Rexx Language Coding Techniques

Tracy Dean
IBM
tld1@us.ibm.com

October 2019

IBM

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
- ➢ Instructions, functions, and subroutines
- ➢ Variable visibility
- ➢ Parsing
- ➢ 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

Rexx Language Coding Techniques

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

Rexx Language Coding Techniques

Part 1

Rexx Interpreter and Libraries

- ➤ A procedural language
- ➤ Indicator to operating system that it's Rexx - **first line** contains:
  `/* Comment (if any) */`
- ➤ 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 **not** 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:

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
    - Trace S provides limited syntax checking
      - ▪ For example: missing END statements
      - ▪ Does not catch syntax errors in If Then statements regarding value comparisons)
  - 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 © 2014, 2019 IBM Corporation

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
    - The default environment in a non-TSO/E address space
    - Example:
        ```
        Address MVS "EXECIO * DISKR MYINDD (FINIS STEM MYVAR"
        ```

- **Many more**

Other z/OS Host Command Environments

- ISPF services
- ISPF edit macros
- CONSOLE
- LINK, LINKMVS, LINKPGM, ATTACH, ATTCHMVS, ATTCHPGM
- SYSCALL
- SDSF
- DSNREXX

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"
    ```

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
```

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
…
```

IBM Z

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 Ress

  - 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

Rexx Language Coding Techniques

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

Multiple Methods to Specify External Environment

- Initial value, later changed:
    - MYTEST is another Rexx EXEC I'm calling from this program:

```
/* Sample Rexx program */
Address Command
…
"EXEC MYTEST"
…
Address CMS
…
"MYTEST"
```

> **Run under Address Command**

> **Run under Address CMS**

- All future commands are treated as **Address CMS** unless specified otherwise

Multiple Methods to Specify External Environment

- Initial value used as default
  - All calls requiring a different value have **Address** statement
  - MYTEST is another Rexx EXEC I'm calling from this program:

```
/* Sample Rexx program */
Address Command
…
"EXEC MYTEST"
…
Address CMS "MYTEST"
…
"EXEC MYTEST"
```

Run under Address Command

Run under Address CMS

Instructions, Functions, and Subroutines

Instructions vs Functions vs Subroutines

➢ 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. often on the **right side of an equal sign**)
  - Built-in - provided as part of the Rexx language
  - Internal - create your own within the same program
  - External – create your own outside this program

➢ Subroutine
  - Called (similar to a function) but may not return data
  - Returns data in special variable: Result

Functions

➢ Must return a single result string

- Often on the **right side of an equal sign**

➢ Built-in functions – too many to list, so a few examples

- Absolute value of a number
  ```
  total = -3
  newtotal = Abs(total)
  ```
  → 3

- Left justify a string
  ```
  fullname = Left('Tracy Dean',20)
  ```
  → 'Tracy Dean   '

- Determine the type of data
  ```
  If Datatype(amtowed,N)=1
    Then amtowed = amtowed * 1.1
    Else Say 'Amount owed is invalid'
  ```

- Today's date or day of the week
  ```
  today = Date()
  ```
  → 28 July 2019

  ```
  todaydow = Date('W')
  ```
  → Friday

- Find a string within another string
  ```
  startcol = Pos('day','Tuesday')
  ```
  → 5
  ```
  startcol = Pos('x','Tuesday')
  ```
  → 0

Subroutines

- ➢ Multi-step task to execute multiple times
  - Write once, use multiple times
  - Make code easier to read
- ➢ Call a subroutine, pass and return variables

```
…
Call CalcInterest amountborrowed
amountdue = Result
…
```

Value from Return
instruction in
special variable
called "Result"

- ➢ Define a subroutine
  - After Exit instruction of main program
  - Start with name of subroutine followed by colon
  - End with Return instruction

```
CalcInterest:
Parse Arg principal
…
total = principal + (principal * rate)
Return total
```

Declaration and Visibility of Variables

Variable Declaration

➢ Rexx is a procedural language

➢ Variables are **not** declared

- Initial value is same as **variable name** in uppercase
- Seen as a string unless perform mathematical operation
  - Requires the value **at the time** be valid for the operation
  - **Type** of data assigned to a variable can **change** within a program
  - Valid:
    ```
    total = 'Here is some text'
    Say 'Here is the original total:' total
    total = 5+3
    newtotal = 1.5 * total
    Say 'Here is the new total:' newtotal
    ```
  - Result:
    ```
    Here is the original total: Here is some text
    Here is the new total: 12.0
    ```

Visibility of Variables

➢ Variables can be visible throughout a program

- Visible within **Functions** and **Subroutines** you create within the program
  - No need to pass, declare or expose them
- Not visible in **Procedures** unless specifically exposed

➢ Programming practice

- Functions, Subroutines, and Procedures use **different** variable names for reusability
- Pass the value of variables on the call
- Function, Subroutine or Procedure will parse the value and assign to its own variables
- Pass values back to main program via Return statement
  - For subroutines, values are visible to calling routine via **Result** variable

Visibility of Variables in **Functions** and **Subroutines**

- ## Using existing variables
  - ### In a function

```
principal = 100
interest = 0.10
total = CalcTotalDue()

…
Exit
…
CalcTotalDue:
total = principal*(1 + interest)
Return total
```

- ## Creating new variables
  - ### In a subroutine

```
loan = 100
rate = 0.10
Call CalcTotalDue(loan rate)
totaldue = Result

…
Exit
…
CalcTotalDue:
Parse Arg principal interest .
total = principal*(1 + interest)
Return total
```

Both are valid for functions and subroutines

More common to use separate variables in the function or subroutine

Visibility of Variables in Procedures

➤ **No visibility** of variables from main program unless specifically requested – **most common**

```
total = 855
…
loan = 100
rate = 0.10
Call CalcTotalDue(loan rate)
Say 'This is your total due:' Result
Say 'Principal & interest:' principal interest
…
Exit
…
CalcTotalDue: Procedure
Parse Arg principal interest .
total = principal*(1 + interest)
Return total
```

Variable used for something else in the program

Value from Return statement in Procedure

Does not effect variable of same name in main program

Indicates no exposure of variables from main program

Value of anything here becomes value of "Result" variable

Visibility of Variables in Procedures

➤ **No visibility** of variables from main program unless specifically requested

total = 855

> Variable used for something else in the program

Output:

    This is your total due: 110.00

    Principal & interest: PRINCIPAL INTEREST

> Doe...
> effect v...
> of same name
> in main
> program

```
...Exit
...
CalcTotalDue: Procedure
Parse Arg principal interest .
total = principal*(1 + interest)
Return total
```

> Value of anything here becomes value of "Result" variable

Visibility of Variables in Procedures

> **Exposing** variables from main program to a Procedure

```
loan = 100
rate = 0.10
Call CalcTotalDue(loan rate)
Say 'This is your total due:' total
Say 'Principal & interest:' principal interest
…
Exit
…
CalcTotalDue: Procedure Expose total
Parse Arg principal interest .
total = principal*(1 + interest)
Return
```

Never set in main program, only set in Procedure

Brings in value of this variable from main program

BEWARE: Also changes value in main program

No need to Return "total" as "Result"

Visibility of Variables in Procedures

➢ **Exposing** variables from main program to a Procedure

```
loan = 100
rate = 0.10
```

> Note never set in main program, only set in Procedure

> Same output:
>   This is your total due: 110.00
>   Principal & interest: PRINCIPAL INTEREST

> BEWA...
>   Als...
>   chang...
>   value i...
>   progr...

> ...value
>   ...able
>   ...in
>   ...m

```
total = principal*(1 + interest)
Return
```

> No need to Return "total" as "result"

Returning Variables from Functions and Procedures

➢ **Returning** variables from a Procedure to the main program

```
loan = 100
rate = 0.10
Call CalcTotalDue(loan rate)
totaldue = Result
Say totaldue

…

Exit

…

CalcTotalDue: Procedure
Parse Arg principal interest .
total = principal*(1 + interest)
Return 'Total due:' total
```

All data from "Return" statement is assigned as a string to "Result" variable

Can return any expression (literals, variables, etc.)

Rexx Language Coding Techniques

© 2014, 2019 IBM Corporation

Returning Variables from Functions and Procedures

➢ **Returning** variables from a Procedure to the main program

```
loan = 100
rate = 0.10
Call C
totald
Say to
…
Exit
…
CalcTo
Parse A
total = principal*(1 + interest)
Return 'Total due:' total
```

All data from "Return" statement is assigned a string to "result" variable

Output:

Total due: 110.00

Can return any expression (literals, variables, etc.)

Rexx Language Coding Techniques

Parsing

Keyword Instruction: 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 secondvar thirdvar fourthvar

firstvar  -> 'Write'
secondvar -> 'the'
thirdvar  -> 'blank-delimited'
fourthvar -> '   string  '
```

Parse Templates – Blank Delimiter

➤ Simple template

- A period (aka a dot) is a placeholder in a template
  - A "dummy" variable used to collect unwanted data
  - Notice the consecutive single quotes so the single quote is recognized as part of the string

```
datastring = 'Last one gets what''s left'
Parse Var datastring firstvar . secondvar

firstvar -> "Last"
secondvar -> "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 secondvar .

firstvar -> "Last"
secondvar -> "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 secondvar thirdvar fourthvar .
firstvar -> 'Write'
secondvar -> 'the'
thirdvar -> 'blank-delimited'
fourthvar -> 'string'
```

Parse Templates – Literal or Variable Delimiter

➢ 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 '-' secondvar .
```

Literal delimited

Blank delimited

Variable:

```
delim = '-'
Parse Var datastring firstvar (delim) secondvar .
```

Result (the same in both cases):

```
firstvar -> '  Write the   blank'
secondvar -> 'delimited'
```

Parse Templates – Positional Delimiter

➢ 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           '
```

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

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

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

Part 2

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

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
```

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

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

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 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
```
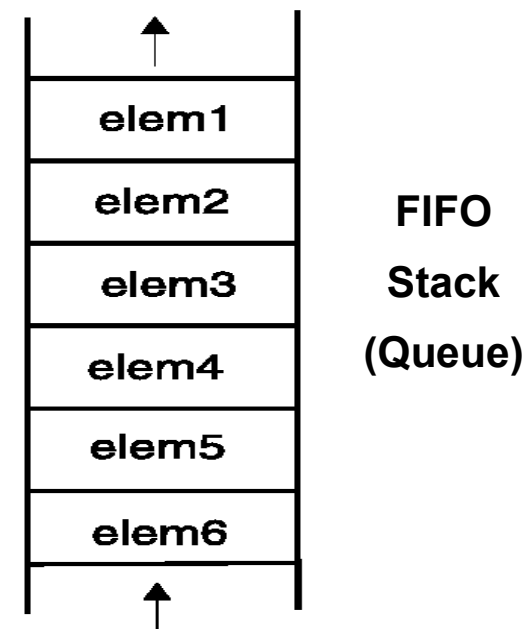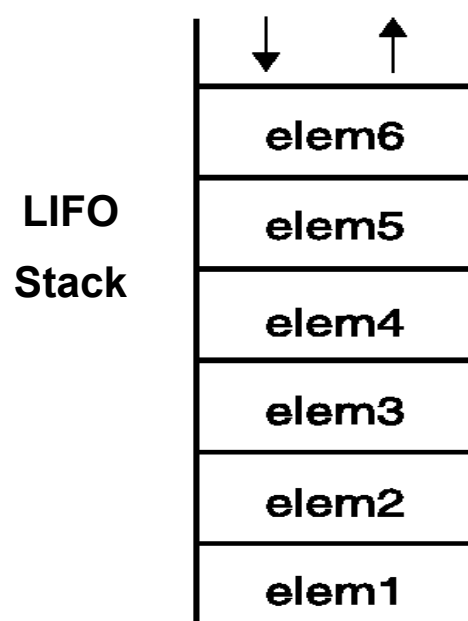
Rexx Language Coding Techniques

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 **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

LIFO

Stack

elem6
elem5
elem4
elem3
elem2
elem1

elem1
elem2
elem3
elem4
elem5
elem6

FIFO

Stack

(Queue)

Manipulating the Data Stack

➢ 3 basic Rexx <u>instructions</u>

- **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 nextthing
  ```

- Result:

  ```
  nextthing → 'new top element'
  ```

➢ 1 Rexx <u>function</u>

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

  ```
  num_elems = Queued()
  ```

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

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
…
```

Full parsing capability

Rexx Language Coding Techniques

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 changes where the Queue instruction inserts new elements

• Remember **Queue** inserts at the **"bottom"** of the stack (or **buffer**)
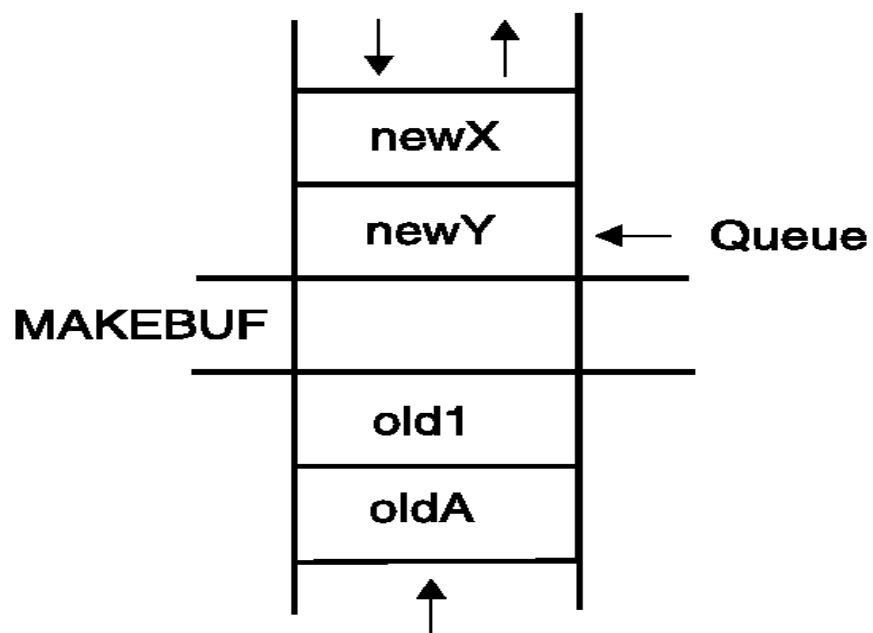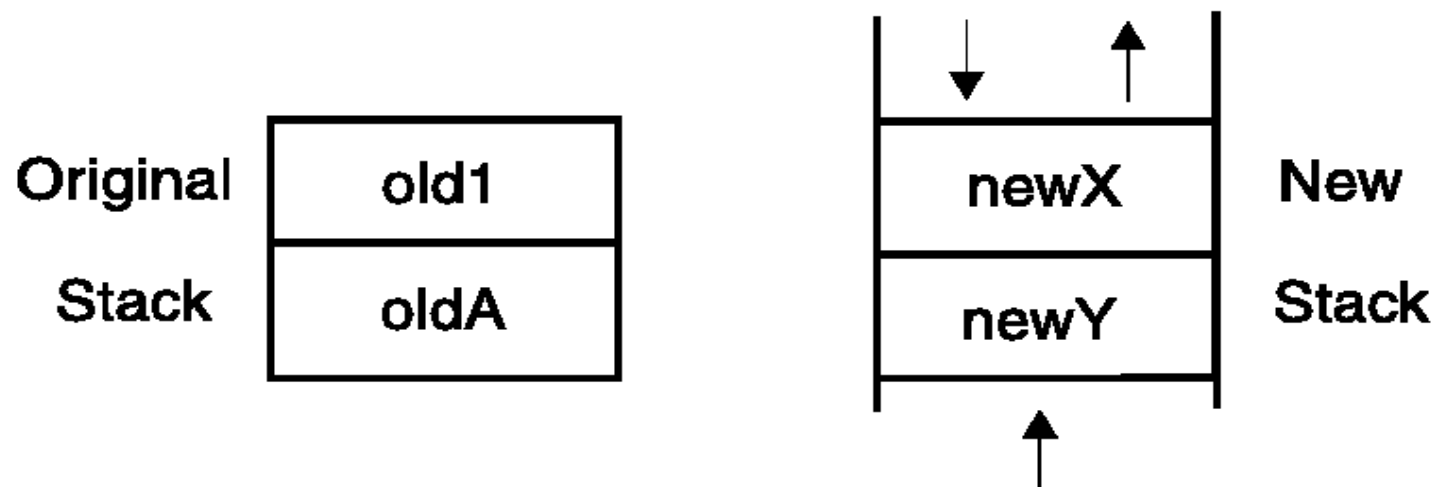
Using Buffers in the Data Stack . . .

➢ 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 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
- 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

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 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)

Original Stack	old1
	oldA

	newX	New
	newY	Stack

Rexx Language Coding Techniques   © 2014, 2019 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.

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

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

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

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 commands are **operating system dependent**

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

Special Variables

➤ RC variable

- • Return code from external commands and special Rexx commands/statements

➤ RESULT variable

- • Value of an expression returned by a subroutine

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

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 - ontains the command 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)`

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──────────┘
```
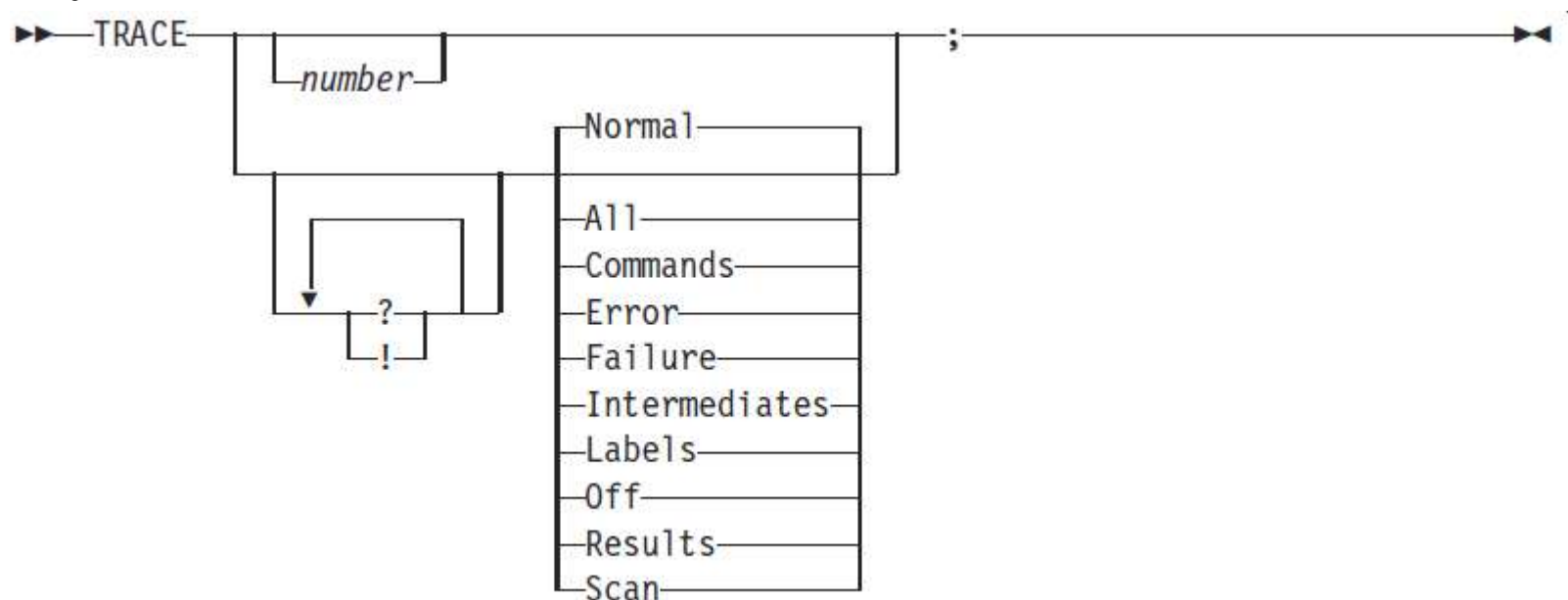
Troubleshooting – Trace Facility . . .

➤ Code example:

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

Trace
Results

➤ Result:

```
 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.
```

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.'
```

**Trace
Intermediates**

➢ 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

  - Issue Trace command 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

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

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

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 …
  ```

  Extra blank

- 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 "\>" "\<=

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 ...
      ```                        ⬅ Rexx error
    - Can be avoided by nesting IF statements:

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

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'`

Rexx Language Coding Techniques

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

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)

Open Object REXX

➢ Open Object REXX is available via open source community

  • Runs on Linux on z Systems

  • 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

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

➢ IBM Rexx Website
https://www.ibm.com/us-en/marketplace/compiler-and-library-for-rexx-on-ibm-z

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

Rexx Language Coding Techniques   © 2014, 2019 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

Rexx Language Coding Techniques