Advanced Rexx Topics

Phil Smith III
Voltage Security, Inc.
SHARE 111
August 2008
Session 8331

Copyright Information

Agenda

- Why Rexx?
- “Power” uses for Rexx
- Hygiene
- Robust Rexx
- Non-IBM Rexx Implementations
- Coding for portability
- Conclusions

Why Rexx?

- Extremely powerful
- Everyone with VM, TSO has Rexx
  - Available for diverse environments
- Lots of products support Rexx macros
- Easy to learn—similar to PL/I
  - Very “forgiving”, easy to debug
- Compiler improves performance, control

Few fully exploit Rexx capabilities!

Power Uses for Rexx

- System debugging (with privileges)
- System monitor
  - Summarize performance data
- Front-end to end-user commands
  - Simplify (or alter!) parsing
  - Set up default environment
- Application prototyping
  - Robust, sophisticated applications!
- Full applications
  - Entire vendor products are written in Rexx

Rexx Power Tools

- Logical variables
- PARSE
- Stemmed variables and arrays
- Nested function calls
- Interactive tracing
- SIGNAL ON
- Internal vs. external functions
- Special function uses
- Obscure functions
- Word manipulation functions
- INTERPRET and ADDRESS
- The stack
- DROPping variables

Logical Variables

- Logical variables have values of 1 or 0
- Are tested implicitly:

```
if logical_variable then ...
```

  - Faster to execute, easier to read than:

```
if non_logical_variable = 'YES' then …
```

- All comparisons resolve to logical values

Logical Variables

- Exploit for efficiency, readability:
  - Return 1 if rc is 3, else return 0:

```
return (rc = 3)
```

  - Return 0 if rc is 0 or rc is 3:

```
return (rc <> 3) * rc
```

  - Assign to variables:

```
var = (a = 3)
```

  - Not:

```
if a = 3 then var = 1
else var = 0
```

Logical Variables

- Can use logical values in arithmetic
  - Example: determine length of year in days

```
days = 365 + (substr(date('O'), 1, 2)//4 = 0)
```

  - Versus:

```
if substr(date('O'), 1, 2)//4 = 0 then
days = 366
else days = 365
```

PARSE

- One of Rexx's most powerful features
  - Can parse variables, stacked data, function responses…
  - Can parse on blank-delimited tokens, specific strings, variable strings, absolute column positions, relative column positions
- PARSE ***usually*** faster than multiple SUBSTR
  - Possible exceptions include huge strings

PARSE Examples

- Parse date into month, day and year:

```
parse var date mm '/' dd '/' yy
```

or:

```
parse var date mm 3 . 4 dd 6 . 7 yy
```

not:

```
mm = substr(date, 1, 2)
dd = substr(date, 4, 2)
yy = substr(date, 7, 2)
```

PARSE Examples

- PARSE VALUE avoids intermediate variables:

```
parse value diag(8, 'QUERY FILES') with ,
   . rdrfiles . ',' .
```

not:

```
files = diag(8, 'QUERY FILES')
parse var files . rdrfiles ',' .
```

  - Of course, if `FILES` will be used later, intermediate variable better than multiple function calls

Stemmed Variables and Arrays

- Rexx vectors/arrays have **stem** and **tail**:

```
stem.tail = 27
```

  - `STEM.` is the stem, including the dot; `TAIL` is the tail
- May have non-numeric tails
  - Allows “associative memory”, relating variable structures by suffix value
  - Saves searching arrays: refer to compound name, Rexx finds it

Stemmed Variables and Arrays

- Easy to build and process list of values:
  - Build list of tails in variable (if simple keys)
  - Otherwise build numeric-tailed key list
  - Build arrays with tails as suffixes
  - Break down list of tails, process each value
- Can set default array variable values:

```
things. = 0       /* Set all counts to 0 */
…
things.user = things.user + 1   /* Next */
```

Compound Variable Examples

- Enables simple variable management:

```
parse var line user count comments
things.user = things.user + count
if find(users, user) = 0 then
  users = users user          /* New user */
```

- Scan array by parsing tail list:

```
users = users          /* Copy list of IDs */
do while users <> ''    /* Run the list */
   parse var users user users   /* Next */
   say user 'has' things.user 'things'
end
```

Nested Function Calls

- Nest function calls when appropriate:

```
last = right(strip(substr(curline.3,1,72)) 1)
```

  to get last non-blank byte of a line, not:

```
line = substr(curline.3, 1, 72)
line = strip(line)
last = right(line, 1)
```

  - Avoids useless intermediate variables
- Within limits, nesting cheaper, easier
  - Occasionally two steps more readable, maintainable—use judgment

Internal vs. External Functions

- Rexx supports internal and external user functions
  - Internal functions are subroutines in program
  - External functions are separate programs, called exactly like built-in functions
- External commands can examine/set Rexx variables
  - Search SHVBLOCK in *Rexx Reference* for details

Internal vs. External Functions

- PARSE SOURCE returns call type:
  - `COMMAND`, `FUNCTION`, `SUBROUTINE`
- PROCEDURE instruction in internal functions provides isolation of variables
  - `PROCEDURE EXPOSE` *`varlist`* allows limited variable sharing in internal functions
  - Can EXPOSE compound variable stem
  - Use compound variables to avoid long EXPOSE lists

Internal vs. External Functions

- External functions can simplify maintenance of mainline
- External functions must use GLOBALV command (or equivalent) to share variables
- Freeware GLOBALV implementations for TSO:
  - `www.wjensen.com` (Willy Jensen's page)
  - `www.btinternet.com/~ashleys/`
    (Ashley Street, FADH)
  - `www.searchengineconcepts.co.uk/mximvs/rexx-functions.shtml`
    (Rob Scott's STEMPUSH and STEMPULL)
- Tradeoff of function and performance vs. readability/maintainability/commonality

Function: TRANSLATE

- TRANSLATE does what it sounds like
- With no translation tables, TRANSLATE uppercases string:

```
mvariable = 'woof'
uvariable = translate(mvariable)
/* Now UVARIABLE = 'WOOF‘ */
```

- UPPER theologically incorrect
  - Not in Rexx language specification
  - Won’t go away, but not in some implementations

Function: TRANSLATE

- Also use TRANSLATE to rearrange strings
  - Input string, translate table specify old, new order:

```
d = '01/31/90'
d = translate('78612345', d, '12345678')
/* Now D = '90/01/31' */
```

  - Much faster, easier than multiple SUBSTR calls
  - May be harder to read, though!

Translate: How Does This Work??

```
date = '01/31/99'
date = translate('99/01/31', date, '12345678')
```

Key points:

- Strings must be unique (or expect confusion!)
- Evaluation is atomic, so:

  `translate('12', '2x', '12')`

  Evaluates to “2x”, not “xx”

Function: SPACE

- SPACE can remove blanks from a string
- For example, squeeze user input:

```
line = 'This is a line'
line = space(line, 0)     /* Lose any blanks */
/* Now LINE = 'Thisisaline' */
```

- Or (with TRANSLATE), remove “/” from date:

```
date = date('U') /* Assume today is 01/31/90 */
date = space(translate(date, ' ', '/'), 0)
/* Now DATE = '013190' */
```

- Remove commas from number:

```
n = '199,467,221'
n = space(translate(n, ' ', ','), 0)
/* Now NUMBER = '199467221' */
```

Obscure Functions

Rexx has many, many built-in functions

Often multiple ways to perform a given task

- **Know the functions, avoid reinventing the wheel**

ABBREV:
Test if token is keyword abbreviation

COMPARE:
Determine where two strings differ

COPIES:
Copy string specified # of times

DELSTR:
Delete part of a string

INSERT:
Insert a string into another

LASTPOS:
Determine last occurrence of string

RANDOM:
Generate pseudo-random number

REVERSE:
Reverse string

XRANGE:
Generate hex value range

Word Manipulation Functions

- CMS, TSO commands are blank-delimited words
- Many Rexx functions manipulate blank-delimited words:
  - SUBWORD, DELWORD, WORDINDEX, WORDLENGTH, WORDS, WORD, WORDPOS
  - Useful in building and processing commands

The INTERPRET Instruction

- Evaluates variables, then executes result
  - Adds extra level of interpretation
- Enables variables as compound variable stems (although VALUE does, too)
- Eschewed by Rexx aficionados except where absolutely required

The INTERPRET Instruction

- Enables powerful test program:

```
/* Rexx EXEC to execute from the terminal */
   signal DO_FOREVER     /* Skip error handler */
SYNTAX:                 /* Here if syntax error */
   say 'SYNTAX ERROR' rc': ' errortext(rc)
DO_FOREVER:                          /* Main loop */
   signal on SYNTAX     /* Catch syntax errors */
   do forever                /* Loop until EXIT */
      parse external cmd          /* Read a line */
      interpret cmd          /* Execute the line */
   end
```

- Invoke program, enter lines to be executed
  - Type EXIT to terminate

The ADDRESS Instruction

- Controls execution environment for non-Rexx commands
  - Can pass single command to another environment:

```
address tso
...some code...
address ispexec 'some command'
...some more code... /* ADDRESS TSO in effect */
```

- ADDRESS operand ***not*** normally interpreted:

```
address tso
```

  - Same result whether variable TSO set or not
  - Quotes add apparent significance, have no value
- You can ***force*** an operand interpretation:

```
address value tso   /* Use value of variable TSO */
address (tso)       /* Use value of variable TSO */
```

The ADDRESS Instruction

- With no operands, returns to previous environment:

```
address tso
...some code...
address    /* Back to environment before TSO */
```

  - But ***usually*** you know the previous environment
  - Better to be explicit—more readable/maintainable
- Null operand meaningful in some environments:

```
address '' /* Same as ADDRESS COMMAND in CMS */
```

  - As with omitted operand, confusing for readers—avoid

The Stack

- Rexx supports a data stack
  - Lines of data that can be “pulled” and “pushed”
  - Concept came from VM/CMS
- Many Rexx programs manipulate stack
- Programs should tolerate pre-stacked lines
  - Not just tolerate, but not consume inadvertently
  - Failure to do so causes breakage in nested calls
- “Leave the toys [lines] where you found them”

The Stack

- Commands exist to aid in stack management
- NEWSTACK/DELSTACK/QSTACK control stack isolation
  - Each stack is completely separate from others
- MAKEBUF/DROPBUF/QBUF/QELEM manage current stack
  - MAKEBUF creates “stack level”
  - DROPBUF deletes lines stacked in a stack level
  - QBUF returns number of stacks
  - QELEM returns number of lines in current stack
- Manage stack effectively and carefully

DROPping Variables

- DROP destroys variables
  - One or more variables can be DROPped per statement
  - DROP stem destroys array
- SYMBOL function returns LIT after variable DROPped
- Releases storage for compound variables
  - Useful for reinitializing in iterative routines
  - Can conserve storage in complex applications

Performance: Quoting Calls

- Built-in functions can be quoted:

```
line = 'SUBSTR'(line, 1, 8)
```

  - Avoids search for local function with same name
- Appears to offer improved performance
  - Untrue: function search parses entire program, builds function lookaside entry
  - Quoted calls do not generate lookaside
  - Repeated quoted calls require extra resources
  - Avoid even if only used once—little or no savings

Performance: Semicolons

- C, PL/I programmers find Rexx ";" statement delimiter familiar
  - Many programmers end all statements with ";"
  - Bad idea: generates null Rexx statement internally, requires additional processing

Hygiene

- Use PROCEDURE to isolate subroutines
- Always specify literal strings in quotes:
  - Know which commands are part of Rexx, specify non-Rexx commands in quotes
  - Specify quoted strings in capitals unless mixed-case desired
  - Specify interpreted function operands in quotes: `date('U')` not `date(u)`
  - Consider SIGNAL ON NOVALUE to enforce
- Good Rexx hygiene pays off in improved reliability, readability, maintainability

Debugging: Interactive Tracing

- Stop after each statement (*?* operand)
- Examine variables, change values
- Re-execute line with new values (= operand)
- Suppress command execution (*!* operand)
- Suppress tracing for *n* statements
- Trace without stopping for *n* statements

Debugging: Interactive Tracing

- Enabled by TRACE instruction in program
  - Subroutine tracing can be reset without affecting mainline
- In large programs, consider conditional tracing:

```
if (somecondition) then trace ?r
```

Debugging: SIGNAL ON

- SIGNAL ON similar to PL/I ON condition
  - Transfers control when condition raised
- Five conditions:
  1. ERROR: Trap non-zero RC
  2. FAILURE: Trap negative RC
  3. SYNTAX: Trap Rexx syntax errors
  4. HALT: Trap HI Immediate command (CMS, TSO)
  5. NOVALUE: Trap uninitialized variable reference
- Can specify label, or use default (condition name):

```
signal on syntax
signal on syntax name SomeLabel
```

Debugging: SIGNAL ON

- SIGNAL allows graceful recovery from errors
  - Special variable SIGL contains line number where execution interrupted
  - Use `SOURCELINE(SIGL)` to extract source of error
- For syntax errors, RC contains error code
  - Use `ERRORTEXT(RC)` to extract syntax error text
- CONDITION function adds details about condition
  - Variable name for `NOVALUE` conditions, for example

Using SIGNAL ON HALT

- Use SIGNAL ON HALT whenever there's cleanup to be done:

```
Halt:
   call Halted              /* Go close files, etc. */
   say 'We got halted!'                    /* Notify */
   call Quit 999            /* Go do normal cleanup */
```

Using SIGNAL ON SYNTAX

- In any “serious” program, prepare for the “impossible”:

```
Syntax:                  /* Syntax error: tell people */
   signal off syntax       /* Don't want to recurse */
   badline = sigl    /* Remember where it happened */
   emsgtxt = errortext(rc)                /* And why */
   sourcel = GetSourceLine(badline)  /* Get source */

/* Go allow debugging, say what happened */
   call DebugLoop 'SYNTAX ERROR' rc 'in' g._Efn':', ,
       emsgtxt, 'Error occurred in line' badline':', ,
       sourcel
   call Quit 20040                          /* And exit */
```

- Use variable names not used in rest of program…

Using SIGNAL ON NOVALUE

- Get error details, display on console

```
NoValue:          /* Undefined variable referenced */
   signal off novalue    /* Don't want to recurse */
   badline = sigl   /* Remember where it happened */
   var = condition('D')                 /* And why */
   sourcel = GetSourceLine(badline)
   call DebugLoop 'NOVALUE of "'var'" raised in', ,
      g._Execname, ,
     'Error occurred in line' badline':', sourcel
   call Quit 20040                     /* And exit */
```

GetSourceLine Function

- Get ***entire*** source line, even if continued

```
GetSourceLine: Procedure   /* Note no EXPOSE list! */
   arg l
   if sourceline(l) = '' then
   return '(compiled, no source available)'
   line = ''
   do forever
      temp = sourceline(l)
      line = line temp
      if right(strip(temp), 2) = '*/' then
      if pos('615c'x, temp) > 0 then
      parse var temp temp '615c'x
      if right(strip(temp), 1) <> ',' then
        return line
      l = l + 1
   end
```

DebugLoop: Interactive Debugging

- Example of common subroutine for interactive debugging:

```
DebugLoop:
   do DebugI = 1 to arg()
      say arg(DebugI)    /* Display the message(s) */
   end

/* Go into interactive trace */
   trace ?r
   do forever
      nop
   end
   call Quit 20040                /* And exit nicely */
```

Alternative: Rexx “Dump” & Traceback

- For end-users, interactive debugging is just confusing
  - A “dump” is much more useful
- Easy if Pipelines available (z/VM example):

```
'PIPE rexxvars | > DUMP FILE A'
say 'Dump is in DUMP FILE A'
```

- More difficult if no Pipelines
  - Could write program to traverse Rexx variable tree
  - Fetch SHVBLOCKs (referenced earlier)
- Force traceback, too

```
say 'Forcing traceback:'
signal off syntax    /* Should be off already */
signal value ''              /* Force traceback */
```

First Failure Data Capture in Rexx

- FFDC: IBM term for “Get enough data the first time”
  - “Reboot and see if it happens again” is ***not*** FFDC
- Rexx facilities enable FFDC
  - SIGNAL ON
  - CONDITION()
  - SOURCELINE()
  - Interactive debugging
- With these “smarts”, you can (sometimes) fix a problem and continue running
  - Especially valuable in servers or programs with significant startup cost

FFDC Example: SIGNAL ON SYNTAX

- Error trapped by `SIGNAL ON SYNTAX`:

```
bad
Entering interactive debug mode
SYNTAX ERROR 40 in BAD EXEC A2
Incorrect call to routine
Error occurred in line 2315:
yesterday = space(translate(date('O', date('B', ,
  yy'/'mm'/'dd, 'O') - 1, 'B'), ' ', '/'), 0)
+++ Interactive trace.  TRACE OFF to end debug,
  ENTER to continue. +++
```

FFDC Example: SIGNAL ON SYNTAX

- Now let's figure out what broke:

```
say yy
07
say mm
12
say dd
91
```

- Ah, we have a bad date (…or maybe it was December 7, 1991, or July 12, 1991, and we parsed it wrong?)
  - Enough information to at least start real debugging!
  - Obviously much more interesting examples will occur
  - Failing program might send email about the error as well or instead

FFDC Example: SIGNAL ON NOVALUE

```
whatever
WHATEVER running at 2:46 on Tuesday, July 31, 2007
NOVALUE of "N" raised in WHATEVER EXEC A1
Error occurred in line 39:
g._ConsoleToFor = substr(g._ConsoleToFor, 1, n)
+++ Interactive trace. TRACE OFF to end debug, ENTER to
   continue. +++
say g._consoletofor
TO PHSDEV   RDR DIST PHSDEV    FLASHC 000 DEST OFF
say n
N
```

- We now know which variable was not set!

Other Rexx Implementations

- Personal Rexx — Quercus (was Mansfield)
  - Rexx for PC-DOS/MS-DOS, OS/2
  - www.quercus-sys.com
- Regina — Open Source
  - Freeware Rexx for *ix, Windows, etc.
  - regina-rexx.sourceforge.net
- UniRexx — The Workstation Group
  - Rexx for VMS, UniXEDIT also available
  - www.wrkgrp.com
- Object Rexx
  - IBM product; recently Open Sourced
  - www.oorexx.org

uni-REXX

Portability

- Rexx is well-defined
  - Leads to consistency across implementations
  - Slight implementation differences make testing important
- Issues when writing portable programs:
  - System commands
  - Filenames
  - Utilities such as `EXECIO`, CMS Pipelines
- Use stream I/O functions instead for portability
  - Sometimes less convenient
  - Sometimes slower

Portability

- Use PARSE SOURCE to determine platform:

```
parse upper source g._Host . g._Efn g._Eft
...
/* Set the output fileid */
  select
   when g._Host = 'CMS' then file = 'MY FILE A'
   when g._Host = 'WIN32' then file =
  'C:\TEMP\MY.TXT'
   when g._Host = 'UNIX' then file = '/tmp/my.txt/'
   otherwise say 'Unsupported platform "'g._Host'"'
   exit 24
  end
...
if lineout(file) then … /* Write, handle errors */
```

Portability

- When performance critical, multipathing may be worthwhile:

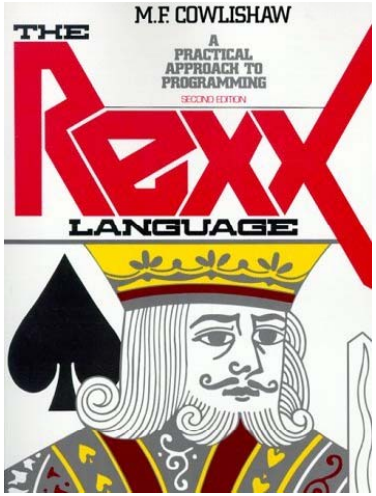
```
/* Process the xxx,xxx input variables */
select
   when g._Host = 'CMS' then
   'PIPE …'                    /* Do it in a Pipe */
   when g._Host = 'WIN32' then
   call TheHardWay /* Call pure Rexx subroutine */
   when g._Host = 'UNIX' then
   'awk …'                     /* Do it in awk */
```

Portability Example

- Tournament scheduler written in Rexx
  - OT: `www.firstlegoleague.org` — Check it out!
- Input parameters in flat file:
  - *somename* `FLL A` on CMS
  - *somename*`.fll` in current directory (or FQN) on Windows, Linux
- 1448 lines of Rexx
  - Five tests for host type, all related to file management
  - One of the five is just make a message environment-specific
- About as portable as anything! (Java, eat my…)
  - Doesn't handle blanks in directories; could if needed

Conclusions

- Rexx is powerful, rich in function
  - Offers more function than most people use
- Inexperienced users can add skills easily
  - Learning more about it increases productivity
- Once you feel proficient, read reference manual
  - Note unfamiliar facilities
  - Try them, discover their power
  - Experimenting is fun and easy!
- What Rexx tips can you share?



Questions?

Phil Smith III

(703) 476-4511

phil@voltage.com