REXX Language Coding Techniques

Tracy Dean, IBM
tld1@us.ibm.com
August 2015

Disclaimers

- **The information contained in this presentation is provided for informational purposes only.**

- **While efforts were made to verify the completeness and accuracy of the information contained in this presentation, it is provided "as is", without warranty of any kind, express or implied.**

- **In addition, this information is based on IBM's current product plans and strategy, which are subject to change by IBM without notice.**

- **IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this presentation or any other documentation.**

- **Nothing contained in this presentation is intended to, or shall have the effect of:**

  – Creating any warranty or representation from IBM (or its affiliates or its or their suppliers and/or licensors); or

  – Altering the terms and conditions of the applicable license agreement governing the use of IBM software.

Agenda

- **REXX products**

- **External environments and interfaces**

- **Key functions and instructions**

- **REXX data stack vs. compound variables**

- **I/O**

- **Troubleshooting**

- **Programming style and techniques**

- **REXX Enhancements (z/OS)**

- **Hands-on Lab Exercises**
  - REXX Lab: Parts 1 - 2
    - Tuesday, starting at 10:00am

REXX Language Coding Techniques

REXX Interpreter and Libraries

- **The Interpreter executes (interprets) REXX code "line by line"**
  - Included in all z/OS and z/VM releases
- **A REXX library is required to execute compiled programs**
  - Compiled REXX is <u>not</u> an LE language
- **Two REXX library choices:**
  - (Runtime) Library – a priced IBM product
  - Alternate library – a free IBM download
    - Uses the native system's REXX interpreter
- **At execution, compiled REXX will use whichever library is available:**
  - (Runtime) Library
  - Alternate Library

The REXX Products

- **IBM Compiler for REXX on zSeries Release 4**

  – z/VM, z/OS: product number 5695-013

- **IBM Library for REXX on zSeries Release 4**

  – z/VM, z/OS: product number 5695-014

- **z/VSE**

  – Part of operating system

- **IBM Alternate Library for REXX on zSeries Release 4**

  – Included in z/OS base operating system (V1.9 and later)

  – Free download for z/VM (and z/OS)

    http://www.ibm.com/software/awdtools/rexx/rexxzseries/altlibrary.html

- **REXX Interpreter**

  – Included in all z/OS and z/VM releases

Why Use a REXX Compiler?

- **Program performance**
  - Known value propagation
  - Assign constants at compile time
  - Common sub-expression elimination
  - stem.i processing
- **Source code protection**
  - Source code not in deliverables
- **Improved productivity and quality**
  - Syntax checks all code statements
  - Source and cross reference listings
- **Compiler control directives**
  - %include, %page, %copyright, %stub, %sysdate, %systime, %testhalt

REXX External Environments

External Environments

- **ADDRESS instruction is used to define the external environment to receive host commands**

  – For example, to set TSO/E as the environment to receive commands

  ```
  ADDRESS TSO
  ```

- **Several host command environments available in z/OS**

- **A few host command environments available in z/VM**

REXX Language Coding Techniques

Host Command Environments in z/OS

- TSO
  - Used to run TSO/E commands like ALLOCATE and TRANSMIT
  - Only available to REXX running in a TSO/E address space
  - The default environment in a TSO/E address space
  - Example:

    `Address TSO "ALLOC FI(INDD) DA('USERID.SOURCE') SHR"`

- MVS
  - Use to run a subset of TSO/E commands like EXECIO and MAKEBUF
  - The default environment in a non-TSO/E address space
  - Example:

    `Address MVS "EXECIO * DISKR MYINDD (FINIS STEM MYVAR"`

Host Command Environments in z/OS

- ISPEXEC
  - Used to invoke ISPF services like DISPLAY and SELECT
  - Only available to REXX running in ISPF
  - Example:
    ```
    Address ISPEXEC "DISPLAY PANEL(APANEL)"
    ```

- ISREDIT
  - Used to invoke ISPF edit macro commands like FIND and DELETE
  - Only available to REXX running in an ISPF edit session
  - Example:
    ```
    Address ISREDIT "DELETE .ZFIRST .ZLAST"
    ```

- Many more!

Host Command Environments in z/OS ...

– CONSOLE

– LINK, LINKMVS, LINKPGM, ATTACH, ATTCHMVS, ATTCHPGM

– SYSCALL

– SDSF

– DSNREXX

REXX Language Coding Techniques

Host Command Environments in z/OS ...

– CONSOLE

- Used to invoke MVS system and subsystem commands
- Only available to REXX running in a TSO/E address space
- Requires an extended MCS console session
- Requires CONSOLE command authority
- Example:

```
"CONSOLE ACTIVATE"
Address Console "D A"  /* Display system activity */
"CONSOLE DEACTIVATE"
```

Result:

```
IEE114I 04.50.01 2011.173 ACTIVITY 602
 JOBS      M/S      TS USERS     SYSAS     INITS    ACTIVE/MAX VTAM    OAS
00002     00014      00002       00032     00005     00001/00020      00010
```

REXX Language Coding Techniques

Host Command Environments in z/OS ...

- **LINK, LINKMVS, LINKPGM, ATTACH, ATTCHMVS, ATTCHPGM**
  - Host command environments for linking to and attaching unauthorized programs
  - Available to REXX running in any address space
  - LINK & ATTACH – can pass one character string to program
  - LINKMVS & ATTCHMVS – pass multiple parameters; half-word length field precedes each parameter value
  - LINKPGM & ATTCHPGM – pass multiple parameters; no half-word length field
  - Example:
    ```
    "FREE FI(SYSOUT SORTIN SORTOUT SYSIN)"
    "ALLOC FI(SYSOUT)   DA(*)"
    "ALLOC FI(SORTIN)   DA('VANDYKE.SORTIN') REUSE"
    "ALLOC FI(SORTOUT)  DA('VANDYKE.SORTOUT') REUSE"
    "ALLOC FI(SYSIN)    DA('VANDYKE.SORT.STMTS') SHR REUSE"
    sortparm = "EQUALS"
    Address LINKMVS "SORT sortparm"
    ```

Host Command Environments in z/OS ...

– SYSCALL
  - Used to invoke interfaces to z/OS UNIX callable services
  - The default environment for REXX run from the z/OS UNIX file system
  - Use syscalls('ON') function to establish the SYSCALL host environment for a REXX run from TSO/E or MVS batch
  - Example:

```
Call Syscalls 'ON'
Address Syscall 'readdir / root.'
Do i=1 to root.0
 Say root.i
End
```

Result:

```
…
bin
dev
etc
…
```

Host Command Environments in z/OS ...

– SDSF

- Used to invoke interfaces to SDSF panels and panel actions
- Use isfcalls('ON') function to establish the SDSF host environment
- Use the ISFEXEC host command to access an SDSF panel
- Panel fields returned in stem variables
- Use the ISFACT host command to take an action or modify a job value

- Example:

```
rc=ISFCalls("ON")
Address SDSF "ISFEXEC ST"
Do ix = 1 to JNAME.0
  If Pos("MYREXX",JNAME.ix) = 1 Then
  Do
    say "Cancelling job ID" JOBID.ix "for MYREXX"
    Address SDSF "ISFACT ST TOKEN('"TOKEN.ix"') PARM(NP P)"
  End
End
rc=ISFCalls("OFF")
Exit
```

Host Command Environments in z/OS ...

  – DSNREXX

  • Provides access to DB2 application programming interfaces from REXX
  • Any SQL command can be executed from REXX
    – Only dynamic SQL supported from REXX
  • Use RXSUBCOM to make DSNREXX host environment available
  • Must CONNECT to required DB2 subsystem
  • Can call SQL Stored Procedures

  • Example:
```
RXSUBCOM('ADD','DSNREXX','DSNREXX')
subSys = 'DB2PRD'
Address DSNREXX "CONNECT" subsys
owner = 'PRODTBL'
recordkey = 'ROW2DEL'
sql_stmt = "DELETE * FROM" owner".MYTABLE" ,
           "WHERE TBLKEY = '"recordkey"'"
Address DSNREXX "EXECSQL EXECUTE IMMEDIATE" sql_stmt
Address DSNREXX "DISCONNECT"
```

REXX Language Coding Techniques

Other External Environments in z/OS

- **IPCS**

  - Used to invoke IPCS subcommands from REXX

  - Only available when run from in an IPCS session

- **CPICOMM, LU62, and APPCMVS**

  - Supports the writing of APPC/MVS transaction programs (TPs) in REXX

  - Programs can communicate using SAA common programming interface (CPI) communications calls and APPC/MVS calls

Other "Environments" and Interfaces in z/OS

- **System REXX**
  - A function package that allows REXX EXECs to be executed outside of conventional TSO/E and Batch environments
  - Can be invoked using assembler macro interface AXREXX or through an operator command
  - Easy way for Web Based Servers to run commands/functions and get back pertinent details
  - EXEC runs in problem state, key 8, in an APF authorized address space under the MASTER subsystem
  - Two modes of execution
    - TSO=NO   runs in MVS host environment
                address space shared with up to 64 other EXECs
                limited data set support
    - TSO=YES   runs isolated in a single address space
                can safely allocate data sets
                does not support all TSO functionality

Other "Environments" and Interfaces . . .

- **RACF Interfaces**

  - IRRXUTIL

    - REXX interface to R_admin callable service (IRRSEQ00) extract request
    - Stores output from extract request in a set of stem variables

    ```
    myrc=IRRXUTIL("EXTRACT","FACILITY","BPX.DAEMON","RACF","","FALSE")
      Say "Profile name: "||RACF.profile
      Do a=1 to RACF.BASE.ACLCNT.REPEATCOUNT
        Say " "||RACF.BASE.ACLID.a||":"||RACF.BASE.ACLACS.a
      End
    ```

  - RACVAR function

    - Provides information from the ACEE about the running user
    - Arguments: USERID, GROUPID, SECLABEL, ACEESTAT

    ```
    If racvar('ACEESTAT') <> 'NO ACEE' Then
        Say "You are connected to group " racvar('GROUPID')"."
    ```

Other "Environments" and Interfaces . . .

- **Other ISPF Interfaces**

  - Panel REXX
    - Allows REXX to be run in a panel procedure
    - *REXX statement used to invoke it
    - REXX can be coded directly in the procedure or taken from a SYSEXEC or SYSPROC DD member
    - REXX can modify the values of ISPF variables

  - File Tailoring Skeleton REXX
    - Allows REXX to be run in a skeleton
    -)REXX control statement used to invoke it
    - REXX can be coded directly in the procedure or taken from a SYSEXEC or SYSPROC DD member
    - REXX can modify the values of ISPF variables

Host Command Environments in z/VM

- **CMS (default)**
  - Commands treated as if entered on the CMS command line
    - Translation of parameter list
      - Uppercasing and tokenizing
  - Same search order as CMS command line

- **COMMAND**
  - Basic CMS CMSCALL command resolution
    - No translation of parameter list
      - No uppercasing of tokenized parameter lists
    - To call an EXEC, prefix the command with the word EXEC
    - To send a command to CP, use the prefix CP

- **CPICOMM, CPIRR, OPENVM**

- **Generally, best practice is to use "Address Command" at the top of REXX EXECs that will be run in CMS environment**

Key Instructions and Functions

Instructions vs. Functions

- **Keyword instruction**
  - One or more clauses
  - First word is a keyword that identifies the instruction
    ```
    ARG, DO, IF, PARSE, …
    ```
- **Instruction**
  - Statement that performs an assignment of a value to a variable
    ```
    counter = 1
    ```
- **Function**
  - Must return a single result string (i.e. must be on the right side of an equal sign)
  - Built-in - provided as part of the REXX language
  - Internal - create your own
  - External – create your own or use platform unique functions
- **Subroutine**
  - Called like a function, but may not return data

Key Instructions – PARSE

- **PARSE**
  - Allows the use of a template to split a source string into multiple components
  - Syntax:

```
>>-PARSE--+-------+--+-ARG---------------------+-----------> 
          '-UPPER-'  +-EXTERNAL----------------+
                     +-NUMERIC-----------------+
                     +-PULL--------------------+
                     +-SOURCE------------------+
                     +-VALUE-+-----------+-WITH-+
                     |       '-expression-'     |
                     +-VAR--name---------------+
                     '-VERSION-----------------'


>--+---------------+--;------------------------------------><
   '-template_list-'
```

- **Short forms to some of these commands exist**
  - **NOT RECOMMENDED**
  - But you may see them in another user's code you must maintain
    - ARG
      - Short form for PARSE UPPER ARG
    - PULL
      - Short form for PARSE UPPER PULL

PARSE Templates

- **Simple template**
  - Divides the source string into **blank-delimited** words and assigns them to the variables named in the template
    - The last variable gets the rest of the string exactly as entered

```
datastring = '  Write the    blank-delimited   string  '
Parse Var datastring firstvar 2ndvar 3rdvar 4thvar

firstvar   -> 'Write'
2ndvar     -> 'the'
3rdvar     -> 'blank-delimited'
4thvar     -> '   string  '
```

PARSE Templates

- **Simple template**
  - A period is a placeholder in a template
    - A "dummy" variable used to collect unwanted data
    - Notice the double quotes so the single quote is recognized as part of the string

    ```
    datastring = "Last one gets what's left"
    Parse Var datastring firstvar . 2ndvar

    firstvar -> "Last"
    2ndvar -> "gets what's left"
    ```

    - Often used at the end of PARSE statement to take "the rest of the data"

    ```
    datastring = "Last one gets what's left"
    Parse Var datastring firstvar 2ndvar .

    firstvar -> "Last"
    2ndvar -> "one"
    ```

    - Causes the last variable to get the last word without leading and trailing blanks

    ```
    datastring = '  Write the    blank-delimited   string  '
    Parse Var datastring firstvar 2ndvar 3rdvar 4thvar .
    firstvar -> 'Write'
    2ndvar -> 'the'
    3rdvar -> 'blank-delimited'
    4thvar -> 'string'
    ```

PARSE Templates . . .

- **String pattern template**
  - A **literal or variable string pattern** indicating where the source string should be split
  - Assumes blank-delimited if no other pattern specified

```
datastring = '  Write the   blank-delimited   string '
```

Literal:

```
Parse Var datastring firstvar '-' 2ndvar .
```

String delimited

Blank delimited

Variable:

```
dlm = '-'
Parse Var datastring firstvar (dlm) 2ndvar .
```

Result (the same in both cases):

```
firstvar -> '  Write the   blank'
2ndvar -> 'delimited'
```

PARSE Templates . . .

- **Positional pattern template**
  - Use numeric values to identify the **character positions** at which to split data in the source string
  - An <u>absolute</u> positional pattern is a number or a number preceded by an equal sign

```
                    ----+----1----+----2----+----3----+----4----+
Datastring = 'Cowlishaw           Mike            UK          '
Parse Var datastring =1 surname =20 chrname =35 country =46 .

surname -> 'Cowlishaw        '
chrname -> 'Mike           '
country -> 'UK          '
```

  - A <u>relative</u> positional pattern is a number preceded by a plus or minus sign
    - Plus or minus indicates movement right or left, respectively, from the last match

```
                   ----+----1----+----2----+----3----+----4----+
datastring = 'Cowlishaw           Mike            UK          '
Parse Var datastring =1 surname +19 chrname +15 country +11 .

surname -> 'Cowlishaw        '
chrname -> 'Mike          '
country -> 'UK           '
```

REXX Language Coding Techniques

PARSE Templates . . .

- **Positional pattern template – removing blanks**
  - Specify an <u>absolute</u> positional pattern
  - Insert periods to strip blanks

```
              ----+----1----+----2----+----3----+----4----+
datastring = 'Cowlishaw          Mike           UK           '
Parse Var datastring =1 surname . =20 chrname . =35 country .

surname -> 'Cowlishaw'
chrname -> 'Mike'
country -> 'UK'
```

  If data starts in column 1 and is blank-delimited, this is the same as
  `Parse Var datastring surname chrname country`

  - **Warning –** won't work if any of the data has more than one "word"

```
              ----+----1----+----2----+----3----+----4----+
datastring = 'Cowlishaw, Jr.     Mike           UK           '
Parse Var datastring =1 surname . =20 chrname . =35 country .

surname -> 'Cowlishaw,'
chrname -> 'Mike'
country -> 'UK'
```
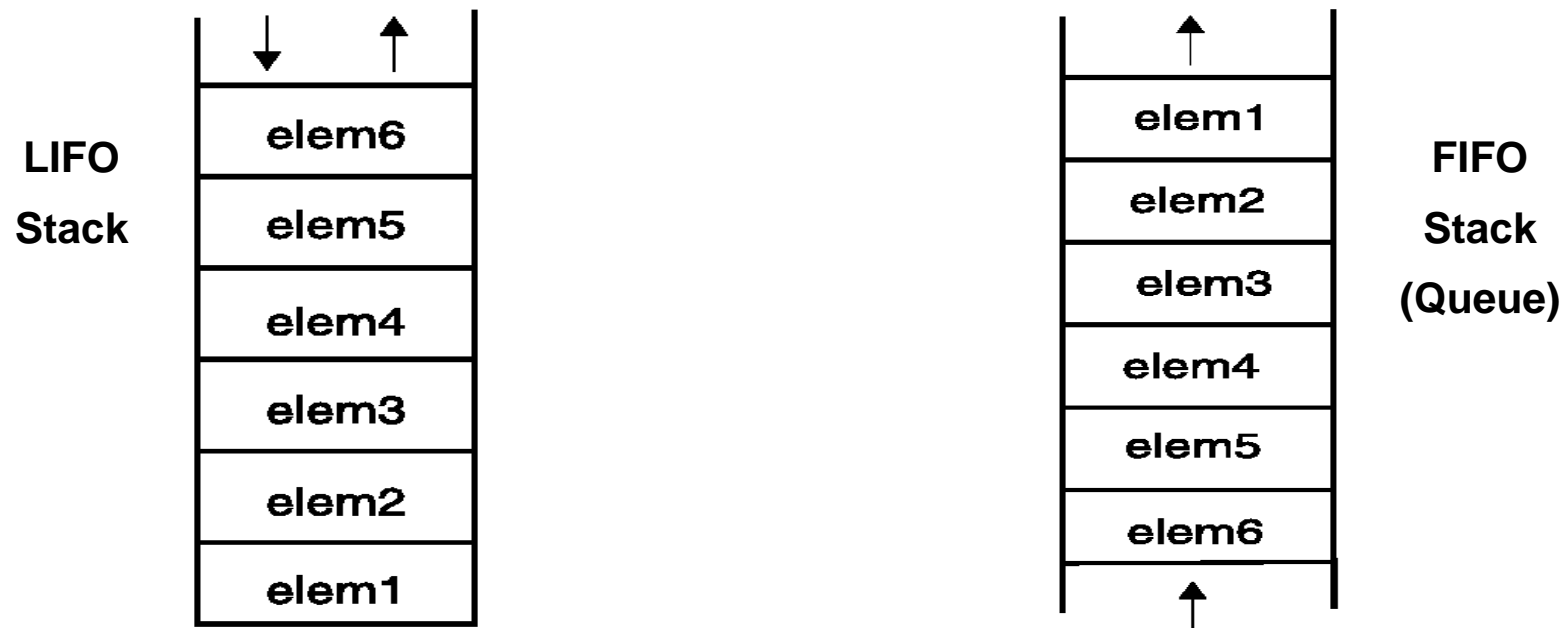
Blank delimited

Data Stack and Compound Variables

What is a Data Stack?

- **An expandable data structure used to temporarily hold data items (elements) until needed**

- **When an element is needed it is <u>always</u> removed from the <u>top</u> of the stack**

- **A new element can be added either to the top (LIFO) or the bottom (FIFO) of the stack**

  - FIFO stack is often called a queue

LIFO

Stack

elem6
elem5
elem4
elem3
elem2
elem1

elem1
elem2
elem3
elem4
elem5
elem6

FIFO

Stack

(Queue)

Manipulating the Data Stack

- **3 basic REXX instructions**
  - PUSH - put one element on the <u>top</u> of the stack

    ```
    elemone = 'new top element'
    Push elemone
    ```

  - QUEUE - put one element on the <u>bottom</u> of the stack

    ```
    elemtwo = 'new bottom element'
    Queue elemtwo
    ```

  - PARSE PULL - remove an element from the (top) of the stack

    ```
    Parse Pull elemthree
    ```

  - Result:

    ```
    elemthree → 'new top element'
    ```

- **1 REXX function**
  - QUEUED() - returns the number of elements in the stack

    ```
    num_elems = Queued()
    ```

REXX Language Coding Techniques

Why Use the Data Stack?

- **To store a large number of data items of virtually unlimited size for later use**

- **Pass a large or unknown number of arguments between EXECs or routines**

- **Specify commands to be run when the EXEC ends**
  - Elements left on the data stack when an EXEC ends are treated as commands

  ```
  Queue "TSOLIB RESET QUIET"
  Queue "ALLOC FI(ISPLLIB) DA('ISP.SISPLOAD' 'SYS1.DFQLLIB') SHR REUSE"
  Queue "TSOLIB ACTIVATE FILE(ISPLLIB) QUIET"
  Queue "ISPF"
  ```

- **Pass responses to an interactive command that runs when the EXEC ends**
  - Example: z/VM DDR program

Quick Example of Processing the Data Stack

- **Receiving (or called) program processing data on the stack**

  – Passed from sending/calling program

```
/* Sample stack processing */
Address Command
element.0 = Queued()
Do i = 1 To element.0
  Parse Pull element.i          Full parsing capability
  …
End
…
```

More Stack Functions and Options

- **Buffers**

- **Additional stacks**

- **Some functions are z/OS only**

- **See more details in the handout**

Using Buffers in the Data Stack

- **An EXEC can create a buffer in a data stack using the MAKEBUF command**

- **All elements added after a MAKEBUF command are placed in the new buffer**
  - MAKEBUF basically changes where the QUEUE instruction inserts new elements
    - Remember QUEUE inserts at the "bottom" of the stack (or buffer)
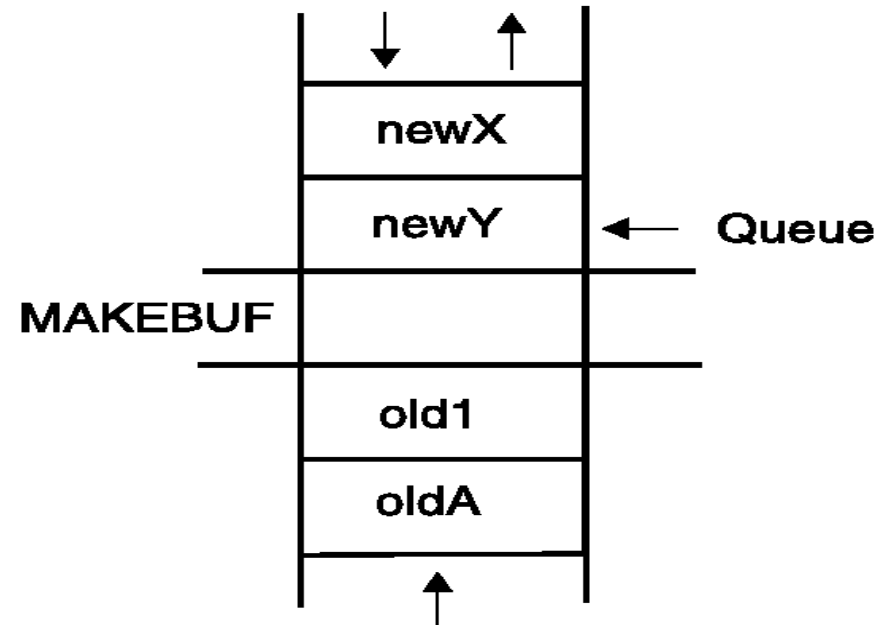
Using Buffers in the Data Stack . . .

- **An EXEC can use MAKEBUF to create multiple buffers in the data stack**
  - MAKEBUF returns in the RC variable the number identifying the newly created buffer
- **DROPBUF command is used to remove a buffer from the data stack**
  - Allows an EXEC to easily remove temporary storage assigned to the data stack
  - A buffer number can be specified with DROPBUF to identify the buffer to remove
    - Default is to remove the most recently created buffer
  - DROPBUF 0 results in an empty data stack (use with caution)
- **z/OS only**
  - The QBUF command is used to find out how many buffers have been created
  - The QELEM command is used to find out the number of elements in the most recently created buffer

Using Buffers in the Data Stack . . .

- **Important notes**
  - When an element is removed from an empty buffer, the buffer <u>disappears with no error or indication</u>
  - Keep track of where you are in buffers within the stack
  - Used QUEUED() to keep track of the total number of elements in the stack
  - To remove a buffer
    - Issue DROPBUF (the recommended approach)
      or
    - Remove an element (via PARSE PULL) when the buffer is already empty
  - The next request to remove an element will move
    - To the next buffer if there is one (including buffer 0)
    - To the external input queue if the stack (all buffers) are empty

REXX Language Coding Techniques

Protecting Elements in the Data Stack – z/OS Only

- **REXX code can use the stack, but protect itself from inadvertently removing someone else's data stack elements**
  - Create a new private data stack using the NEWSTACK command

- **All elements added after a NEWSTACK command are placed in the new data stack**
  - Elements on the original data stack cannot be accessed by an EXEC or any called routines until the new stack is <u>removed</u> (not just emptied)
  - When there are no more elements in the new data stack, information is taken from the terminal (<u>not the original data stack</u>)

Original Stack: old1 / oldA

New Stack: newX / newY
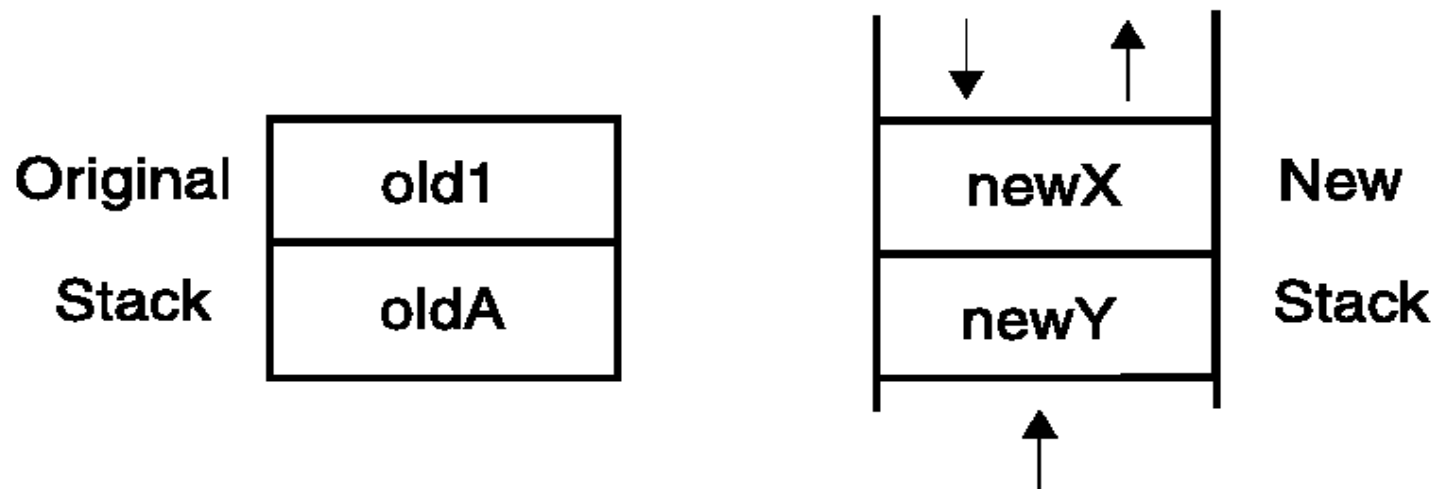
Protecting Elements in the Data Stack - z/OS Only

- **DELSTACK - removes a data stack**

  - Removes the most recently created data stack

    - Including all remaining elements in the stack
  - <u>CAUTION</u>

    - If no stack previously created with NEWSTACK, then DELSTACK removes all the elements from the <u>original stack</u>

- **QSTACK - returns the number of data stacks**

  - Including the original stack

  - Puts the value in the variable RC

- **<u>NOTE</u>: For z/OS, the QUEUED() function returns the number of elements in the <u>current</u> data stack.**

Data Stack vs Buffers

- **Data Stack**

  - Advantages

    - Protects data in the original stack
      - Never defaults back to the "previous" stack in the chain
      - Must specifically delete current stack to move to previous stack
      - Can easily request terminal input if also have items in the stack
        - > Just create a new stack with nothing on it and issue "Pull"

  - Disadvantages

    - Only available on z/OS
      - z/VM must issue "Parse External" to request terminal input if data is in the stack

REXX Language Coding Techniques

Data Stack vs Buffers

- **Buffers**

  - Advantages

    - Create a stack on top of the existing stack for new list of items

    - Use "QElem" (z/OS only) to keep track of how many items in this buffer

  - Disadvantages

    - No guaranteed protection of previous stack in the chain
      - If current stack is empty, will proceed to next one automatically

REXX Language Coding Techniques

What is a Compound Variable?

- **A way to reference a collection of related values**
    - Also called a *stem variable* or *stem array*
- **Variable name is *stem* followed by zero or more *tail*s**
    - *stem* must be simple variable ending in a period
    - *tail* must be simple variable or decimal integer
    - Multiple *tail*s are separated by periods
- **Each *tail* variable is replaced by its value**
    - Default value of *stem* and *tail* is the variable names used for *stem* and *tail*
    - Each *tail* references a dimension of the collection
- **The resulting *derived name* is used to access a specific value from the collection**
- **Tails which are variables are replaced by their respective values**
    - If no value assigned, takes on the uppercase value of the variable name

```
day.1                                    stem:      DAY.
                                         tail:      1

array.i                                  stem:      ARRAY.
                                         tail:      I

name = 'Smith'
phone = 12345

employee.name.phone                      stem:      EMPLOYEE.
                                         tail:      Smith.12345
```

Compound Variable Values

- **Initializing a stem to some value automatically initializes every compound variable with the same stem to the same value**

```
Say month.15    ──────▶   MONTH.15
month. = 'Unknown'
month.6 = 'June'
month.3 = 'March'

Say month.15        ──────▶     Unknown
monthnum = 3
Say month.monthnum  ──────▶     March
```

- **Easy way to reset the values of compound variables**

```
month. = ''
Say month.6    ──────▶    ''
```

- **DROP instruction can be used to restore compound variables to their uninitialized state**

```
Drop month.
Say month.6    ──────▶   MONTH.6
```

REXX Language Coding Techniques

Processing Compound Variables

- **Compound variables provide the ability to process one-dimensional arrays**
  - Use a numeric value for the tail
  - <u>Good practice</u> - store the number of array entries in the compound variable with a tail of 0 (zero)
  - Often processed in a DO loop using the tail as the loop control variable

```
invitee.0 = 10
Do i = 1 to invitee.0
   Say 'Enter the name for invitee' i
   Parse Pull invitee.i
End
```

- **Stems can be used with I/O functions to read data from and write data to a file on z/VM or data set on z/OS**
  - Stream I/O
  - EXECIO
  - PIPE

- **Stems can also be used with the external function OUTTRAP (z/OS) or PIPE (z/VM) to capture output from commands**

Processing Compound Variables . . .

- **The tail for a compound variable can be used as an index to related data**

- **The tail (index) and data can contain blanks**

- **Given the following input data:**

```
----+----1----+----2----+----3----+----4----+
Employee# Name          Location
A1234     M Cowlishaw   United Kingdom
B5678     T Dean        Portland
C9012     V Hein        Austin

...
```

- **The unique employee number value can be used as the tail of compound variables that hold the rest of the person's data**

```
'PIPE < EMPLOYEE INFO A | STEM rec.'
Do i = 2 To rec.0
  Parse Var rec.i =1 empnum name.empnum =25 location.empnum
End i
Say 'Which employee number do you want to learn about?'
Parse Upper Pull empnum
Say 'The name of employee' empnum 'is' Strip(name.empnum)'.'
Say 'The location of employee' empnum 'is' Strip(location.empnum)'.'
Exit
```

Data Stack vs Compound Variables

- **Data Stack**

  - Advantages

    - Can be used to pass data to external routines
    - Able to specify commands to be run when the EXEC ends
    - Can provide response(s) to an interactive command that runs when the EXEC ends

  - Disadvantages

    - Program logic required for stack management
    - Processing needs 2 steps
      - Take data from input source and store in stack
      - Read from stack into variables
    - Stack attributes and commands are OS dependent

Data Stack vs Compound Variables . . .

- **Compound Variables**
  - Advantages
    - Basically variables - REXX will manage them like other variables
    - Only one step required to assign a value
    - Provide opportunities for clever and imaginative processing
  - Disadvantages
    - Can not be used to pass data between external routines

- **Conclusion**
  - Try to use compound variables whenever appropriate
    - They are **<u>simpler</u>**

I/O and Troubleshooting

EXECIO Command – z/OS

- **A TSO/E REXX command that provides record-based processing**
  - Used to read and write records from/to a z/OS sequential data set or z/OS partitioned data set member
  - Requires a DDNAME to be specified
    - Use ALLOC command to allocate data set or member to a DD
- **Records can be read into or written from compound variables or the data stack**
- **Can also be used to:**
  - Open a data set without reading or writing any records
  - Empty a data set
  - Copy records from one data set to another
  - Add records to the end of a sequential data set
  - Update data in a data set, one record at a time

EXECIO Command – z/VM

- **CMS EXECIO command provides record-based processing**
  - Recommend using CMS Pipelines (PIPE command) instead
    - Simpler to use

      ```
      'EXECIO * DISKR EMPLOYEE INFO A (STEM REC. FINIS'
      vs
      'PIPE < EMPLOYEE INFO A | STEM rec.'
      ```

    - PIPEs has much more function

REXX Language Coding Techniques

REXX Stream I/O

- **Available natively in z/VM**
  - Again, learn PIPEs instead

- **Function package shipped with z/OS**
  - Also shipped with the IBM Library for REXX on zSeries
  - Allows REXX EXECs to use stream I/O functions to process sequential data sets and partitioned data set members

- **Process data**
  - Character by character, or
  - Line by line

- **Why use stream I/O?**
  - Extends and enhances I/O capabilities of REXX for TSO/E
  - A familiar I/O concept
  - Provides better portability of REXX between OS platforms

Troubleshooting – Condition Trapping

- **SIGNAL ON and CALL ON instructions can be used to trap exception conditions**

- **Syntax:**

```
►►──────SIGNAL ON ──┬─ERROR────┬── NAME labelname──►◄
                    ├─FAILURE──┤
                    ├─HALT─────┤
                    ├─NOVALUE──┤
                    └─SYNTAX───┘


►►──────CALL ON ──┬─ERROR────┬── NAME trapname──►◄
                  ├─FAILURE──┤
                  └─HALT─────┘
```
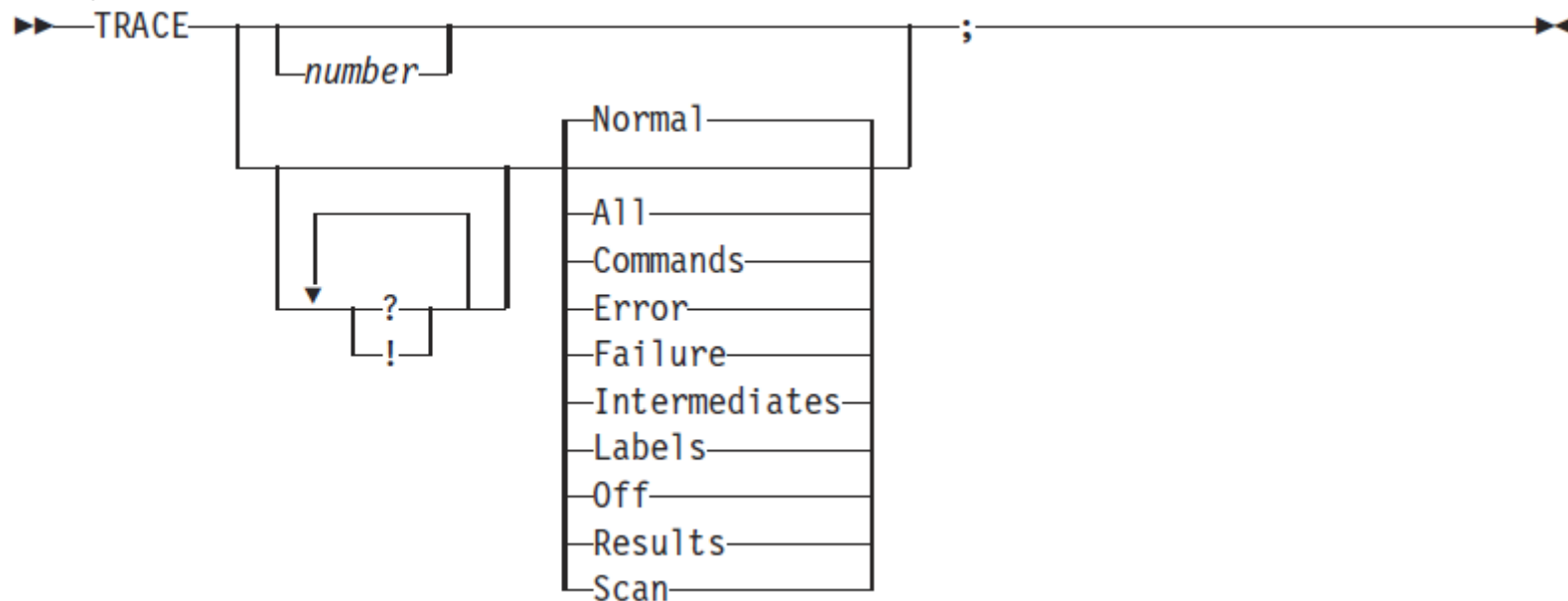
- **Condition types:**

  - ERROR — - error upon return (positive return code)
  - FAILURE — - failure upon return (negative return code)
  - HALT — - an external attempt was made to interrupt and end execution
  - NOVALUE — - attempt was made to use an uninitialized variable
  - SYNTAX — - language processing error found during execution
  - NOTREADY — - z/VM only. Error during input or output operation

Troubleshooting – Condition Trapping. . .

- **Good practice to enable condition handling to process unexpected errors**
  - Specifically `Signal On NoValue`
- **Use REXX provided functions and variables to identify and report on exceptions**
  - CONDITION function – returns information on the current condition
    - Name and description of the current condition
    - Indication of whether the condition was trapped by SIGNAL or CALL
    - Status of the current trapped condition

  - RC variable – return code
    - Contains the command return code for ERROR and FAILURE
    - Contains the syntax error number for SYNTAX

  - RESULT variable – value of an expression returned by a subroutine

  - SIGL variable – line number of the clause that caused the condition

  - ERRORTEXT function – returns REXX error message for a SYNTAX condition
    `Say ErrorText(rc)`

  - SOURCELINE function – returns a line of source from the REXX EXEC
    `Say Sourceline(sigl)`

REXX Language Coding Techniques

Troubleshooting – Trace Facility

- **Provides powerful debugging capabilities**
  - Displays the results of expression evaluations
  - Displays the variable values
  - Follows the execution path
  - Interactively pauses execution and runs REXX statements
- **Activated using the TRACE instruction and function**
- **Syntax:**

```
►►──TRACE──┬──────────┬───────────────────────────────;──────────────────────►◄
           └─number──┘
                    ┌──────────┐      ┌─Normal──┐
           └────────┤ ◄────────┤──────┤         ├──────┘
                    │   ┌─?─┐  │      ├─All──────
                    └───┤   ├──┘      ├─Commands─
                        └─!─┘         ├─Error────
                                      ├─Failure──
                                      ├─Intermediates─
                                      ├─Labels───
                                      ├─Off──────
                                      ├─Results──
                                      └─Scan─────
```

Troubleshooting – Trace Facility . . .

- **Code example:**

```
A = 1
B = 2
C = 3
D = 4
Trace I
If (A > B) | (C < 2 * D) Then
   Say 'At least one expression was true.'
Else
   Say 'Neither expression was true.'
```

- **Result:**

```
 6 *-* If (A > B) | (C < 2 * D)
      >V>    "1"
      >V>    "2"
      >O>    "0"
      >V>    "3"
      >L>    "2"
      >V>    "4"
      >O>    "8"
      >O>    "1"
      >O>    "1"
      *-*    Then
 7 *-*   Say 'At least one expression was true.'
      >L>     "At least one expression was true."
At least one expression was true.
```

Troubleshooting – Trace Facility . . .

- **Interactive trace provides additional debugging power**
  - Pause execution at specified points
  - Insert instructions
  - Re-execute the previous instruction
  - Continue to the next traced instruction
  - Change or terminate interactive tracing

- **Starting interactive trace**
  - ? option with the TRACE instruction
  - In TSO, use EXECUTIL TS command (Trace Start)
    - Code in your REXX EXEC
    - Issue from the command line to debug next REXX EXEC run
    - Cause an attention interrupt and enter TS

Programming Style and Techniques

- **Be consistent with your style**
  - Helps others read and maintain your code
  - Having style rules will make the job of coding easier

- **Indentation**
  - Improves readability
  - Helps identify unbalanced or incomplete structures (DO-END pairs)

- **Comments**
  - Provide them!
  - Choices:
    - In blocks
    - To the right of the code

- **Capitalization**
  - Can improve readability
  - Suggestions
    - Use all lowercase for variables
    - Use mixed case (captilize the first letter) for keywords, labels, calls to internal subroutines
    - Use upper case for calls to external routines (commands)

Programming Style and Techniques . . .

- **Variable names**
  - Try to use meaningful names – helps understanding and readability
  - Avoid 1 character names – easy to type but difficult to manage and understand
  - Avoid ending names with letter O or lowercase L – hard to distinguish between numbers 0 and 1

- **Subroutines**
  - Try to avoid the over use of subroutines or functions
  - Subroutines are useful, but have performance impact
  - If it's called only once, does it need to be a subroutine?

- **Comparisons**
  - REXX supports *exact* (e.g. "==") and *inexact* (e.g. "=") operators
  - Only use *exact* operators when appropriate
    ```
    if a == "SAVE" then …
    ```
  - Above comparison will fail if **a** is **"SAVE "**
  - Avoid using non-standard NOT characters: "¬" and "/"
    - Portability problem when transferring code to an ASCII platform
    - Use "\=", or less commonly used "\>" "\<=

Programming Style and Techniques . . .

- **Semicolons**

  - Can be used to combine multiple statements in one line
    - DON'T – detracts from readability
  - Languages like C and PL/I require a ";" to terminate a line
  - Can also be done in REXX
    - DON'T – doubles internal logic statement count for interpreted REXX

- **Conditions**

  - For complex statements, REXX evaluates all Boolean expressions, even if first fails:

  `If 1 = 2 & 3 = 4 & 5 = 5 Then Say 'Impossible'`

    - Divide-by-zero can still occur if a=0

    `If a \== 0 & b/a > 1 Then ...`

    - Can be avoided by nesting IF statements:

    `If a \== 0 Then`
    `   If b/a > 1 Then ...`

REXX Language Coding Techniques

Programming Style and Techniques . . .

- **Literals**

  - Important to use literals where appropriate

    - For example: external commands

  - Lazy programming can lead to unfortunate results

    - For uninitialized variables: value=name
      `control errors cancel`

    - This usually works

      - Breaks if any of the 3 words is a variable with value already assigned

    - Also a performance cost for unnecessary variable lookups (20%+ more CPU)

    - Instead enclose literals in quotation marks
      `"CONTROL ERRORS CANCEL"`

Programming Style and Techniques . . .

- **External commands**

  – Best practices

    • Enclose in quotation marks

    • Use uppercase

    • Fully spell out the command

      – Don't assume any abbreviations that may not be present if the EXEC is moved to another system

      – Preface with the external environment as needed

REXX Enhancements in z/OS V2.1

REXX Enhancements in z/OS V2.1 and later

- **EXECIO enhanced to support I/O with RECFM=U, VS, VBS**

- **RECFM=U,VS,VBS support also added to callable I/O interface**

- **New TRAPMSG function allows IRX... messages, if issued from a command invoked by the EXEC, to be captured via OUTTRAP**

- **STORAGE function now supports 64-bit addresses for both reading from and writing to storage.**

- **Empty sequential data set can be part of a concatenation accessed by EXECIO,CLIST I/O, PRINTDS if it is SMS managed**

- **LISTDSI enhanced (REXX and CLIST)**

- **RACF/NORACF operand**

- **Multi Volume Support**

- **Handles data sets with extended attributes**

- **Other smaller requirements**

Related Programs

CMS and TSO Pipelines

- **A powerful method of processing or manipulating data**

- **Can be called within REXX programs**

- **A collection of data processing elements connected in a series**

  – Output of one element becomes the input to the next element

  – For example, on z/VM

  ```
  'PIPE cp query dasd | stem dasd.'
  ```

  • Issues the CP command QUERY DASD and response is written into the pipeline
  • The next stage (STEM) receives the input and places it into the stem variable DASD, setting DASD.0 to the number of lines of data

- Included in all current releases of z/VM

- Available as a separate product for TSO

  – **Batchpipes (5655-D45)**

Open Object REXX

- **Open Object REXX is available via open source community**

  – Runs on Linux on System z

  – Many other platforms

- **www.oorexx.org**

  – Managed by REXX Language Association

- **99% compatible with other System z REXX programs**

- **Informal testing with SLES on memory and CPU constrained system**

  – Compare PERL and OOREXX – OOREXX is much faster!

  – Memory footprint of OOREXX is similar to PERL with several modules loaded

(Open Source) NetRexx

- **An object oriented Rexx for the Java virtual machine (JVM)**
  - Write in REXX (or REXX-like)
  - Compiler converts to Java source statements and bytecode
- **Available via open source community since 2011**
- **netrexx.org**

  – Managed by REXX Language Association

Additional Information and Contacts

- **IBM REXX Website
  http://www.ibm.com/software/awdtools/rexx**

- **Additional IBM Contacts**

  – Virgil Hein, vhein@us.ibm.com

    • Compiler and Library for REXX on zSeries

  – John Ehrman, ehrman@us.ibm.com

    • REXX Compiler

- **Hands-on Lab Exercises**

  – REXX Lab: Parts 1 - 2

    • Tuesday, starting at 10:00am

References

- Compiler and Library for REXX User's Guide and Reference (SH19-8160)
- REXX/VM User's Guide (SC24-6222)
- REXX/VM Reference (SC24-6221)
- TSO/E REXX Reference (SA22-7790)
- SA22-7781-08, z/OS TSO/E CLISTs
- SA22-7786-12, z/OS TSO/E Messages
- ISPF Services Guide
  - z/OS V1, SC19-3626
  - z/OS V2, SC34-4819
- ISPF Dialog Developer's Guide and Reference
  - z/OS V1, SC19-3619
  - z/OS V2, SC34-4821
- ISPF Edit and Edit Macros
  - z/OS V1, SC19-3621
  - z/OS V2, SC28-1312
- Using REXX and z/OS UNIX System Services (SA22-7806)
- SDSF Operation and Customization (SA22-7670)
- DB2 Application Programming and SQL Guide (SC19-4051)
- MVS IPCS Commands (SA22-7594)
- MVS Programming Authorized Assembler Services Guide (SA22-7605)
- Security Server RACF Macros and Interfaces (SA22-7682)

धन्यवाद
Hindi

多謝
Traditional Chinese

감사합니다
Korean

Спасибо
Russian

Gracias
Spanish

Thank
You
English

Obrigado
Brazilian Portuguese

شكرا
Arabic

Danke
German

Grazie
Italian

多谢
Simplified Chinese

Merci
French

நன்றி
Tamil

ありがとうございました
Japanese

ขอบคุณ
Thai