What's wrong with Rexx?

Walter Pachl, IBM Retiree

Last summer when I visited my former (temporary) working place, I learned about the forthcoming celebration of Rexx' 25th birthday here in Sindelfingen and Boeblingen. Months passed by - time flies for retirees - and then somebody sort of intrigued me to submit a proposal for a presentation at this event. I gave it some thoughts – retirees are very busy indeed - and came up with an idea just one day after the submission deadline.
Yes, I have been using Rexx for many years now and I am actually very happy doing so within my limits: Classic Rexx on the host (MVS/TSO) and my good old OS/2 at home. One of my favourite jobs in IBM was testing the Rexx Compiler. It was then when I learned to know Rexx inside out by studying its specifications and outside in by setting up an automated test environment. Self-checking test cases were generated using Rexx (the Interpreter being used as a yardstick) and run on demand or regularly.

In my second career, the programming department of an Austrian bank, I could luckily continue to use Rexx on MVS/TSO. I even found our Rexx compiler on the system. The work stations use Windows/NT and there I miss Rexx but found not enough reason to ask for it being installed.

In my using Rexx there has been a task that I have to do over and over again which is related to the

Novalue condition.

Rexx has a wonderful feature which allows you to trap references to variables that haven't been initialized or set before. Signal on Novalue is the magic key to this feature and the boring fact is that one has to explicitly enable this condition instead of it being the default for the experienced (and thus sloppy?) programmer. The boring task I am talking about is the framing of each and every Rexx program with lines like this:

```
/* REXX ***************************************************************
* Description of the program's purpose
* Change activity:
* 02.05.2004 Walter Pachl  new for the Rex Symposium 2004
*********************************************************************/
  Signal On Halt                     /* enable the conditions and  */
  Signal On Novalue                  /* point at reasonable        */
  Signal On Syntax                   /* condition handling         */

/* My program to be inserted here                                  */
```

```
/*********************************************************************
* Condition handling: pretty self-explanatory
*********************************************************************/
Novalue:
  Say 'Novalue raised in line' sigl    /* line in error  number      */
  Say sourceline(sigl)                 /*            and text        */
  Say 'Variable' condition('D')        /* the bad variable reference */
  Signal lookaround                    /* common interactive code    */

Syntax:
  Say 'Syntax raised in line' sigl     /* line in error  number      */
  Say sourceline(sigl)                 /*            and text        */
  Say 'rc='rc '('errortext(rc)')'      /* the error code and message */

halt:
lookaround:                            /* common interactive code    */
  If fore() Then Do                    /* when running in foreground */
    Say 'You can look around now.'     /* tell user what he can do   */
    Trace ?R                           /* start interactive trace    */
    Nop                                /* and cause the first prompt */
    End
  Exit 12                              /* exit with "bad" return code*/
```

To have this work, there must be an external function FORE that returns the value '1' when running in the foreground and '0' otherwise. When compiling the program one must use the SOURCELINE and the HALT compiler option.
On my OS/2, I have, of course, fore.cmd returning '1' in my path (I never run Rexx programs in OS/2 batch) and when I send such a little program to a friend, he or she will probably call me about the function FORE that they don't have on their computer.

SIGH - that was the background to the catchy title but there is some more to come.

First of all the reason that Novalue is NOT the default was explained to me many years ago: The casual user (not a programmer) should not be bothered with string quotes so that the Hello world program could be as simple as this:

```
/* REXX */
Say What is your name?
Pull name
Say Hello name
```

The fact that the question comes out in uppercase may be disturbing but is probably not noticed by our casual user.

Entering Walter will result in HELLO WALTER which is consistent with the casing of the question. If the user's name is Günther, the computer will display
HELLO GüNTHER.
Not quite perfect (to the professional programmer).

Phrasing the question in colloquial English as What's your name? (note the apostroph) leads, however, to the first error.
And Say Hello name! (with an exclamation mark) will give a surprising result.

Now that we have enabled the Novalue condition (or "turned on the condition trap" in Rexx nomenclature) the Bible (Mike Cowlishaw's book) tells us that this condition is raised if (I'd prefer WHEN here) a symbol (other than a constant symbol) is used as

- a term in an expression
- the name following the VAR sub-keyword of a PARSE instruction
- a variable reference in a parsing template, a PROCEDURE instruction, or a DROP instruction.

but does not have an assigned value.

When preparing this paper I was shocked by the third bullet above because I thought that I knew everything about (classic!) Rexx and this was news to me. What is actually meant, I think, is that Novalue is raised when a variable name is enclosed in parentheses in any of these uses thus serving as variable pattern or as a subsidiary list. I claim that this fact is not very clearly stated in the literature but I would hope that the ANSII Standard states this explicitly and so confirms my belief.

A bullet that is missing above is that variable references in tails of compound variables are NOT trapped when the variable used has no assiged value but the symbol's default value (text in uppercase) is used without mercy... An omission (I think) that was never fixed.

If we could turn back the clock or rather the calendar for 25 years, we could try to convince the author to modify the language specifications but with millions of lines of Rexx-code out there such endeavour is nigh impossible.

By the way: Perfectly running Rexx programs may cause error messages when being compiled or when used with Object Rexx because these implementations perform a complete syntax check and diagnose errors in program parts that would possibly never be executed -- oops -- encountered in practical program runs.

This would be the end of the 15 minute tutorial presentation that I suggested. The program committee has, however, scheduled me for a full hour's presentation and so I had to perform some additional research to find and voice some other wrongs of Rexx.

Are compound variables equivalent to arrays?

Yes, unless the variables making up the tail contain periods:

```
a=1.2; b=3  ; s.a.b=17
u=1  ; v=2.3; Say s.u.v /* is actually identical to s.a.b */
```

Comparisons may not yield the expected result:

- when the comparands are interpreted as numbers: 000000 = 000e00
- when the operands are equal numbers under the Numeric Digits setting
- when leading and trailing blanks are ignored

The remedy is to either use strict comparison (x == y) or to enforce string comparison by something like '$'x = '$'y

Oops--- another pitfall: the x is interpreted to make '$' a hex literal which is, of course invalid. So one has always to be careful when using the variable names we are used to from high school: a, b, x, y,

Some language features are rather hard to understand: I have never found a practical use of Numeric Fuzz and teaching how Parse works is not at all easy.

```
Exercise: a='abcdefghijk'
          Parse Var a 'def' x +2 y
What is the value of x and y?
```

In my archive I found a foil (dated 9/1987) that asked the ARB (Architecture Review Board) for resolution:

```
max(123456789.3,123456789.7) gave 123456789
```

This was meanwhile corrected to give 123456790.

```
Numeric Digits 5
x=10000.0-9000.51  /* gives 1000 - I expected 999 */
```

The Book describes that the operands are extended to digits+1 but doesn't mention that they may also be truncated. In this case 09000.51 is truncated to 09000.5. The subtraction gives 00999.5 which is rounded to 1000.
I would suggest that using more than one guard digit or not truncating the number would significantly reduce the number of surprising results.

roo, by the way, shows the results 123456789.7 and 999, respectively. (More on roo to come later)

A few other trouble spots:

- External Procedures cannot share variables with the caller or with each other.
- Special Variable sigl must be exposed when used in Procedure
- Code point | vs. ! (the not character I avoid by using <>
- Input/Output: Call lineout fid to close an input file :-((lineout for input?)

Host Programs (MVS/TSO)

When I moved from IBM to my recent employee, I was sort of a pioneer with using Rexx. Some colleagues had attended a Rexx course in the past but apparently that course had not included I/O and what use is a programming language unable to read and write data. Rexx was therefore used to some extent by the system programming people but not at all by the community of application programmers.

Since I came from an IBM development lab with access to a wealth of tools on VM the move to a rather naked MVS/TSO development environment left me with the desire for all the good EXECs and macros that I was used to. Over the first few months I then reinvented some wheels by implementing the most important of these tools in the new environment. The switch from XEDIT to the ISPF editor was particularly challenging.

The invocation of programs as TSO commandname instead of just entering the commandname as in VM is a nuisance one gets used to. (This corresponds to commandname on OS/2 vs. rexx commandname on Windows.)

As far as documentation of these tools is concerned, I continued the good old practice to show some explaining text when the user enters the command or edit macro with a question mark as argument. The command TSO GUT (we are a German environment) lets the user view a little documentation data set with a one-liner for every tool. This online help is an inexpensive way of helping the user to find his or her way to the available tools and to tell them how to use them.
My other obvious use of Rexx was in the area of data manipulation.
Two lessons I had to learn with respect to performance were that EXECIO and invocation of external procedures are extremely slow.

For EXECIO, I use, therefore, explicit buffering:

```
/* REXX ***************************************************************
* Demo of buffered Input/Output (when running on the host)
*********************************************************************/
g.0host=host()                          /* 1: running on the host     */
If g.0host Then Do                      /* running on the host        */
  i.=0                                  /* Input buffer is empty      */
  i=1                                   /* Index into input buffer    */
  o.0=0                                 /* Output buffer is empty     */
  If fore()=1 Then Do                   /* Running in foreground      */
    "ALLOC FI(IN) DA('CT.PACHL.INPUT') SHR REUSE"  /* input          */
    "ALLOC FI(OU) DA('CT.PACHL.OUTPUT') SHR REUSE" /* output         */
    End                               /* in batch we use DD-statements */
  End
Else Do                                 /* running on the PC (at home)*/
  fid='input.txt'                       /* input                      */
  oid='output.txt'                      /* output                     */
  'erase' oid                           /* erase the output file      */
  End
xff=copies('ff'x,100)                   /* end-of-file indication     */
Do Until inline=xff                     /* until the end of file      */
  inline=read()                         /* read an input line         */
  If inline<>xff Then Do                /* not at the end             */
    oline=process(inline)               /* process the line           */
    Call o oline                        /* write to output file       */
    End
  End

If g.0host Then Do                      /* running on the host        */
  'EXECIO 0 DISKR IN ( FINIS'           /* close the input file       */
  'EXECIO' o.0 'DISKW OU (STEM O. FINIS' /* write last buffer and     */
  'EXECIO 0 DISKR IN ( FINIS'           /* close the output file      */
  End
Else Do
  Call lineout fid                      /* close the input file       */
  Call lineout oid                      /* close the output file      */
  End

Exit pgmrc
```

```
read:
/*********************************************************************
* Input routine returns one line from input file
*********************************************************************/
If g.0host Then Do                      /* running on the host        */
  If i>=l.0 Then Do                     /* Index out of range         */
    'EXECIO 1000 DISKR IN ( STEM L.'    /* read up to 1000 lines      */
     If rc<>0 Then Do                   /* not enough records         */
      z=l.0+1                           /* add an end-of-file         */
      l.z=xff                           /* indicator.                 */
      l.0=z                             /* index of end-of-file       */
      End
    i=0                                 /* reset input index          */
    End
  i=i+1                                 /* increment input index      */
  Return l.i                            /* Return current line        */
  End
Else Do                                 /* running on the PC          */
  If lines(fid)>0 Then                  /* still data in input file   */
    Return linein(fid)                  /* Return next line           */
  Else                                  /* input file exhausted       */
    Return xff                          /* Return end-of-file         */
  End

o:
If g.0host Then Do                      /* running on the host        */
  If o.0>=1000 Then Do                  /* output buffer full         */
    'EXECIO' o.0 'DISKW OU (STEM O.'    /* empty the output buffer    */
    o.0=0                               /* reset output index         */
    End
  z=o.0+1                               /* Increment output index     */
  o.z=arg(1)                            /* place line in output buffer*/
  o.0=z                                 /* maintain buffer contents   */
  End
Else                                    /* running on the PC          */
  Call lineout oid,arg(1)               /* just write the line        */
Return

process: Return arg(1)
```

Subroutines such as pd2 (packed to decimal) I copy into the program that uses them. Forgetting to do so is punished by unbearably bad performance (and so quickly recognized). (I never used the compiler-supported /*%INCLUDE member */ directive because I run programs usually interpreted.)

The little use of Rexx was indicated by an incident when I returned to the bank from my first vacation: the Rexx compiler had been archived. A fate for all data sets that haven't been used for some time!

For some applications I even managed to access DB2 thanks to some SQL interface for Rexx that was on the system as part of some vendor's software package. When this ceased to work at some point in time I abandoned accessing DB2 from Rexx and turned to intermediate steps of unloading DB2 data to sequential data sets.

The main programming language in the bank being PL/I, I found some other uses for Rexx:

- A syntax-checking edit macro (PLICHK) that diagnoses errors not nicely diagnosed by the PL/I compiler.
- A casing tool: I hate to maintain code written in uppercase
- A generator for creating DB2 access modules based on the table definitions
- A point-and-shoot macro for displaying the contents of %INCLUDE members
- And a set of PL/I-procedures that implement some of the Rexx built-in functions such as C2X, WORDS, WORD, STRIP, etc.

Object Rexx for Windows

Two years ago I was invited to attend a lecture series given by Rony Flatscher at the University of Economics and Business Administration, Vienna, Austria, where he taught object oriented programming using OORexx.
I could follow the lecture, understood some principles, even found a little bug in the implementation (I don't remember what it was), but never became fluent with Object Rexx.
It's a little like swimming: I can do the breast stroke but will never get to butterfly or dolphin style. A few attempts to create some object oriented Rexx code (using predefined methods or methods of my own) usually ended in some disappointments caused by error messages about unknown methods caused by my sloppiness or by an incorrectly set up environment.
Still I use Object Rexx for my Classic Rexx programs. The upward compatibility is just wonderful. The one problem that still bothers me is that Interactive Trace does not work as expected when running a Rexx program in the Object Rexx workbench.

With Object Rexx for Windows there come a number of **function packages.**

RxMath implements a number of mathematical functions with a specified precision of up to 16 digits. Since I have programmed these functions in Rexx with specified arbitrary precision it was an easy task to test the functions in RxMath and luckily (?) I found another wrong: Most of the functions deviate in the last digit from my 'true' results in 30 % of the invocations.

Deviations in the last digit when using a precision of 16:

Function	too small	OK	too large	Percent inaccurate
sqrt	16	72	12	28.00
exp	19	167	15	16.92
log	87	47	66	76.50
log10	80	60	60	70.00
sinh	28	158	15	21.39
cosh	20	363	18	9.48
tanh	60	281	60	29.93
power	38	141	22	29.85
sin	24	105	22	30.46
cos	40	88	23	41.72
tan	26	102	23	32.45
pi	0	15	0	0.00
arcsin	10	74	17	26.73
arccos	10	74	17	26.73
arctan	24	105	22	30.46
Total	482	1852	392	32.06

Here's why I consider "my" values correct: one can see that the higher precision value is always correctly rounded to the lower precision. The trick is that for getting a certain precision I compute to a (much) higher precision and then round the over-precise result to the desired precision.

```
Precision sine(1,precision)
        5 0.84147
        6 0.841471
        7 0.8414710
        8 0.84147098
        9 0.841470985
       10 0.8414709848
       11 0.84147098481
       12 0.841470984808
       13 0.8414709848079
       14 0.84147098480790
       15 0.841470984807897
       16 0.8414709848078965
       17 0.84147098480789651
       18 0.841470984807896507
       19 0.8414709848078965067
```

Another minor problem with RxCalcSin is that it sometimes returns numbers in exponential representation when it should not:

```
   RxCalcSin(0.007,16) => 6.999942833473391e-003
              instead of 0.006999942833473392
```

Why the invocation of my pi(prec) function no longer works after loading the RxMath function package, I don't know.

And while at it: the definition of RxCalcSinH(number,prec) as "returning the hyperbolic sine of number, expressed in radians" appears to be a thoughtless copy from RxCalcSin ... where number is the angle size expressed in degree (D), radian (R), or grade (G) units.
I don't think that sinh has anything to do with radians, or does it.

This leads to the next observation: The syntax diagram for RxCalcSin should show the codes under quotes!

The order in the deviation table above is the one given in RxMath's documentation. I cannot see any logical reason for it.

Another high-precision implementation of the mathematical functions I found in the roo-package from Kilowatt Software (Keith Watts).
The testing method applied to RxMath gives comparable incorrectness for this implementation (sine(1) computed for precision 5 through 100):

```
 Precision  sine(1) under Numeric Digits p
              (no extra argument required)
          5  0.84147    "my" value
         >   0.84146    roo's value (when different)
          6  0.841471
          7  0.8414710
          8  0.84147098
          9  0.841470985
         >   0.841470984
         10  0.8414709848
         >   0.8414709847
         11  0.84147098481
         >   0.84147098480
…
         20  0.8414709848078965066525
         >   0.8414709848078965066526
         21  0.84147098480789650665
         >   0.84147098480789650666
         22  0.841470984807896506525
         >   0.841470984807896506526
-----------------------------------------------
        59  roo-values are too small
        24  roo-values are ok
        13  roo-values are too large
    -> 75 percent of the values are incorrect.
```

00**

While looking at mathematics I stumbled over another surprise:
0**0 gives 1 in all of IBM's Rexx implementations that I have access to. The aforementioned roo gives 0. When teaching elementary mathematics I tell my pupils that division by zero and 0**0 are undefined expressions. Therefore, I consider both results to be wrong.

PL/I, by the way, is well behaving. 0**0 results in this error message:

```
IBM0682I  ONCODE=1550  X in EXPONENT(X) was invalid.
   At offset +0000062C in procedure with entry T00
```

With respect to roo I can gladly report that Keith Watts fixed the problems within hours when I told him about them. sine gives the exact values and 0**0 raises the error condition (he chose error 26 invalid whole number). I dare not ask IBM for a comparably quick fix.
And Keith did not fix the arithmetic problems mentioned above – he claims that the specs are insufficient.

Back to Object Rexx for Windows:

There are a number of samples provided. Among them a class for **complex numbers**. While I understand the arithmetic operations that can be used with them, I had to study the implementation of integer division and remainder for two complex numbers. What the creator of these methods did is apparently

```
a+ib   (a*c-b*d) + i*(a*d+b*c)
---- = ----------------------
c+id         (c**2+d**2)
```

The classic integer division and remainder operators are now applied to the real and imaginary part of the numerator and denominator.

Does this make any sense?? I don't think so. In particular the equation

(u%v)*v + u%v

does not hold for the results of these operations as it does for classic Rexx.

Last but not least: What's really wrong with Rexx is its little use.
One should see many more uses of Rexx for

- personal computing,
- documenting algorithms,
- prototyping, and
- all the things presented at this symposium and the ones before and after it.

A generally available test suite would be useful to help all implementors with providing consistent functionality.

Let me close with three **aphorisms** of my own:

Every program contains at least 2 errors:
one anyway and where there is one error, there is a second one.

Programming needs 3 things:

- Some intelligence,
- much diligence,
- and lots of luck,

Data:
is wrong as soon as you look at it

Some URLs:

The Rexx Language Association:
http://www.rexxla.org/

Rony Flatscher's lectures notes and Symposium papers :
http://wi.wu-wien.ac.at/rgf/rexx/

Keith Watts' roo and other software
www.kilowattsoftware.com

Vladimir Zabrodsky's Album of Algorithms:
www.geocities.com/SiliconValley/Garage/3323/aat/

The author's email address:
pachl@chello.at