IBM **Z**

Rexx Language Coding Techniques

Part 2

Tracy Dean
IBM Product Manager, z/VM Tools and IMS Tools

June 2023

Agenda

- Part 1
  - Rexx products
  - External environments and interfaces
  - Instructions, functions, and subroutines
  - Variable visibility
  - Parsing
- Part 2
  - Rexx compound variables vs. data stack
  - I/O
  - Troubleshooting
  - Programming style and techniques
  - Other Rexx products and projects
- Additional material included in hand-out, not covered in session

© 2014, 2023 IBM Corporation

Compound Variables and Data Stack

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.j                      stem:   ARRAY.
                             tail:   J

name = 'Smith'
phone = 12345

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

 © 2014, 2023 IBM Corporation

Compound Variable Values

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

```
Say month.12 → MONTH.12
month. = 'Unknown'
month.3 = 'March'
month.6 = 'June'

Say month.12 → 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
```

- Useful in memory constrained environments
- Avoids "No Value" errors

 © 2014, 2023 IBM Corporation

Processing Compound Variables

- Compound variables provide the ability to process one-dimensional arrays
  - Use a numeric value for the tail
  - **Good practice** - 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 j = 1 to invitee.0
  Say 'Enter the name for invitee' j
  Parse Pull invitee.j
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

© 2014, 2023 IBM Corporation

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     J Smith       Austin
. . .
```

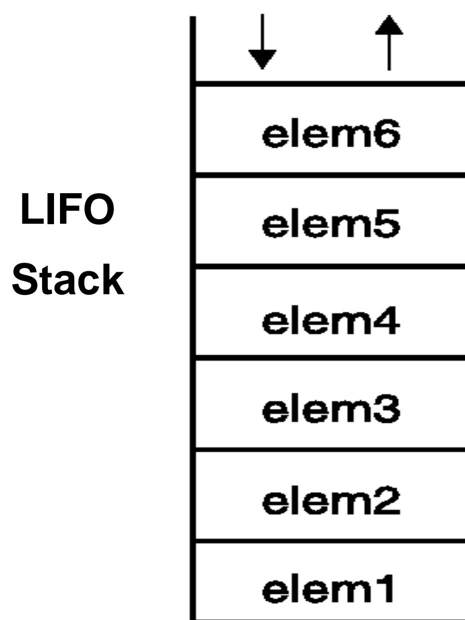
- The unique employee number value can be used as the tail of compound variables that hold the rest of the person's data

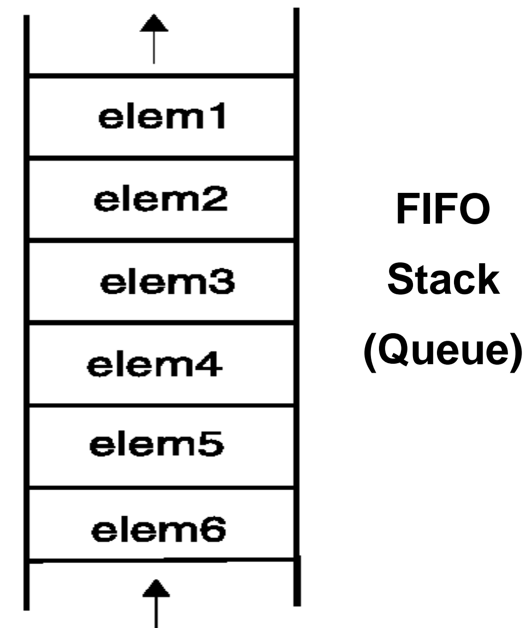
```
'PIPE < EMPLOYEE INFO A | STEM rec.'
Do j = 2 To rec.0
  Parse Var rec.j =1 empnum name.empnum =25 location.empnum
End j
Say 'Which employee number do you want to learn about?'
Parse Upper Pull empnum
Say 'The name of employee' empnum 'is' Strip(name.empnum)'.'
Say 'The location of employee' empnum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


© 2014, 2023 IBM Corporation

What is a Data Stack?

- An expandable data structure used to temporarily hold data items (elements) until needed
- Technically provided by the operating system with Rexx instructions to use it
- When an element is needed it is **always removed** from the **top** 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





© 2014, 2023 IBM Corporation

Manipulating the Data Stack

- 3 basic Rexx instructions
  - **Push** - put one element on the top of the stack

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

  - **Queue** - put one element on the bottom of the stack

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

  - **Parse Pull** - remove an element from the (top) of the stack

```
Parse Pull nextthing
```

  - Outcome:

```
nextthing → 'new top element'
```

- 1 Rexx function
  - **Queued()** - returns the number of elements in the stack

```
num_elems = Queued()
```


© 2014, 2023 IBM Corporation

Why Use the Data Stack?

- Pass a large or unknown number of arguments between EXECs or routines
- To store a large number of data items for later use
  - Size may be unpredictable or unknown
- 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

© 2014, 2023 IBM Corporation

Quick Example of Processing the Data Stack

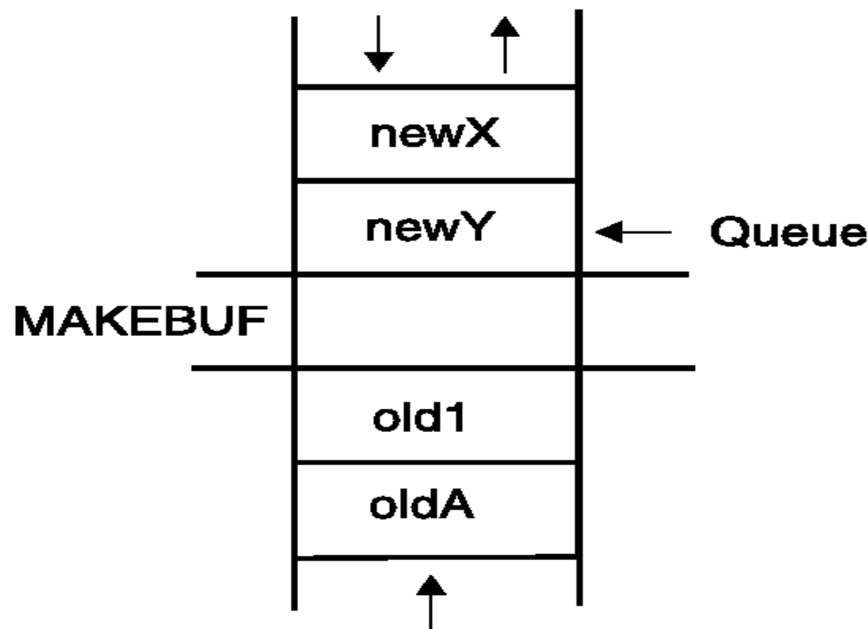
- A receiving (or called) program collects data from the stack
  - Passed from sending/calling program

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

 © 2014, 2023 IBM Corporation

Using Buffers in the Data Stack

- An EXEC can create a buffer in a data stack using the **Makebuf** instruction
- All elements added after a Makebuf instruction are placed in the **new buffer**
  - Makebuf changes where the Queue instruction inserts new elements
    - Remember **Queue** inserts at the **"bottom"** of the stack (or **buffer**)



 © 2014, 2023 IBM Corporation

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
  - Buffer 0 is the main stack so the first Makebuf creates Buffer 1
- **Dropbuf** instruction is used to **remove** a buffer from the data stack
  - Allows an EXEC to easily remove all items in a buffer that is no longer needed
  - 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** instruction is used to find out **how many buffers** have been created
  - The **Qelem** instruction is used to find out the **number of elements** in the most recently created buffer

© 2014, 2023 IBM Corporation

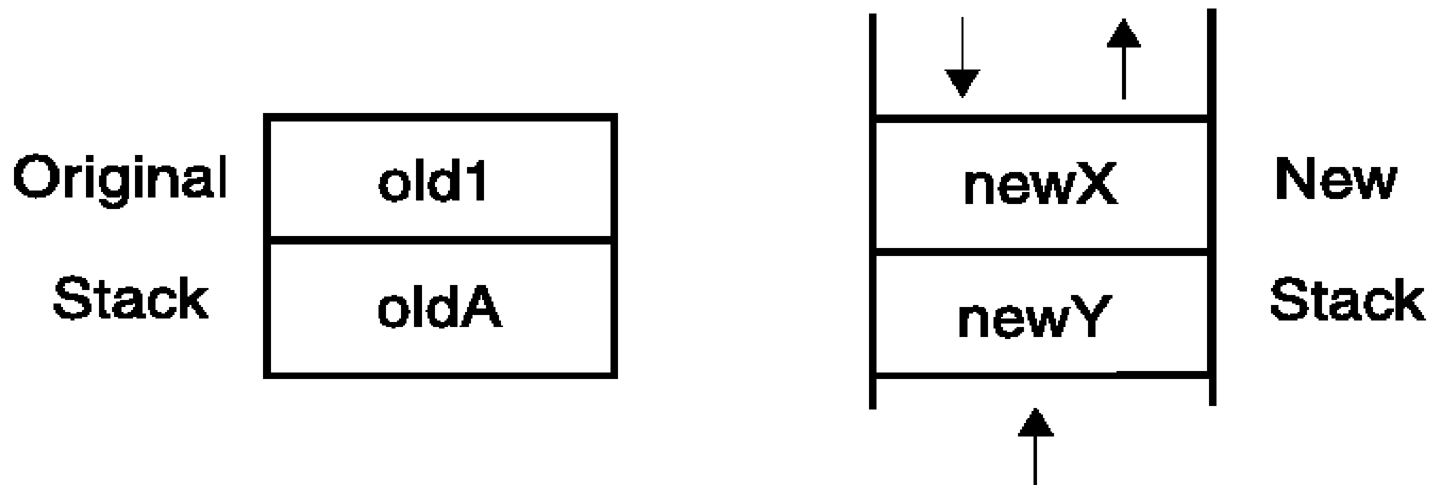
Using Buffers in the Data Stack . . .

- Important notes
  - **When the buffer is empty**
    - Next Pull automatically pulls the next item on the stack
    - Technically that item was in the next buffer
    - No error or indication
  - Creating a buffer only changes the insert point of a FIFO stack
    - Items are still pulled from the top of the stack/buffer
  - Keep track of where you are in buffers within the stack
    - Use Queued() to find the total number of elements in the stack
  - To **remove a buffer** that still contains elements
    - Issue Dropbuf
    - The next request to pull 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

© 2014, 2023 IBM Corporation

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** z/OS 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 removed (not just emptied)
  - When there are no more elements in the new data stack, information is taken from the terminal (not the original data stack)



© 2014, 2023 IBM Corporation

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
  - Caution
    - If no stack previously created with NEWSTACK, then DELSTACK removes all the elements from the **original stack**
- QSTACK - returns the number of data stacks
  - Including the original stack
  - Puts the value in the variable RC
- Note: For z/OS, the QUEUED() function returns the number of elements in the current data stack

© 2014, 2023 IBM Corporation

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

© 2014, 2023 IBM Corporation

Data Stack vs Buffers

- Buffers
  - Advantages
    - z/VM and z/OS supported
    - Create a “stack” on top of the existing stack for new list of items
    - Ability to insert at the bottom of the new “stack”
    - **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

© 2014, 2023 IBM Corporation

Compound Variables vs Data Stack

- 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 simpler

© 2014, 2023 IBM Corporation

Compound Variables vs Data Stack

- 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 instructions are **operating system dependent**

© 2014, 2023 IBM Corporation

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

© 2014, 2023 IBM Corporation

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

 © 2014, 2023 IBM Corporation

Special Variables

- **RC** (or **rc**) variable
  - Return code from external commands and special Rexx instructions
- **Result** variable
  - Value of an expression returned by a called subroutine

© 2014, 2023 IBM Corporation

Troubleshooting – Condition Trapping

- Signal On and Call On instructions
  - Used to trap exception conditions
  - **Signal On** goes to label and does not return
  - **Call On** goes to label and returns
- 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

© 2014, 2023 IBM Corporation

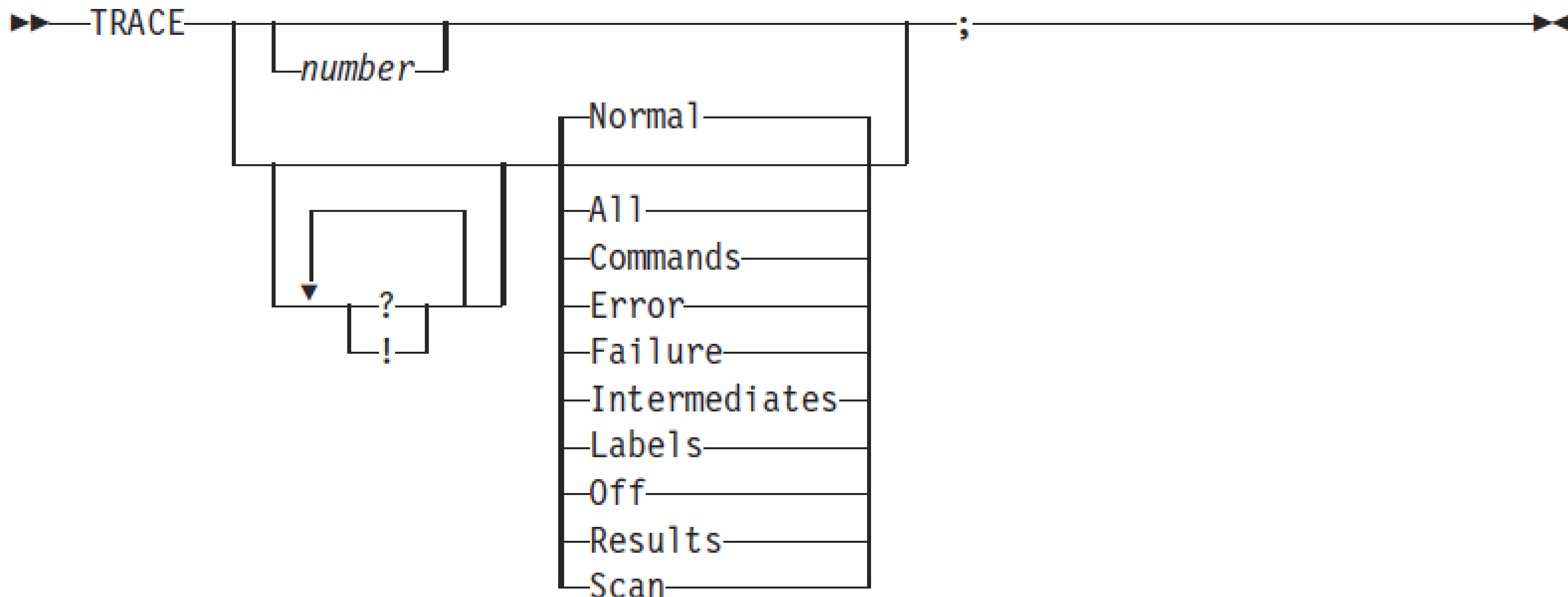
Troubleshooting – Condition Trapping. . .

- Good practice to enable condition handling to process unexpected errors
  - Specifically `Signal On NoValue Name error-routine`
- 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
    - For ERROR and FAILURE - contains the instruction and return code
    - For SYNTAX - contains the syntax error number
  - 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)`

© 2014, 2023 IBM Corporation

Troubleshooting – Trace Facility

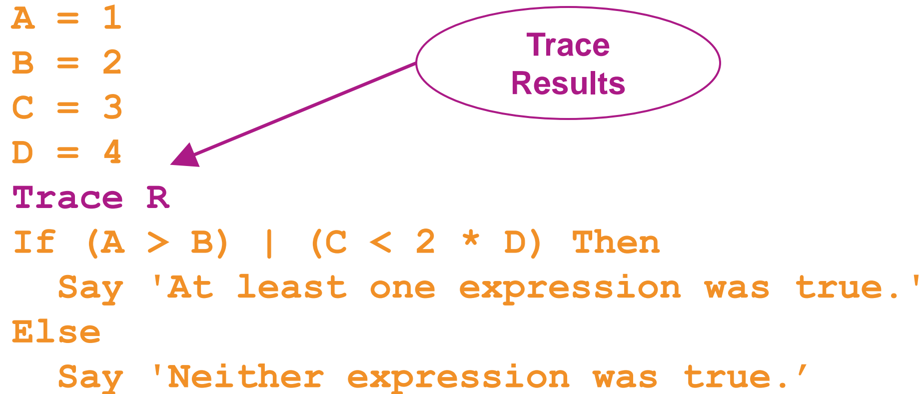
- Provides powerful debugging capabilities
  - Displays the outcome of expression evaluations
  - Displays the variable values
  - Follows the execution path
  - Interactively pauses execution and runs Rexx statements
- Activated using the **Trace** instruction and function
- Syntax:



© 2014, 2023 IBM Corporation

Troubleshooting – Trace Facility . . .

- Code example:




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

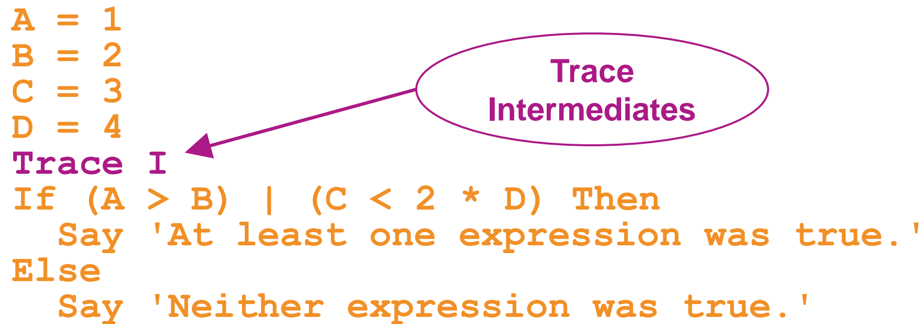
- Output:

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

 © 2014, 2023 IBM Corporation

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.
```

 © 2014, 2023 IBM Corporation

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
    - Issue Trace instruction with desired parameters at next prompt
- 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

 © 2014, 2023 IBM Corporation

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

© 2014, 2023 IBM Corporation

Programming Style and Techniques . . .

- Capitalization
  - Can improve readability
  - Suggestions
    - Use all lowercase for variables
    - Use mixed case (capitalize the first letter) for keywords, labels, calls to internal subroutines
    - Use upper case for calls to external routines (commands)
- Variable names
  - Try to use meaningful names
    - Helps understanding and readability
  - Avoid 1 character names
    - Easy to type but difficult to manage and understand
    - Exception – indices to compound variables
  - Avoid ending names with letter O or lowercase L
    - Hard to distinguish between numbers 0 and 1

 © 2014, 2023 IBM Corporation

Programming Style and Techniques . . .

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

 © 2014, 2023 IBM Corporation

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 ...
```

 © 2014, 2023 IBM Corporation

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
      - Up to 20%+ more CPU
    - Instead enclose literals in quotation marks
      `'CONTROL ERRORS CANCEL'`

© 2014, 2023 IBM Corporation

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

© 2014, 2023 IBM Corporation

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
    - Response is written into the pipeline
    - 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)

 © 2014, 2023 IBM Corporation

Open Object REXX

- Open Object REXX is available via open source community
  - Runs on Linux on Z
  - Many other (distributed) platforms
- www.oorexx.org
  - Managed by Rexx Language Association
- 99% compatible with other IBM Z Rexx programs
- Comparison of PERL and OOREXX
  - Informal testing with SLES on memory and CPU constrained system
    - OOREXX is much faster!
    - Memory footprint of OOREXX similar to PERL with several modules loaded

© 2014, 2023 IBM Corporation

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

© 2014, 2023 IBM Corporation

Additional Information

- IBM Rexx Website
  https://www.ibm.com/products/compiler-and-library-for-rexx-on-ibm-z

© 2014, 2023 IBM Corporation

Summary

- Part 1
  - Rexx products
  - External environments and interfaces
  - Instructions, functions, and subroutines
  - Variable visibility
  - Parsing
- Part 2
  - Rexx compound variables vs. data stack
  - I/O
  - Troubleshooting
  - Programming style and techniques
  - Other Rexx products and projects
- Additional material included in hand-out, not covered in session

© 2014, 2023 IBM Corporation

Tracy Dean
Product Manager, z/VM Tools and IMS Tools
tld1@us.ibm.com

© 2014, 2023 IBM Corporation

धन्यवाद
Hindi

多謝
Traditional Chinese

감사합니다
Korean

Спасибо
Russian

Ndzi khense ngopfu
Tsonga

Gracias
Spanish

شكراً
Arabic

Thank You
English

Obrigado
Brazilian Portuguese

Grazie
Italian

Danke
German

Ke a leboha
Tswana

多谢
Simplified Chinese

Merci
French

நன்றி
Tamil

ありがとうございました
Japanese

ขอบคุณ
Thai

© 2014, 2023 IBM Corporation