Using REXX for IBM Mainframe Application Development

Liam Doherty
IBM Corporation

Wednesday March 4th, 2015
Session 16722

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides **education**, professional **networking** and industry **influence.**

Important REXX Compiler Disclaimer

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

- What use REXX?
- The Product
- The REXX compiler
- REXX External environments
- Common code constructs used
- Less Common code constructs

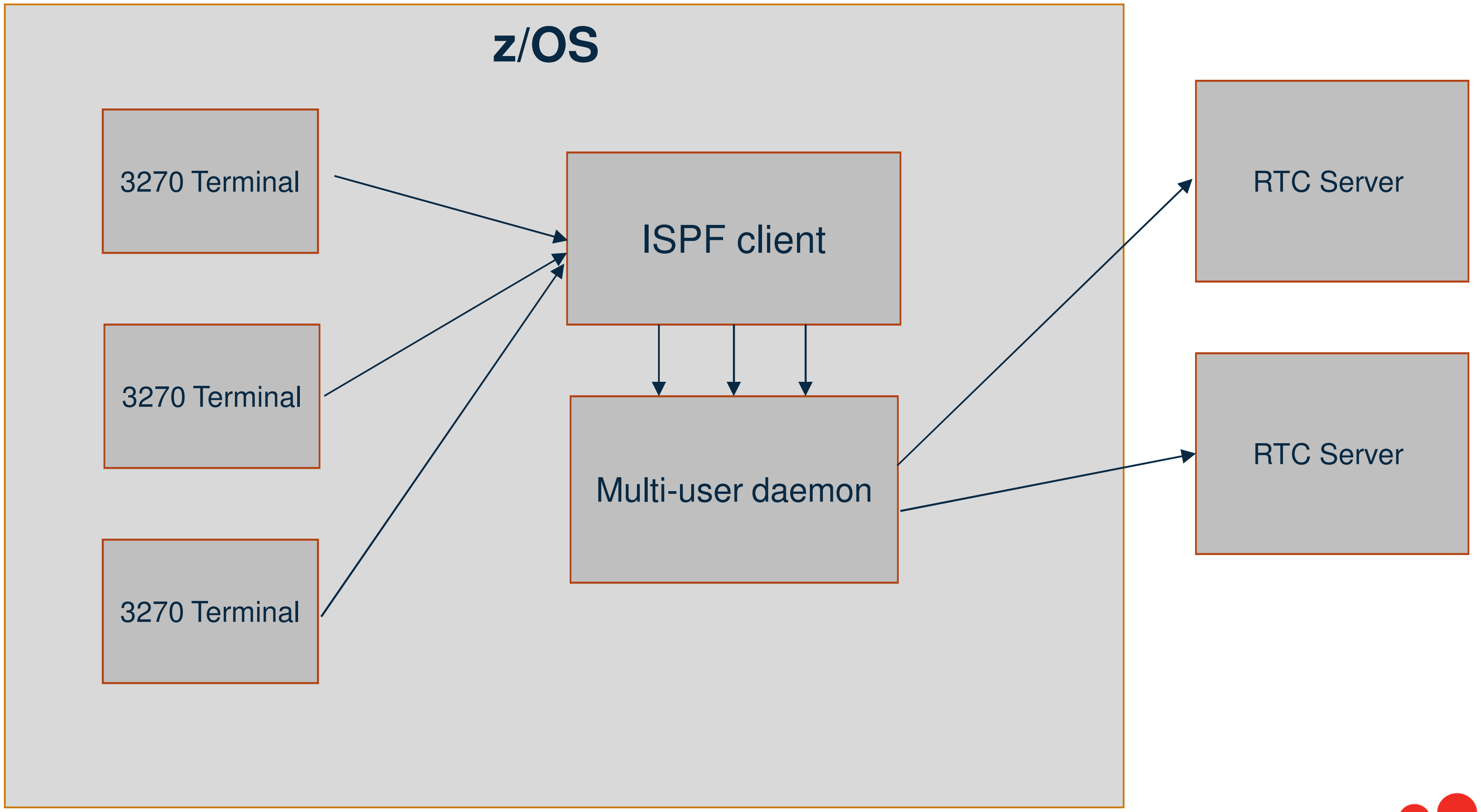
Why use REXX?

- Easy to pick up if you are unfamiliar with mainframe languages
- Easy to "get runs on the board".
  - In an Agile environment with short sprints we could prototype and show progress very quickly
- Interfaces with other environments very well
  - In particular z/OS Unix
- Is the perfect language when paired with ISPF Dialog development
- Ideal for parsing data
- Because I love it…

The Product

- Rational Team Concert was originally offered as an Eclipse Client and a Web Client
  - However a number of customers expressed an interest in there being an ISPF Client
- The ISPF Client would need to send requests to a Java Daemon. The daemon would send/receive the data to the RTC Server. Then send a response in the form of a JSON string back to the ISPF client.
- In a later release a Configuration Utility and IVP were also provided

Architecture



The REXX Compiler

- **Why use the REXX compiler?**
  - Source Code Protection
    - *Protects your intellectual property*
    - *Protects your code from manipulation*
    - *Keeps your code maintainable*
  - Meet IBM packaging rules to include a copyright in the load
  - Program performance
    - *Approximately 30% performance gain unless you are using the REXX Alternate library*
  - Compiler syntax checks the code without the need for execution
    - *Lists all errors, rather than stopping at the first error when run interactively*

The REXX Compiler

- The REXX library is required to run compiled programs
- Compiled REXX is not an LE Language
- Compiled with run one of 2 ways
  - Run-Time Library : Purchased Program Product
  - REXX Alternate : Installed on z/OS
    - *Will use the native systems REXX interpreter*
  - Compiled REXX will use whichever library is available at execution time
- Compiled code runs in 31-bit mode
  - Uses old opcodes such as BALR. Can run on old hardware
  - No z/Architecture in plan today.

The REXX Compiler - %STUB

- The %STUB compiler directive provides interfaces with various parameter-passing conventions
- The only two we use are:
  - CALLCMD
    - *When the CALL program_name command is issued from the TSO/E command line, or when the CALL program_name host command is issued from within an EXEC executing under TSO/E.*
    - *In simple terms, any "Main" program invoked by:*
      - *SELECT CMD(program_name)*
    - *First parm will be the module name*
  - *CPPLEFPL*
    - *It contains the logic to determine if the REXX program is being invoked as a TSO/E command or as a REXX external routine. Once this has been determined, the compiled REXX program is given control with the appropriate parameters.*
    - *In simple terms, subroutines called using the CALL statement or invoked as a function call*
- See IBM Compiler and Library for REXX on System z: User's Guide and Reference

  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.rexa100/toc.htm?lang=en

REXX external environments

- One of the main reasons REXX is so powerful as a development language
- ADDRESS instruction is used to define the external environment to receive host commands
  - Address <environment name>
- There are many different environments that provide many different functions
  - TSO
  - ISPEXEC
  - ISREDIT
  - CONSOLE
  - LINK, LINKMVS, LINKPGM, ATTACH, ATTCHMVS, ATTCHPGM
  - SYSCALL
  - SDSF
  - DSNREXX
  - …and more

Address TSO

- One of the most common environments, and the default if running in a TSO/E address space
- Used to run TSO/E commands like ALLOCATE and TRANSMIT
- See: *TSO/E REXX Reference* –

  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ikja300/toc.htm
- Examples:

```
Address TSO "ALLOC F(SYSIN) NEW RECFM(F B) LRECL(80) SP(1 1) TRACKS"
Address TSO "EXECIO * DISKR SYSPRINT (STEM cmdout. FINIS "
Address TSO "TSOEXEC CALL *(BLZPASTK) '"USER APPLID PASSTCKT"''
Address TSO "DELETE 'DOHERTL.MY.DATASET'"
```

Address ISPEXEC

- Used to invoke ISPF services like DISPLAY and SELECT
- Only available to REXX running in ISPF
- See: *ISPF Services Guide* –

  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.f54sg00/toc.htm

Examples:

```
Address ISPEXEC
"ADDPOP ROW(2) COLUMN(4)"
"DISPLAY PANEL(BLZ@ABOU)"
"REMPOP"
```

REXX external environments

- Environment can be set for a single command, or for all non-REXX commands from a certain point
- Last set environment will be used if REXX does not recognise a command
- Example

```
Address ISPEXEC
“VGET (BLZTRACE) SHARED”

Address TSO "ALLOC F(INDD) DA(‘DOHERTL.EXEC(COND)’) SHR“
Address TSO "EXECIO * DISKR INDD (STEM cmdout. FINIS “

"DSINFO DATASET(‘DOHERTL.EXEC')“

Address TSO
"ALLOC F(OUTDD) DA(‘DOHERTL.EXEC(OUTPUT)’) SHR“
"EXECIO * DISKW OUTDD (STEM cmdout. FINIS “
“ISPEXEC VPUT (BLZVAR) PROFILE”
```

Address 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
- See: Using REXX and z/OS UNIX System Services -

  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxb600/toc.htm

  Examples:

```
xx = syscalls('ON')
Address Syscall 'readdir /tmp tmp. tmpst.`
Do sb = 1 to tmp.0
  If Substr(tmp.sb,1,1) = '.' then
  Do
    If Pos(substr(tmpst.sb.ST_MODE,1,1),'2467') = 0 |,
       Pos(substr(tmpst.sb.ST_MODE,2,1),'2467') = 0 |,
       Pos(substr(tmpst.sb.ST_MODE,3,1),'2467') = 0 then
    Do
      Say ‘/tmp is not writable’
```

Address ISREDIT

- Used to invoke ISPF edit macro commands like FIND and DELETE
- Only available to REXX running in an ISPF edit session
- See: ISPF Edit and Edit Macros –

  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.f54em00/toc.htm

  Examples:

```
Address ISREDIT "MACRO (PARMSTR)"
Do while (PARMSTR /= '')
  Parse var PARMSTR cmd ';' PARMSTR
  Address ISREDIT cmd
End
Address ISREDIT 'END'


Address ISREDIT "MACRO"
Address ISPEXEC "VGET (CONFLICT) SHARED"
Address ISPEXEC "VGET (BLZMOD) SHARED"
Address ISREDIT "COMPARE '"CONFLICT"' X"
```

Address 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
- See: *TSO/E REXX Reference –*
  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ikja300/toc.htm
- Examples:

```
Address TSO "ALLOC F(NEWDD) DA('DOHERTL.V6DEV.SOURCE') SHR"
Address TSO "ALLOC F(OUTDD) NEW SP(1 1) CYL"
Address TSO "ALLOC F(SYSIN) NEW SP(1 1) CYL"

Sysin.0 = 2
Sysin.1 = "SRCHFOR  'BLZDEMON'"
Sysin.2 = "SELECT BLZBKPZP,BLZBUILD,BLZCIVP,BLZCKHIS,BLZCLMM,BLZCNVRT"
Address TSO "EXECIO * DISKW SYSIN (STEM SYSIN. FINIS)"

PARMS = "SRCHCMP,ANYC"

Address LINKMVS 'ISRSUPC PARMS'

"EXECIO * DISKR OUTDD (FINIS STEM outdd."
```

Address 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
- See: **SDSF Operation and Customization**
  http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.isfa500/toc.htm
- And: **Implementing REXX Support in SDSF**
  http://www.redbooks.ibm.com/abstracts/sg247419.html?Open
- Examples:

```
rc=isfcalls('ON')
mycmd.0=1
mycmd.1="D IKJTSO,AUTHPGM"
Address SDSF ISFSLASH "("mycmd.") (WAIT)"
Say RC
/* List any error messages */
Say "isfmsg is:" isfmsg
Say "isfmsg2.0 is:" isfmsg2.0
if datatype(isfmsg2.0) = "NUM" then
  do ix=1 to isfmsg2.0
    Say "isfmsg2."ix "is:" isfmsg2.ix
  end
rc=isfcalls('OFF')
/* get output */
do i = 1 to ISFULOG.0
  say ISFULOG.i
end
```

Address SDSF (cont)

```
rc=isfcalls('ON')
/* Access the ST display */
isfprefix = 'BLZGO1'
Address SDSF "ISFEXEC ST"
/* Loop for all running BLZJMON jobs */
Do ix=1 to JNAME.0
  if JNAME.ix = "BLZGO1" & QUEUE.ix = "EXECUTION" & ACTSYS.ix <> "" then
  Do
    /* Issue the ? (JDS) action against the  */
    /* row to list the data sets in the job. */
    Address SDSF "ISFACT ST TOKEN('"TOKEN.ix"') PARM(NP ?)" ,
       "( prefix jds_"

    /* Find the SYSOUT data set and allocate it   */
    /* using the SA action character              */
    Do jx=1 to jds_DDNAME.0
      if jds_DDNAME.jx = "STDOUT" then
      Do
        Address SDSF "ISFACT ST TOKEN('"jds_TOKEN.jx"')" ,
          "PARM(NP SA)"

        /* Read the records from the data set and list them. */
        /* The ddname for each allocated data set will be in */
        /* the isfddname stem.  Since the SA action was done */
        /* from JDS, only one data set will be allocated.    */
        Do kx=1 to isfddname.0
          Say "Now reading" isfdsname.kx
          "EXECIO * DISKR" isfddname.kx "(STEM line. FINIS"
          Say "  Lines read:" line.0
          Do lx = 1 to line.0
            Say "  line."lx "is:" line.lx
          end
        end
      end
    end
  end
end
rc=isfcalls('OFF') "
```

Address 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
- See: DB2 Application Programming and SQL Guide –
  http://www-01.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/com.ibm.db2z11.doc.apsg/src/apsg/db2z_apsg.dita
- Examples:

```
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"
```

Other External Environments

- MVS
  - Use to run a subset of TSO/E commands like EXECIO and MAKEBUF
  - The default environment in a non-TSO/E address space, for example
  - See: TSO/E REXX Reference

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

- IPCS
  - Used to invoke IPCS subcommands from REXX
  - Only available when run from in an IPCS session
  - See: MVS IPCS Commands

- 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
  - See: TSO/E REXX Reference

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*

See: *Security Server RACF Macros and Interfaces –*

http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.icha300/toc.htm

```
class = 'STARTED'
profile = ‘BLZBFA.*'
rc=IRRXUTIL("EXTRACT",class,profile,"RACF","","FALSE")
racfUser  = RACF.STDATA.USER.1
racfGroup = RACF.STDATA.GROUP.1
```

Common code constructs used

- PARSE
- Compound variables

PARSE – Key Instructions : ARG

- ARG
  - Retrieves the argument strings provided to a program or internal routine and assigns them to variables
  - ARG by itself is a short form of PARSE UPPER ARG
  - Examples:

```
Arg HLQ TEAMHLQ LANG TR
```

```
Parse Arg WSUUID BLZZHUID BLZZLNME BLZZHNME
```

```
Parse Arg EditDsn '(' EditMem ')'
```

```
Parse arg Module '"'manifest'"' '"'ziploc'"' '"'pkgzip'"`,
                 '"'restore'"' '"'builddefVersion'"',
                 '"'runtimeVersion'"' '"'tstamp'"`,
                 '"'traceOption'"'.
```

PARSE – Key Instructions : PULL

- PULL
  - Reads a string from the head of the external data queue
  - PULL by itself is a short form of PARSE UPPER PULL
  - Example

```
Say 'Unable to determine HLQ for the SBLZEXEC installed dataset'
Say 'Please enter correct HLQ ? (eg: BLZ.V5)'
Pull hlq
If hlq = '' Then
Do
  Say 'No High Level Qualifier entered .. processing ends'
  Exit 8
End
```

PARSE – Key Instructions : Templates

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

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

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

PARSE – Key Instructions : Templates

- Simple Templates
  - Divides the source string into blank-delimited words and assigns them to the variables named in the template
  - A period is a placeholder in a template – a “dummy” variable used to collect unwanted data

```
ex 'dohertl.v6dev.sblzexec(blzcinit)' 'dohertl.v6dev dohertl.v6conf enu'
```

```
Parse Upper Arg parms
Parse Var parms hlq usrhlq LANG .
If LANG = '' Then LANG = 'ENU'
```

parms	= ‘DOHERTL.V6DEV DOHERTL.V6CONF ENU’
hlq	= ‘DOHERTL.V6DEV’
usrhlq	= ‘DOHERTL.V6CONF’
LANG	= ‘ENU’

PARSE – Key Instructions : Templates

- String Pattern Template
  - A literal or variable string pattern indicating where the source string should be split
  - Literal example

```
string = `  Parse the    blank-delimited   string'
parse var string var1 `-' var2 .
```

  - Variable example

```
dlm = `-'
parse var string var1 (dlm) var2 .
```

  - Result

var1 → '   Parse the   blank'
var2 → 'delimited'

PARSE – Key Instructions : Templates

- Positional Pattern Templates
  - Use numeric values to identify the character positions at which to split data in the source string
  - An absolute positional pattern is a number or a number preceded with an equal sign

```
          ----+----1----+----2----+----3----+----4----+
string = ‘Doherty            Liam           Australia  ’
parse var string 1 surname 20 chrname 35 country 46 .
surname -> ‘Doherty            ’
chrname -> ‘Liam           ’
country -> ‘Australia  ’
```

  - A relative 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----+
string = ‘Doherty            Liam           Australia  ’
parse var string 1 surname +19 chrname +15 country +11 .
surname -> ‘Doherty            ’
chrname -> ‘Liam           ’
country -> ‘Australia  ’
```

PARSE – Key Instructions : Templates

- RTC ISPF Client usage
  - RTC used numerical positioning to pull the value of “special” characters out of a returned JSON string. Then use those characters as variable substitution in subsequent parsing instructions:

```
Parse Var Result junk 'returnValue' var2 +23 Specials +6
Specials = Substr(Specials,1,6)

BLZSQBOP = Substr(Specials,1,1)
BLZSQBCL = Substr(Specials,2,1)
BLZCUBOP = Substr(Specials,3,1)
BLZCUBCL = Substr(Specials,4,1)
BLZTSLAC = Substr(Specials,5,1)
BLZDBQTE = Substr(Specials,6,1)

Parms = ''
Result = BLZDEMON('BLZC010' 'getApplId' Parms)
Parse Var Result APPLI_RC . BLZDBQTE'returnValue'BLZDBQTE':',
                 Result
```

What is a Compound variables

- A series of symbols (simple variable or constant) separated by periods.
- Made up of 2 parts – the stem and the tail.
- The stem is the first symbol and the first period. The symbol must be a name. Sometimes called the stem variable.
- The tail follows the stem and comprises one or more symbols separated by periods.
  - Variables take on previously assigned 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 variables 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
val = 3
say month.val             → March
```

- Easy way to reset the values of compound variables

```
month.   = ''
say month.6               → MONTH.6
```

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

```
drop month.
say month.6               → MONTH.6
```

Processing Compound variables

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

```
invitee.0 = 10
do i = 1 to invitee.0
  SAY ‘Enter the name for invitee’ i
  PARSE PULL invitee.i
end
```

Processing Compound variables

- Stems can be used with the EXECIO command to read data from and write data to a data set

```
sqlin.0 = 3
sqlin.1 = "  SELECT *"
sqlin.2 = "    FROM SYSIBM.SYSUSERAUTH"
sqlin.3 = "   WHERE GRANTEE = 'DOHERTL'"
"ALLOC F(SYSIN) NEW RECFM(F B) LRECL(80) SP(1 1) TRACKS"
"EXECIO * DISKW SYSIN (FINIS STEM sqlin."
```

```
"EXECIO * DISKR SYSPRINT (STEM cmdout. FINIS "
```

- Stems can also be used with the OUTTRAP external function to capture output from commands

```
x = outtrap(profile.)
Address TSO "PROFILE"
x = outtrap('OFF')
Parse var profile.1 . 'PREFIX(' BLZTSOPR ')' .
```

Processing Compound variables

- The tail for a compound variable can be used as an index to related data
- Given the following input data:

```
Symbol  Atomic#   Name        Weight
H       1         Hydrogen    1.00794
HE      2         Helium      4.002602
LI      3         Lithium     6.941
```

- The unique symbol value can be used as the tail of compound variables that hold the rest of the symbol's values

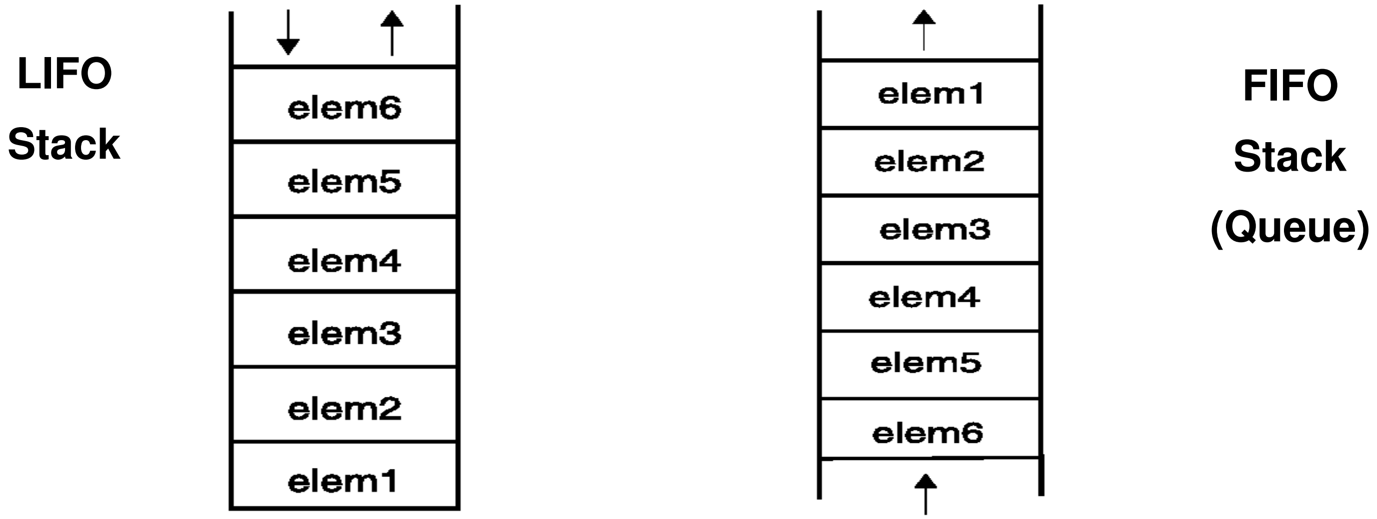
```
Address TSO "ALLOC F(INDD) DA('DOHERTL.EXEC(ATOMIC)') SHR"
"EXECIO * DISKR INDD (STEM rec. FINIS"
Do i = 2 To rec.0
  Parse Var rec.i symbol atomic#.symbol name.symbol weight.symbol
End
Say "Which atomic symbol do you want to learn about?"
Parse Pull symbol
symbol = Translate(symbol)
Say "The name of" symbol "is" name.symbol"."
Say "The atomic number for" symbol "is" atomic#.symbol"."
Say "The atomic weight of" symbol "is" weight.symbol"."
```

Less common code constructs used

- Data Stacks
- Interpret
- Sockets

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



Manipulating the Data Stack

- 3 basic REXX instructions
  - PUSH- put one element on the top of the stack

```
elem = 'new top element'
PUSH elem
```

  - QUEUE - put one element on the bottom of the stack

```
elem = 'new bottom element'
QUEUE elem
```

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

```
PARSE PULL top_elem .
```

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

```
num_elems = QUEUED()
```

Data Stacks usage examples #1

- Running SQL and displaying output

```
subSys   = 'DSNA'
sqlin.0 = 3
sqlin.1 = "   SELECT NAME"
sqlin.2 = "     FROM SYSIBM.SYSPLAN"
sqlin.3 = "    WHERE NAME LIKE 'DSN%'"
"ALLOC F(SYSIN) NEW RECFM(F B) LRECL(80) SP(1 1) TRACKS"
"EXECIO * DISKW SYSIN (FINIS STEM sqlin."
"ALLOC F(SYSPRINT) TRACKS SPACE(5,5) UNIT(VIO) NEW"

/* Run DSNTEP2 to run SQL */
"NEWSTACK"
queue "RUN PROGRAM(DSNTEP2) PLAN(DSNTEP10) LIB('DBAWK1.RUNLIB.LOAD')"
queue "END"
"DSN SYSTEM ("subSys")"
db2_rc = rc
"DELSTACK"

"EXECIO * DISKR SYSPRINT (STEM cmdout. FINIS "
Do co = 1 to cmdout.0
  say cmdout.co
End
"FREE F(SYSPRINT)"
"FREE F(SYSIN)"
```

Data Stacks usage examples #2

- Running DB2 commands and displaying output

```
Command = '-DIS DDF'

X = OUTTRAP('db2out.')
queue Command
queue "End"
Address TSO "DSN SYSTEM(DSNA)"
If rc /= 0 Then
Do
  /* Clear the command queue so we don't exit with an error */
  Do x = 1 to Queued()
    Pull oldStuff
  End
End
X = OUTTRAP('OFF')
Do i = 1 to db2out.0
  Say db2out.i
End
```

Data Stacks usage examples #3

- Reading a file, uppercase certain words, rewrite file

```
Address TSO "ALLOC F(GMLUP) DA('"ToDataset"("Member")') SHR"
Do queued();Pull;End       /*Insure a clean stack */
'EXECIO * DISKR GMLUP (STEM INPUT. FINIS'
Do i = 1 to input.0
   If Pos('<:ENTITY',input.i) <> 0 Then
   Do
      Parse var input.i '<:ENTITY' entity sysstr transtg
      If strip(sysstr) = 'system' | strip(entity) = '%' Then
         Upper input.i
      Else
      Do
         transtg = sysstr transtg
         Upper transtg
         input.i = '<:ENTITY' entity transtg
      End
      Queue input.i
   End
   Else
   Do
      Upper input.i
      Queue input.i
   End
End
'EXECIO' queued() 'DISKW GMLUP (FINIS'
```

Other Data Stack functions

- **MAKEBUF**
  - All elements added after a MAKEBUF command are placed in the new buffer
  - MAKEBUF basically changes the location the QUEUE instruction inserts new elements
- **DROPBUF**
  - Removes a buffer from 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
- **QBUF**
  - The QBUF command is used to find out how many buffers have been created
- **QELEM**
  - The QELEM command is used to find out the number of elements in the most recently created buffer
- **NEWSTACK**
  - protect data stack elements from being inadvertently removed by creating a new private data stack
- **DELSTACK**
  - Removes the most recently created data stack
- **QSTACK**
  - returns in the variable RC the number of data stacks (including the original stack)

Data Stacks

- **Advantages**
  - Can be used to pass data to external routines
  - Able to specify commands to be run when an exec ends
  - Can provide response 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, then read from stack into variables
  - Stack attributes and commands are OS dependent

- Try to use compound variables whenever appropriate. They are **simpler**.

Interpret

- Expression specified with the INTERPRET instruction is evaluated and then the resulting value is processed (interpreted)
- Given the following input data read from a file:
  `<cfg:InclCond>SHARE = "Seattle"</cfg:InclCond>`

```
Share = 'Seattle'

Address TSO "ALLOC F(INDD) DA('DOHERTL.EXEC(COND)') SHR"
"EXECIO * DISKR INDD (STEM rec. FINIS"
Address TSO "FREE  F(INDD)"

Parse var rec.1 "<cfg:InclCond>"cond"</cfg:InclCond>"

continue = 'N'
Cond = 'If 'Cond 'then continue = "Y"'

/* Execute the statement just created */
Interpret Cond

If continue /= 'Y' Then
  Exit 0
Else
  Say 'Running at SHARE Seattle'
```

Interpret

- Interpret Can provide powerful test and debugging capabilities

```
Address ISPEXEC
'VGET (BLZTRACE) SHARED'

Parse var BLZTRACE TraceOn '(' TraceMod ')'
TraceCmd = ''
If Substr(TraceOn,1,5) = 'TRACE' Then
Do
  modname = 'BLZAWI'
  If TraceMod = 'ALL' | TraceMod = modname Then
  Do
    Say "*** Tracing activated for "modname "on "Date('N') ||,
        " at "Time()" ***"
    Select
      When (TraceOn = 'TRACE?I') Then TraceCmd = 'Trace ?i'
      When (TraceOn = 'TRACEI')  Then TraceCmd = 'Trace i'
      Otherwise NOP
    End
  End
End
Interpret TraceCmd
```

REXX Sockets

- **Requirement for Socket protocol**
  - RTC needed to remove it's dependency on cURL
  - We used a Java daemon, running on a Port, to serve requests to an RTC server using the cURL request
  - In replacing cURL, Sockets were the obvious candidate
  - There are already plenty of materials, also from SHARE, on REXX sockets, just google "REXX Sockets".
  - See the z/OS Communications Server: IP Sockets Programming interface Guide and Reference:

    http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.hala001/toc.htm

  - The ISPF Client would build a request for an internal API and send it to the daemon running on a port, via the socket.
  - The daemon would return the result in a JSON string read from the socket

REXX Sockets

- Open Socket

```
Host = '127.0.0.1'
Port = '7333'
/*--------------------------------------------------------------*/
/* Initialize the socket                                        */
/*--------------------------------------------------------------*/
Result = Socket('Initialize',USER)

/*--------------------------------------------------------------*/
/* Establish the socket                                         */
/*--------------------------------------------------------------*/
Result = Socket('Socket', 'AF_INET', 'STREAM', 'TCP')

/*--------------------------------------------------------------*/
/* Set Socket descriptor                                        */
/*--------------------------------------------------------------*/
Sdesc = sres

/*--------------------------------------------------------------*/
/* Enable EBCDIC-ASCII conversion                               */
/*--------------------------------------------------------------*/
Result = Socket('SetSockOpt',sdesc,'SOL_SOCKET','SO_ASCII','On')

/*--------------------------------------------------------------*/
/* Connect the socket                                           */
/*--------------------------------------------------------------*/
Result = Socket('Connect', sdesc, 'AF_INET' Port Host)
```

REXX Sockets

- Send Request

```
Command = Method Service "HTTP/1.1" ||CRLF
Command = Command"User-Agent: ISPF" ||CRLF
Command = Command"Host: "BLZLPBAK":"Port ||CRLF
Command = Command"Accept: */*"||CRLF
Command = Command"X-Secret-Key:"Key||CRLF

if (Method = "POST") Then
Do
  Command = Command"Content-Type:application"
  Command = Command"/x-www-form-urlencoded"||CRLF
  Command = Command"Content-Length:"lgParameters||CRLF||CRLF
  Command = Command||Parameters
End
Command = Command||CRLF
Result = Socket('Send', sdesc, Command,'')
```

REXX Sockets

- Receive response

```
/*-------------------------------------------------------------*/
/* Receive response in a non-blocking socket                   */
/*-------------------------------------------------------------*/
Result = Socket('Fcntl',sdesc,'F_SETFL','NON-BLOCKING')
mask = 'READ 'sdesc
Result = Socket('SELECT',mask, Timeout)
numTotal = 0;receivedTotal = '`;messagelength = -1
Do forever
  fc = SOCKET('READ',sdesc,'8000')
  parse var fc read_rc num_read_bytes received_string
  /* Complete message involved */
  if read_rc <> 0 | numTotal = messagelength then leave
  else
  do
    numTotal = numTotal + num_read_bytes
    receivedTotal = receivedTotal||received_string
    if messagelength = -1 Then
    do
      parse var received_string . "Content-Length: "contentlength (CRLF) .
      headerlength = Length(received_string)
      Do While received_string <> ''
        Parse Var received_string Line (LF) received_string
        if length(Line) <= 1 Then Leave
      End
      headerlength = headerlength - Length(received_string)
      messagelength = headerlength + contentlength
    end
  End
End
```

REXX Sockets

- Close Socket

```
Result = Socket('CLOSE', sdesc)
parse var Result src sres
If src <> 0 then
  return

Result = Socket('Terminate',USER)
```