tables). The next word of the filter definition is passed to the look-up routine. It returns the resolved entry point address, or zero when the entry point cannot be found.

rex A REXX program that has been processed by the PIPGREXX filter to generate an in-storage program. The next word in storage is the length of the list that follows, in bytes. The program list (pairs of addresses and lengths) follows.

pip The entry address is the beginning of a program descriptor.

The entry point resolved by a second level of entry point table or by a look-up routine is inspected for an entry descriptor. These can be nested to any depth.

Program Descriptor: The program descriptor defines a built-in program to *CMS Pipelines*. It specifies attributes of the program that allow the pipeline parser to:

- Perform checks that are done by the program itself in a traditional implementation. For instance, does the program require arguments, must there not be arguments, or are arguments optional? Checking syntax before starting the pipeline means that the complete pipeline can be aborted when an error is found in the parameters to a single program.
- Allocate storage for all programs with one call to the host system storage management. The descriptor states the amount of storage to be allocated on the initial entry. Work areas for neighbouring invocations of programs are allocated adjacent; this may reduce the working set.
- Call a syntax exit, if specified, to perform further argument scan. For instance, the syntax exit can ensure that a disk file to be read does exist.
- Obtain the address of the main entry to call when no syntax check fails.

Commit Level

The commit level is an integer. A program starts on a particular commit level. The program advances its commit level to co-ordinate its progress with other programs. When a program returns on its original invocation, the return code is inspected and an aggregate return code is computed for the pipeline specification.

The programs that start at the lowest commit level are invoked first. This set of programs run until each of them returns or issues a COMMIT request to increase its commit level. The commit level is increased when there are no programs left at the original commit level. Programs on the new commit level are started only if the aggregate return code is zero at the time the commit level is reached; programs that start on a commit level are abandoned if any program has returned with a non-zero commit code at a lower level of commit. Programs that were started at a lower commit level receive the aggregate return code as the return code for the commit when the requested commit level is reached.

The convention for all *CMS Pipelines* built-in programs is that they transport data on commit level 0; most of the built-in programs start on commit level 0 as well.

The syntax exit can be considered to be commit level minus infinity.

The syntax exit must not allocate resources (for instance open files or obtain storage) because these resources are not released if some other syntax exit fails. On the other hand a program can allocate a resource on, for instance, commit level -1. It can then increase its commit level to 0. If the return code on the commit is not zero, the program can de-allocate the resource and exit; it can continue if the return code is zero.

When a subroutine pipeline commits to a level that is higher than the one of its caller, the caller commits to this higher level before the subroutine's commit completes. A subroutine pipeline can be abandoned before it commits its caller when there are errors in the subroutine; the return code can cause the caller's pipeline to be abandonned too.

Most built-in programs process records of any length. To do this, they typically process a record this way. Processing stops when a non-zero return

code is received. A positive return code indicates end-of-file; a negative one indicates a stall (deadlock).

- Preview the input record. The address and length of the record is provided. The record is not moved in storage.
- Process the record. If the output record is a subset, the address and length from the preview are modified without moving the record. A record that is modified must be loaded into a buffer in the program that processes it.
- Write the output record. An unmodified record is written from the producer's buffer; a modified record is written from the program's own buffer.
- Release the input record with a read into a buffer of length zero. This lets the producer continue.

Data Transport

CMS Pipelines transports records between pipeline programs without buffering. A record is moved in the pipeline when the left-hand side of a connection is writing and the right-hand side is reading.

The most important functions of the device-independent interface are:

- Write a line. The program provides the address and length of a buffer where the record is stored. The program is suspended until the right-hand side performs a read operation. The number of bytes read by the other side of the connection is returned.
- Read a line, moving it into a buffer or work area. The program specifies the address and length of the area into which the next input record is stored. The program is suspended until the left-hand side performs a write operation. The number of bytes stored is returned.
- Preview the next line. The address and length of the next line are returned. The program is suspended until the left-hand side performs a write operation. This function does not read a line; successive previews return the same record. The program on the left-hand side remains suspended in its write call until the record is read into a buffer or released with a read call for zero bytes.
- Select a particular stream for subsequent reads or writes, or both. The program can also select whichever input stream has a record available; in this case, it is suspended if no input stream has a record available.
- Sever a stream. The connection to the other side is broken. End-of-file is reflected on the other side.
- Short-circuit the currently selected input and output streams. The streams on the left-hand and right-hand neighbour are connected directly as if the program has never been in the pipeline. This is convenient for programs that inspect the beginning of a file to determine if any particular processing is required. Shorting the connections avoids the overhead of copying the rest of the file.

REXX Interfaces

CMS Pipelines supports REXX in two ways:

- REXX programs can process pipeline data. In this case, the program issues commands to transmit data to and from the pipeline. Such programs are started on commit level -1; they are committed to level 0 when they issue a pipeline command to transport data, or an explicit COMMIT pipeline command. Thus, if the program discovers an error in its arguments, it can return with a return code before the implied commit; this causes the pipeline to be abandoned. Likewise an error that causes a subroutine pipeline to be abandoned can be propagated to the calling pipeline which can then also be abandoned.
- Device drivers can access variables in a REXX environment that is active at the time the pipeline specification is parsed. The REXX program is passive; it performs no action to make this happen.

REXX Pipeline Commands: Because filters run as co-routines, REXX filters do not in general return to the caller in the reverse of the order they are started. REXX filters are invoked by a branch to the address in AFXEC in NUCON instead of an SVC (or CMSCALL); thus, all REXX programs in a pipeline run on the same SVC level. On MVS, REXX filters run in re-entrant environments.

This is the reason why `readto` and `peekto` (which previews the next record) are commands with side effects rather than function calls: REXX calls an external function with SVC (or CMSCALL).

Commands in the REXX filter are processed using *Non-SVC Subcommand Invocation*. REXX programs use the Address instruction to issue commands to other environments.

As REXX programs are dispatched, *CMS Pipelines* maintains the CMS subcommand stack to ensure that the topmost EXECCOMM represents the running program.

Access to REXX Variables: The address of the most current EXEC or REXX environment is obtained (using SUBCOM) when a pipeline specification is parsed. This is the base environment for all device drivers that access REXX variables. To avoid interference from REXX stages in the pipeline, device drivers branch directly to EXECCOMM using this environment (or an earlier one if requested).

Dispatcher Strategy

At the current commit level, the dispatcher maintains a stack of programs that have not started, or are ready to run. Programs that are committed to a higher level than the current one are kept on a separate list; they are moved to the dispatch stack when the dispatcher commit level is increased to the level that the program are committed to.

Initially the dispatcher stack has the rightmost program in the pipeline specification at the bottom; the leftmost program is started first.

A program runs until it calls the pipeline dispatcher to transport data or perform some other function. As an example, refer to the pipeline in Figure 1 on page 2.

cp issues the command to CP and gets the response in a buffer. It calls the pipeline dispatcher to write the first line into the pipeline. The dispatcher checks the program at the other end of the connection to see if it is ready to read the line. *split* is not waiting for input, it is ready to run and not started, so *cp* is suspended (waiting for output to be consumed) and *split* is started.

split calls the pipeline dispatcher to get the address and length of the next input line. (The line is not moved in storage.) The line is available to the dispatcher, so the information is returned and *split* is resumed. It locates the first comma in the input line and calls the dispatcher to write the part of the line up to the comma.

In the same way, *locate* is started. It inspects the line. Assuming the first line is for a connected user, *locate* calls the dispatcher to indicate that it has finished with the input line. The dispatcher makes both programs ready to run. To pump data out of the pipeline as quickly as possible, the dispatcher puts the right-hand program last on the ready stack, so *locate* is resumed once more. It calls the dispatcher to get another record and is suspended waiting for input to be made available because *split* has not yet written the next line.

split is resumed to provide the second record. This process is repeated for each record in the input file.

cp returns on the initial invocation when all lines are processed. The pipeline dispatcher severs all streams available to a program (in this case there is only the primary output stream). Severing the stream which *split* is waiting for sets return code 12 and makes the program ready to run.

split is resumed. It notes the return code meaning end-of-file and returns as well. This reflects end-of-file to *locate* which also returns. *count* gets end-of-file and writes a line with the count on its primary output stream. *console* is finally started to process the line and write the response to the terminal.

How CMS Pipelines Evolved

CMS Pipelines evolved over the 1980s. The first implementation ran on VM/System Product Release 1; the parser used the tokenised parameter list—the untokenised command string was not available to a CMS command in those days. The first built-in programs supported the console, disk files, and virtual unit record output devices. Filters were resolved from a few built-in device drivers and the CMS loader tables. A pipeline was run by calling the parser (with a BALR instruction). This implementation was convenient to write CMS user area modules.

VM/System Product Release 2 introduced NUCXLOAD to load relocatable modules from a LOADLIB into free storage as commands. A

command interface was written to support this. Because NUCXLOAD was a transient module originally, it was not practical to have a bootstrap module; an EXEC was used instead. It ensured that the pipeline module was installed in storage before invoking it.

By early 1982 it was clear to insiders that REXX would be part of VM/System Product Release 3. An interface was quickly written when it was realised that:

- The language is attractive to process data.
- The interpreter is re-entrant.
- The mechanism for Non-SVC Subcommand Invocation allows subcommands to be issued on one CMS nesting level.
- The system interfaces (after some tweaking) are suitable to maintain concurrent invocations of REXX programs.

The parser was rewritten to use the extended parameter list on VM/System Product Release 3.

There were several attempts at multistream pipelines and dynamic reconfiguration at this time. After some experimentation, the pipeline specification found its current form in the summer of 1985.

VM/System Product *CMS Pipelines* Program Offering (5785-RAC) was announced on October 6, 1986.

NUCXLOAD was made nucleus resident in VM/System Product Release 4. The PIPE bootstrap was written to avoid going through an EXEC to run a pipeline.

The program descriptor was introduced in Modification Level 2 which shipped in November 1987. XA toleration was shipped in Modification Level 3 in December 1988. Modification Level 4, shipped in October 1989, provided XA exploitation and support of PL/I and IBM C/370. Modification level 5, shipped in August 1990, provided support for commit levels and VM/ESA.

Virtual Machine CMS Pipelines RPQ P81059 was announced October 31, 1989.

Conclusion

CMS Pipelines moves CMS away from the single-task single-program model. *CMS Pipelines* is attractive because it:

- Makes the system more efficient and responsive. Passing data (in storage) between programs saves I/O operations. Running co-routines saves processor time relative to calling subroutines.
- Makes the programmer more efficient. The user and the programmer can often plug functional building blocks together without having to worry about procedural code. A solution is often expressed as a subroutine pipeline that can be called from other programs. Filters are easily added to tailor existing solutions.
- Makes programs more robust. A filter is tested out of context, and often exhaustively. It is easy to perform a regression test.
- Supports REXX as a programming language both to write command procedures that use pipelines for processing, and as programs in the pipeline processing data.
- Provides multistream pipelines. Selection filters can split a file in streams that are processed in different ways. Programs using multiple streams can be cascaded.
- Supplies a library of more than 100 built-in programs to access host interfaces and operate on data.

References

CMS Pipelines Tutorial, GG66-3158, explains *CMS Pipelines* in 15 easy chapters with many examples.

CMS Pipelines User's Guide and Filter Reference, SL26-0018, has a task-oriented guide to *CMS Pipelines* and a reference section describing built-in programs and messages.

CMS Pipelines Toolsmith's Guide and Filter Programming Reference, SL26-0020, describes multistream pipelines, the REXX interface, and the original Assembler programming interface.

CMS Pipelines Installation and Maintenance Reference, SL26-0019, describes maintenance procedures, and how to generate a filter package.

EXPERT SYSTEM DESIGN IN REXX

MARC VINCENT IRVIN
NORDEN SYSTEMS

EXPERT SYSTEM DESIGN IN REXX

By Marc Vincent Irvin

Because REXX has user friendly syntax, is lightly typed, and handles symbols well it was an ideal medium for a Knowledge Engineering systems experiment named REXRULES.

Expert System Design In REXX

Introduction

- WHAT ARE EXPERT SYSTEMS
  - history
  - players
  - structure
  - paradigms
  - justifications
  - applications
- REXRULES: ES INFERENCE IN REXX
  - assets
  - facts
  - rules
  - chainings
  - attributes
  - pros/cons
- ES/REXX PROMISE

WHAT ARE EXPERT SYSTEMS?

- HISTORY
  from games to chemistry to medicine to value seen
- PLAYERS
  hosts, tools, shells, experts, engineers, & users
- STRUCTURE
  knowledge, inference engines, and heuristics
- PARADIGMS
  Emycin, Prolog, OPS5, TK-Solver, and Expert Choice
- QUALITIES
  high ROIs by saving, promoting, and enforcing work
- APPLICATIONS
  DASD, Software, JOB, Network, and Help Desk management

DEFINITIVE DEFINITION

- An EXPERT SYSTEM is one that assists facillitates, or replaces expert(s).
- ASSISTS - When the expert calls on the system to improve his/her performance.
- FACILLITATES - When the expert's skills improve the performance of non-experts.
- REPLACES - When expert(s) develop expertise exceeding human capabilities.

What Are Expert Systems?

HISTORY

- PARLOR GAMES
  Turing asked, can it pass a line up?
  Parlor game provided measuring rod.
  Eliza was therapist, PARRY was patient.
- NASA's ROBOT Chemist
  DENDRAL uncoded molecules.
  DENDRAL out did the experts.
- Bacterial Infection Diagnosis
  MYCIN gave RXs and explained reasoning.
  MYCIN led to EMPTY MYCIN or EMYCIN.
- NEED SEEN: Future Shock's Answer
  MIT, computers didn't up productivity.
  10-15 yrs to get expert in something.
  Experts decode & sift $K into solutions.
  ES are industrial complexity pills.
- BEST AI SOLUTION AMONG MANY
  NL, Robotics, Neural Nets, & Fuzzy Logic.

What Are Expert Systems?

PLAYERS

- Construction Materials
  computers, languages, shells, and boxes.
- Architect (Cogpsych/CS/AI)
  Knowledge Engineers put it together.
  Trained in extraction & compaction.
- Domain Expert
  Provides subjective functional analysis.
  KA blues, busy silent vague and distant.
- ES Users
  They get advised, directed, or corrected.
  Potential next generation of experts.
  Complex work with little or no training.
- ES Support
  Keep facts and rules up to date.
  Done by any of the above, or others.

What Are Expert Systems

STRUCTURE

- KNOWLEDGE
  Facts
  Rules
  Frames
  Attributes
- INFERENCE ENGINE
  Interpreter
  Scheduler
  Reporter
- HEURISTICS
  Expert observation
  Metabase experience

What Are Expert Systems

STRUCTURE: Knowledge

- FACTS ie. Can is bent, top is not off.
  data element or multivalue variables
  monotonic, nonmonotonic, or uncertain
  discrete or inheritable via attributes
- RULES ie. If can is bent then can is b
  has rulename for infer, fuzz, & reporting
  name: LHS/antecedent RHS/consequen
  can use old facts to make new facts
- FRAMES (objects attributes values)

CONCEPT: part
SLOT1: name
SLOT2: condition

INSTANCE: part
NAME: can
CONDITION: bent
PROCEDURE: part_fixer

What Are Expert Systems

STRUCTURE: Inference Engine

- Inference Engines contain strategies and controls that KE use to manipulate facts and rules. Its 3 main functions are to interpret, schedule, and explain facts and rules to and for its users.
- INTERPRETER
  Maps attributes against facts and rules.
  Sets and stores processing options.
  Does var/memory inits and runs profiles.
  Does syntax checking and writes errors.
- SCHEDULER
  Seeks goals, fires rules, finds unknowns.
  Sets firing priorities and tracks steps.
  Interfaces procedurally to environment.
  Does message sending and retrieval.
- EXPLAINER
  Tells in english how answers were made.

What Are Expert Systems

STRUCTURE: Attributes

- ATTRIBUTES - Used for query, process controls, and english status reporting.
- USER QUERY
  Usually, if knowbase has no answer it will ask user for answer using A) a KE made text or B) an invented text.
- PROCESS CONTROLS
  Things like value and range checking, logic tracking, fuzzy or confidence factors, and defaults are often found.
- ENGLISH STATUS REPORTING
  Sometimes, IE will provide english like responses about outstanding RULES and set values. They may be KE supplied, but better IEs will give current reasoning and value settings.

What Are Expert Systems

STRUCTURE: Chainings

- Demo Ruleset for chaining
  GOAL = END -OR- RETE = ON
  R1: IF A = J & C = 2 THEN END = 1
  R2: IF D = M & C = 1 THEN C = 2
  R3: IF C = 2 THEN A = J
  R4: IF D = M THEN C = 1
  R5: IF C ^= 1 THEN D = M

- forward chaining FIRE sequences
  R1 R2 R3 R4 (R5)
  R1 R2 R3 (R4) R5
  R1 (R2) R3 -R4- -R5-
  R1 R2 (R3) -R4- -R5-
  (R1) R2 -R3- -R4- -R5-

- backward chaining FIRE sequence
  R1 R3 R2 (R5) R2 (R4) (R2) (R3)
  (R1)END

What Are Expert Systems

PARADIGMS

- M.1
  Mycin backward chained
  certainty factors

- PROLOG
  declare/proc Logic base
  clause driven

- OPS5
  RETE forward chained
  data driven popular in MIS

- TK-SOLVER
  ESS w/o cells

- EXPERT CHOICE
  DSS w/o PHD

What Are Expert Systems

QUALITIES

- 1000 pct return on investment
  Northrop, ESP - 14 to 4.3 hrs
  DEC nets $40M/yr on XCON
- VALUABLE SKILLS RETAINED
  retirees knowledge coded
  career changes painless
- USEFUL SKILLS PROMOTED
  Easy, 24 hr, error free access
  Automatic history of reasoning
- STANDARDS ENFORCEMENT
  critical tasks
  routine tasks
- COMPLEXITY MANAGEMENT
  information overload
  overboard technology

What Are Expert Systems

MIS APPLICATIONS

- DASD DEFINITIONS
  performance/security enforcing
  parameter optimizing
- SOFTWARE MAINTENANCE
  advice and error recovery
- JOB MANAGEMENT
  scheduling and prioritizing
- NETWORK SUPPORT
  EP/VTAM/NETVIEW line servers
  DASD and SPOOL monitoring
- HELP DESK
  expert directory & basic fixups
  problem logs and tracks
- AUTOMATED OPERATIONS
  Operators/Tech-support replaced
  Light's Out has become common

Expert System Design In REXX

REXRULES: Expert Systems In REXX

- ASSETS
  untyped, pseudo code, symbolic, and portable
- FACTS
  literals, variables, stored, filed, & iterative
- RULES
  value, clause, data, formula, clock, & self driven
- CHAININGS
  backward, forward, mixed, and custom(depth/breadth)
- ATTRIBUTES
  basic, truth, dynamic, imprecise, and fuzzy values
- PROS/CONS
  learning, I/O, math, connects, and procedural

Expert Systems In REXX

REXX ASSETS

- UNTYPED MEANS:
  no time defining fields
  unlimited mixing
  SAY 'Value of' X 'is' N*3'.'
- PSEUDO CODE MEANS:
  coders need little training
  code can be self documenting
  IF A = B THEN SAY 'A=B'
- SYMBOLIC MEANS:
  words & phrases paramount
  unknown symbols can be found
  IF SUNNY & WARM THEN 'SWIM'
- PORTABLE MEANS:
  PC code can run on VM & MVS
  one day REXX IBM on REXX DEC
- CONNECTED MEANS:
  excellent host interfaces?

Expert Systems In REXX

REXRULES FACTS

- LITERAL
  IF IDEA = 'GOOD' THEN JOKE = 'FUNNY'
  INITGOAL: IDEA = GOOD; TIME = LATE

- ASSIGNED (FALSE = 0)
  IF DAY OR SUNSET THEN NIGHT = FALSE
  rules/tasks can set true/false values
  popular in diagnostic systems

- SYMBOLIC
  PUT(Tim sees Ann); PUT(Ted sees Sue not)
  UNKNOWNS = 'WHO'; GET(Tim sees who)
  multiple UNKNOWN solutions sep'd by space
  advanced pattern match, GET(RANGE'>')

- FILE BASED
  X = READ(filename,seq,[key]) for basic files
  RC = WRITE(filename,seq,data) to write recs
  OPS(MAKE,'','UID1' DAY TIME)
  IF OPS('','UID1 = "UID1"') THEN 'IDFND'

- ITERATIVE - experimental='Tim Ted Sue Ann'
  One pass done for each name.

Expert Systems In REXX

REXRULES RULES

- VALUE DRIVEN
  unset values Q'd for OOS resolution
  FORM(s) => RULE(S) => ASK.X
  Examples...
  RULE_8: IF SPEED = LOW AND,
    PRT_QUALITY = HIGH THEN DO
     PRT_SET='MODL-1'; RUN(SAY_MODL); END
  RULE_2: IF ABBREV('YES',NEEDFAST,1) AND,
    PRTPAGES > 5 THEN SPEED = HIGH

- CLAUSE DRIVEN
  unfound patterns/GET() Q'd for solution
  RULE(s) => ASK.
  Examples... unknowns = 'sport'
  put(hockey has contact soccer has contact,3)
  put(hockey is fast hockey played_with puck,3)
  likes_sport: if nop(name) & get(sport has action),
      & get(sport played_with puck),
      then put(name likes_sport sport)
  has_action: if get(sport has contact) &,
      get(sport is fast),
       then put(sport has action)

Expert Systems In REXX

REXRULES RULES

- DATA DRIVEN
  matched objects/OPS() direct reasoning
  Examples...

```
OPS('FIL NAM A',QUEST,'NAME TEXT')
OPS(MAKE,QUEST,'QUES1 WHAT IS TEMP?')
OPS_RULE:
IF OPS(QUEST,'NAME = "QUES1"') THEN,
   OPS(SET,QUEST); SAY NAME TEXT
   OPS(REMOVE,QUEST)
   END
```

- FORMULA DRIVEN
  Unset values in formulas/EQU. get solved
  Examples...

```
MAINGOAL   = 'SHOTS'
EQU.BACBOOZ= (150/WGHT)*(PCT/50)*SHOTS,
           * .025
EQU.PCT    = PROOF/ 2
ASK.WGHT   = 'WHAT DO YOU WEIGH' NAME'?'
* Implied via rules of algebra...
EQU.PROOF  = PCT* 2
EQU.SHOTS=BACBOOZ/(150/WGHT*PCT/50*.025)
```


Expert Systems In REXX

REXRULES RULES

- CLOCK DRIVEN
  rules|forms|tasks on date|time|elapsed
  Examples...

```
/* in 30 minutes run CK_LINES once */
CLK.CK_LINES = 'M30'
/* on 3/3 run every 2min from 12 to 6*/
CLK.SUBMIT='91/03/03 12:00 0.M2*18:00'
CK_LINES:  'EXEC LINESCAN'
ACCTRULE: IF CLK.SUBMIT THEN 'SUBMIT X'
```

- SELF DRIVEN
  recursion done on rules|tasks|patts|masks
  recursion via FIRE|RUN|GET|OPS commands
  Example...

```
HANOI: RUN('MOVE 6 LEFT MIDDLE RIGHT')
MOVE:  PARSE VAR RUNSTR ?N ?A ?B ?C
  IF ?N ^= 1 THEN DO
     ?M  = ?N - 1
     RUN('MOVE ?M ?A ?C ?B')
     SAY 'MOVE DISK' ?N 'TO' ?C
     RUN('MOVE ?M ?B ?A ?C')
     END
```

Expert Systems In REXX

REXRULES CHAININGS

- BACKWARD using literals & assigned facts

```
maingoal = 'animal'
rule1: if mammal & carnivore & striped,
          then animal = tiger
rule2: if haired then mammal = true
rule3: if meateater then carnivore = true
ask.striped = 'is animal striped? Y/N'
equ.haired  = 1=1/1=0-false /* ans = 1 */
```

- FORWARD using file based facts

```
FIRE('PICK HOLD DROP STOP',FOR 100)
PICK: IF OPS(GOAL,'TASK="ADD"') AND,
         OPS(BRICK,'PLACE="HEAP"'),
        THEN OPS(MODIFY,BRICK,'A 10 HAND')
HOLD: IF OPS(GOAL,'TASK="ADD"') AND,
         OPS(BRICK,'PLACE="HAND"'),
        THEN OPS(MODIFY,GOAL,'DROP')
DROP: IF OPS(GOAL,'TASK="DROP"') AND,
         ETC...
```

- MIXED, unset value in fired rule starts GOAL. Once GOAL set fired rule resumes. Fires in GOALs takeover & when done GOAL resumes.

Expert Systems In REXX

REXRULES ATTRIBUTES

- STATIC

Sets query, process, & english info controls.

```
ASK.VIEW = "Select VIEW from:@" CHK.VIEW
DFT.VIEW = "FOR"       /* default reply */
CHK.VIEW = "FOR AGAINST"
FMT.VIEW = 1 7 ALPHABETIC 3
WHY.VIEW = "Need to know side you're on,",
           "OK!"  /* continuation example */
DOC.VIEW = "Only reply is FOR or AGAINST."
IDK.VIEW = "AGAINST"   /* ie. I don't know */
```

- TRUTH

Uses boolean true/false logic for REXX vars.

```
YES = 1; NO = 0; DONTKNOW = '@'
ASK.EATSMEAT = 'Does animal eat meat? Y/N'
CHK.EATSMEAT = 'CHK_YESNO:' /*set 1 | 0 */
IF EATSMEAT = NO THEN HERBIVORE = TRUE
IF EATSMEAT=DONTKNOW THEN EATSMEAT=1
IF EATSMEAT THEN DO
  HERBIVORE = NO; CARNIVORE = YES; END
 ELSE CARNIVORE = FALSE
```

REXRULES ATTRIBUTES

- DYNAMIC

Most attributes can execute tasks or rules.

```
ASK.NAME = "ASK_NAME:" /* use name task */
DFT.NAME = "DFT_NAME:" /* default task */
CHK.NAME = "CHK_NAME:" /*check name igc*/
PNL.NAME = "PNL_NAME: name addrs phone"
ASK_NAME: /* msges below shown to user */
  SAY "It's" TIME()", please enter name."
  SAY "Thank you."
DFT_NAME: /*assume William's using system*/
  if am then rspns = BOB; else rspns = WILL
CHK_NAME:
  IF AM THEN,
    IF FIND('BOB TIM JAY',RSPNS) > 0,
      THEN RUNCC = 1; ELSE RUNCC = 0
PNL_NAME:
 !    --- FULLSCREEN INFO ENTRY ---
  ENTER NAME: %NAME          !
    ADDRESS: %ADDRS                    !
     PHONE: %PHONE       !
EOP:
```

For @ panel var invalid dialog asks for input.

Expert Systems In REXX

REXRULES ATTRIBUTES

- CHRONOLOGIC

Rules can be run after N hrs, mins, or secs.
Rules can be run based on "date" or "time".
Rules can be run every N hrs, mins, or secs.
Variables used to signal if time has come.

```
CLK.PAYRULE = "M30" /*Do 30 mins from now*/
CLK.CHKDASD = "H1*21:00" /*@ hr, stop 9pm*/
* On May 30th tell operator's to go home.
CLK.MSGOPER = "91/05/30 09:00 0.M10*12"
* Issue clock request for 10 PM shutdown.
CLK.STOPRUN = DATE(O) '22:00 0'
PAYRULE: "MSG ALL IT'S PETTY CASH TIME."
* test of CLK. val needed to ctl miss fires.
CHKDASD:
  IF CLK.CHKDASD THEN FIRE(DASDRULES)
SPECIAL_RULE_CHECKS_CLOCK_STUFF:
  SELECT
    WHEN CLK.MSGOPER THEN RUN(TELLOP)
    WHEN CLK.STOPRUN THEN EXIT 000
    OTHERWISE NOP
    END
```

If no rule then only CLK. switch is set.

(C)Copyright 05/91 By Marc Vincent Irvin

Expert Systems In REXX

REXRULES ATTRIBUTES

- IMPRECISE

```
IF RUN(BIG: A) OR RUN(MED: A),
   AND ^RUN(LOW: A) THEN A_OK = TRUE
BIG: PARSE VAR RUNSTR X
IF X >= 9 THEN RUNCC = 1; ELSE RUNCC = 0
MED: PARSE VAR RUNSTR X; RUNCC = 0
IF X < 10 AND X > 5 THEN RUNCC = 1
```

- FUZZY (FUZ. is certainty factor if sw on)

```
CHK.QUALITY = " HIGH GOOD FAIR POOR "
FUZ_QUALITY = "0   1  .75   .5  .25 "
Output example:
   MEDIA_TO_CONSIDER      = 80%  FOILS
   MEDIA_TO_CONSIDER      = 92%  SLIDES
```

- EPISODIC

```
Assume FUZ. for quality/ ability/ quantity set
CHK.RATING = " QUALITY ABILITY QUANTITY "
DSS_RATING = "0   .345    .243     .161 "
Output example:
   RATING                 = .376 HAYAT, F
   RATING                 = .234 NISS, T
```


Expert Systems In REXX

REXRULES PROS/CONS

- PROS

Needs no compile, but compiler possible.
Can mimic most PROLOG & OPS5 code well.
Perfect medium for teaching AI skills.
Grade schoolers could learn this in hours.
Intensely flexible parse ability kept.
Ideal for LADDER like NL implementations.
Relatively little cost to buy & maintain.
IBM's SAA stand is less support overhead.
Embodies six plus popular IE paradigms.
Highly flexible calculator ability.
Infers using both dialogs and panels.

- CONS

Not very fast CPU wise, nor IO wise.
Lacks high math capabilities.
Bleeding edge...
Few people use or have heard of REXX.
MVS version lacks EXECIO's index feature.
VM REXX Compiler can't do INTERPRET cmds.
Does not do LISP like list processing.
Has few syntax checking features yet.
Has little to no documentation yet.

(C)Copyright 05/91 By Marc Vincent Irvin

Expert Systems In REXX
Towers of Hanoi in LISP

```
(defun tower-of-hanoi (disks from to spare)
  (unless (endp disks)
    (tower-of-hanoi (rest disks) from spare to)
 (format t "~%Move ~a from ~a." (first disks) from to)
    (tower-of-hanoi (rest disks) spare to from)))
```


Expert Systems In REXX
Towers of Hanoi in Prolog

```
  loc = right; middle; left
predicates
  hanoi(integer)
  move(integer, loc, loc, loc)
  inform(integer, loc, loc)
clauses
  hanoi(N) if
     move(N, left, middle, right).
  move(1, A, _, C) if
     inform(1, A, C), !.
  move(N, A, B, C) if
     M = N - 1,
     move(M, A, C, B),
     inform(N, A, C),
     move(M, B, A, C).
  inform(Disk, Loc1, Loc2) if
     write("\nMove disk ", Disk, " from ",
           Loc1, " to ", Loc2, ". ").
```

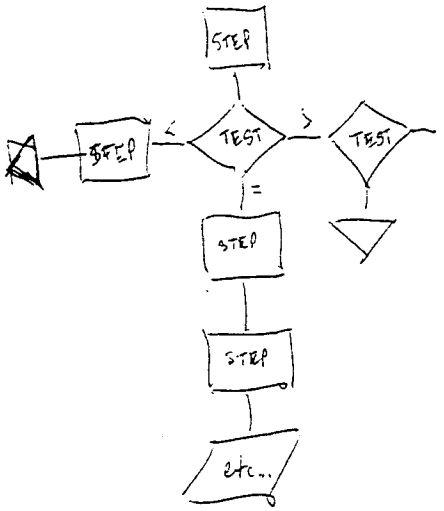
Expert Systems in REXX
Towers of Hanoi in REXX

```
HANOI: RUN('MOVE' SIZE 'LEFT MIDDLE RIGHT')

MOVE: PARSE VAR RUNSTR ?N ?A ?B ?C .
   IF ?N = 1,
      THEN SAY 'MOVE DISK' ?N 'TO' ?C
      ELSE DO
       ?M = ?N - 1
       RUN('MOVE ?M ?A ?C ?B ?N')
       SAY 'MOVE DISK' ?N 'TO' ?C
       RUN('MOVE ?M ?B ?A ?C)
       END
```

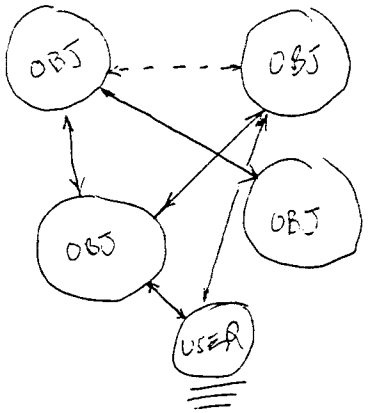

EXPERT SYSTEMS IN REXX

PROCEDURAL



Take out any step or test and logic collapses.

DECLARATIVE OR OBJECTES



Goes to whatever object will provide needed information

For example if any of the above were removed it would just try to get ANSWER FROM the user.

REVIEW OF ANSI AND OTHER LANGUAGE DISCUSSIONS

BRIAN MARKS
IBM

Preamble added April 1991, was not part of the original handout.

In addition to the caveats embedded in this document, note that:

The reaction of SHARE members included rejection of some of these suggestions and ideas for variations which were improvements.

The examples tend to be examples of the simplest case. It is a reasonable rule to assume that the generalizations which seem natural to you were also intended.

This Handout.

This handout accompanies the SHARE session number A636 - REXX Design Dialogue, speaker Dr Brian Marks, on Tuesday February 26th 1991.

Preamble

The Procedures Language Architecture Review Board is an IBM committee that defines the programming language and the interfaces that make up Procedures Language. The Procedures Language Interface Owner, Linda Green, selects particular parts of the committee's output as the SAA Procedures Language levels; so far there have been two levels, SAA level 1.0 announced in March 1987 and level 2.0 announced in June 1990. The committee members all work for IBM and I am the chairman.

The work of the committee naturally divides into work on reconciliation, (wherever there is doubt about what implementations should do to honour the architectural definition), and work to continue the original design principles of REXX into extensions. Today we are going to discuss the design of extensions. The design I will present is at a very early stage; it is in the minds of the Board members. There are no implementation plans and only one part of it is being prototyped. So IBM is making no commitments that any of this will be delivered, or that if it is it will be in the form that we are going to discuss today.

It is a fair question to ask "Why do any work on extensions?". REXX was well designed in the first place, so perhaps extension will do more harm than good by making REXX more complicated.

The argument in favor of change is that the world of computing has changed - what was an optimum design earlier may not be optimum for today or for the next decade. REXX has already benefited from some evolution over the years. The Review Board feels that further evolution may be justified by the trends of the 1990s. Such an evolution is prompted by, and built around, the views expressed by our customers.

The purpose of today's session is to begin a dialog with you about the Board's view of what might be appropriate; we call this 'Architected REXX'. The remainder of this session will be split into parts. In each of the parts I will describe a component of 'Architected REXX' and you will have the opportunity to discuss it. In order to keep to the timetable it may be necessary to guillotine discussion but I am sure there will be other opportunities.

The first of these parts relates to the trend for the problems that programmers are solving to be more complex. Although programmers today have better equipment than ever before, there is still a challenge in programming because our ambitions have increased. In the case of REXX, writing big programs exposes some limitations in the way variables are handled. As the number of variables in a program increases it becomes difficult to control the scope of variables with simple PROCEDURE EXPOSE statements. The recent extension of EXPOSE to allow a list of variables to be named is an improvement but users still tell us that better facilities for sharing and scoping are required.

Architected REXX postulates the addition of external procedures that share variables and parameter passing 'by reference'. Here is a foil with syntax. In the example the variable ABC is shared by the main program and the subroutine ALPHA.

```
/* Main Program */                    /* External procedure */
  .                                   ALPHA:PROCEDURE EXPOSE ABC
  call ALPHA                             .
  say ABC                                ABC=66
  .                                      .
  exit                                  return
```

This is the same syntax and meaning as is currently used for internal subroutines, but external routines today cannot start with a PROCEDURE statement. So this is a 'clean' extension - no correct existing program is 'broken', ie given a different meaning, by the extension. Also it introduces little in the way of new terminology and concepts.

By-reference addressing is not so 'clean'.

```
/* Calling code */                    /* Called routine */
  call BETA MYVAR.                    BETA:
  say MYVAR.33                         use arg GAMMA.
                                       GAMMA.33='Something'
                                       return
```

This program would do something today, but nothing very useful - because there is no 'use' statement today, that line would issue a command. That is not a serious breakage problem because the coder who wanted to do that would almost certainly have used quote signs around the word 'use'.

The use statement introduces a name for an argument, like PARSE ARG does, with the difference that the argument IS NOT COPIED. The name introduced is a second name for the same variable. In this example MYVAR and GAMMA are the same variable. This aliasing is a powerful feature and also a source of pitfalls for the unwary. There are alternative designs, mostly involving multiple values on the return statement, but the Board feels that passing By-Reference is the correct choice for execution speed, since return of multiple values could require more copying of data.

This new sort of external routine, that starts with a procedure statement, is also more like an internal procedure in the way that internal values (like the current number of NUMERIC DIGITS) are handled. Today's external routines reset these internal values when the routine starts; the new sort of procedure inherits the caller's settings in the way that an internal procedure does today.

We will take our first discussion period now. I believe the essential questions are:

- Does the expectation of more complex programming justify additions to REXX?
- Is adding External-like-Internal and By-Reference-Arguments enough to alleviate the difficulties in sharing that people have experienced?
- Are there better designs of language with the same power?

National Language Sensitivity.

Our second area for design dialog is National Language Sensitivity. Our meeting today has a majority of people whose natural language is English, and REXX is optimized to people who know American-English, so this may seem seem a minor design issue. However, there are two trends of the nineties that make it increasingly important. The first is an increasing number of non-English speaking programmers. The second is the explosion in communications which is making our world into a village and making possible individual applications which have widely spread parts.

We all know this is a hard problem to tackle - the complexities of code pages and character sets together with the variety of dialects and customs makes a daunting challenge. Fortunately we are not on our own - all the Programming Languages, the operating systems, and the components like SQL are involved. An IBM architecture is emerging - the Character Data Representation Architecture which you can hear more about at other SHARE sessions.

REXX has the advantage over some programming languages that it is defined in terms of characters rather than bytes, and the definition stands up whether the characters are physically represented as one, two or a variable number of bytes. The extensions in Architected REXX provide for:

1. Source programs written in the characters sets identified by CDRA. (Those that implementations support - we would expect that to be a large number.)
2. A set of rules for coping with the specialities of particular character sets - eg which characters are allowed in names, how substitutes are used for unavailable character sets, what Uppercasing means. (By the way, Lowercasing is in the design.)
3. Existing keywords, function results remain in English. To do otherwise would cause a lot of breakage.
4. New variations on the builtin functions allow, for example the day of the week to be returned in French. (This by retaining the same names for builtin functions but adding variety to the arguments, eg DATE('W') for the local form of weekday.)
5. Run time data which is not in the same character set as the source program is permitted. However there are no automatic conversions between character sets.

This design follows CDRA in the idea that data is 'tagged' with identification of the character set that the data is in. Whether these 'tags' or 'attributes' are actually present will depend on their operating system support. Our design allows for the character set to be given in a REXX-specific way if the operating system support for tags is not present.

Some of the NLS questions:

- Do the trends justify adding these features?
- Is the extent of the support appropriate? Or maybe we need keywords in non-English? Automatic conversions at runtime to some Universal character set?
- What is the best design, given the extent of the support?

Message Driven Processing.

Our next area for design dialog is Message Driven Processing. It will be characteristic of the nineties that many applications will be distributed, with parts of the applications on different machines and often geographically far apart. Such divisions need a clear way of specifying what data and functions belong to one part of the application as opposed to another. The message driven paradigm, also known as 'Object Oriented', has proved to be good for this. And of course object orientation has also proved good for other things, like manipulating windows on a screen, for the same reason as it is good for distributed applications - because of the 'data encapsulation'.

Architected REXX favors the Object Oriented style developed by the OO-REXX team. Simon Nash talked about this prototyping effort at SHARE74 and is giving an update on Wednesday at 0930. There will be an opportunity then for a detailed discussion. Right now, I will recap on the main features so that we can discuss how this fits with other parts of Architected REXX.

These Message Driven Programming facilities introduce only insignificant breakage so a programmer who does not want to use them need not know about them. Such a programmer can continue to program using non-object-oriented features and terminology. It will be a choice for the programmer whether to adopt the OO-REXX style.

Programs in the OO-REXX style use Methods, analogous to external procedures, with a new METHOD statement analogous to PROCEDURE. Methods provide two related facilities, the encapsulation of data (variables on a METHOD EXPOSE are shared only across the methods associated with an object) and a unit of execution that can be paralleled (the method invocation).

Methods are invoked by a new form of REXX term, the 'message term', or a new instruction, the 'message instruction'. Each of these has the syntax (in the simplest case) of a function call preceded by a term and the tilde character. eg rectarea = myrect~area; mystack~push('Bill Brown')

In the Object Oriented terminology, the object on the left hand side of the tilde responds to the message (on the right hand side) by returning a result object. The objects may be strings, in which case the newness may be solely in terminology and syntax.
'ABC'~REVERSE = = 'CBA' = = REVERSE('ABC')

However, the objects need not be strings. Objects are characterised by the methods that can be applied to them, and there are Builtin methods which will create objects and associate methods with them. In this way the usual object-oriented features of powerful objects, inheritance etc. are established mainly by the programmer. Only the essential primitives have been added to REXX in this enhancement. Any particular problem oriented solution, eg a windowing scheme, could be provided as a package of pre-programmed objects but will not be part of this extension.

The parallel nature of object activity is achieved by the addition of a REPLY statement analogous to the RETURN statement. REPLY does what RETURN does but additionally continues execution (with the statement following the REPLY). Where this might lead to unsynchronized shared access to variables the programmer should make use of 'guards'. The guard statement, with syntax GUARD expression, blocks execution until the expression evaluates to '1'.

Some of the high level message driven processing questions:

- Do the trends justify adding these features?
- Is the extent of the support appropriate? Too high because it adds a whole new set of concepts and extends the character set required for REXX? Too low because it only provides mechanisms, and does not define a comprehensive set of useful objects as a part of REXX?
- Should it be viewed as a different language, analogous to the relation of C and C++, or is it right to design it as a compatible component of architected REXX?

Calling non-REXX code - the Generic Binding.

It has always been possible to call non-REXX code from REXX code; the necessary interfaces are defined and publicized. But it is not the easiest thing to do - it requires a knowledge of parameter passing details and requires some low-level programming. The difficulty hampers the development of applications in which REXX is used to harness other facilities. This applies whether the facilities are IBM supplied, like SQL, or developed by a customer.

The design the board favors has three features:

1. A set of conventions about how to pass arguments to packages. For example, if an array is to be passed the elements of the array should be assigned to SomeName.1, SomeName.2, SomeName.3, etc. and the stemmed variable SomeName. passed.
2. A language in which the developer of a package can describe the entry points of the package. This language is essentially declarations in the programming language 'C'.
3. A mechanism in the REXX implementation to convert REXX arguments to non-REXX format and pass them to non-REXX procedures, without the need for anyone to program these conversions.

The user of a package only needs to know the conventions. Such a user will not even be aware of the language in which the package is written. The developer of a package needs to describe the entry points and make use of a utility program to convert the specification of the package into a table to be used when the package is used. Only the developer of a REXX interpreter or compiler needs to know about how arguments are actually converted and passed.

Some key questions are:

- Is the investment in such a general solution justified, or are the packages and system components that will need to be accessed sufficiently few to assume they should hand-craft their own interfaces?

- How far can the infinite variety of non-REXX arguments be accommodated? eg should it be possible to pass a REXX procedure name to non-REXX code and have the non-REXX code subsequently call that REXX procedure?
- If there have to be restrictions on what can be passed, does that make the whole approach unjustified?

Debugging paradigms.

At this point I am going to mention something that is not part of architected REXX, but is a suitable subject for dialog. There is a world of difference between debugging with current REXX trace facilities and the debugging schemes available with some other languages. The latter may have multiple windows showing relevant source, variable values, tracebacks, breakpoints - you know the sort of thing I mean. I have no specific proposal, but we can discuss:

- Should there be an ambition to debug REXX in this way?
- Should it be regarded as something for the system to provide, for all programming languages, or a REXX facility?
- What would the relation to existing TRACE be? A replacement, or in some way an evolution?
- What are the implications for existing proposals to embellish TRACE?

Other items.

Our final area for design dialog covers a selection of smaller items which are not so much driven by changes in the computing scene but are more a matter of filling gaps in the general data processing capabilities of REXX. I think you will recognise them as SHARE requirements although they may not match the exact form of submitted requests.

1. Iterating over associative arrays. Builtin function TAILS returns the number of tails. NEXTTAIL returns a tail, or the successor to a given tail. The sequence produced by NEXTTAIL is guaranteed to include all tails just once, if there is no intervening creation or deletion of tails. An example loop to traverse the tails:

```
if TAILS('Mystem.')>0 then do
 Given = NEXTTAIL('Mystem.');Current=Given
 do until Current=Given /* Process Current */
   Current=NEXTTAIL('Mystem.',Current)
 end
end
```

   (Design of this feature was made more difficult by the fact that there is no string value which cannot be a tail.)

2. String functions more symmetric; negative values for positions are no longer errors; they define the position as counted from the right end of the string. This sets direction, and lengths are counted in that direction.
3. More situations are introduced in which the result of an expression is used as a symbol:

   a) `call (expression)`

Expression evaluates to the symbol called. Note that this is not extended to functions because of the breakage; (abc)(def) is concatenation.

b) `(expression)=rhs`

Neater than using VALUE. Expression evaluates to the symbol that is the target. There is breakage in theory, but who passes 0 and 1 to the host system?

c) `A.(J+1)=99 /* Same as T=J+1;A.T=99 */`

There is breakage, but who uses procedure names that end with a dot? I should point out that this item is not as solidly supported by the board as the rest of architected REXX is. (An arbitarily complex symbol doesn't fit well with the structure of existing interpreters; there is a risk that even those who don't use the feature may suffer a performance penalty from its existence.)

4. DATE() and TIME() builtin functions are extended to have conversion, allowing time arithmetic. Syntax is DATE(outputformat,inputvalue,inputformat). There is no builtin help for 'carry' from time calculation into date calculation.

Some of the questions relevant to these features:

- Does the extra complication outweigh their usefulness?
- Is the breakage tolerable?
- And since this is the last discussion period, it would be an appropriate time for you to voice opinions on the total architected REXX design.

IBM REXX COMPILER

BERT MOSER
IBM

IBM REXX Compiler

Bert Moser

IBM Vienna Software Development Lab
Wien 1, Cobdengasse 2

c/o IBM Austria
Obere Donaustrasse 95
A-1020 Austria
EUROPE

MOSER@VABVM1.IINUS1.IBM.COM

(+431) 21145-4476

May/91

Past

REXX and Interpreters

Present

CMS REXX Compiler COMPLEMENTs SPI

Future

REXX Compiler Improvements and Requirements

'79 Mike Cowlishaw becomes father of REXX

'83 Command Language for IBMs VM/CMS
'87 SAA Procedures Language
'89 REXX supported on MVS
'90 REXX supported on OS/2 and OS/400

2/89 IBM announces the CMS REXX Compiler

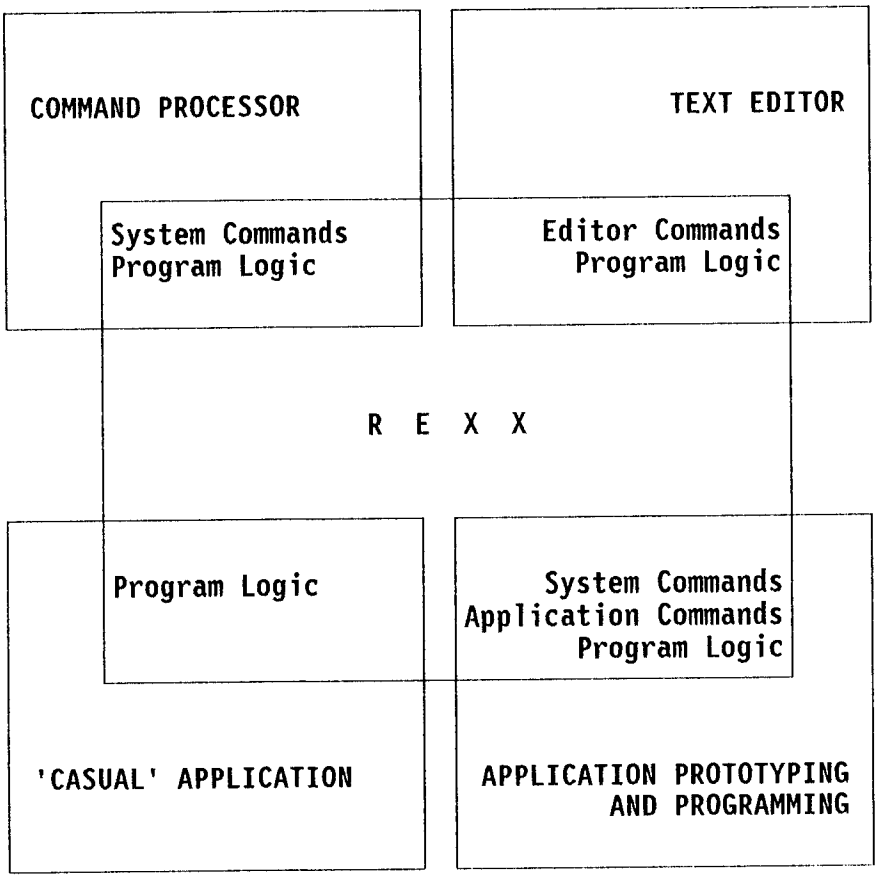
Available since 7/89

Developed by IBM Vienna Software Development Lab
Based on IBM Israel Scientific Center's feasibility study

- Compilability of REXX
- Appropriate run-time performance improvements

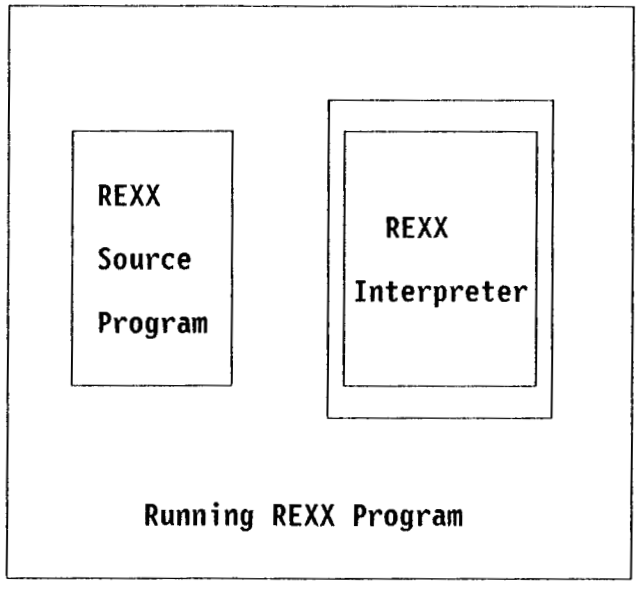
11/89 Library becomes a separate product

REXX Positioning



- REXX initially implemented by Interpreters
  - Excellent debugging features
  - Very short and appealing edit/run cycle
- HOWEVER
  - Better performance desirable

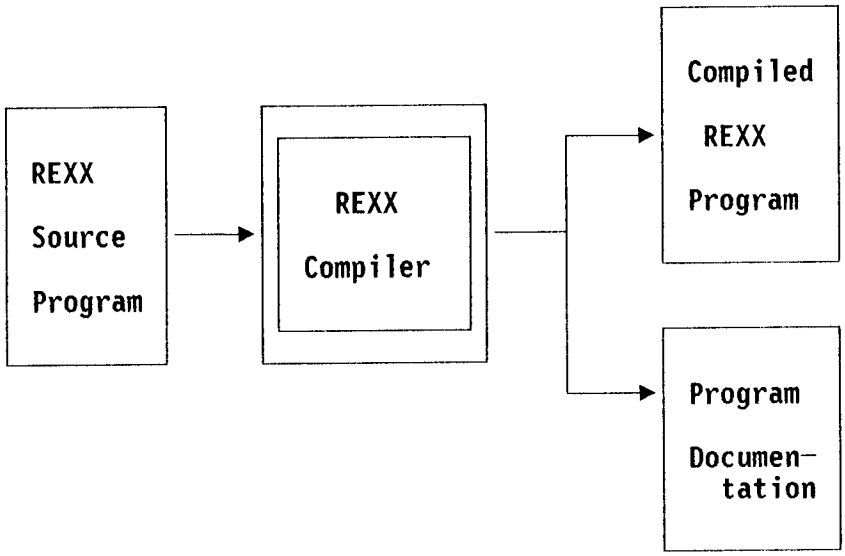
Single Step Approach



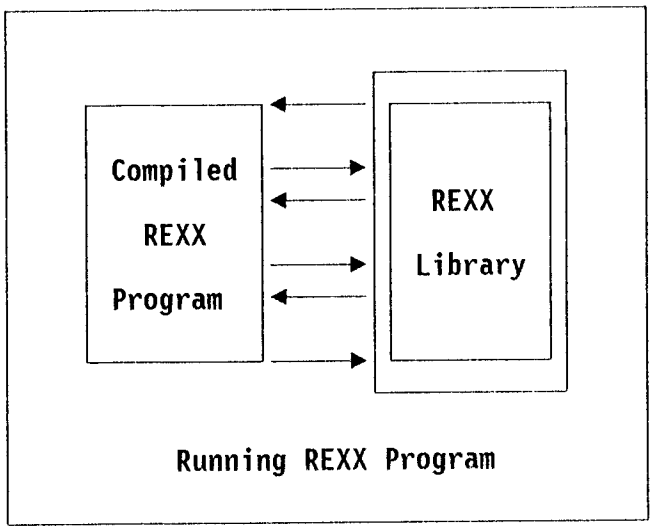
- Everything needs to be done at run-time
- On every REXX program invocation
- REXX source must be made available to every user

Two Step Approach

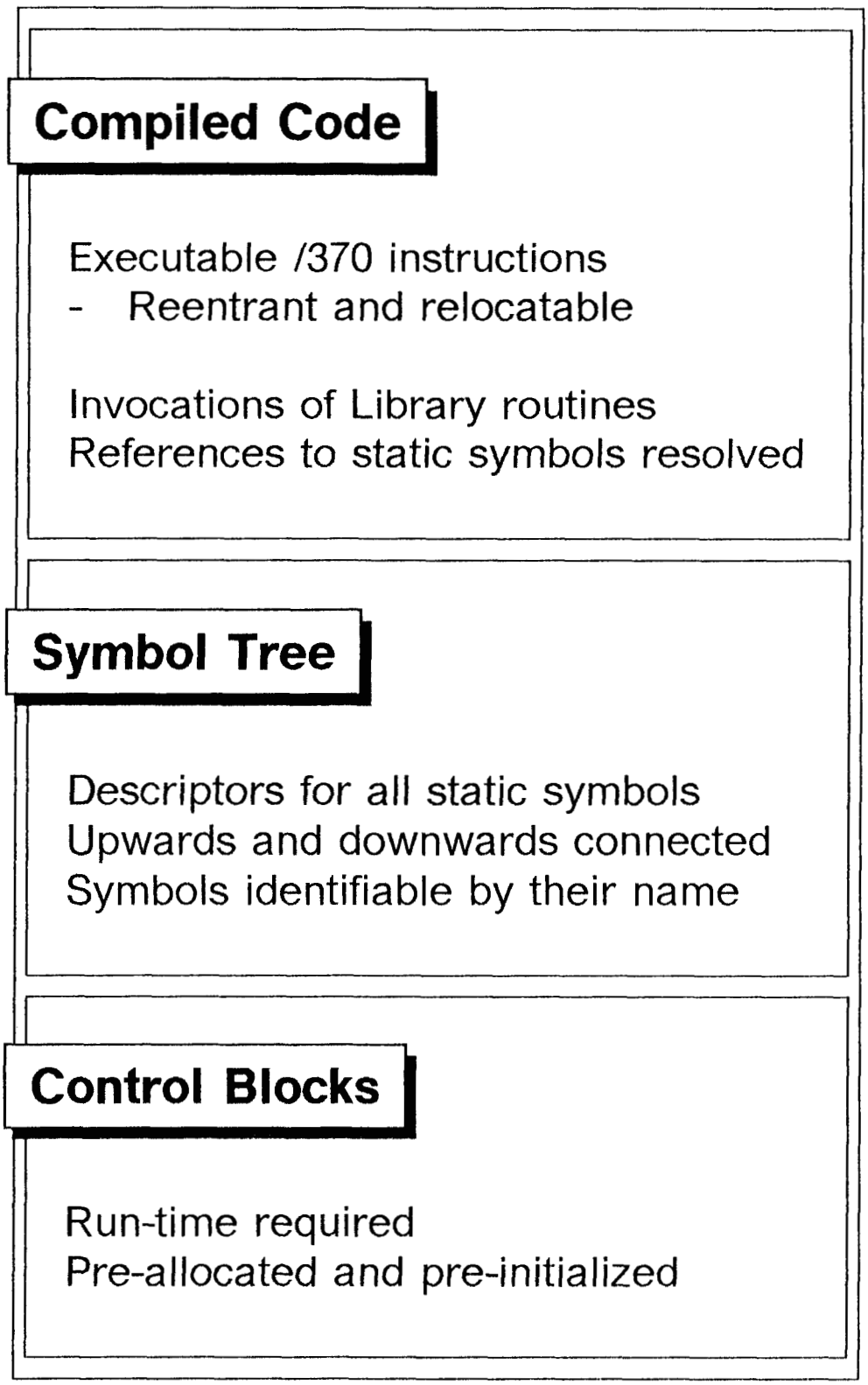
Compile



Run



Compiled REXX Program



EXEC-type

Same behavior as interpreted - "transparently" replace

- Same way of invocation and search sequence
- EXECLOADable
- Shared segment capability

Module-type

Other HLL compilers' object format (ESD, RLD, TXT,..)

- Linkable to other object programs
- Need to be LOADed - can be GEN'd into a module

- Search order is different
- CMS restriction: SVC-type arg/parmlist (PLIST)

Compiler

Set of phases performing all compilation tasks

- Compiled with IBM SAA C/370 Compiler
- Prerequisite when compiling:
  IBM C/370 Library V1 (5668-039) or later

Run-time Library

Routines invoked from compiled REXX programs

- Common to every compiled program

  Initialization, Termination, ...

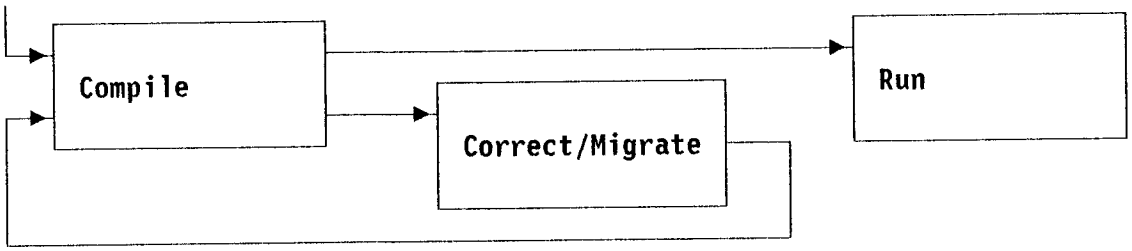
- Too bulky as to be copied to every program

  String Arithmetic, Conversions, Comparisons, Built-in Functions, Compound Variable Access/Handling, ...

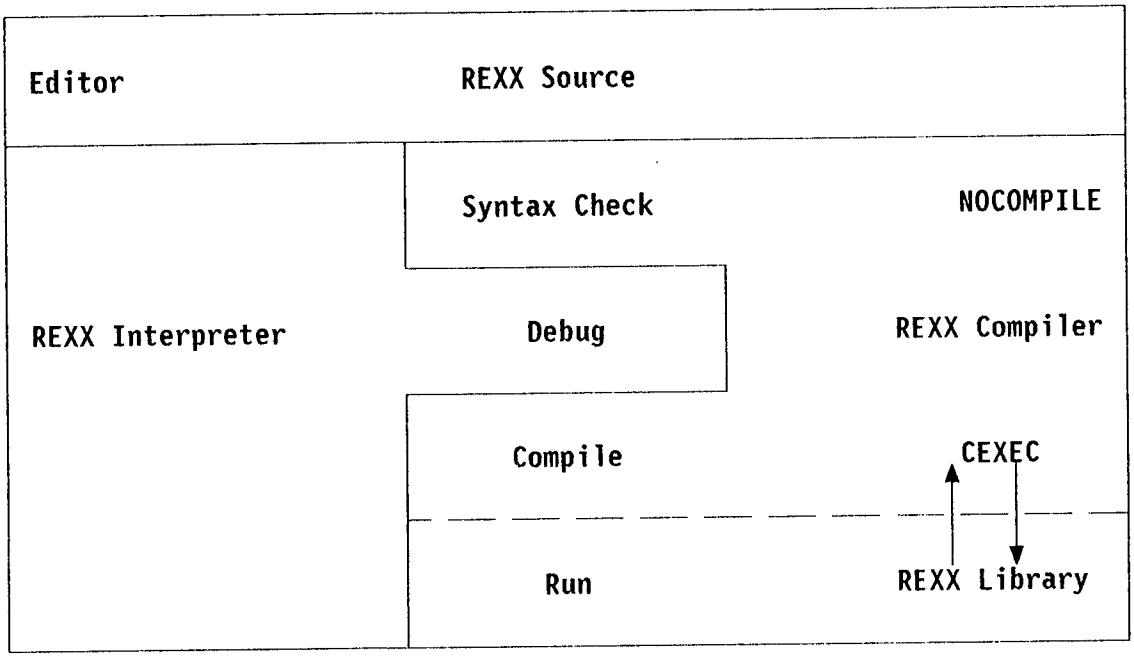
- Extremely time critical --> written in Assembler

Compiler/Interpreter COMPLEMENT Each Other

Existing REXX Programs



Newly Developed REXX Programs



Significantly Faster than Interpreted

"Plug-compatible" with Interpreted

Language Equivalent to Interpreter

"Unreadable" REXX Programs

Comprehensive Program Documentation

Comprehensive REXX Program Documentation

Source and Cross-Reference Listings
Syntax check whole program
More accurate messages
Begin debugging with more-correct programs

- Improve program quality
- Increase developer productivity

"Unreadable" Compiled REXX

Executable /370 code

- Provide program integrity
- Improved maintainability
- Protect REXX-coded assets

Source Listing Example

```
                                        SAMPLE   EXEC     G1

CMS REXX COMPILER 1.1.0      TIME: 11:35:02              DATE: 30 May 1989

 IF  DO SEL  LINE C ----+----1----+----2----+----3----+----4----+----5----

                1   /* SAMPLE incorrect REXX program */
                2
                3   Parse Arg Tmp
                4   val. = TRANSLATE(tmp)
                5   line.2 = LEFT(val.,2,'40')
                                              |
+++EAGGA00771S Invalid or missing argument(s) on built-in function
                6   $ = EXTFUNC(line.2)
                7   Call INTFUNC 2
                8   Exit
                9
               10   INTFUNK: Procedure Expose x. i
               11      Signal on NOVALUE NAME my_value
                                                 |
+++EAGGA00072S Label not found
               12      Do x.i
     1         13         If x.i//2 /= 0 then
  1  1         14            say "Odd: " x.i
     1         15         End
               16      Return
               17
               18   my_valu: Say "NOVALUE raised at: " sigl
               19      Return
               20   /* end of program SAMPLE
                    |
+++EAGGA00654S Unmatched "/*"
```

```
ITEM                            ATTRIBUTE LINE REFERENCES

------ LABELS, BUILT-IN FUNCTIONS, EXTERNAL RTNS ------

EXTFUNC                         EXT RTN   6
INTFUNC                         EXT RTN   7
INTFUNK                         LABEL     10(d)
LEFT                            BUILT-IN  5
MY_VALU                         LABEL     18(d)
MY_VALUE                        LABEL+++  11(u)
TRANSLATE                       BUILT-IN  4

------ CONSTANTS ------

'NOVALUE raised at: '           LIT STR   18
'Odd: '                         LIT STR   14
0                               NUMBER    13
2                               CONST SYM 5 6
2                               NUMBER    5 7 13
'40'                            LIT STR   5

------ SIMPLE VARIABLES ------

$                               SIMP VAR  6(s)
I                               SIMP VAR  10 12 13 14
SIGL                            SIMP VAR  18
TMP                             SIMP VAR  3(s) 4

------ STEMS AND COMPOUND VARIABLES ------

LINE.2                          COMP VAR  5(s) 6
VAL.                            STEM      4(s) 5
X.I                             COMP VAR  10 12 13 14
```

Language Equivalence with REXX Interpreter

NO compiler-specific language features !

- Minimize migration effort
- Almost all REXX programs run unchanged
  - except those with INTERPRET instructions

Flag Non-SAA Items - optional

Support SAA Procedures Language level 1.1

- Ease programming for multiple SAA environments

"Plug-Compatibility" with Interpreted Programs

Identical external interfaces - invocation and use

- "Transparently" replace interpreted
- No restriction on mutual invocation

31-Bit Capability

Compiler, Library, and Compiled Code **run** and **use storage** above the 16 Mega-byte line

- Make room for others below the line

No Conventional Block Structure

PROCEDURE is an executable instruction
- Not a syntactic boundary

Variables' life-time is dynamic
- Depends on calling sequence
- "Exposure" among procedures

No denotation of the END of a procedure
- Logical end is an executed RETURN

SIGNAL

Control can be transferred to everywhere
- Even into "procedure" and loop bodies

Computed GOTO - SIGNAL VALUE

No data types

All data is "character string"

Sometimes contents must be "numeric",
- Whole number", or "Boolean"

No declarations

Variables come and go - EXECCOMM/DROP

Can be shared with external programs

Names of variables can be computed
- Tails of compound variables

Value of variables only limited by storage
- Storage for values must be allocated dynamically

Arithmetic precision can be set dynamically
- NUMERIC DIGITS

Run-Time Performance Improvements

Performance gains depend on program mix

Programs with a lot of ...	TIMES faster than Interpreter	Performance Category
Default-precision Arithmetic	6 - 10+	VERY HIGH
String Arithmetic	4 - 25	
Assignments	6 - 10	
Changes to Variables' Values	4 - 6	HIGH
Constants and Simple Variables	4 - 6	
Reuse of Compound Variables	2 - 4	MEDIUM
Host Commands	1 -	LOW

- Up to 30% CPU load reduction reported - "... better than last CPU upgrade"
- On average 10% - 15%
- Savings example

```
Interpreted program runs     60     times a day (12 sec's)
                  using      12     min's CPU
   assume improvement of      6     times

            runs compiled      2     min's
```

INTERPRET Instruction not Supported

Rarely used

- Compiler diagnoses - no code generated

- Try to avoid

  Interpret target'='expr

  Call SETVAR target,expr

  RXSETVAR sample Assembler program
  User's Guide & Reference SH19-8120

- Restructure the program

  Isolate interpretative part
  Make it a separate program, and
  Let the Interpreter handle it

TRACE Instruction and Function not Supported

Does not change the semantics of a REXX program

- No need to change REXX program

  TRACE instruction - NOP instruction
  TRACE built-in function - "O"
  Interpreter default - "N"

- Diagnosed with an informational message

Save CPU Time & Reduce System Load

Improve Program Quality

Increase Developer Productivity

Protect REXX-Coded Assets

Allow to Keep Applications in REXX

Save Expensive Rewrites to Other HLL's

Attract to Write Even More REXX Applications

Reduce Storage Needed at Compile-Time

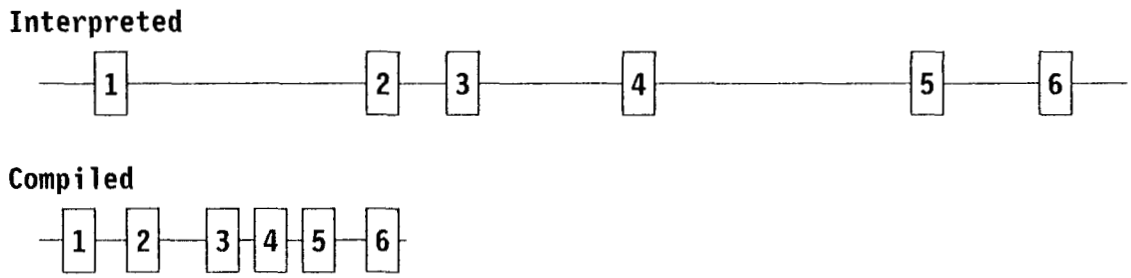
Improve Compiler's Performance

Improve Access/Handling of Compound Vars

- Binary tails
- Faster algorithms

Improve Built-in Functions

Compiled REXX Increases Paging



Running compiled on an I/O bound systems makes it WORSE

Compiled REXX Scatters Storage

Sorry for this one - was a bug
4 Bytes of a control block left over - sometimes

Compiled REXX Needs More Storage when Run

NO
Both implementations show similar storage consumption

Reduce Disk Space Needed by Compiled REXX

Remember: Code + Symbol Tree + Control Blocks

- Improve Assignment

  Special casing by Compiler = lot of code

  - Trade-off between performance and storage
  - Move case distinction to Library

- Compress Compiled Output

  Compiler option

  - Reduce med/large to size of source
  - Automatically de-compress
  - Reduce expensive I/O

Static Binding

- Allow to link external subroutines and functions

  For Module-type output only

  Compiler option
  Dualism - resolved address/dynamic invocation

  Tie together REXX-written application
  Function-package capability

Tailorable Cross Reference Listing

- Make Xref of CONSTANTs and LITERALs optional
- Compiler option XREF(S)

Library as "Test" Shared Segment

- Test new Library in parallel with "production" version

Relax Restriction on INTERPRET

- SEVERE ERROR - NO code generated

- Diagnose as ERROR - produce code

  Code should raise ERROR when executed
  - Allow to program around
  - Parse Version & REXXC370

  - Support development of multi-environmental programs

Implement INTERPRET

Long Range Consideration

Provide an MVS REXX Compiler

Accepted

MVS Compiler - Requirements

- Same language level as TSO/E Interpreter

- Same external interfaces - invocation and run

- Similar behavior and benefits as CMS REXX Compiler

- Cross compile ?

  - REXX code could run unchanged in VM and MVS
  - No need to re-compile

- Support MVS Parameter List Conventions

  - EFPL for external functions
  - CPPL for TSO/E commands
  - MVS JCL parameters for batch programs
  - CALL command parameters for foreground programs

PRACTICAL APPLICATION OF REXX IN THE UNIX ENVIRONMENT

ED SPIRE
THE WORKSTATION GROUP

PRACTICAL APPLICATION OF REXX IN THE UNIX ENVIRONMENT

1. COMMERCIAL USERS MIGRATING TO UNIX.

2. UNIX HAS A LARGE LEARNING CURVE.

3. REXX CAN EASE THE TRANSITION BY PROVIDING A FAMILIAR FACILITY.

4. REXX BRINGS A NEW LEVEL OF FUNCTIONALITY TO UNIX.

Types of REXX applications in Unix:

1. Unix command macros
2. Macros for other utilities which support REXX directly
3. General purpose programming in REXX
4. Embedded REXX applications

UNIX COMMAND MACROS:

1. RECORD RESEARCHED TECHNIQUES FOR FUTURE USE
2. SIMPLIFY UNIX COMMAND SYNTAX
3. AUTOMATE REPEATED USAGE OF RELATED UNIX COMMAND SEQUENCES
4. PROVIDE ACCESS TO FEATURES THAT ARE OTHERWISE DIFFICULT TO USE
5. EXTEND THE OPERATING SYSTEM'S FACILITIES

UNIX COMMAND MACROS: RECORD RESEARCHED TECHNIQUES FOR FUTURE USE

INSTEAD OF	CAT <FILE> | RSH SCOTTY LPR
ALLOW	RLP <FILE>

```
#!/usr/local/bin/rxx
/*
 * rlp - print on a printer on another machine
 *
 * rlp filename machine traceopt
 *
 *      filename is the name of the file to be printed.
 *      machine is the machine that has the desired printer (defaults
 *              to scotty)
 *      traceopt is a rexx trace option, defaults to no tracing.
 */
parse arg fn machine traceopt
trace value traceopt
if machine="" then machine="scotty"
"cat" fn "| rsh" machine "lpr"
```

UNIX COMMAND MACROS: SIMPLIFY UNIX COMMAND SYNTAX

INSTEAD OF FIND /USR -NAME <THINGY> -PRINT

ALLOW FI <THINGY>

```
#!/usr/local/bin/rxx
/*
 * fi - run find on just /usr, where everything is anyway.
 *
 * this helps you not run find on the root, which would go out and
 * look through all your nfs mounts.  It also helps you not have to
 * remember the find command's syntax...
 */
parse arg name
"find /usr -name" name "-print"
```

UNIX COMMAND MACROS: AUTOMATE REPEATED SEQUENCES

INSTEAD OF	PS -U <USERID> (VISUALLY LOOK FOR A LINE REFERRING TO <PGM> AND REMEMBER ITS <PROCESS ID>) DBX -A <PROCESS-ID>
ALLOW	DBXW <PGM>

```
#!/usr/local/bin/rxx
/*
 * dbxw - run dbx on the program running in another window.
 * This is useful when the program in the other window is a curses
 * application and the dbx output would mess up its "screen" display.
 *
 * dbx programname
 *
 * will run a ps -u userid and look for a process running programname,
 * and then dbx -a processid.
 *
 * Note that you should probably cd to the directory where the program
 * resides before you dbxw.
 */
parse arg programname traceopt
trace value traceopt
call popen "ps -u" userid() "| grep" programname
select
  when queued()=0
   then say "can't find" programname
  when queued()=1
   then do
        parse pull processid .
        "dbx -a" processid
        end
  when queued()>1
   then say "more than one" programname "running!"
  end
```

UNIX COMMAND MACROS: ACCESS TO OTHERWISE HARD TO USE FEATURES

INSTEAD OF	?????? (TO SET <TITLE> AS THE TITLE OF AN X WINDOW AND ITS ICON)
ALLOW	SETNAMES <TITLE>

```
#!/usr/local/bin/rxx
/*
 * setnames - change the name associated with an X window.
 *
 *    The name of the window or its icon can be changed by sending
 *    a specific escape sequence to the terminal window...
 */
escape = x2c("1b")
parse arg name                                    /* charout avoids the */
call charout , escape||"]l"||name||escape  /* unwanted 'cr' that */
call charout , escape||"]L"||name||escape  /* lineout would send */
```

Typical usage of setnames:

```
#!/usr/local/bin/rxx
/*
 * rl - rlogin to another system, changing the names in the cterm window
 *      and icon to reflect that system's name
 */
parse arg system                                        /* who to rlogin to */
"setnames" system                                        /* put his name up */
"rlogin" system                                            /* rlogin to him */
call popen "hostname"                                        /* who are we? */
parse pull hostname
"setnames" hostname                          /* restore this system's name */
```

UNIX COMMAND MACROS: SIMPLIFY UNIX COMMAND SYNTAX (LARGE SCALE)

REXX UTILITY TO PARSE LARGE NUMBERS OF OPERANDS

(CALLING SEQUENCE SHOWN)

```
/******************** PARSE SEQUENCE PATTERN ****************************/
/* MODIFY THE THIRD LINE AS YOUR "PROTOTYPE" SHOWING PARMS AND DFLTS  */
/******************** START OF PARSE SEQUENCE ****************************/
parse arg a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18
interpret cparse(,
"p1 p2(*) ( nk1 nk2 k1(k1v) k2() abc",
||" )" a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18)
/******************** END OF PARSE SEQUENCE ****************************/
```

UNIX COMMAND MACROS: SIMPLIFY UNIX COMMAND SYNTAX (LARGE SCALE)

SOME UNIX COMMANDS (ESPECIALLY THOSE ASSOCIATED WITH THE X WINDOWS SYSTEM) CAN HAVE LOTS OF OPERANDS...

```
xterm [ -ah] [ -ar] [ -b NumberPixels] [ -bd Color] [ -bg Color]
      [ -bw NumberPixels] [ -ccCharRange:Value[,...]]
      [ -cr Color] [ -cu] [ -display Name:Number] [ -dw]
      [ -fb Font] [ -fg Color] [ -fn Font] [ -fr Font]
      [ -fullcursor] [ -geometry Geometry] [ #Geometry] [ -help]
      [ -i] [ -ib File] [ -j] [ -keywords] [ -lang Language] [ -l]
      [ -leftscroll] [ -lf File] [ -ls] [ -mb] [ -mc Number]
      [ -ms Color] [ -n IconName] [ -name Application]
      [ -nb Number] [ -po Number] [ -ps] [ -reduced] [ -rv]
      [ -rw] [ -s] [ -sb] [ -sf] [ -si] [ -sk]
      [ -sl NumberLines] [ -sn] [ -st] [ -suppress] [ -T Title]
      [ -ti] [ -tm String] [ -tn TerminalName] [ -ut]
      [ -v] [ -vb] [ -W] [ -xrm String] [ -132] [ -e Command]
```

Examples

The following example can be used to create an xterm, specifying the size and location of the window, using a font other than the default, and also specifying the foreground color to be used of the text. It then runs a command in that window.

```
xterm -geometry 20x10+0+175 -fn Bld14.500 -fg DarkTurquiose
                            -e /tmp/banner_cmd &
```

UNIX COMMAND MACROS: SIMPLIFY UNIX COMMAND SYNTAX (LARGE SCALE)

SIMPLIFIED XTERM, WITH NEW DEFAULTS AND EASY SPECIFICATION OPERANDS

```
#!/usr/local/bin/rxx
/*
 * xt - start an xterm window
 *
 * The positional parms comprise a Unix command that is to be run in this
 * window.  If none is specified, then your normal shell is run instead.
 *
 * optional parms:
 *   l #  - number of lines (default 25)
 *   x #  - x component of window location
 *   y #  - y component of window location
 *   i    - if present, window starts as an icon.
 *   fr # - reduced screen font size (default 14, the typical default
 *          for normal aixterm windows.)
 *   fn # - normal screen font size (default 10, small than typical for
 *          aixterm windows).
 *             The above two default settings make for normally small
 *             windows, which can be temporarily enlarged back to their
 *             traditional size by selecting "reduced" from the alt-
 *             left button menu.
 *   s    - if present, xterm is run synchronously.
 *   test - if present, the options line is shown on the screen and
 *          aixterm invocation is supressed.
 */
/******************** PARSE SEQUENCE PATTERN ****************************/
/* MODIFY THE THIRD LINE AS YOUR "PROTOTYPE" SHOWING PARMS AND DFLTS  */
/******************** START OF PARSE SEQUENCE ***************************/
parse arg a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18
interpret cparse(,
"c1() c2() c3() c4() c5() c6() ( l(25) x() y() i fr(14) fn(10) test s",
||" )" a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18)
/******************** END OF PARSE SEQUENCE *****************************/
cmd=c1 c2 c3 c4 c5 c6                          /* build the desired command */
if cmd="" then command = ""                 /* build the required xterm option */
         else command = "-e" cmd
if s="" then amp="&"  /* build the required background execution option */
        else amp=""
options="-fullcursor -sb -sl 999 -ar -ls"  /* initial xterm options set */
options=options "-geometry 80x"l                  /* number of lines option*/
if x<>"" | y<>"" then do                         /* if position specified, */
     if x="" then x=0                        /* fill in remaining defaults */
     if y="" then y=0
     options=options||"+"||x||"+"||y                  /* position option */
     end
options=options "-fn Rom"||fn||".500"                 /* normal font option*/
options=options "-fb Rom"||fn||".500"                   /* bold font option*/
options=options "-fr Rom"||fr||".500"               /* reduced font option */
if i="i" then options=options "-i"      /* if req, then start as an icon */
call chdir("/u/ets")                             /* back to home directory */
if test=""                                          /* show it or do it */
then "/usr/lpp/X11/bin/aixterm" options command amp
else say options command amp
```

UNIX COMMAND MACROS: SIMPLIFY UNIX COMMAND SYNTAX (LARGE SCALE)

EXAMPLE USAGE OF XT:

```
#!/usr/local/bin/rxx
/*
 * xi - initialize xterm environment
 *
 * this just creates my standard set of windows, with their normal
 * positions, but leaves them all as icons at first.
 */
"xt            ] x 0   y 0   i l 64"              /* large primary window */
"xt            ] x 680 y 0   i"                   /* 1st alternate window */
"xt            ] x 680 y 395 i"                   /* 2nd alternate window */
"xt rl wrkgrp  ] x 0   y 390 i"     /* window on wrkgrp (Sun-3) */
"xt rl drwho   ] x 575 y 630 i"     /* window on drwho (Sparc) */
"xt rl scotty  ] x 600 y 590 i"     /* window on scotty (SCO/Unix) */
"xt rl orac    ] x 0   y 545 i"     /* window on orac (HP-9000/300 HP-UX */
"xt rl worf    ] x 0   y 545 i"     /* window on orac (HP-9000/300 Domain-OS*
/*
 * orac and worf are in the same spot, since they are the same machine,
 * and only one will be up at a time.  The other will die quitely
 * after a few attempts to rlogin
 */
```

UNIX COMMAND MACROS: EXTEND UNIX FACILITIES

INSTEAD OF UNDERSTANDING YOUR LOCAL NFS NETWORK AND HOW TO TRANSLATE LOCAL FILENAMES ON ONE SYSTEM TO THE CORRESPONDING LOCAL FILENAME ON YOUR SYSTEM...

ALLOW FILENAME SYNTAX OF

NODE:/LOCAL/FILE/NAME

THROUGHOUT A SET OF UNIFORM UTILITIES.

Filesystem	Total KB	free	%used	iused	%iused	Mounted on
/dev/hd4	49152	11356	76%	1146	9%	/
/dev/hd2	225280	37716	83%	6367	11%	/usr
/dev/hd3	32768	31700	3%	26	0%	/tmp
/dev/hd1	249856	223444	10%	1292	2%	/u
wrkgrp:/home	47946	5832	87%	-	-	/sun
wrkgrp:/usr	213313	18127	91%	-	-	/sunusr
wrkgrp:/	7608	2635	65%	-	-	/sunroot
drwho:/home	326519	66998	79%	-	-	/drwho
scotty:/	99037	4694	95%	-	-	/odt
drwho:/usr	183439	47344	74%	-	-	/whousr

UNIX COMMAND MACROS: EXTEND UNIX FACILITIES

INSTEAD OF UNDERSTANDING YOUR LOCAL NFS NETWORK AND HOW TO TRANSLATE LOCAL FILENAMES ON ONE SYSTEM TO THE CORRESPONDING LOCAL FILENAME ON YOUR SYSTEM...

ALLOW FILENAME SYNTAX OF

NODE:/LOCAL/FILE/NAME

THROUGHOUT A SET OF UNIFORM UTILITIES.

```
/*
 * fn filename
 *
 * fn accepts a filename in a system independent form, and generates
 * a local filename which will provide (probably NFS) access to the
 * desired file.
 *
 * filename has the form
 *
 *   host:/filename/on/that.host
 *
 * note:  if host: is omitted, then no translation is done, assuming
 * that a local filename was really specified in the first place
 */
parse arg host ':' file
call popen 'hostname'
parse pull currenthost
select
when file=""                /* if no "host:", parse will have put it all */
then o=host                                  /* in host, and file will be null. */
when host=currenthost /* if explicitly referring to a file on this host */
then o=file                                          /* just use that file name */
otherwise do
 call popen 'df'
 lm=""                    /* will become the saved df line that matches. */
 do while queued()>0
  parse pull l
  parse var l dfhost ':' dfhostdir dfjunk '/' dflocaldir
  if length(dfhostdir)>0 ,                    /* weeds out header and locals */
      & dfhost==host                               /* weeds out other systems */
  then do
   if dfhostdir=="/"                                    /* for root file system */
   then lm=l                    /* save this line in case we find no other */
   else if left(file,length(dfhostdir))==dfhostdir       /* right line? */
   then do
    lm=l
    leave
    end
   end
  end
 if length(lm)>0                                       /* if we found something, */
 then do
  parse var lm dfhost ':' dfhostdir dfjunk '/' dflocaldir
  if dfhostdir^=='/'                                   /* if not root filesystem, */
  then file=right(file,length(file)-length(dfhostdir))  /* trim rmt dir */
  o='/'||dflocaldir||file                              /* add correct local dir */
  end
 else do                  /* if we found nothing, fail with error message */
      say 'sorry, no path from here to' host':'file
      return                          /* return with no value is a failure */
      end
 end
end
return o                                                /* non-failure return */
```

UNIX COMMAND MACROS: EXTEND UNIX FACILITIES

INSTEAD OF UNDERSTANDING YOUR LOCAL NFS NETWORK AND HOW TO TRANSLATE LOCAL FILENAMES ON ONE SYSTEM TO THE CORRESPONDING LOCAL FILENAME ON YOUR SYSTEM...

ALLOW FILENAME SYNTAX OF

NODE:/LOCAL/FILE/NAME

THROUGHOUT A SET OF UNIFORM UTILITIES.

Typical usages of fn allow the user to access files on other systems without knowing the details of the NFS links that connect these systems.

```
#!/usr/local/bin/rxx
/*
 * nfl - invoke flist with system independent filename
 */
parse arg dir
ldir=fn(dir)
say 'flist' ldir
'flist' ldir
```

```
#!/usr/local/bin/rxx
/*
 * nxe - invoke xedit with a system independent filename
 */
'xe' fn(arg(1))
```

Macros for other utilities that support REXX

1. Record researched techniques for future use
2. Extend the features of that utility
3. Integrate the utility with other Unix operations

MACROS FOR OTHER UTILITIES: RECORD RESEARCHED TECHNIQUES

TO PRINT PART OF THE CURRENT XEDIT FILE,

INSTEAD OF	!RM TEMP PUT <TARGET> TEMP !RLP TEMP
ALLOW	RLP <TARGET>

```
/*
 * rlp.xedit - an xedit macro to print part of an xedit file
 *
 * rlp target
 *
 *   is the same as
 *
 *        !rm temp
 *        put target temp
 *        !rlp temp
 *
 */
parse arg target
address unix  'rm temp'
address xedit 'put' target 'temp'
address unix  'rlp temp'
```

MACROS FOR OTHER UTILITIES: EXTEND FEATURES

PROVIDE PARAGRAPH REFORM CAPABILITIES IN XEDIT

```
/* FLOW MACRO.
This macro aligns two or more lines of a text-type file
being edited (such as a NOTE). It tries to place as many
words as possible on a line, within the right margin
defined by XEDIT SET TRUNC.

USE:

    FLOW <target>

where <target> is a standard Xedit target defining the first line
not to be flowed. Typically, the alignment process will result in
there being fewer lines in the block than there were before alignment.
This will not always be true.

UNIQUE CAPABILITY OF THIS PROGRAM:

This macro can, unlike other parts of XEDIT, shorten lines. If you
SET TRUNC to a value shorter than some of the lines in your file,
they will be handled correctly by this macro. Elsewhere in XEDIT,
results are unpredictable and will likely involve data loss.

MODIFICATION HISTORY:

11/16/86 - Roger Deschner - Original version
01/02/88 - Roger Deschner - Replace call to "JOIN", for performance
02/14/88 - Roger Deschner - Allow lines to be shortened; use PUTD
10/24/89 - Roger Deschner - Protect from LINEND character
10/04/90 - Roger Deschner - Changed to FLOW; moved to RS/6000
*/

/* Do it to it */

doit:
PARSE ARG targ
```

```
doit:
PARSE ARG targ
tempfile = 'JJ.TEMP'
ADDRESS UNIX 'rm -f' tempfile
'PUTD' targ tempfile
'UP 1'
'EXTRACT /TRUNC'

rotbuf = ''  /* initialize rotating buffer */
DO FOREVER
  IF (LINES(tempfile) = 0) THEN LEAVE    /* EOF? */
  ibuf = LINEIN(tempfile)
  IF (SUBSTR(ibuf,1,1) = ' ') THEN DO /* Paragraph break, either kind */
    IF (rotbuf ^= ' ') THEN DO    /* Anything left in buffer? */
      'INPUT' rotbuf    /* put it out */
      rotbuf = ''
    END
  END
  IF (ibuf = ' ') THEN 'INPUT   '    /* Blank line */
  ELSE DO                          /* duit tuit */
    /* concatenate the new stuff */
    IF (rotbuf = '') THEN rotbuf = STRIP(ibuf,'T')
                     ELSE rotbuf = rotbuf STRIP(ibuf,'T')
    SELECT
      WHEN (LENGTH(rotbuf) = trunc.1) THEN DO /* perfect fit */
        'INPUT' rotbuf
        rotbuf = ''
      END
      WHEN (LENGTH(rotbuf) > trunc.1) THEN DO /* more than enough */
        DO FOREVER
          /* Find last blank, starting at TRUNC.1+1, working backwards */
          i = trunc.1+1
          DO WHILE (SUBSTR(rotbuf,i,1) ^= ' ')
            i = i - 1
            IF (i = 0) THEN SIGNAL word_too_long /* word > trunc */
          END
          'INPUT' SUBSTR(rotbuf,1,i-1)
          rotbuf = STRIP(SUBSTR(rotbuf,i),'B')
          IF (LENGTH(rotbuf) < trunc.1) THEN LEAVE /* Split enough? */
        END
      END
      OTHERWISE NOP /* not long enough - read another line */
    END       /* end of select */
  END
END
IF (rotbuf ^= ' ') THEN DO        /* Anything left in buffer? */
  'INPUT' rotbuf          /* put it out */
  rotbuf = ''
END
/* Clean up our toys and go home */
ADDRESS UNIX 'rm -f' tempfile
RETURN

/* Error routines */

word_too_long:
'INPUT ************************************************************'
'INPUT *ERROR* Justify encountered word longer than TRUNC setting.'
'INPUT Split word manually and restart justification from there.'
'INPUT Delete these error message lines.'
'INPUT ************************************************************'
CALL EXIT 13

EXIT:
PARSE ARG orc .
EXIT orc
```

MACROS FOR OTHER UTILITIES: INTEGRATE WITH UNIX OPERATIONS

PROVIDE BACKGROUND COMPILATION INITIATED FROM THE EDITOR, WITH THE RESULTING COMPILER ERROR MESSAGES DISPLAYED IN A POP-UP X WINDOW

```
/*
 * mk.xedit
 *
 * Runs make out of an xedit session.
 *
 * Default name is taken from source filename assumed to be of
 * the form "name.something".  So if you are editing key.c, this
 * routine will kick off "make key".  You can also issue "mk else"
 * if you want to make another target.
 *
 * The real work is done in a background task, and its output is
 * presented in a separate window.
 */
parse arg name
if name='' then do                  /* if name not specified, generate the default */
     "extract /fname"
     name=left(fname.1,pos(".",fname.1)-1)
     end
"save"                              /* make sure the disk file is up to date */
address unix "xemake" name "&"      /* kick off the background task */


#!/usr/local/bin/rxx
/*
 * xemake - run a make and display the results in a window.  Normally
 *          invoked from with xedit via mk.xedit.
 */
parse arg name                                  /* get name of make target */
"make" name ">"||name||".makeout 2>&1"          /* run make, output to a file */
"xt xe" getcwd()"/"||name||".makeout"           /* display the results */
```

General purpose programming in REXX

1. Reusable filters written in REXX vs. in-line AWK or SED programming

2. Small applications can be crafted by pulling together existing system facilities, integrated through REXX programming.

3. No high productivity language normally available in Unix. Alternatives are usually C and FORTRAN.

4. REXX applications can be ported to Unix from other platforms.

```
#!/usr/local/bin/rxx
/*
 * both - find lines containing both strings within a specific number
 *        of words.
 */
parse arg first second distance      /* two strings and a min. distance */
do while lines()>0
   line=linein()
   fpos=wordpos(first,line)             /* position of first word or 0 */
   spos=wordpos(second,line)           /* position of second word or 0 */
   if fpos>0 & spos>0 & abs(spos-fpos)<=distance
   then call lineout(,line)                      /* write matching lines */
   end
call lineout()                                    /* close output file */
exit


#!/usr/local/bin/rxx
/*
 * mult - find lines containing all input strings
 */
parse arg strings                          /* all words to be searched for */
do x=1 while lines()>0           /* x= only for leave instruction below */
   line=linein()                    /* line is a candidate to be tested */
   do i=1 to words(strings)                 /* try all words in string. */
      if wordpos(word(strings,i),line)=0          /* 0 means not found */
      then leave x                             /* terminates outer loop */
      end                                        /* end of all tests */
   call lineout(,line)                  /* if all found, write it out. */
   end
call lineout()                                    /* close output file */
exit
```

GENERAL PURPOSE PROGRAMMING: INTEGRATION OF EXISTING FACILITIES

THIS SAMPLE IMPLEMENTS A "PHONE DIRECTORY" BY USING XEDIT, DRIVEN BY A REXX PROGRAM. "PH <NAME>" POPS UP AN X WINDOW SHOWING AN EDIT SESSION THAT HAS BEEN PRE-POSITIONED ON THE FIRST LINE IN THE DATASET THAT CONTAINS <NAME>.

```
#!/usr/local/bin/rxx
parse arg name                          /* get the name he wants to find */
"rxx ph2" name "&"                     /* pass it along to the background */
```

```
#!/usr/local/bin/rxx
parse arg string                        /* get the name he wants to find */
"cp $HOME/.profile.xedit ph.xedit"          /* copy his .profile.xedit */
call lineout 'ph.xedit', '"cl/'string'"'  /* add a search command to it */
call lineout 'ph.xedit'                                /* close new profile */
"xt xe -p ph $HOME/phone/dir ] s"          /* xe phone/dir in a window */
"rm ph.xedit"            /* cleanup after synchronous window terminates */
```

General purpose programming: High productivity language

No high productivity alternative is usually present in Unix.
Alternatives are usually limited to C and FORTRAN.

```
#!/usr/local/bin/rxx
/*
 * vptrim - a utility to trim ventura publisher markup from a word
 *          processing file.
 *
 * vptrim infile traceopt
 *
 *        infile required, specifies the input file containing a
 *               word processing file that contains ventura publisher
 *               markup string.
 *
 *        traceopt optional, a trace instruction operand to turn on
 *                 REXX tracing.
 *
 * The output is sent to STDOUT, and may be redirected to a file.
 *
 * Example:  vptrim xehelp > xehelp2
 *
 * Most "@... = " and <...> sequences are simply removed from the
 * file.
 *
 * <T> is changed into three blanks.
 *
 * @FUNCTION = text is appended to the start of the next line, with
 * a " - " placed between the two chunks of text.
 *
 * << and >> are translated to < and > respectively.
 *
 * Room for improvement:
 *
 *   We could define our own set of tab stops and try to handle (T)
 *   in some smarter way.
 *
 *   @FUNCTION trick should maybe be extended to handle multiple such,
 *   through a table of special functions.
 *
 */
```

NO HIGH PRODUCTIVITY ALTERNATIVE IS USUALLY PRESENT IN UNIX.
ALTERNATIVES ARE USUALLY LIMITED TO C AND FORTRAN.

```
parse arg fn traceopt
trace value traceopt

if fn=""
then do
     say "usage: vptrim fn traceopt"
     exit
     end

lag=""

do while lines(fn)>0        /* push the entire file through this loop */

   line = linein(fn)

   do while pos("<T>",line)>0    /* turn <T> into white space */
      line=overlay("   ",line,pos("<T>",line))
      end

   do while pos("<<",line)>0     /* turn << into x'01' to hide them */
      line=left(line,pos("<<",line)-1)||'01'x||,
             right(line,length(line)-pos("<<",line)-1)
      end

   do while pos(">>",line)>0     /* turn >> into x'02' to hide them */
      line=left(line,pos(">>",line)-1)||'02'x||,
             right(line,length(line)-pos(">>",line)-1)
      end

   do while pos("<",line)>0      /* take out all other <..anything..> */
      if pos(">",line)>0
      then line=left(line,pos("<",line)-1)||,
             right(line,length(line)-pos(">",line))
      else do
           say '********* VPTRIM ERROR: Unmatched "<" in the following line.'
           leave
           end
      end

   line=translate(line,"<>","0102"x) /* unhide translated << and >> */

   if left(line,1)="@"               /* check for paragraph tag */
   then do
        type=left(line,10)           /* remember tag type */
        line = right(line,length(line)-pos("=",line)-1)  /* remove it */
        if type="@FUNCTION "
        then do                      /* for @FUNCTION tag */
             lag = line "-"          /* save text for next line */
             iterate                 /* and skip putting it out now */
             end
        end

   say lag line    /* put out current line plus any lag data */
   lag=""
   end
```

General purpose programming: Portability

REXX on Unix allows for porting applications developed on other platforms. Specific areas of concern:

- OS commands
- I/O facilities

For large programs, these areas can easily be a minor part of the code.

```
doit:
PARSE ARG targ
tempfile = 'JJ.TEMP'
ADDRESS UNIX 'rm -f' tempfile
'PUTD' targ tempfile
'UP 1'
'EXTRACT /TRUNC'

rotbuf = ''  /* initialize rotating buffer */
DO FOREVER
  IF (LINES(tempfile) = 0) THEN LEAVE    /* EOF? */
  ibuf = LINEIN(tempfile)
  IF (SUBSTR(ibuf,1,1) = ' ') THEN DO /* Paragraph break, either kind */
    IF (rotbuf ^= ' ') THEN DO    /* Anything left in buffer? */
      'INPUT' rotbuf    /* put it out */
      rotbuf = ''
    END
  END
  IF (ibuf = ' ') THEN 'INPUT   '    /* Blank line */
  ELSE DO                          /* duit tuit */
    /* concatenate the new stuff */
    IF (rotbuf = '') THEN rotbuf = STRIP(ibuf,'T')
                     ELSE rotbuf = rotbuf STRIP(ibuf,'T')
    SELECT
      WHEN (LENGTH(rotbuf) = trunc.1) THEN DO /* perfect fit */
        'INPUT' rotbuf
        rotbuf = ''
      END
      WHEN (LENGTH(rotbuf) > trunc.1) THEN DO /* more than enough */
        DO FOREVER
          /* Find last blank, starting at TRUNC.1+1, working backwards */
          i = trunc.1+1
          DO WHILE (SUBSTR(rotbuf,i,1) ^= ' ')
            i = i - 1
            IF (i = 0) THEN SIGNAL word_too_long /* word > trunc */
          END
          'INPUT' SUBSTR(rotbuf,1,i-1)
          rotbuf = STRIP(SUBSTR(rotbuf,i),'B')
          IF (LENGTH(rotbuf) < trunc.1) THEN LEAVE /* Split enough? */
        END
      END
      OTHERWISE NOP /* not long enough - read another line */
    END       /* end of select */
  END
END
IF (rotbuf ^= ' ') THEN DO        /* Anything left in buffer? */
  'INPUT' rotbuf          /* put it out */
  rotbuf = ''
END
/* Clean up our toys and go home */
ADDRESS UNIX 'rm -f' tempfile
RETURN

/* Error routines */

word_too_long:
'INPUT ************************************************************'
'INPUT *ERROR* Justify encountered word longer than TRUNC setting.'
'INPUT Split word manually and restart justification from there.'
'INPUT Delete these error message lines.'
'INPUT ************************************************************'
CALL EXIT 13

EXIT:
PARSE ARG orc .
EXIT orc
```

Embedded applications

1. Business applications
2. Utility software

Embedded applications

...require a robust API.

uni-REXX's API was modeled after that used by TSO/E REXX.

1. REXX program invocation from a C-based application
2. Ability to create addressable environments (i.e., SUBCOM)
3. Variable pool interface
4. C-based external functions (yet to be delivered)

Practical Application of REXX in the Unix Environment

- Supports migration of existing staff to Unix
- Brings new levels of integration and ease of use to Unix

Practical Application of REXX in the Unix Environment

2nd Annual SLAC REXX Symposium
Asilomar Conference Center
Pacific Grove, California

May 9, 1991

Presented by:
Ed Spire
The Workstation Group
Rosemont, Illinois

As commercial users migrate from proprietary IBM mainframes to Unix, they are often bringing REXX with them. REXX not only aids in the migration process, but also brings new functionality to the experienced Unix user. In many cases, you can use a single REXX program or macro where a native Unix solution would require a combination of tools (one of several shells, awk, grep, sed, etc.) each of which has its own syntax and ideosyncracies.

This presentation will discuss the applicability of REXX to various tasks in the Unix environment, and show examples where appropriate.

General types of applications:

1. Unix command macros

2. Macros for other utilities that use REXX, perhaps in combination with Unix facilities (primarily XEDIT)

3. General purpose programming in REXX

   custom filters

   entire applications

4. Embedded REXX applications

UNIX COMMAND MACROS are written to provide shorthand notations for often used, hard to remember, or lengthy sequences.

For example, sending something to a printer may require some local rain dance. Once you figure out what that particular rain dance is, you could "can" that research in a simple REXX program

```
#!/usr/local/bin/rxx
/*
 * rlp - print on a printer on another machine
 *
 * rlp filename machine traceopt
 *
 *      filename is the name of the file to be printed.
 *      machine is the machine that has the desired printer (defaults
 *              to scotty)
 *      traceopt is a rexx trace option, defaults to no tracing.
 */
parse arg fn machine traceopt
trace value traceopt
if machine="" then machine="scotty"
"cat" fn "| rsh" machine "lpr"
```

The command syntax for some Unix commands can be less than obvious. Once again, when you figure out a command syntax, you might want to "cover" the command with a REXX program that has a syntax that you find more intuitive.

Further, sometimes you find Unix commands that can be "misused". "find /" (i.e., find starting from the root directory) is often a really bad idea, since it will be looking through many relatively slow (i.e., remotely mounted NFS) file systems.

```
#!/usr/local/bin/rxx
/*
 * fi - run find on just /usr, where everything is anyway.
 *
 * this helps you not run find on the root, which would go out and
 * look through all your nfs mounts.  It also helps you not have to
 * remember the find command's syntax...
 */
parse arg name
"find /usr -name" name "-print"
```

The above are perhaps trivial to the experienced Unix user, but provide a handy shortcut for new Unix users. Once a method is researched, it is "canned" for future use. This also helps avoid one's shooting yourself in the foot as in find (root)...

Users who are migrating to Unix from proprietary IBM mainframe platforms are normally conversant enough with REXX to use it in the above manner as part of their migration efforts. Or, a thorough effort on the part of a support group could easily provide a series of such utilities that could ease a migration.

Some often used sequences are a bit more complex. In this case REXX is useful even for the experienced Unix user.

For example, when debugging an application which is using a terminal window in a fullscreen mode (perhaps via the curses terminal I/O package), it is nice to run the trace/debug package (dbx) from another window, so the debugging information won't interfere with the "fullscreen" I/O from the application. Normally, to do this, you display the list of active processes, scan it to find the process which represents the fullscreen application, and then invoke DBX supplying it with that process ID number. A fairly simple REXX program can automate that task for you.

```
#!/usr/local/bin/rxx
/*
 * dbxw - run dbx on the program running in another window.
 * This is useful when the program in the other window is a curses
 * application and the dbx output would mess up its "screen" display.
 *
 * dbx programname
 *
 * will run a ps -u userid and look for a process running programname,
 * and then dbx -a processid.
 *
 * Note that you should probably cd to the directory where the program
 * resides before you dbxw.
 */
parse arg programname traceopt
trace value traceopt
call popen "ps -u" userid() "| grep" programname
select
  when queued()=0
   then say "can't find" programname
  when queued()=1
   then do
        parse pull processid .
        "dbx -a" processid
        end
  when queued()>1
   then say "more than one" programname "running!"
  end
```

Some facilities are present but not easily used without special support. REXX is a convenient way to provide that support.

For example, the X windows system lets you control various aspects of the system by sending special character sequences to the terminal. It's pretty unlikely that someone is going to remember the special sequence that changes a window's title, for instance. But a REXX program could make it very easy to use this feature.

```
#!/usr/local/bin/rxx
/*
 * setnames - change the name associated with an X window.
 *
 *   The name of the window or its icon can be changed by sending
 *   a specific escape sequence to the terminal window...
 */
escape = x2c("1b")
parse arg name                                    /* charout avoids the */
call charout , escape||"]l"||name||escape  /* unwanted 'cr' that */
call charout , escape||"]L"||name||escape  /* lineout would send */
```

Typical usage of setnames:

```
#!/usr/local/bin/rxx
/*
 * rl - rlogin to another system, changing the names in the cterm window
 *      and icon to reflect that system's name
 */
parse arg system                                        /* who to rlogin to */
"setnames" system                                        /* put his name up */
"rlogin" system                                            /* rlogin to him */
call popen "hostname"                                        /* who are we? */
parse pull hostname
"setnames" hostname                          /* restore this system's name */
```

After a while you will have assembled a series of "local tools", building one upon the other, which can vastly improve the usability of Unix; and without climbing the rather steep learning curve associated with the various Unix utilities and filters which you would have to put together to accomplish all this without REXX.

Sometimes you might want to cover a Unix command that has many operands, with a syntax that makes the most often used operands more accessible. Now we're starting to write REXX programs with more than just one or two operands.

Before we proceed, let me mention how we write REXX macros that accept many operands, so that the user can invoke them with the same flexibility usually found in native commands.

Complex REXX programs at our installation use CPARSE to bring in operands...

```
/******************* PARSE SEQUENCE PATTERN ***************************/
/* MODIFY THE THIRD LINE AS YOUR "PROTOTYPE" SHOWING PARMS AND DFLTS  */
/******************* START OF PARSE SEQUENCE ***************************/
parse arg a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18
interpret cparse(,
"p1 p2(*) ( nk1 nk2 k1(k1v) k2() ",
||" )" a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18)
/******************* END OF PARSE SEQUENCE ***************************/
```

Cparse is an external REXX subroutine that accepts a string describing the model syntax of the REXX main program's parameter list, followed by the parameters that were actually passed to the main program. Cparse returns a string of assignment statements which, when interpreted by the main program, will place the appropriate symbols in the main program's symbol table to reflect the arguements that were passed to the main program (as parsed against the model syntax.)

Note that CPARSE was ported directly from CMS with no changes (other than case considerations), and hence implements a CMS-style command operand syntax. It could easily be modified to support a Unix-style syntax.

CPARSE makes it easy to write REXX programs that accept several operands, providing the flexibility found in native commands. This allows a REXX program to easily "cover" a basic Unix facility, reorganizing the parameter structure to the user's liking.

XTERM is one such command, with many operands, most of which you want to have standard values for, and a few which you might want to be able to change easily. XTERM has **lots** of operands...

```
xterm [ -ah] [ -ar] [ -b NumberPixels] [ -bd Color] [ -bg Color]
      [ -bw NumberPixels] [ -ccCharRange:Value[,...]]
      [ -cr Color] [ -cu] [ -display Name:Number] [ -dw]
      [ -fb Font] [ -fg Color] [ -fn Font] [ -fr Font]
      [ -fullcursor] [ -geometry Geometry] [ #Geometry] [ -help]
      [ -i] [ -ib File] [ -j] [ -keywords] [ -lang Language] [ -l]
      [ -leftscroll] [ -lf File] [ -ls] [ -mb] [ -mc Number]
      [ -ms Color] [ -n IconName] [ -name Application]
      [ -nb Number] [ -po Number] [ -ps] [ -reduced] [ -rv]
      [ -rw] [ -s] [ -sb] [ -sf] [ -si] [ -sk]
      [ -sl NumberLines] [ -sn] [ -st] [ -suppress] [ -T Title]
      [ -ti] [ -tm String] [ -tn TerminalName] [ -ut]
      [ -v] [ -vb] [ -W] [ -xrm String] [ -132] [ -e Command]
```

Examples

The following example can be used to create an xterm, specifying the size and location of the window, using a font other than the default, and also specifying the foreground color to be used of the text. It then runs a command in that window.

```
xterm -geometry 20x10+0+175 -fn Bld14.500 -fg DarkTurquiose
                            -e /tmp/banner_cmd &
```

Now if you want to establish local defaults for some of these and make the few operands you would normally use more accessible, you can cover XTERM with a REXX program like this...

```
#!/usr/local/bin/rxx
/*
 * xt - start an xterm window
 *
 * The positional parms comprise a Unix command that is to be run in this
 * window.  If none is specified, then your normal shell is run instead.
 *
 * optional parms:
 *   l #  - number of lines (default 25)
 *   x #  - x component of window location
 *   y #  - y component of window location
 *   i    - if present, window starts as an icon.
 *   fr # - reduced screen font size (default 14, the typical default
 *          for normal aixterm windows.)
 *   fn # - normal screen font size (default 10, small than typical for
 *          aixterm windows).
 *             The above two default settings make for normally small
 *             windows, which can be temporarily enlarged back to their
 *             traditional size by selecting "reduced" from the alt-
 *             left button menu.
 *   s    - if present, xterm is run synchronously.
 *   test - if present, the options line is shown on the screen and
 *          aixterm invocation is supressed.
 */
/******************** PARSE SEQUENCE PATTERN ****************************/
/* MODIFY THE THIRD LINE AS YOUR "PROTOTYPE" SHOWING PARMS AND DFLTS  */
/******************** START OF PARSE SEQUENCE ***************************/
parse arg a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18
interpret cparse(,
"c1() c2() c3() c4() c5() c6() ( l(25) x() y() i fr(14) fn(10) test s",
||" )" a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18)
/******************** END OF PARSE SEQUENCE *****************************/
cmd=c1 c2 c3 c4 c5 c6                          /* build the desired command */
if cmd="" then command = ""           /* build the required xterm option */
          else command = "-e" cmd
if s="" then amp="&"  /* build the required background execution option */
        else amp=""
options="-fullcursor -sb -sl 999 -ar -ls"  /* initial xterm options set */
options=options "-geometry 80x"l                /* number of lines option*/
if x<>"" | y<>"" then do                       /* if position specified, */
     if x="" then x=0                      /* fill in remaining defaults */
     if y="" then y=0
     options=options||"+"||x||"+"||y                 /* position option */
     end
options=options "-fn Rom"||fn||".500"              /* normal font option*/
options=options "-fb Rom"||fn||".500"                /* bold font option*/
options=options "-fr Rom"||fr||".500"             /* reduced font option */
if i="i" then options=options "-i"     /* if req, then start as an icon */
call chdir("/u/ets")                          /* back to home directory */
if test=""                                          /* show it or do it */
then "/usr/lpp/X11/bin/aixterm" options command amp
else say options command amp
```

Typical use of xt, others further below.

```
#!/usr/local/bin/rxx
/*
 * xi - initialize xterm environment
 *
 * this just creates my standard set of windows, with their normal
 * positions, but leaves them all as icons at first.
 */
"xt           ] x 0   y 0   i 1 64"              /* large primary window */
"xt           ] x 680 y 0   i"                   /* 1st alternate window */
"xt           ] x 680 y 395 i"                   /* 2nd alternate window */
"xt rl wrkgrp ] x 0   y 390 i"     /* window on wrkgrp (Sun-3) */
"xt rl drwho  ] x 575 y 630 i"     /* window on drwho (Sparc) */
"xt rl scotty ] x 600 y 590 i"     /* window on scotty (SCO/Unix) */
"xt rl orac   ] x 0   y 545 i"     /* window on orac (HP-9000/300 HP-UX */
"xt rl worf   ] x 0   y 545 i"     /* window on orac (HP-9000/300 Domain-OS*
/*
 * orac and worf are in the same spot, since they are the same machine,
 * and only one will be up at a time.  The other will die quitely
 * after a few attempts to rlogin
 */
```

Sometimes, you might want to extend the operating system's facilities. Here's an example associated with Network File System (NFS) usage. NFS can make remote machine file systems appear as local systems on your machine. You can see the relationships between local aliases for remote file systems in the df command output:

Filesystem	Total KB	free	%used	iused	%iused	Mounted on
/dev/hd4	49152	11356	76%	1146	9%	/
/dev/hd2	225280	37716	83%	6367	11%	/usr
/dev/hd3	32768	31700	3%	26	0%	/tmp
/dev/hd1	249856	223444	10%	1292	2%	/u
wrkgrp:/home	47946	5832	87%	-	-	/sun
wrkgrp:/usr	213313	18127	91%	-	-	/sunusr
wrkgrp:/	7608	2635	65%	-	-	/sunroot
drwho:/home	326519	66998	79%	-	-	/drwho
scotty:/	99037	4694	95%	-	-	/odt
drwho:/usr	183439	47344	74%	-	-	/whousr

So, in this example, if you are told that the file you need is available on machine drwho as /usr/local/bin, you would need to access the file as /whousr/local/bin. The aliases in use might vary from machine to machine.

We use the concept of a "system independent" filename, and have a routine that will compare the filename you specify to a 'df' command output, returning the local filename that will access the desired file from this machine. We then cover typical utility programs with a REXX invocation program that translates the filename through this routine, so that the user could refer to the above file as drwho:/usr/local/bin from any platform on the network.

```
/*
 * fn filename
 *
 * fn accepts a filename in a system independent form, and generates
 * a local filename which will provide (probably NFS) access to the
 * desired file.
 *
 * filename has the form
 *
 *   host:/filename/on/that.host
 *
 * note:  if host: is omitted, then no translation is done, assuming
 * that a local filename was really specified in the first place
 */
parse arg host ':' file
call popen 'hostname'
parse pull currenthost
select
when file=""                /* if no "host:", parse will have put it all */
then o=host                                 /* in host, and file will be null. */
when host=currenthost /* if explicitly referring to a file on this host */
then o=file                                          /* just use that file name */
otherwise do
 call popen 'df'
 lm=""                    /* will become the saved df line that matches. */
 do while queued()>0
  parse pull l
  parse var l dfhost ':' dfhostdir dfjunk '/' dflocaldir
  if length(dfhostdir)>0 ,                    /* weeds out header and locals */
      & dfhost==host                               /* weeds out other systems */
  then do
   if dfhostdir=="/"                                    /* for root file system */
   then lm=l                  /* save this line in case we find no other */
   else if left(file,length(dfhostdir))==dfhostdir       /* right line? */
   then do
    lm=l
    leave
    end
   end
  end
 if length(lm)>0                                       /* if we found something, */
 then do
  parse var lm dfhost ':' dfhostdir dfjunk '/' dflocaldir
  if dfhostdir^=='/'                                   /* if not root filesystem, */
  then file=right(file,length(file)-length(dfhostdir))  /* trim rmt dir */
  o='/'||dflocaldir||file                               /* add correct local dir */
  end
 else do                  /* if we found nothing, fail with error message */
      say 'sorry, no path from here to' host':'file
      return                          /* return with no value is a failure */
      end
 end
end
return o                                                /* non-failure return */
```

Typical usages of fn allow the user to access files on other systems without knowing the details of the NFS links that connect these systems.

```
#!/usr/local/bin/rxx
/*
 * nfl - invoke flist with system independent filename
 */
parse arg dir
ldir=fn(dir)
say 'flist' ldir
'flist' ldir
```

```
#!/usr/local/bin/rxx
/*
 * nxe - invoke xedit with a system independent filename
 */
'xe' fn(arg(1))
```

MACROS FOR OTHER UTILITIES THAT USE REXX can serve similar purposes.

We will use XEDIT as an example of a utility that uses REXX as its macro processor.

Once again, when you have figured out how to do something like print part of the edit file, you can "can" that procedure in a trivial REXX program.

```
/*
 * rlp.xedit - an xedit macro to print part of an xedit file
 *
 * rlp target
 *
 *   is the same as
 *
 *       !rm temp
 *       put target temp
 *       !rlp temp
 *
 */
parse arg target
address unix  'rm temp'
address xedit 'put' target 'temp'
address unix  'rlp temp'
```

A utility like XEDIT can also have significant functional extensions added to it via REXX programming. A typical requirement for a text editor is the ability to reform a modified paragraph to more nicely fit within it's margins. Here's a REXX macro that adds this function to XEDIT. Note that it was ported from CMS with minimal changes (to exactly three out of 66 lines of code.)

```
/* FLOW MACRO.
This macro aligns two or more lines of a text-type file
being edited (such as a NOTE). It tries to place as many
words as possible on a line, within the right margin
defined by XEDIT SET TRUNC.

USE:

    FLOW <target>

where <target> is a standard Xedit target defining the first line
not to be flowed. Typically, the alignment process will result in
there being fewer lines in the block than there were before alignment.
This will not always be true.

UNIQUE CAPABILITY OF THIS PROGRAM:

This macro can, unlike other parts of XEDIT, shorten lines. If you
SET TRUNC to a value shorter than some of the lines in your file,
they will be handled correctly by this macro. Elsewhere in XEDIT,
results are unpredictable and will likely involve data loss.

MODIFICATION HISTORY:

11/16/86 - Roger Deschner - Original version
01/02/88 - Roger Deschner - Replace call to "JOIN", for performance
02/14/88 - Roger Deschner - Allow lines to be shortened; use PUTD
10/24/89 - Roger Deschner - Protect from LINEND character
10/04/90 - Roger Deschner - Changed to FLOW; moved to RS/6000
*/

/* Do it to it */

doit:
PARSE ARG targ
tempfile = 'JJ.TEMP'
ADDRESS UNIX 'rm -f' tempfile
'PUTD' targ tempfile
'UP 1'
'EXTRACT /TRUNC'

rotbuf = ''  /* initialize rotating buffer */
DO FOREVER
  IF (LINES(tempfile) = 0) THEN LEAVE    /* EOF? */
  ibuf = LINEIN(tempfile)
  IF (SUBSTR(ibuf,1,1) = ' ') THEN DO /* Paragraph break, either kind */
    IF (rotbuf ^= ' ') THEN DO    /* Anything left in buffer? */
      'INPUT' rotbuf    /* put it out */
      rotbuf = ''
    END
  END
  IF (ibuf = ' ') THEN 'INPUT   '    /* Blank line */
  ELSE DO                          /* duit tuit */
    /* concatenate the new stuff */
    IF (rotbuf = '') THEN rotbuf = STRIP(ibuf,'T')
                     ELSE rotbuf = rotbuf STRIP(ibuf,'T')
    SELECT
```

```
      WHEN (LENGTH(rotbuf) = trunc.1) THEN DO /* perfect fit */
        'INPUT' rotbuf
        rotbuf = ''
      END
      WHEN (LENGTH(rotbuf) > trunc.1) THEN DO /* more than enough */
        DO FOREVER
          /* Find last blank, starting at TRUNC.1+1, working backwards */
          i = trunc.1+1
          DO WHILE (SUBSTR(rotbuf,i,1) ^= ' ')
            i = i - 1
            IF (i = 0) THEN SIGNAL word_too_long /* word > trunc */
          END
          'INPUT' SUBSTR(rotbuf,1,i-1)
          rotbuf = STRIP(SUBSTR(rotbuf,i),'B')
          IF (LENGTH(rotbuf) < trunc.1) THEN LEAVE /* Split enough? */
        END
      END
      OTHERWISE NOP /* not long enough - read another line */
    END       /* end of select */
  END
END
IF (rotbuf ^= ' ') THEN DO         /* Anything left in buffer? */
  'INPUT' rotbuf           /* put it out */
  rotbuf = ''
END
/* Clean up our toys and go home */
ADDRESS UNIX 'rm -f' tempfile
RETURN

/* Error routines */

word_too_long:
'INPUT ************************************************************'
'INPUT *ERROR* Justify encountered word longer than TRUNC setting.'
'INPUT Split word manually and restart justification from there.'
'INPUT Delete these error message lines.'
'INPUT ************************************************************'
CALL EXIT 13

EXIT:
PARSE ARG orc .
EXIT orc
```

The use of extensive REXX macros in a setting such as XEDIT becomes more viable on fast RISC processors, where the relatively low speed of its interpretive execution is not a problem. We expect to see REXX macros adapt XEDIT to many widely varied tasks (such as true word processing with automatic paragraph reform, LEXX-style live parsing, etc.) Such applications would be far too slow on the previous generation of computing platforms.

When using a utility that supports REXX, you also have the opportunity to integrate the work being done through the utility with work to be done in Unix itself.

When using XEDIT to write programs, the following REXX macro set will make it very easy to "kick off" a compilation, and put that compilation in the background, so that you can continue editing while the compiler checks your syntax. Once the compilation is completed, an X window is presented showing the output from the compiler.

```
/*
 * mk.xedit
 *
 * Runs make out of an xedit session.
 *
 * Default name is taken from source filename assumed to be of
 * the form "name.something". So if you are editing key.c, this
 * routine will kick off "make key". You can also issue "mk else"
 * if you want to make another target.
 *
 * The real work is done in a background task, and its output is
 * presented in a separate window.
 */
parse arg name
if name='' /* if name not specified, generate the default */
then do
     "extract /fname"
     name=left(fname.1,pos(".",fname.1)-1)
     end
"save" /* make sure the disk file is up to date */
address unix "xemake" name "&" /* kick off the background task */
```

```
#!/usr/local/bin/rxx
/*
 * xemake - run a make and display the results in a window. Normally
 *          invoked from with xedit via mk.xedit.
 */
parse arg name /* get name of make target */
"make" name ">"||name||".makeout 2>&1" /* run make, output to a file */
"xt xe" getcwd()"/"||name||".makeout" /* display the results */
```

GENERAL PURPOSE PROGRAMMING IN REXX fits quite nicely with the Unix philosophy of small, reusable pieces of code. Custom filters are easily written in REXX. These programs read STDIN and write part of the output file to STDOUT.

The Unix style of programming often uses data sources and filters in a multi-tasking "pipe" to achieve a particular result. Often custom filters are required, and in many cases they are created "on the spot" with utilities like sed, awk or perl. REXX can be used to write general purpose filters which can be documented and retained for future use.

Here are two examples, which filter a file, passing only lines that contain specific string combinations.

```
#!/usr/local/bin/rxx
/*
 * both - find lines containing both strings within a specific number
 *        of words.
 */
parse arg first second distance      /* two strings and a min. distance */
do while lines()>0
   line=linein()
   fpos=wordpos(first,line)             /* position of first word or 0 */
   spos=wordpos(second,line)           /* position of second word or 0 */
   if fpos>0 & spos>0 & abs(spos-fpos)<=distance
   then call lineout(,line)                        /* write matching lines */
   end
call lineout()                                      /* close output file */
exit


#!/usr/local/bin/rxx
/*
 * mult - find lines containing all input strings
 */
parse arg strings                          /* all words to be searched for */
do x=1 while lines()>0           /* x= only for leave instruction below */
   line=linein()                    /* line is a candidate to be tested */
   do i=1 to words(strings)                    /* try all words in string. */
      if wordpos(word(strings,i),line)=0              /* 0 means not found */
      then leave x                              /* terminates outer loop */
      end                                            /* end of all tests */
   call lineout(,line)                    /* if all found, write it out. */
   end
call lineout()                                      /* close output file */
exit
```

Often, small applications can be crafted by pulling together pieces of REXX code which bring existing system facilities together.

Here's a sample which implements a "phone directory" using XEDIT, driven by a REXX program. "ph name" pops up an X window showing an edit session that has been pre-positioned on the first line in the dataset that contains "name".

```
#!/usr/local/bin/rxx
parse arg name                             /* get the name he wants to find */
"rxx ph2" name "&"                        /* pass it along to the background */


#!/usr/local/bin/rxx
parse arg string                          /* get the name he wants to find */
"cp $HOME/.profile.xedit ph.xedit"              /* copy his .profile.xedit */
call lineout 'ph.xedit', '"cl/'string'"'  /* add a search command to it */
call lineout 'ph.xedit'                                /* close new profile */
"xt xe -p ph $HOME/phone/dir ] s"            /* xe phone/dir in a window */
"rm ph.xedit"            /* cleanup after synchronous window terminates */
```

And, as always, low volume applications can be developed and maintained much more cost effectively with REXX than with many other languages. This is especially true in the Unix world where the only universally available language is C, and the most likely alternate is FORTRAN, and BASIC is not usually present.

```
#!/usr/local/bin/rxx
/*
 * vptrim - a utility to trim ventura publisher markup from a word
 *          processing file.
 *
 * vptrim infile traceopt
 *
 *        infile required, specifies the input file containing a
 *               word processing file that contains ventura publisher
 *               markup string.
 *
 *        traceopt optional, a trace instruction operand to turn on
 *                 REXX tracing.
 *
 * The output is sent to STDOUT, and may be redirected to a file.
 *
 * Example:  vptrim xehelp > xehelp2
 *
 * Most "@... = " and <...> sequences are simply removed from the
 * file.
 *
 * <T> is changed into three blanks.
 *
 * @FUNCTION = text is appended to the start of the next line, with
 * a " - " placed between the two chunks of text.
 *
 * << and >> are translated to < and > respectively.
 *
 * Room for improvement:
 *
 *   We could define our own set of tab stops and try to handle (T)
 *   in some smarter way.
 *
 *   @FUNCTION trick should maybe be extended to handle multiple such,
 *   through a table of special functions.
 *
 */

parse arg fn traceopt
trace value traceopt

if fn=""
then do
     say "usage: vptrim fn traceopt"
     exit
     end

lag=""

do while lines(fn)>0       /* push the entire file through this loop */

   line = linein(fn)

   do while pos("<T>",line)>0   /* turn <T> into white space */
      line=overlay("   ",line,pos("<T>",line))
      end

   do while pos("<<",line)>0    /* turn << into x'01' to hide them */
```

```
   line=left(line,pos("<<",line)-1)||'01'x||,
          right(line,length(line)-pos("<<",line)-1)
   end

do while pos(">>",line)>0     /* turn >> into x'02' to hide them */
   line=left(line,pos(">>",line)-1)||'02'x||,
          right(line,length(line)-pos(">>",line)-1)
   end

do while pos("<",line)>0      /* take out all other <..anything..> */
   if pos(">",line)>0
   then line=left(line,pos("<",line)-1)||,
          right(line,length(line)-pos(">",line))
   else do
        say '********* VPTRIM ERROR: Unmatched "<" in the following line.
        leave
        end
   end

line=translate(line,"<>","0102"x) /* unhide translated << and >> */

if left(line,1)="@"               /* check for paragraph tag */
then do
     type=left(line,10)           /* remember tag type */
     line = right(line,length(line)-pos("=",line)-1)  /* remove it */
     if type="@FUNCTION "
     then do                      /* for @FUNCTION tag */
          lag = line "-"          /* save text for next line */
          iterate                 /* and skip putting it out now */
          end
     end

say lag line    /* put out current line plus any lag data */
lag=""
end
```

REXX Applications that were developed on the mainframe can be ported to Unix without a complete rewrite in another language. You still have to pay close attention to the system interfaces, such as OS commands and I/O facilities.

The largest such application we have seen ported to Unix is a "silicon compiler" used to design microchips - a **very** large REXX program that was ported with minimal difficulties to Unix, and would have required a major rewrite in C or Fortran had REXX not been available.

APPLICATIONS THAT EMBED REXX are being ported to (or developed for) Unix. We have seen a refinery simulation system that embedded REXX as the simulator's control language ported to Unix with minimal effort. More than one vendor is working on automated operations and/or network control systems which embed REXX as the controlling language. REXX has been specified as the controlling language for other commercial Unix applications as well.

These applications require a robust API, similar to that found in VM/CMS or MVS/TSO. The uni-REXX API includes REXX program invocation, the ability to create an addressable environment (i.e., SUBCOM), and the Variable Pool Interface. Still to come are external functions written in C.

SUMMARY:

The availability of REXX in the Unix environment not only supports the migration of existing staff from proprietary mainframes to Unix, but also brings a new level of integration and ease of use to the Unix environment.

REXX LANGUAGE PARSING CAPABILITIES

KEITH WATTS
KILOWATT SOFTWARE

REXX language parsing capabilities

Keith Watts (Kilowatt Software)

Abstract

The REXX language has several powerful features which distinguish it from other programming languages that are generally available. Among these are the language's intricate collection of parsing capabilities. These enable the programmer to easily divide character strings by a diversity of methods. Herein, the syntax and semantics of these methods are described in detail. This paper is intended to help programmer's of varying proficiency gain a commanding grasp of these concepts. Many examples are also provided.

One of the more powerful features of REXX is its ability to parse. However, if you are like many others who are learning REXX you are unfamiliar with the word "parse". Webster's New World Dictionary contains the definition:

> **parse** *vt*, *vi* **parsed, pars'ing** [Now Rare]
>
> 1.to separate (a sentence) into its parts, explaining the grammatical form, function, and interrelation of each part 2. to describe the form, part of speech , and function of (a word in a sentence)

The above definition has little in common with the REXX parsing capability. The key phrase is: "to separate into its parts". For the word "parse" is computer science parlance for the act of separating computer input into meaningful parts for subsequent processing actions.

REXX is one of few languages which provides parsing as a fundamental statement. Most languages merely provide lower level string separation capabilities, leaving the preparation of parsing capabilities as user developed enhancements. Within REXX, these capabilities are immediately available, and as you will soon find, very powerful.

Let us learn about parsing by analyzing the following:

```
phrase = "  I think, therefore   I am [I  think].  "
```
[1]

If you scrutinize the above, you will notice that there are extra blanks at various points within the phrase. These extra blanks and the punctuation characters within the phrase complicates the parsing process.

The words within the phrase could be traditionally extracted as follows:

```
 /* try1 [the brute force approach] */
/* trace ?i */                  /* turn on the trace to see how this code works */
remaining_words = phrase
do i=1 by 1 while remaining_words <> ""
  remaining_words = strip( remaining_words, "Leading" )      /* skip lead blanks */
  end_pos = pos( " ", remaining_words )            /* locate blank after current word*/
  word.i = substr( remaining_words, 1, end_pos )                /* extract a word */
  remaining_words = substr( remaining_words, end_pos )        /* step over word */
 end
```

[1]This adaptation of Descartes famous insight is from "On the Threshold of a Dream", by the Moody Blues.

Alternatively, REXX contains built-in functions which are excellent for extracting words from phrases, as follows:

```
 /* try2 */
/* trace ?i */                  /* turn on the trace to see how this code works */
do i=1 for words( phrase )
  word.i = word( phrase, i )
 end
```

Finally, an approach which uses REXX parsing is:

```
 /* try3 */
/* trace ?i */                  /* turn on the trace to see how this code works */
rest = phrase
do i=1 while rest <> ""
  parse var rest word.i rest
 end
```

Of the 3 approaches shown above, the second is clearly the best choice for separating a string into words. However, the second approach is specifically capable of accessing words, it is inadequate for other parsing tasks. The third approach is slightly more intricate than the second, and is occasionally preferrable. All that can be said about the first approach is that it successfully obtains individual words, and the method used is familiar to those who have programmed with other languages; though the subroutine names are probably different.

Let REXX know what you mean

Notice that words within the phrase above are generically captured by relative position into the set of **word.i** symbols. Now you will see how phrases can be parsed into symbols which are syntactically significant.

For example, we can divide our phrase into meaningful constituents as follows:

```
parse var phrase precondition ',' consequence '[' qualifier ']'
```

This results in the following symbol assignments:

```
precondition          "I think"
consequence           "therefore  I am "
qualifier             "I  think"
```

[Please observe that there are extra spaces within the `consequence` and `qualifier` symbols].

Notice how easy it was to divide our phrase with the parse statement.. This partitioning can not be done by modifying the `try2` example shown earlier. The `try1` example can be modified with considerable effort to extract the precondition, consequence, and qualifier symbols based on syntactic dividers. However, the resulting code would be far more intricate than the simple parse statement above. Furthermore, revision of the brute force method requires similar complexity and effort as other parsing challenges arise.

For a more familiar example consider the following:

```
parse value "Sam likes green chili pizzas" with subject verb entree
```

The result of this parsing operation leads to the following symbol assignments:

```
subject     "Sam"
verb        "likes"
entree      "green chili pizzas"
```

Syntactic elements are now associated with meaningful symbols, instead of generic symbols **word.1**, **word.2**, etc.

How does parsing work

The REXX parse statement divides a *source string* into constituent parts and assigns these to symbols as directed by the governing parsing *template*. The parse statement has the following general form:

```
parse[2] [upper] source_identification  symbol_and_rule_template
```

Where:

upper

This is an *optional* keyword. When it is present, all values assigned to symbols are converted to upper case.

source_identification

This identifies where the *source string* for parsing is obtained. This is one of the following:

◊ ARG

Example: `parse arg a1.1 a1.rest , a2, a3.1 . , a4`

The ARG keyword indicates that one or more procedure arguments are to be processed as source strings. This is the only keyword which can have multiple source strings. Each argument passed to an internal or external procedure corresponds to the clauses separated by commas in the template pattern above. However, only 1 argument string is available for processing by the topmost procedure level associated with an EX command.

In the example above, the first word in the first argument string is assigned to symbol *a1.1* and the remainder of the first argument string is assigned to symbol *a1.rest*. The

---

2The PARSE keyword itself is omitted in ARG and PULL statements, which are actually shorthand forms for PARSE UPPER ARG ... and PARSE UPPER PULL ... respectively. Both of these forms assign uppercase values to associated symbols. The longer forms PARSE ARG and PARSE PULL must be used when you want to preserve mixed case values during assignment.

entire second argument string is assigned to symbol *a2*. The first word in the third argument string is assigned to symbol *a3.1*. The entire fourth argument string is assigned to symbol *a4*. Additional argument strings are ignored.

Empty strings [""] are assigned to all remaining symbols that appear in the template when insufficient source argument strings are available.

◊ LINEIN

Example: `parse linein first_letter 2 0 whole_line`

The LINEIN keyword indicates that the source string is obtained by reading one line from the default input stream.

In the example, the first letter within the line is assigned to symbol *first_letter* and the entire line, including the first letter, is assigned to symbol *whole_line*. When an empty line is read, then the empty string "" is assigned to *first_letter*.

◊ PULL

Example: `parse pull qline`

The PULL keyword indicates that the source string is obtained by extracting the topmost line from the external data queue.

If there are NO lines within the queue a line is obtained from the default input stream instead. This can be troublesome in numerous ways. First, if your program uses other stream functions to process lines from the default input stream, it is easy to overlook lines that are accidentally acquired by a PARSE PULL or PULL request.

Second, in many REXX environments, there is no indication that your REXX program is expecting input from the keyboard. This will cause you to MISTAKENLY believe that your session is HUNG ! Rather than automatically restarting your session, always try to type characters at the keyboard first. If you can type, your program is reading keyboard input. It is strongly recommended that you always precede keyboard input requests with prompt messages. And, you should assert that lines remain in the queue before performing PARSE PULL and PULL requests as follows:

```
if queued() = 0
  then
    if lines() = 0
      then do
         say "No more terminal input is available for parsing"
         exit 86
       end
    else
      say a_meaningful_prompt_message

parse pull keyboard_wd1 etc
```

Finally, when end of file has already been reached in the default input stream, the source string for parsing is the empty string [""]. This assigns the empty string to all

symbols that appear in the template. This can lead to unusual difficulties later during your program's execution.

You should activate the trace facility when you are developing programs that use the PARSE PULL and PULL statements, or perform other default input stream operations. Then, a helpful trace message will let you know that your program is waiting for keyboard input to complete.

◊ SOURCE

Example:

```
parse source environment proc_kind src_file proc_name implementation
```

The SOURCE keyword indicates that the source string is internally prepared by REXX with information describing the procedure's execution environment.

Within Portable/REXX™ the following symbol assignments can be expected:

environment	PCDOS | PCWIN
proc_kind	COMMAND [top level procedure] FUNCTION [procedure executing as function] SUBROUTINE [procedure was invoked by CALL statement] CALLONTRAP [procedure servicing CALL ON error handler]
src_file	the name of the file containing procedure source statements
proc_name	the current procedure's name
implementation	Portable/REXX [always]

◊ VALUE expr WITH

Example: `parse value getkey() with scan_code 2 key_code`

The "VALUE expr WITH" form establishes the result of any REXX expression as the source string to parse.

In the example above, the special Portable/REXX™ Getkey built-in function is used to obtain the double-byte code for a keyboard input action. The parse template indicates that the first byte is assigned to symbol *scan_code* and the second to symbol *key_code*. If the user had pressed function key "F1" then *scan_code* would be "3B"x and *key_code* would be "00"x.

◊ VAR variable_name

Example: `parse var rest word1 rest`

The VAR keyword indicates that the value of a symbol is the source string to parse.

In the example above, the source string is the value of symbol *rest*. The first word in this string is assigned to symbol *word1*. The remainder of the string is reassigned to symbol *rest*. Thus, every time this statement is executed, the first word is extracted, and the number of words associated with symbol *rest* is reduced by one.

◊ VERSION

Example: `parse version lang_ident lang_level release_date`

The VERSION keyword indicates that the source string is internally prepared by REXX with information which distinguishes the language implementation.

Within Portable/REXX™ for MS-DOS® the symbol assignments of the following form can be expected:

lang_ident	REXX-KilowattSoftware-Portable-BV112 [or later release]
lang_level	4.00 [or higher]
release_date	9 May 1991 [or later]

symbol_and_rule_template

This is the *template* which specifies how to parse the source string, so that symbol values can be assigned. The template can be omitted from the parsing statement. When the template is absent, the source string to parse is STILL prepared! This preparation may remove a line from the external data queue, perform a file input operation, or compose associated values when PULL, LINEIN, and VALUE are requested.

The parsing template has the following general form. Some templates can be significantly different. For example, the leading item can be a division specifier, and multiple division specifiers can appear without an intervening symbol name.

parse template form

```
symbol_1  division_specifier_1  symbol_2  division_specifier_2   etc...
```

The first character of each parsing template element is sufficient to distinguish whether it is a symbol name or a division specifier. The element is a symbol name, when the first character is an eligible symbol name character. Division specifiers are one of the following, with examples of each shown underneath:

◊ space_delimiter

Example: subject verb entree

When spaces separate symbol names within a template, then each word of the source is assigned to each corresponding symbol identified in the template. If there are more words in the source than there are names in the template, then the remainder of the source is assigned to the last symbol. All spaces within the remaining portion of the source string are preserved in this last symbol's value. If there are insufficient words in the source string for all template symbols, then words are assigned on a one-to-one basis to the leading symbols, and the empty string "" is assigned to all remaining template symbols.

Tabs are considered equivalent to spaces with respect to the *space_delimiter* division specifier. Tabs are preserved by all other division specifiers.

◊ literal_pattern

Example: "," consequence '[' qualifier ']'

Literal patterns are quoted strings within the pattern. These strings usually contain a single character, but may include many characters as well as spaces. In the example above, these are separated from other template items by spaces. However, these spaces are not necessary. Literal patterns can be immediately adjacent to other terms, as in the following example. Presume:

```
time()          15:27:14
```

Then

```
parse value time() with hour":"minute":"second
```

Causes the following symbol assignments:

```
hour            15
minute          27
second          14
```

The source string is searched from the current position until an exact match with the literal pattern is located. If the literal pattern is found within the source string, then the prior symbol is assigned all characters, including spaces, up to the last character preceding the matching source position. The characters in the source which match the literal pattern are skipped. The next character to be assigned is that which immediately follows the last character in the source string that matched the literal pattern.

◊ variable_pattern

Example: `before (delim) after`

Variable patterns are very similar to literal patterns. The only difference is that the pattern to match is the value of the parenthesized symbol name.

In the example, the value of symbol *delim* is used as a pattern. The part of the source string which precedes the pattern is assigned to symbol *before* , and the part which follows the pattern is assigned to symbol *after*. Presume the following:

```
rel_date        19 Dec 1990

delim           Dec
```

Then

```
parse var rel_date before (delim) after
```

Causes the following symbol assignments:

```
before          19_
after           _1990
```

[The '_' characters above indicate invisible spaces in assigned symbol values].

◊ column#

Examples:

```
hour 3 4 minute 6 7 second 9                    [parses: 12:44:37]

first_letter =2 1 whole_line

head =(offset) tail
```

Absolute columns are distinguished as numbers within the template, numbers preceded by an equals marker [=], or a variable reference which is also preceded by an equals marker. Absolute column 1 prepares for subsequent access to the 1st character in the source string, column 2 for the second character, etc. A column specification of 0, causes the 1st character to be accessed. Column specifications which exceed the source string length are truncated to the number of characters within the source string.

In the first example above, the following assignments occur:

```
hour      12
minute    44
second    37
```

In the third example, the value of symbol offset identifies where the source string is partitioned for assignment to symbols head and tail .

◊ relative_column

Examples:

```
hour +2 +1 minute +2 +1 second +2                    [parses: 12:44:37]

first_letter +1 0 whole_line

item1 +(width1) item2 +(width2) item3 +(width3)
```

Relative columns are distinguished as signed numbers within the template, or variable references preceded by plus and minus signs. Negative relative column motions can not access character positions less than the first, and positive motions can not access characters after the last.

In the first example above, the following assignments occur:

```
hour      12
minute    44
second    37
```

In the last example, symbols item1 , item2 , and item3 receive values from the source string according to the values of the corresponding width variables.

◊ period

Example: `file_name . revision_date .`

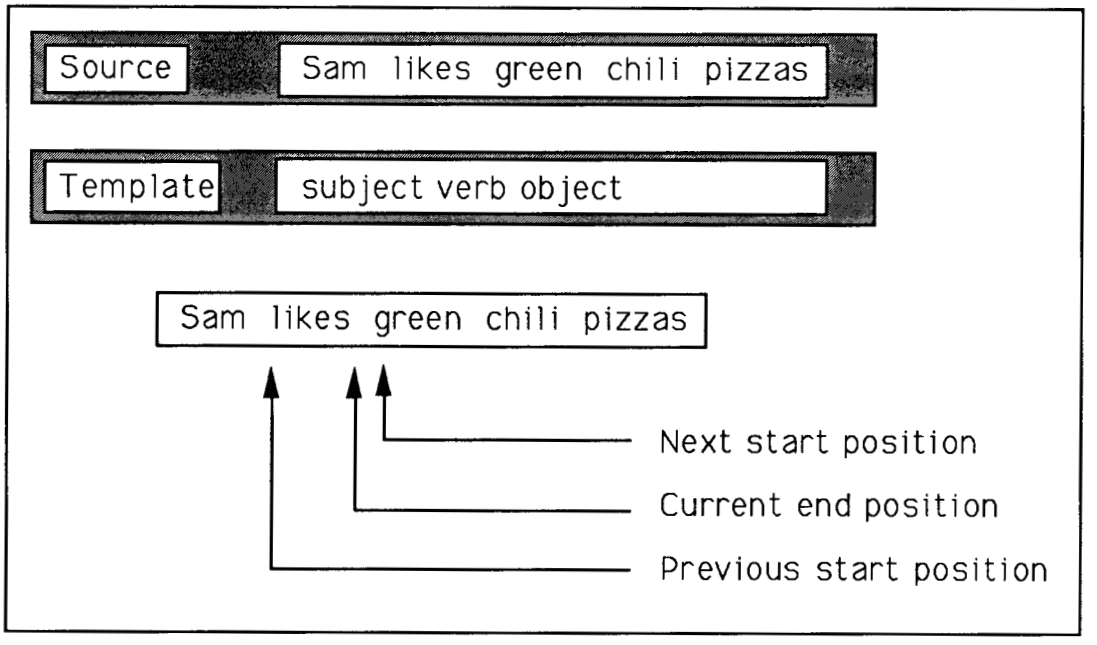
Periods within the template act as *placeholder* symbol names. These absorb values which would have been assigned to symbol names instead. A trailing period absorbs the remainder of the source string.

◊ comma

Example: `arg1_wd1 arg1_wd2 ., arg2_wd1 etc...`

Commas within the template are only used when multiple argument strings are processed by internal and external procedures. Hence, these are only valid when the ARG *source_identification* keyword is in effect. Only one argument source string is available for the topmost REXX procedure level. A comma in the template indicates that parsing of the current argument source string is to be discontinued, and processing ensues from the beginning of the next argument source string.

The following picture may help you to understand how parsing is performed.



While the *template* is processed from left to right, current positions in the *source string* are maintained. The motion of these positions is guided by the division specifiers within the template. This motion is toward the right, except when an absolute position or negative relative motion is specified. The initial start position is position 1, which corresponds to the first character at the leftmost end of the source string. An absolute position less than 1 is revised to be 1, as are negative relative motions which would precede the first source character. Likewise, the highest end position is the rightmost end of the source string.

The above picture shows positions associated with the *space_delimiter* which separates the *verb* and *object* symbol names in the template. The previous start position locates the "l" in "likes". The current end position locates the space between "likes" and "green". The next start position locates the "g" in "green". With these positions established, the word "likes" is assigned to the symbol name "verb". As only the object name remains in the template, the remainder of the source string from the next start position is assigned to symbol name *object*. This is the phrase "green chili pizzas". If there had been multiple spaces between the words

"likes" and "green" then the next start position would have located the second intervening space.

Power parsing

Now two common applications of parsing will be studied. The first shows how to meaningfully extract variable length text information from MS-DOS® files. The second shows how to extract fixed length information from files.

Parsing variable width text fields

Assume that you want to analyze information in a name&address file. Each line of information contains multiple fields of varying length. The fields are separated by tab characters ["09"x]. The file could have been obtained by extracting rows from a database or spreadsheet program. Alternatively, it could have been created by a REXX program which wrote lines with the following request.

```
tab = "09"x

call lineout "nad", ,
  fname ||tab|| mname ||tab|| lname ||tab|| company ||tab|| addr_line1 ||tab|| addr_line2 ||tab|| ,
  city ||tab|| state ||tab|| zip ||tab|| phoneno
```

The parsing of input lines into meaningful fields has the same structure, and uses the tab symbol as a variable pattern specification. Fields can be obtained as follows:

```
tab = "09"x

parse value linein( "nad" ) with ,
  fname (tab) mname (tab) lname (tab) company (tab) addr_line1 (tab) addr_line2 (tab) ,
  city (tab) state (tab) zip (tab) phoneno
```

Fixed width binary data analysis

Instead of a file containing variable width fields, suppose you have a file containing fixed width character fields and binary-encoded numbers This file could have been created by a REXX program which wrote lines with the following request.

```
call charout "tranfile.db", ,
  left( partno, 8 ) || left( serialno, 8 ) || d2c( unit_price, 2 ) || d2c( quantity, 2 ) || ,
  d2c( subtotal, 4 ) || d2c( tax, 4 ) || d2c( total, 4 )
```

The parsing of this information into meaningful fields has a similar structure, with an extra step to convert each binary-encoded value to a corresponding decimal value. Fields can be obtained as follows:

```
parse value charin( "tranfile.db",, 32 ) with ,
  partno +8 serialno +8 unit_price +2 quantity +2 subtotal +4 tax +4 total +4

unit_price    = c2d( unit_price )
quantity      = c2d( quantity )
subtotal      = c2d( subtotal )
tax           = c2d( tax )
total         = c2d( total )
```

This concludes the description of how to perform parsing operations in REXX. To fortify your understanding of parsing you should now try some experiments of your own choosing. You should also read the section titled "Parsing for ARG, PARSE, and PULL" in "The REXX Language".

This paper is an excerpt from:
Learning to Program with Portable/REXX™

which is published by Kilowatt Software at the following address:

1945 Washington St, #410 San Francisco, CA 94109-2968 (415) 346-7353

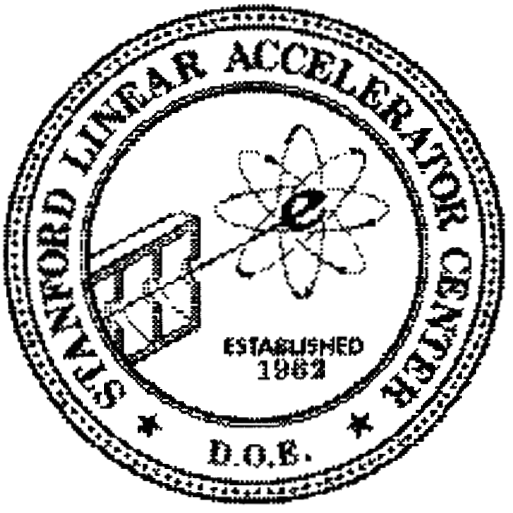
USING REXX TO TEACH PROGRAMMING

BEBO WHITE
SLAC

Using REXX

to Teach

Programming



Bebo White
SLAC
2nd REXX Symposium
Asilomar Conference Center
May 8, 1991

Why? (IMHO)

- The programming education community needs:
  - a flexible, interactive, powerful language with emphasis on basic programming concepts
  - to separate programming instruction from "the language wars"

- Programming students need :
  - a meaningful first exposure to the elements and art of programming
  - the positive feedback of being able to write code quickly and "watch it work"
  - not to be intimidated or bored by concepts couched in language specifics

Typical(?) Goals of a Beginning Programming Course

- To teach that programming can be fun and something to take pride in
- To provide experience with systematic design processes
- To provide early experience with program documentation development
- To provide a knowledge of general principles applicable to many programming languages
- To provide experience with software tools
- To teach attention to style

Typical Beginning Programming Course Curriculum

- Smooth transition from everyday planning experiences to formal design of programs
- Early use of sufficiently complex problems where algorithmic solutions are not immediately obvious - motivates PDM
- Early treatment of issues arising from "large" problems
- Logical introduction to control structures ("structured design")
- "Gentle" introduction to data types, variables and parameters
- Discussion of data structures and data abstraction

What REXX Has To Offer

- An "algorithmic" language "close" to pseudocode
- Allows "self-documenting" code
- Macro capability allows "getting something done fast"
- Modern control structures which are customizable; exceptions allowed in well-defined cases
- Generalized data types; undefined variables
- Generalized/simplistic data structures; user-defined data structures

What REXX Has To Offer (cont.)

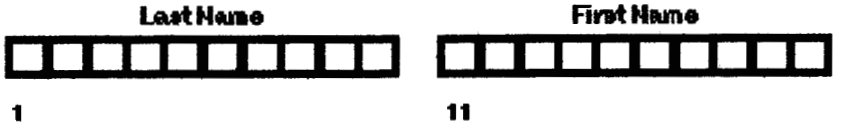
- Generalized/simplistic data abstractions
- Generalized I/O
- Function libraries
- Trace - for debugging and a learning aid
- Sophisticated features/capabilities "under the covers" (e.g., hex manipulation, recursion)

Teaching Data Structures

- Data Structure concepts should go from the most generalized (i.e., a familar analog) to the most specific
- Data Structures should be perceived as a viable entity which can be easily taken apart and manipulated
- "Algorithms + Data Structures = Programs"

Records

- To the "layperson" records look like lines in an application form:



- To the "computer_person" records are "values of various datatypes of differing lengths appended to one another in a specific order"

REXX Knows Both Records

"Layperson" Records as:

parse var NameInfo.1 LastName 11 FirstName 21

"Computer_person" Records as:

NameInfo.1.LastName = ...
NameInfo.1.FirstName = ...

Data Abstraction

- **There is a Share requirement to:**

  Allow an expression/variable to be the target of an assignment statement

- **To the "layperson" this is a _____?**

We Need To...

- Continue to develop the REXX language following its philosophical tradition
- Develop major applications using REXX
- Promote REXX as a mainstream programming language
- Insure the availability of REXX on as many computing platforms as possible

REXX AND UNIX PANEL DISCUSSION

JEFF LANKFORD, NORTHRUP; SAM DRAKE, IBM; JONATHAN JENKINS, AMDAHL;
SCOTT OPHOF, CONSULTANT; ALAN THEW, UNIVERSITY OF LIVERPOOL

REXX and UNIX Panel Discussion

Moderator: Jeff Lankford
Panel: Sam Drake, Jonathan Jenkins, Scott Ophof, Alan Thew

Moderator's Note: The complete session was taped, but the recordings are being withheld pending filing of formal charges. What follows is a collection of prepared notes, not necessarily as full of wit as the delivered presentations; you simply had to be there.

Introduction

Jeff Lankford

Good Afternoon, Ladies and Gentlemen...

On behalf of the participants, welcome to the panel discussion "REXX and the UNIX Environment". We are fortunate today to have panelists outstanding in their fields and who have occasionally been found out standing in other people's fields.

A common theme is addressed by all the panelists: why use REXX with UNIX — and how to do so effectively. The first speaker, Alan Thew of the University of Liverpool Computer Laboratory, will discuss "REXX and awk: how does REXX fit in with existing programming tools?", in which he compares the various interpreted languages commonly used in the UNIX environment and contrasts their capabilities with those of REXX. The next speaker, Jonathan Jenkins of Amdahl Corporation will present a practical perspective on "UNI-REXX use at Amdahl". The third panelist, Scott Ophof, formerly of Delft Hydraulics, will discuss portability concerns when using "REXX on any System". The concluding speaker, Sam Drake of IBM Research in San Jose will raise the question "REXX and UNIX ... what's the point?", and examine the issue of finding REXX's proper niche in the UNIX environment.

Before proceeding, I'd like to exercise my prerogative as moderator to discuss something completely different, by raising a perennial question that has converted many doctors of philosophy into cabbies (and vice versa): "What am I doing here?" The special case is clearly more interesting, namely: "What am I PERSONALLY doing here?"

I am neither a chronic nor habitual user of REXX; my use is best characterized as recreational. In fact, I haven't touched a piece of REXX code in nearly six months; if I stay clean a whole year, the doctors tell me I'll be cured. My first experience with REXX occurred a few years ago when I undertook a project to implement international standard networking protocols in the IBM VM environment. Entering the VM environment from a UNIX background was traumatic, partly due to the paucity of convenient-to-use program development tools.

I soon learned of REXX and found it unlike any other standard VM utility: it was easy to learn and to use, it supported rapid prototyping, it supported personal tailoring of the system command language, and the string processing functions promoted its use as a macro processor. Using REXX, in about a month I built the core of a UNIX-like program development environment that provided networked hierarchical file reference, compatible file, device and interjob I/O, asynchronous job initiation, and implementation of nearly one hundred UNIX-like commands front-ending either VM commands or custom built REXX functions. Without REXX, program development in a heterogeneous networked environment targeting applications for compilation and execution in the VM environment would have been much less productive.

In his 1984 paper published in the IBM Technical Journal, Mike Colishaw succinctly described one of the major reasons for REXX's popularity in the IBM world: "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." While certainly true of many IBM environments, this is less true of the UNIX environment, where several stream editors, command language and macro processors offer complimentary and compatible features.

Hence, there are potential barriers to acceptance of REXX in the UNIX environment. A rudimentary classification scheme distinguishes between barriers of style and barriers of substance. The former category includes the stylistic difference between the UNIX philosophy of making each tool do one thing well, together with the anticipation that tools should interact via "piped" data streams, versus the typical VM practice. Substantive barriers include differences between external I/O models, for example between the UNIX system's three distinct data streams for input, output, and error messages versus REXX's single-threaded data buffer chains or stacks. Also, the lack of regular expression manipulation built-in functions as a standard part of the language could be considered a barrier to the acceptance of REXX. Another barrier is the rudimentary signaling and event handling mechanism. Stylistic barriers can be addressed by acculturation of REXX application programmers to the UNIX environment, but substantive barriers require innovative implementations or even extensions to the evolving standard to provide REXX program accessibility to standard, popular UNIX features. While there are many excellent reasons supporting the use of REXX in the UNIX environment, the real challenge, for you the audience, as much as for the panelists, is to seize this opportunity to cooperate in the uncovering of potential barriers and to begin to formulate reasonable solutions.

On behalf of the sponsors of this second REXX Symposium I want to thank the panelists for subjecting themselves to the mercy of the crowd and most especially to thank you, the

crowd, for making this session memorable.

REXX and AWK/ksh - Can the former learn from the later?

A. J. Thew

I am an applications programmer at the University of Liverpool Computer Laboratory, U. K. We currently run a VM/CMS service for most users but are moving to Unix and by late 1993 will base all our user service on Unix.

I have used REXX for about 5 years. At present this is primarily in conjunction with the SQL/DS RDBMS using RXSQL. REXX is used extensively in data manipulation along with CMSPIPES (a major contribution to REXX on CMS). This represents the bulk of my job. My Unix experience started about 2 years ago and now that I've passed some of the worst part of the learning curve, my main activities here are investigation of public domain (and other) tools, e.g. an e-mail for a new Unix system and editors. I have used all major shells to some degree. Some work with C has been done and familiarity with other tools is increasing. I have also used to some degree all major Unix's (BSD, SunOS, HP-UX, System V Releases 2 and 3).

I want to try to say something about some existing tools on Unix and how they might relate to REXX and vice-versa. These tools are the other interpreted programming languages. This emphasis partly reflects my job and interests at this present time.

My first reactions on seeing the shells from the programming point of view was that they seemed very primitive compared to REXX. It was as if REXX was removed from CMS and EXEC1 was the command interpreter and general programming language. No free format and plenty of chances for error.

David Korn (of AT&T) (1) and Morris Bolsky say "no unquoted spaces or tabs are allowed before the '=' or after". Poor manipulation of strings and almost no arithmetic were additional first impressions. Others coming from a CMS background around the world felt similarly but were met with unsympathetic responses such as "**sh** does all you need" from the existing Unix community. The obvious "problems" were visibly demonstrated by Neil Milsted (2) during last year's symposium.

Times change and I now feel I can use **vi** faster for some operations than XEDIT (I should point out that this was partly out of necessity). In addition I have had plenty of opportunity to look at what interactive programming tools Unix provides. This has mainly centered on AWK abut also the Bourne Shell (**sh**) and the Korn Shell (**ksh**).

My attention was grabbed by AWK since on face value it offers a concise simple syntax like C, but without data-typing, semi-colons, memory management, pointers but with real arrays, *proper* string manipulation functions, arithmetic, simple assignments (no dollar signs in most cases), and some ability to interact with Unix. Arrays are associative which seemed similar to REXX compound variables. It seemed to offer the best of C and shells without any of the pain and with functionality that I'm used to with REXX on CMS.

I have attempted to take a serious look at shells, and have recently standardised on the Bourne Shell, in particular the version dating from System V Release 3. Apart from BSD systems and older System V releases, this is reasonably widespread. It offers more compatibly with other tools, allows redirection on the read statement, more built-ins for better performance and improved parameter checking/substitution and above all functions (though *not* recursion without pain).

In addition David Korn has produced a shell that is largely compatible with the System V Release 3 **sh** but enhanced a great deal. This offers:

- much better performance than existing shells,
- greater functionality,
- more general I/O,
- built in arithmetic,
- some string functions,
- local variables in functions (i.e. recursion),
- limited array capabilities,
- co-processing features,
- and better security.

This shell is gaining in popularity. To illustrate its capabilities, Morris and Korn's book present a powerful subset of the Rand Mail message handler. This is not the "Word processor in FORTRAN" type application but a realistic project. They even claim that the **show** and **next** commands are faster than the C equivalents. However, the code is not nice to look at in my opinion (although it is probably easier to understand than the "real thing" in C) but it's very compact, being less than a 1000 lines of code and comments.

AWK derives its name from its designers Alfred Aho, Brian Kernighan and Peter Weinberger. An additional interest was that the Free Software Foundation provided a version that worked on a PC which was exactly the same apart from pipe support and was constrained by 640K. This is also available for Unix and implements all the latest functionality.

I should stress that I'm referring to what is commonly called "new AWK" or **nawk** and references to AWK will refer to **nawk** unless stated. This version has been available since 1985 and is available as standard (with the old **awk**) from most vendors.

The original language was written in 1977 and its basic action was to "scan a set of input lines in order, searching for lines which match a set of patterns which the user has specified"(3). An action (code section) can be taken when lines match a pattern (default is print). The "patterns may be more general than those in grep, and the actions allowed more involved than merely printing the line". The language was designed for ease of use rather than speed.

AWK provides implicit and explicit data input, the latter not being required for a working program and some books do not present any treatment of the explicit input until toward the end. When given a file to read as an argument AWK reads it sequentially, as records which where the record separator defaults to a newline. The original was designed for short programs of one or two lines. They designers "knew" what the language was designed for but many users, often first time computer users found that the ease of use made it a general programming language and used instead of others. This "shocked and amazed" the designers who assumed that compiled languages (presumably) would be used for anything longer than a few lines. Users often seem willing to sacrifice

performance when ease of use is available, BASIC was/is an example of this.

The AWK users had their demands met and the 1985 version contained dynamic regular expressions, new built in variables and functions, multiple input streams with more explicit I/O, and user functions. The AT&T System V Release 4 version makes some additional enhancements in the spirit of the 1985 release but not so numerous. Function libraries do not have explicit support but are easy to implement.

Examples of major applications which use **awk** are a **nroff** type text formatter (written for early versions of Unix which did not bundle nroff with them) and small Lisp interpreter.

It is interesting that Aho, Kernigan and Weinberger mention REXX in a discussion about "similar" languages (mentioning SNOBOL4 and ICON) (4). This statement was a small prod for my topic, I'll have to admit.

There are built in software limitations (4) [to AWK]:

- 1 open pipe,
- 15 open files,
- 100 fields,
- 3000 chars per input record,
- and 1024 chars per field.

These were designed in or not designed out since the performance would have degraded substantially for one thing. *But* the main thing that AWK users miss/need is that the lack of any debugging facilities. **nawk** and the Free Software Foundation's **gawk** give better error diagnostics when the program fails which is an improvement to old AWK's "bailing out..." error. Debugging has to be done on the lines of any other language without built in tracing or an available interactive debugger.

The Unix shells both provide builtin tracing, **ksh** allows command scanning without execution and something like REXX's *TRACE R* command although no interactive tracing. AWK is poor at replacing the shell command interpreter functions since it's use of pipes is restricted to either input or output (remember the one open pipe limit). It has no interrupt handling facility like the [**sh**] **trap** command, and there are other limitations but this is not "pushing" the language as much as abusing it. AWK's real strength is as a data processing language.

AWK passes arrays to functions by reference and not by value and scalars by reference. It is possible to have local variables by effectively hiding them on the function line:

function fred(a, b, loc1, loc2)

Functions can be recursive.

Users requiring extra performance can buy the **awkcc** program from AT&T which converts the program to C and then compiles it. **awkcc** does not come with any vendor versions of Unix that I know of. A company even provides a proper compiler but only for the PC.

The design goals [of REXX] were/are very different. They make REXX a bigger language than AWK, the latter is often referred to as one of Unix 's little languages. REXX was "designed for generality" (5) which makes it suitable for many tasks, one of which is a command processor:

- readability,
- no explicit data-typing,
- good diagnostics via limited span syntactic units,
- low astonishment factor (predictable results even when features accidentally misused),
- language kept small in sense of number of commands,
- and no defined size or shape limits.

A feature of REXX that has always impressed me as a user was the debugging facilities, just add the required trace command and go.

Goal was to have good performance as well. CMSPIPES as well as performance tips has enabled us to double performance of critical execs on CMS.

REXX 4.0 goals(2): Still to "keep the language small", enhancements chosen on "high power-to-complexity ratio". This last phrase sums up REXX for me in that it is small in some aspects and when interpreted slow on very large programs but otherwise without limit.

Some additional functions are required by REXX to allow it to communicate in the most basic way with the shell which exec'd it and pass commands to a shell to execute [on UNIX]. These were listed by Neil Milsted of the Workstation Group at the last Symposium (2) and provide enough, **cwd**, **getenv/putenv**, etc.

One missing feature is currently regular expression support, a feature of many Unix tools which do some pattern matching. This allows the shells to have some ability to manipulate strings without any inherent string functions. This "lack" is not so apparently bad when one examines the wealth of functions available with REXX, many more than AWK. It was an interesting exercise to see which functions could be implemented as user functions in AWK. Only *JUSTIFY* and *VERIFY* looked hard. REXX's parser is more generalised than AWK's but AWK's use of a simple user definable field separator which itself can be a regular expression should not be underrated. AWK has math functions and substitution functions (type of line edit) that REXX does not have although only the math functions would need a function package to get good performance.

Some shell features are missing such as interrupt handling and some of the advanced features of the **ksh** but these could be provided possibly by function packages without making the core language on Unix non-standard and the manuals twice the size. REXX is easily the language to replace some shell programs and become a major command language for for Unix. However, if REXX becomes *just* another language for scripts that would seem rather limiting since it was designed to be a general purpose language. C and **ksh** (where available) can do a better job in some cases, although a REXX compiler for Unix would be very interesting.

The standard should prevent REXX becoming a monolithic Unix tool which could be tempting but dangerous in my opinion for the above reasons and most importantly would go against proven design objectives. Unix already has a public domain tool which possibly provides this monolithic functionality in Larry Wall's PERL but his design goals were different.

AWK is a programming language in its own right, even taught on some software engineering courses apparently. It was designed to fit in with the Unix philosophy of being a tool to do a job, a tool of many. This philosophy lets other languages do other jobs such as sorting, or better handling of command line arguments (shell wrappers). While REXX

is better suited to cover more ground than AWK and "who wants to learn many programming languages when they can learn one?", REXX could be seen as providing more/better facilities rather than just a replacement language which ignores as much of Unix as possible and re-invents the wheel many times.

References:

(1) Bolsky, M. I. and D. G. Korn. *The Korn Shell - Command and Programming Language*, Prentice-Hall, 1989.

(2) *Proceedings of the 1st REXX Symposium for Developers and Users*, SLAC, 1990.

(3) Aho, A. V., B. W. Kernighan and P. J. Weinburger. *AWK - A Pattern Scanning and Programming Language*, Unix Programmer's Manual, Vol. 2, 1978.

(4) Aho, A. V., B. W. Kernighan and P. J. Weinburger. *The AWK Programming Language*, Addison-Wesley, 1988.

(5) Colishaw, M. F. *The REXX language A practical Approach to programming*, Prentice-Hall, First Edition, 1985.

Uses of REXX under Unix at Amdahl's Corporate Computer Center

J. L. Jenkins

Hi! My name is Jonathon Jenkins and I work for Amdahl Corporation. I'd like to talk to you today about some of the practical applications that we created under our Unix systems using REXX. Please feel free to contact me if you have further questions, or would like to see copies of this code.

Currently 43 uses have been determined for REXX under Unix. Categories include:

- System Monitoring
  - Check for the completion of system accounting
  - Dump Management
  - Checking console logs for system and device errors
  - Daily cleanup of temporary filesystems
  - Continuous montoring of permanent filesystem usage
- Security
  - Checking for unauthorized superuser access
  - Checking for incorrect users in the /etc/passwd file
  - Checking for unauthorized members in system groups
- Disaster Recovery
  - Backing up of critical system files to root and /usr

When I attended the 1990 REXX Symposium, I learned of a product, called Uni-REXX that is a REXX interpreter for Unix systems. After returning to work I recommended that we purchase this product. I was asked to provide a justification as to why we needed this product and how we could use it on our UTS systems at the Corporate Computer Center. I began to make a list of ways to use REXX under UTS. Since that time, the list has grown to contain 43 separate items, some of which I have begun to write. These items seem to fall into three separate categories.

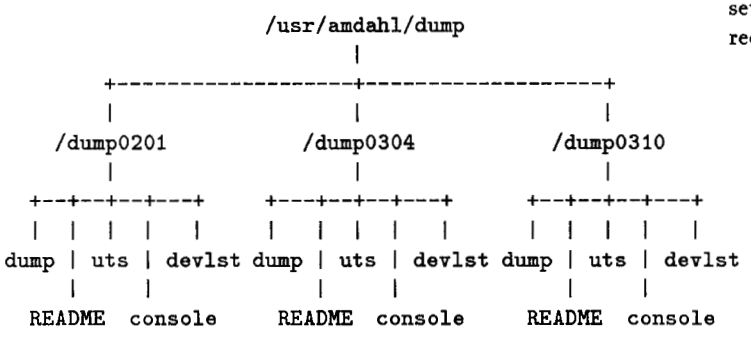
System Monitoring

Periodic checking of the system. Items which fall into this category are:

- Checking the status of System Accounting processing. System accounting generates usage reports on UTS each night. Sometimes the processing software encounters unrecoverable errors. Stewards should check daily to ensure that all of the accounting data from the previous day has been processed. If this is not done, critical charge-back data is not processed to bill CCS UTS customers.
- Check for successful completion of accounting. This can be automatically checked by a program which notifies the system steward only if problems were detected.
- Dump Management:
  The /dump and /usr/amdahl/dump filesystems are used to hold the current dump file and previous system dumps. When a UTS system panics (abends), an image of system storage is written into a file named dump in the /dump directory. Due to the storage sizes of some of the UTS systems, this filesystem can typically hold only one dump at a time. Previous system dumps are copied into the /usr/amdahl/dump filesystem for examination. Here are some of the things that may be checked automatically using REXX are:
  - Make sure that a dump file is on this directory. Create one using the *makedump(1m)* command if dump not found.
  - Make sure that the dump file is the only file in this directory. All other files associated with dumps should be placed in the /usr/amdahl/dump directory. Files found which are not associated with dumps should be removed.
  - Check the %full (blocks and inodes) of the filesystems.
  - Ensure that the following directory scheme is adhered to for each of the files found in the /usr/amdahl/dump directory: a README file that contains a description of why the system was down, the dump file, a copy of the related kernel (/uts at the time of the dump), a copy of the console log (/usr/spool/console/<dump_date>), a copy of the /etc/devicelist.

Following is an example of how the directory structure underneath /usr/amdahl/cump could be organized. Note that the "0304" in the "/dump0304" represents the date of the dump 03/04 of the current year.

- Checking console logs for system and device errors. The system console log contains information about current and significant events on the UTS system. Sometimes it contains sensitive information such as passwords, administrative commands, and system operation information which is not suitable for clients. Some of the important pieces of information contained in the console process stack error messages, unsuccessful logon attempts.



- Unsuccessful login attempts
- Device Data Checks, Equipment Checks, and Unit Checks
- Line timeouts and restarts (PVM, 3274e)
- Missing Interrupts
- Process error messages (ie. Stack too Large)
- Tape mount request information

- Following are some the things which may be checked automatically and responded to/reported on via programs.
  - Permissions, owners, and groups of files in the */usr/spool/* directory. These files/directories may be checked daily and may correct the problem as well as report it to the system steward.
  - System Error Messages such as those listed below, may be collected and delivered to the appropriate groups. Note that the possible groups are listed after each message: Data Checks (UTS/SSS), Unit Checks (UTS/SSS), Equipment Checks (UTS/SSS), Unsuccessful logon attempts (Computer Security), RSCS shutdown/restarts (UTS/TSG), PVM shutdown/restarts (UTS/TSG), Line timeouts (UTS/TSG), Process stack too large (UTS/TSG), Out of paging/swap space (UTS/TSG, UTS/SSS), Ethernet network unavailable (UTS/TSG)
  - The console log contains information about tape mounts. It shows when a mount request was received, when the request was satisfied, and when the tape user completed use of the tape. From this information, we can generate the following types of reports: How many tapes were requested for the day, How long (including average times) tapes were mounted, How many tapes were mounted over 3 shifts (grave, day, swing), How long it took to satisfy mount requests (including average mount times), Flagging of tape mounts which take longer than a predetermined threshold (currently 10 min.).
- Daily cleanup of temporary filesystems like /tmp, /usr/tmp, and /free. These directories are used to hold temporary information on our UTS systems. All users are able write to these directories, and don't always remove their temporary files when they are done. Because of this, these directories run out of space. Following are several things to be checked daily, concerning these directories:
  - Remove all files and directories older than a predetermined amount of time.
  - Warn system stewards and operators when %full (blocks and inodes) is greater than a predetermined threshold and again at 90%.
  - When 90% full (blocks and inodes) and files greater than 3 days old have been removed, remove those over 2 days and then those over 1 day old. If it is still full, then start removing files in reverse time order. (**ls -lt** ... older files first)
- Continuous montoring of permanent filesystem usage. From time to time (at least one per day), filesystems on UTS run out of blocks or inodes. The kernel only places a message on the console once the filesystem is full. Through the use of programs and the **df(1m)** command, we can continually monitor the usage of filesystems and alert the appropriate personnel when they began to become full.

Security

- Checking for unauthorized superuser access. On UTS, the super-user account has complete authority over the system. This user can read or write any file on the system, it can change anyone's password without security restrictions, it can kill any process on the system, modify the kernel and system source code, and write directly to any device on the system. Each of these priviledges is something that should only be available to a select number of system users, therefore access to super-user should be monitored daily to make sure that only authorized users have this ability. Information about superuser access is logged by the system. It is possible to check this log for unauthorized accesses and unsuccessful attempts. This information can be delivered daily to the system stewards for action.
- Checking for incorrect users in the /etc/passwd file. The /etc/passwd file contains a list of the users who are valid to UTS. This file should be checked for inconsistancies and potential security holes.
  - Find users with home directories and .login, .cshrc, and .profiles that are accessible to others. Notify these users of the problem, and of the potential problems of having these files open to others. Change these settings after two weeks/month of notification.
  - Make sure that expired users have a login shell of /usr/dirm/bin/bye
  - Check users no passwords.
- Checking for unauthorized members in system groups. The group permissions of files play a large part in determining who can access them. It is important that the permissions of these files are set correctly and that the members of certain groups are checked regularly.
  - Valid system groups and users kept in the control file.

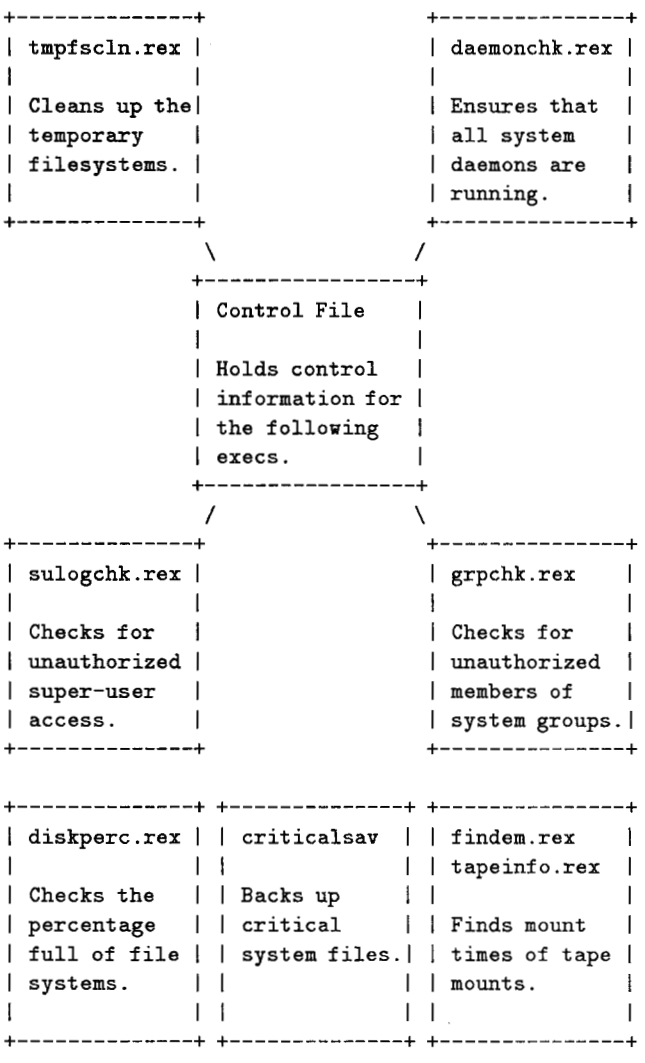
– Groups to check are: bin, mail, adm, oper, sys, tape, dev, uidadm

Disaster Recovery

Backing up of critical system files to root and /usr Due to the importance of some system files, it is necessary to have a backup online in case the file is inadvertantly changed or destroyed. It is possible to have this function done via a program which runs daily. These files should be backed up onto two separate disk volumes to minimize the chances of both copies being destroyed.

- Should be kept backed up both on root and usr volumes. (/.critsave and /usr/dirm/tsg/.critsave)
- The files to be backed up are as follows: /etc/devicelist, /etc/passwd, /etc/group, /etc/identity, /etc/identitydefs, /etc/inittab, /etc/hosts, /etc/services, /etc/netinfo.db, /etc/netstart, /etc/profile, /etc/rc, /etc/bcheckrc, /etc/brc, /etc/local.rc, /usr/spool/cron/crontabs/root
- These should be written to tape weekly.

Examples



The next example demonstrates how the system is put together. These routines are called from the **cron(1m)** command. They are executed at regular intervals and typically produce exception reports. I have included a copy of the source code of the *daemonchk.rex* program which checks to make sure that all of the system daemons listed in the control file are actually running on the system. I have also included the lines from the control file which this exec uses to initialize the daemonlist variable as well as a copy of the *crontab* file used to automatically execute these execs. This exec demonstrates the power of using the functions of Unix along with the power of the REXX language.

Sample crontab file:

```
#
# min     hour    day     month   day_of_week  command
#
#(0-59)   (0-23)  (1-31)  (1-12)  (0-6
#                                 Sunday=0)
#
#-------------------------------------------------
#
# System Steward
#-------------------------------------------------
#
# -> Check disk full percentages and notify contacts.
0 6,16 * * 1-5 rexx /autoops/stewards/diskperc
#
# -> Check disk full percentage every hour during day
0 7-14 * * 1-5 rexx /autoops/stewards/diskperc
0 16-18 * * 1-5 rexx /autoops/stewards/diskperc
#
# Check disk full percent every hour during off shift
0 20,22 * * 1-5 rexx /autoops/stewards/diskperc
0 0,2,4 * * 1-5 rexx /autoops/stewards/diskperc
#
# -> Check to make sure that system daemons are running
# -> every 5 min. during the day.
0,5,10,15 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
20,25,30 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
35,40,45 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
50,55 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
#
# -> Check to make sure that system daemons are running
# -> every 15 min. during off shift.
0,15,30,45 19-5 * * 1-5 rexx /autoops/stewards/daemonchk
#
# Automated Operations
#-------------------------------------------------
0 0 * * 0-6 rexx /autoops/tpmnts/findem
#
# Security
#-------------------------------------------------
31 2 * * 1 rexx /autoops/.security/passwdchk mail nomsg
35 0 * * 1-5 rexx /autoops/.security/criticalsav
0 5 * * 0-6 rexx  /autoops/.security/grpchk
#0 5 * * 0-6 rexx /autoops/.security/tmpfscln
#0 3 * * 0,3,5 rexx /autoops/.security/sulogchk
```

Control file lines:

```
steward jlj50
daemon <swap> <init> <wss_dmn> <wss_steal> <flush>
```

```
daemon <sys_0> <sys_1> <sys_2> <sys_3> osmcat
daemon cron nadaemon admin802 tpdaemon
daemon lpsched reread spls tdmr dbspvsr
daemon portmap
daemon adminllc ipadmin tacomad biod
daemon mountd nfsd sendmail inetd sts
daemon slink mr dbcopyd rscsd
daemon errdemon
```

DAEMONCHK.REX Program Code:

```
/* REXX ***************************************
*                                             *
*  Name: daemonchk                            *
*  Date: 02/04/91                             *
*  Time: 03:59:40                             *
*  Auth: Jonathon Jenkins                     *
*                                             *
*  This exec will check to make sure that all *
*  of the system daemons listed in the control *
*  file are running on the system.  It will   *
*  print them if more than 5 copies are running *
*                                             *
***********************************************
*            Change History                   *
*------------+--------------------------------*
*    Date    |  Description of Changes        *
*------------+--------------------------------*
*                                             *
*                                             *
*                                             *
***********************************************/

address unix
arg .

found.=0
daemonlist=''
control_file='/autoops/control_file'

do queued(); pull; end
x=popen('/bin/grep system_daemons 'control_file)
do while queued()>0
   parse pull keyword . 1 rest_of_line
   if keyword='system_daemons' then do
      parse value rest_of_line with . daemons
      daemonlist=daemonlist daemons
      end
end

do queued(); pull; end
x=popen('/bin/ps -e | /bin/grep -v getty')
do while queued()>0
   parse pull . . . daemon
   daemon=translate(daemon,'_',' ')
   if wordpos(daemon,daemonlist)/=0 then
      found.daemon=found.daemon+1
end

do count=1 to words(daemonlist)
   daemon=word(daemonlist,count)
   if found.daemon=0 then
      say 'daemonchk: 'daemon' was not found',
          ' running on the system.'
   if found.daemon>25 then
      say 'daemonchk: 'found.daemon daemon' were',
          ' found to be running in the system.'
end

exit
```

My management is currently planning to purchase the Unix-REXX product from Wrk/Grp pending the availability of funds. I used a copy of the product that was used to port the interpreter over to UTS to test this code.

REXX on any system

F. S. Ophof

I'm basically a CMS user on the systems maintenance side, trying to find out what UN*X means (kicking and screaming all the way...).

My introduction to REXX was on an IBM 4331 running CMS a few months after starting to program in EXEC2. This was at DELFT HYDRAULICS, my employer at the time. EXEC2 did not really seem an attractive one to write XEDIT macros in, so I took to REXX like a fish to water and never looked back.

Personal REXX and Kedit made the PC a more attractive tool for the CMS users. This led to problems porting applications from CMS to the PC and more people were using *both* versions of REXX, one on the mainframe, the other on their PC.

A new policy at DELFT HYDRAULICS dictated that VMS, CMS, NOS-VE, and PCs were to be replaced by UN*X where possible. The CMS users wouldn't really be happy with **vi** (which they spelled Y-U-K). So from the user support point of view I looked around for a UN*X version of REXX and XEDIT, mainly via e-mail.

It was Alan Thew who mentioned uni-REXX and uni-XEDIT. My hope was the implementors hadn't followed the "include everything but the kitchen-sink" philosophy. Well, the Workstation Group was distributing a very clean, CMS-like version of REXX and XEDIT. Cleaner than I first thought. But it wasn't available yet for the HP 9000 series 800, the machine in use at DELFT HYDRAULICS.

In the meantime Alan and I were discussing REXX (and XEDIT) on UN*X with Ed Spire. My main contention was, and still very much is, that what functionality doesn't belong in a program should not be included. This is in line with the UN*X philosophy of making each tool do its own thing well. I would like to add "and *only* its own thing".

Uni-REXX and uni-XEDIT for the 800 model arrived, were installed at DELFT HYDRAULICS, and have been in use a couple of months. The last I heard is that they were very happy with these UN*X versions.

REXX on any system — What a wonderful idea.

REXX has been implemented on a respectable number of operating systems. And there is no problem using REXX, as long as programs built for a specific operating system stay there.

But when interoperability is needed...

Each operating system has its own peculiarities. So each REXX implementation needs to be adapted to that environment. The result is that no two implementations are identical. The main differences are in:

- I/O models (byte streams on UN*X, records on CMS),
- file specification (*/dir/subdir/part1.part2.etc* on UN*X, *FILENAME FILETYPE FMD* on CMS),
- and operating system philosophy (filters in UN*X, *eierlegende MilchSau* in CMS, "hog all memory and do it *my* way" in DOS).

So when an application is copied from one operating system to another, it comes down to virtually rewriting the whole thing.

This is not my idea of "REXX on any system"...

To be able to have interoperability apply to REXX, the following might need to be done:

- Make REXX programs completely independent of operating systems.
- Modify REXX so it can recognize for which opsys the program was originally written and interpret accordingly.

Opsys independency — Beautiful! Can it be done? If so, what is needed? Externalize the I/O model? Modify the I/O model so the syntax is valid for any conceivable system?

Would a complete rewrite of REXX be necessary? Or could it be done with a number of additions/extensions? What about current users of REXX?

A lot of questions...

Recognize the opsys: this *could* be done with an external set of functions and procedures. Con: Each new implementation of REXX would create the need for new sets of translators (being twice the number of already existent implementations). Pro: One would only need the translation set(s) for those opsyses on expects to translate from.

Any change to REXX itself to achieve this would of course need to be independent of implementation, since one cannot expect the user to buy a new version of REXX for each new implementation to become available.

A logical extension would be to create a "neutral" set of functions and procedures to bring down the number of translations sets. And so we are just about back at the first possibility (independency of opsys).

As to the Uni-REXX implementation, it's quite "clean" (in uni-REXX the **cd** function is necessary). Some of the other implementations could use a bit of clean-up as to modularity.

Statements and functions which are not REXX specific should be relegated to external programs [or] function packages.

The manuals should state clearly which are standard REXX statements/functions, which are implementation dependent, and which are add-ons the implementor offers.

Examples:

- DIAG() in the CMS implementation.
- The whole hardware and DOS groups of functions in Personal REXX.

Add-on due to presentations already here:

- UN*X has multi-tasking. How does this affect the SIGNAL statement? Would SIGNAL need to be enhanced for UN*X? And, how does this affect interoperability? Would implementing the enhancements in UN*X (even as NOP) be a good idea to copy to other implementations?
- Replacing, inserting, deleting a line within a CMS file is very easy without destroying the rest of the file. But using LINEOUT() looses everything after the last line worked on. My reaction was major panic. So on the PC I use EXECIO, not the REXX I/O facilities.
- Since regular expressions are dependent on the operating system, why include it in REXX? It's not part of REXX itself.

REXX on UNIX ... what's the point?

S. Drake

Moderator's note: Sam's well-groomed slides appear separately in these proceedings.

Moderator's Note: The floor was opened for audience comment, and a lively discussion ensued. A splendid time was had by all.

REXX in UNIX

What's the Point?

Sam Drake

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

BITNET: DRAKE at ALMADEN
Internet: drake@almaden.ibm.com

REXX Symposium

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

My Predjudices

- Former "VM Bigot"
  - Used REXX as programming language, macro language under XEDIT and other programs
  - Couldn't survive without it
- Now an "AIX Bigot"
  - Tried to make a "clean break"
  - After four years, I still can't write a shell script
  - And I'm darned proud of it

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

UNIX state-of-the-art

- Two "classic" shell script languages
  - Bourne Shell, C-Shell
- One "classic" data manipulation language
  - AWK
- Two "modern" languages
  - Korn shell
  - Perl
- No unified macro languages

All are powerful, cryptic, unfriendly

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

Korn shell example

```
case $1 in
1)      # keep current dir
   print -r - "$PWD"
   return
   ;;
[2-9]|[1-9][0-9])
   n=x+${1}-1 type=2
   if ((type<3))
   then x=4
   fi
   ;;
*)
   #default
   ;;
esac
```


IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

Perl

- Relatively new language
- By Larry Wall
- Implementation, documentation publically available
- Combines:
  - Good interpreted shell script language
  - Good data manipulation language
  - Excellent access to native UNIX facilities

I can think in REXX and write PERL!!!!!!!!

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

Perl Example (in REXXish Style)

```
$name = "";
while (<>) {
   $line = $_;
   chop($line);
   @words = split($line);
   $lastword = $words[$#words];
   print "Last word = $lastword";
}
```

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

Why REXX in UNIX

- Existing shell script languages are very arcane
- Port existing REXX programs to UNIX
- Universal macro language ... *a REXX exclusive*

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

REXX Symposium

Issues with REXX in UNIX

- Existing shell languages are rich, powerful, universal
- REXX "looks foreign" in UNIX
  - The C heritage of UNIX pervades everything.
  - REXX doesn't look like C
- EXECCOMM ... difficult!?!?!?
- What should the default subcommand environment look like?
- Access to UNIX built-in features

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

Summary

REXX in UNIX can play two key roles:

- Portable, easy to use shell script language
- Common embedded macro language

There is stiff competition for the former.

REXX could dominate the latter.

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

ATTENDEES

Joe Beigel	FTD SCRI, W.P.A.F.B., Dayton, OH 45431
Betty Benson	2149 Ridge Avenue, Evanston, IL 60210 betty@nuacvm.acns.nwu.edu
Gurnie Bowden	2704 Loyola Lane, Austin, TX 78723
Bob Brooks	Box 3511 STNC, Ottawa, Ontario, Canada K1Y 4H7
Pat Buder	P.O. Box 3408, San Jose, CA 95156
Creswell Cole III	1230 East Argues Avenue, Sunnyvale, CA 95051
Larry Cook	Wells Fargo Bank, 201 Third Street, San Francisco, CA 94103
Mike Cowlishaw	IBM UK Labs, Husley Park, Winchester, UK SO21 2JN mfc@ibm.com
George Crane	SLAC, P.O. Box 4349, MS 97, Stanford, CA 94039 crane@slacvm.slac.stanford.edu
Cathie Dager	SLAC, P.O. Box 4349, MS 97, Stanford, CA 94039 cathie@slacvm.slac.stanford.edu
Charles Daney	19567 Dorchester Dr., Saratoga, CA 95070 cgd@well.sf.ca.us
Walt Daniels	Box 704, IBM Research, Yorktown Heights, NY 10598 dan@watson.ibm.com
Chip Davis	10420 Little Patuxent Pkwy., Columbia, MD 21044 chip.davis@amail.amdahl.com
Ken Down	500 Oracle Parkway, 40-876, Redwood Shores, CA 94065 kdown@oracle.com
Sam Drake	IBM, 650 Harry Road, San Jose, CA 95120 drake@ibm.com
Bob Flores	Room 6X08, CIA, Washington, D.C. 20505
Larry Garner	3225 Royal Court, Bedford, TX 76021 garner@msnvm3
Forrest Garnett	IBM, 5600 Cottle Road, San Jose, CA 95120 cmslives@ibm.com
Ann Getoor	8754 Crito Abrazo, La Jolla, CA 92037

Eric Giguere	University of Waterloo, Computer Systems Group Waterloo, Ontario, Canada N2L 361 giguere@csg.waterloo.ca
Suzanne Giguere	541 Woodstock Way, Santa Clara, CA 95054
Earl Goetz	877 Spinosa Drive, Sunnyvale, CA 94087 egoetze@svcs1
Dave Gomberg	7 Gateview Court, San Francisco, CA 94116 gomberg@ucsfvm.ucsf.edu
Linda Green	P.O. Box 6 G921 6C14, Endicott, NY 13760 greenls@gdlvm7.vnet.ibm.com
Rick Haeckel	650 Harry Road, San Jose, CA 95120 haeckel@ibm.com
John Hartmann	IBM FSC, 85 Nymollevej, DK-2800, Lyngby, Denmark john@cphvm1.vnet.ibm.com
Bill Hawes	533 Gleasondale Road, Stow, MA 01775 72230.267@compuserve.com
Jerry Hogsett	IBM, 1530 Page Mill Road, Palo Alto, CA 94304 hogsett@paloalto.vnet.ibm.com
Mare Vincent Irvin	Norden Systems Inc., Norden Place, Norwalk, CT 06906
Jonathan Jenkins	1250 E. Arques, M/S 201, Sunnyvale, CA 94088 jlj50@amail.ccc.amdahl.com
Clifford Johnson	1800 Alexander Bell Dr., Reston, VA 22091 76530.764@compuserve.com
Mike Kay	IBM, 650 Harry Road, San Jose, CA 95120 mhk@ibm.com
Larry Kellogg	IBM D 081, 10401 Fernwood Road, Bethesda, MD 20817 larryk@betasvm2
Peter Koletzke	400 W. 43rd St., 16N, New York, NY 10036
Jeff Lankford	Northrup, One Research Park, Palos Verdes, CA 90254 jlankford@nrtc.northrup.com
Linda Littleton	214 Computer Bldg., Penn State University University Park, PA 16802 lrl@psuvm.psu.com
Dennis Mar	Naval Postgraduate School, CODE 51, Monterey, CA 93943 2001p@navpgs.bitnet

Brian Marks	IBM UK Labs, Winchester, England marks@winvmd.vnet.ibm.com
Neil Milsted	iX Corporation, 575 W. Madison #3610, Chicago, IL 60606 nfnm@wrkgrp.com
Don Moldover	IBM D 081, 01401 Fernwood Rd., Bethesda, MD 20817 moldovr@betvtvm1
Bert Moser	IBM Lab Vienna, c/o IBM Austria Obere Donaustrasse 95, A-1020, Vienna, Austria moser@vabvm1.vnet.ibm.com
Stan Murawski	635 South 16th St., San Jose, CA 95112-2372
JoAnn Malina	SLAC, P.O. Box 4349, MS 97, Stanford, CA 94309 joann@slacvm.slac.stanford.edu
Rick McGuire	Rd. 1, Box 164P, Brackney, PA 18812 mcguire@gdlvm7.vnet.ibm.com
David Morris	IBM MS 36, P.O. Box 10500, Palo Alto, CA 94304 morris@paloalto.vnet.ibm.com
Simon Nash	IBM UK Lab Ltd., Sheridan House 41-43 Jewry Street, Winchester, England S023 8RY nash@winvmb.vnet.ibm.com
Scott Ophof	c/o Rte. 5, Box 418, Russelville, AR
Jerry Pendleton	4245 Technology, Fremont, CA 94538 jerryp@key.amdahl.com
Nell Perry	1250 East Arques M/S 201, Sunnyvale, CA 94088 n2p40@amail.ccc.amdahl.com
Bob Pesner	100 Beekman St., #9H, New York, NY 10038 pesner@well.sf.ca.us
Seth Ratliff	315 Franklin Avenue, Silver Spring, MD 20901 seth@betvtvm1
Sandy Rockowitz	1373 S. Mayfair Ave., Daly City, CA 94015 rock@slacvm.slac.stanford.edu
Tom Sheffield	499 Wolf Run Road, Lewisville, TX 75067
Mark Short	Park A, P.O. Box 9627, Kansas City, MO 64134
David Singer	IBM, 650 Harry Road, San Jose, CA 95120 singer@almaden.ibm.com

Michael Sinz	1200 Wilson Drive, West Chester, PA 19380 mks@cbmvax.commodore.com
Ed Spire	6300 River Road, Rosemont, IL 60018 ets@wrkgrp.com
Karl Swartz	SLAC, P.O. Box 4349, MS 97, Stanford, CA 94309 kls@slacvm.slac.stanford.edu
Alen Thew	Computer Law, Liverpool Univ. P.O. Box 147, Liverpool, UK L69 3BX qq11@liverpool.ac.uk
Keith Watts	1945 Washington #410, San Francisco, CA 94109 bix:kwatts
Jim Weissman	808 Coleman Avenue. #21, Menlo Park, CA 94025-2456 418-1671@mcimail.com
David Wescott	7009 Ansbrough Dr., Citrus Heights, CA 95621
Gary Wesley	3460 Hillview, Palo Alto, CA 94304
Bebo White	SLAC, P.O. Box 4349, MS 97, Stanford, CA 94309 bebo@slacvm.slac.stanford.edu
Dean Williams	8600 West Bryn Mawr, Chicago, IL 60631
Paul Wolberg	461 Glenmoor Road #8, East Lansing, MI 48823 21387P5W@ibm.cl.msu.edu
Rick Wyatt	655 South Fair Oaks, Sunnyvale, CA 94086
Pete Zybrick	20 Dogwood Trail, Kinnelon, NJ 07405

ANNOUNCING

The REXX Symposium for Developers and Users

Annapolis, MD
May 3–5, 1992

- Meet the developers of REXX implementations and products
- Learn tips and techniques on REXX usage
- Dine with REXX pioneers and enthusiasts
- Enjoy historic Annapolis

REXX classes at the introductory and advanced levels will be offered prior to the symposium.

For further information, or to participate as a speaker or panelist, contact:

Cathie Dager
(415) 926-2904
cathie@slacvm.slac.stanford.edu
FAX: (415) 926-3329

or

Bebo White
(415) 926-2907
bebo@slacvm.slac.stanford.edu
FAX: (415) 926-3329

For registration and travel arrangements contact:

Village Travel
(800) 245-3260
FAX (415) 326-0245