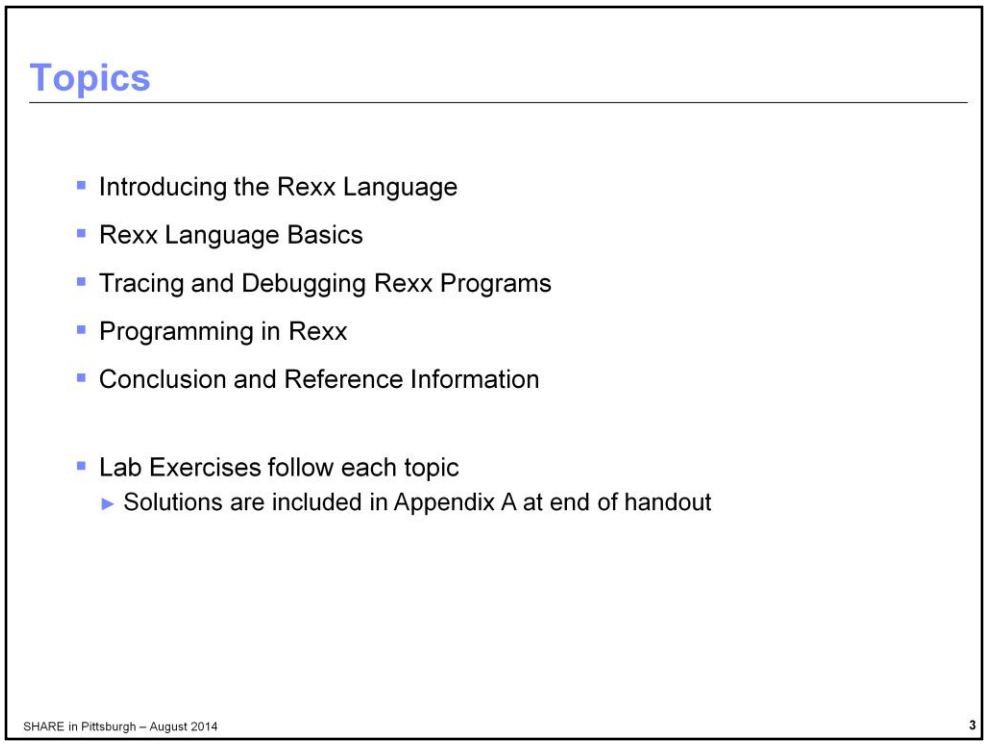
Version 2014.d.1



Welcome to the Introduction to Rexx Workshop. This workshop will cover a series of topics about the Rexx language. At the end of each topic, there will be exercises that use the concepts covered in the charts and text for that topic.

The solutions for the exercises are in Appendix A. This is not a graded class or lab so you are free to refer to the solutions if you would like or if you need help with any of the exercises.

Also, if you need assistance with any of the lab material or exercises, please raise your hand and someone will come to assist you.

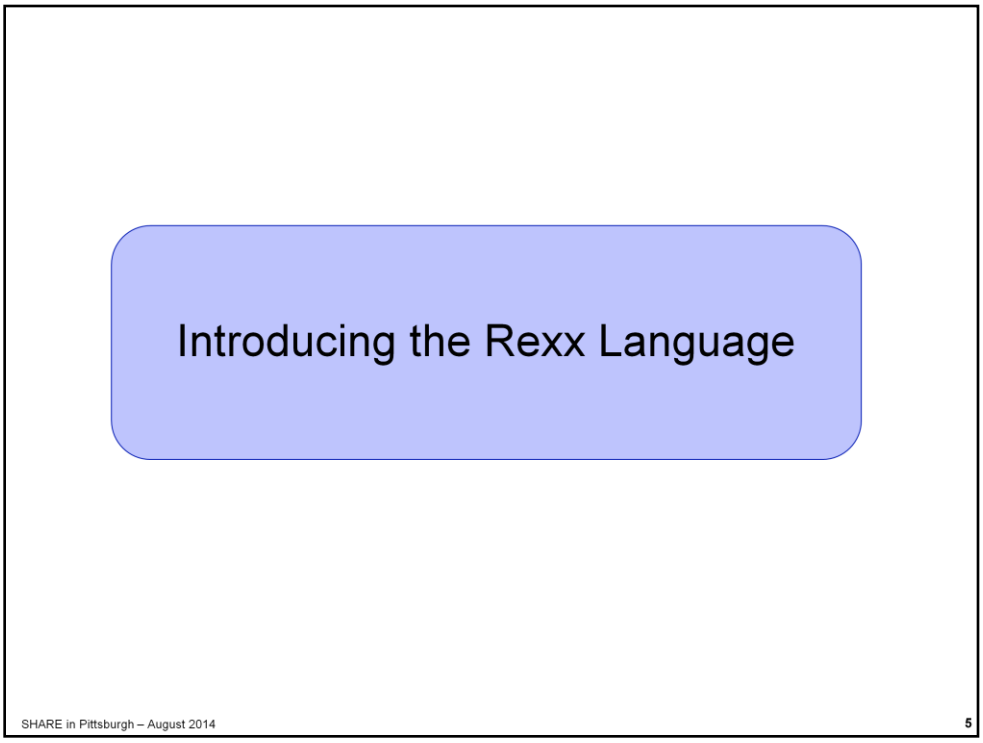
Lab Exercises (following each topic):

1. Run an existing Rexx program to create temporary disk space
2. Write a program to accept an input argument, prompt for data, and display results
3. Trace and Debug existing Rexx programs
4. Write a program to obtain z/VM CP level information
   (issues commands and Diagnose 8)
5. Write a program using a subroutine to issue CMS commands and Pipes to query accessed disks

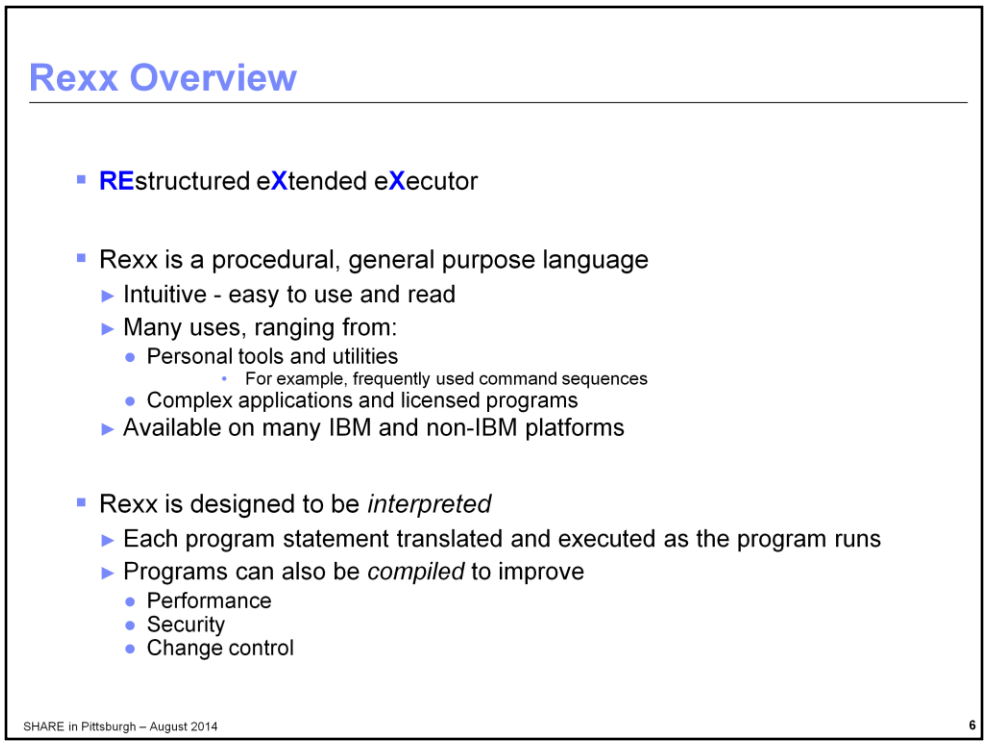
These are the exercises that you will be doing today.

There are 2 PDF files on your desktop that you might find helpful as you work on the exercises:

- REXX Reference will help you with statement syntax and usage
- Appendix H is a summary of XEDIT commands that might be helpful when you are writing and updating your Rexx programs



This next section introduces the Rexx Language and explains how to create and execute Rexx programs on z/VM and TSO systems. It also includes some z/VM hints to help you with today's exercises.



REXX is a nickname for Restructured eXtended eXecutor. The Rexx language was designed to be intuitive, easy to read, and easy to follow a program's logic.

Rexx programs can be very simple, ranging from personal tools and utilities such as "scripts" of commands that you might run every day to set up your system or run test cases, to very complex programs that might be part of a licensed product. Rexx is available on many different IBM and non-IBM platforms.

The Rexx language is designed so that programs can be *interpreted ;* this means that each statement is translated and executed as a Rexx program runs. Rexx programs may also be *compiled* to improve performance and security, as well as to prevent changes to source code. The Rexx Compiler is a Licensed Program. The programs for your lab exercises will be interpreted, not compiled.

Rexx Overview (cont.)

- Few restrictions on program format
  - Indentation
  - 1 or more clauses on a line
  - /* comments can be anywhere and any length */
  - *Implied* semicolon delimiters at end of lines
  - Comma (,) as a continuation character
- Nothing to Declare !
  - Implicit declarations take place during execution

Of course, like any language, Rexx has syntax rules. However, Rexx is very flexible regarding the formatting of your programs:

• You may indent statements or groups of statements however you would like to; there are no column restrictions for Rexx program statements.

• You may include multiple statement clauses on a single line.

• Comments can be placed anywhere in a Rexx program, and can be any length. You can place comments at the end of a Rexx statement line, or on lines all by themselves. You can also create comments that span multiple lines. Remember to always close your comment with a '*/' . If you forget to do this, you will get a surprise when you execute the program because most of the Rexx statements will appear to be part of one very very long comment!

• There is no need to delimit Rexx statements or lines in your program. Rexx places an "implied" semicolon at the end of each line.

• Sometimes a statement in a Rexx program will not fit on one line. When this happens, you may use a comma (',') to indicate that the statement continues on the next line.

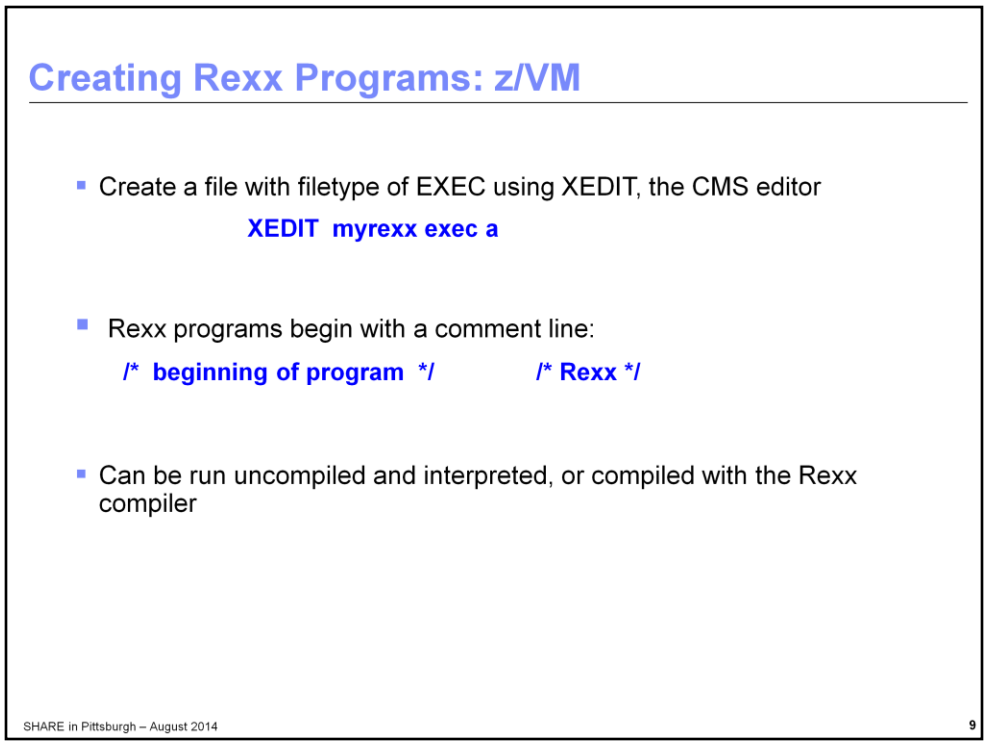
• In Rexx, variables and data areas do not need to be declared. They are declared implicitly based on their usage during program execution.

Rexx Platforms

- IBM Platforms
  - VM
  - TSO/E (z/OS)
  - VSE
  - AIX
  - OS/2
- Object Rexx
  - Object-Oriented Rexx supporting many utilities for a UNIX-type environment, including Linux for System z
- Regina Rexx
  - Rexx interpreter ported to most UNIX platforms, including Linux
- NetRexx
  - Blend of Rexx and Java; compiles into Java classes
- Language concepts are the same on all platforms
  - Minor differences such as file names and structure
  - Operating system-specific tools that support Rexx

(See references page for website information)

As mentioned earlier, Rexx is available on many different platforms.

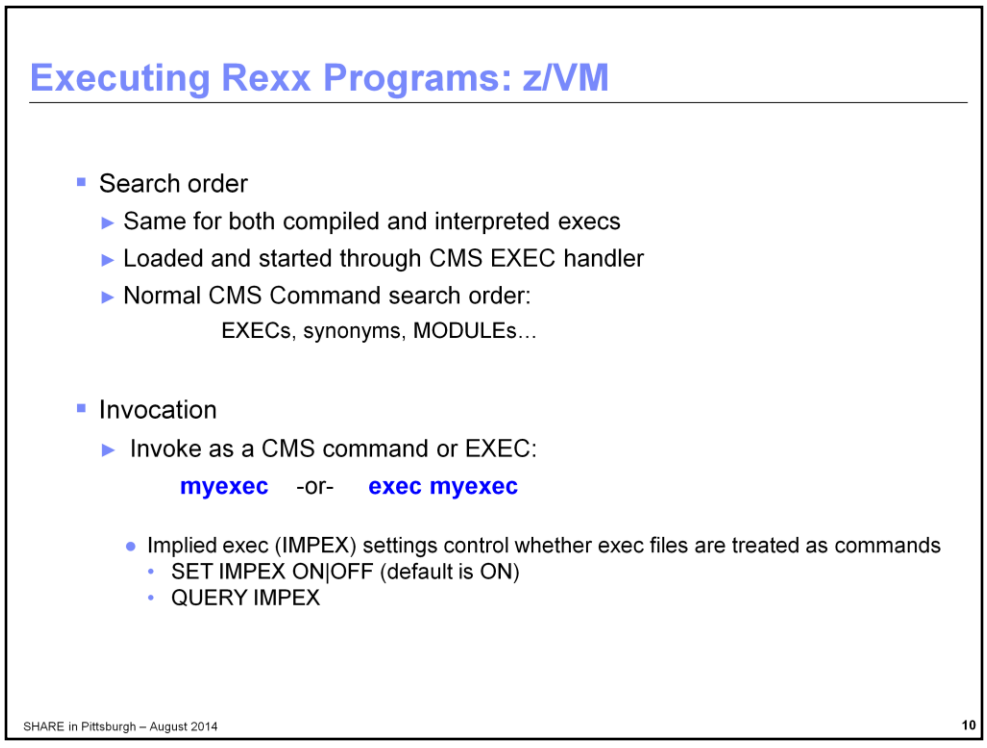


Rexx programs are contained in files on a z/VM users disk, known as a minidisk. To create a Rexx program on z/VM, you use the z/VM – CMS editor called XEDIT. If you invoke the editor for a file that does not exist, XEDIT will create a new file on the designated disk.

Files on a z/VM disk are identified by their:

- name (filename or fn)
  - This is the name of your Rexx program (in the example on the chart, the name of the program is MYREXX)
- type (filetype or ft)
  - A filetype of EXEC indicates that this file contains a Rexx program
- mode (filemode or fm)
  - Indicates the disk that the file resides on. A z/VM user will typically have several disks with file modes a-z. Usually, and for this lab, the 'A' disk is the working disk to create and edit files and programs.

The first line of a Rexx program must be a comment. The comment may be blank or contain any text that you choose. This is a good place to include a description of what the Rexx program does.



z/VM uses a defined search order to execute commands and programs. When a command is issued, if a Rexx program with a name (the filename) that matches the command is found on any disk that the user has accessed, the Rexx program will be executed. The disks are searched in alphabetical order of their filemodes, starting with 'A'. If a matching program is not found on an accessed disk, the command will be treated and executed as a system command.

You may invoke a Rexx program by simply typing its name on the command line. If you want to explicitly tell z/VM that the command you are issuing is the name of a program, you can preface the program name with the keyword EXEC.

Creating and Executing Rexx Programs: TSO/E

- REXX exec can be a sequential data set or a PDS member
- TSO/E EXEC command to invoke a REXX program or a CLIST
- Three ways to use the EXEC command:
  - Explicit execution:
    EXEC dataset(member) 'parameters' operands
  - Implicit execution:
    membername parameters
  - Extended implicit execution:
    %membername parameters
- Search includes:
  //SYSEXEC DD concatenation
  then
  //SYSPROC DD concatenation for **membername** on the command line

In TSO/E, the **EXEC** command is used to invoke a Rexx program or a CLIST. This may be done explicitly, or implicitly as shown in the above chart.

Helpful Hints for Exercises

- List Files on A-disk: **FILELIST * * A** or… **LISTFILE * * A**
- XEDIT a file
  - from command line: **Xedit *<filename> <filetype> <filemode>***
  - from prefix area on FILELIST Screen, PF11 or :

```
x PROFILE  EXEC  A1 V  75  74  1  09/17/07 15:48:18
```

- XEDIT Prefix area commands:

  a add (insert) a single line to the file
  d delete a line (**d5** deletes 5 lines)
  m move a line (**f** following or **p** preceding)
  c copy a line (**f** following or **p** preceding)
  mm…mm block move, **dd…dd** block delete, **cc…cc** block copy

- Leaving XEDIT:
  - Save changes: **FILE**
  - Quit (restore file without changes): **QQUIT**

The exercises will be done on a z/VM system. This chart offers some hints in case you are not familiar with z/VM. We will be happy to help you with any questions.

Helpful Hints for Exercises (cont.)

- Screen execution modes
  - **CP Read**
    - CP is waiting for a command
  - **VM Read**
    - CMS is waiting for a command
  - **Running**
    - System is ready for commands or is working on some
  - **More …**
    - More information than can fit on the screen is waiting to be displayed)
      - Clear screen manually or let CP clear after x seconds determined by TERM command setting
  - **Holding**
    - Waiting for you to clear screen manually
  - **Not Accepted**
    - Too many commands in buffer; wait for executing command to complete)

When you log on to z/VM, you will always see one of these "status" indicators in the lower right hand corner of your session.

Logging on to the z/VM Lab System

- 3270 Session
- Userid
- Password

Logon details will be provided in the lab room during the session.

Exercise 1: Create Temp Disk Space

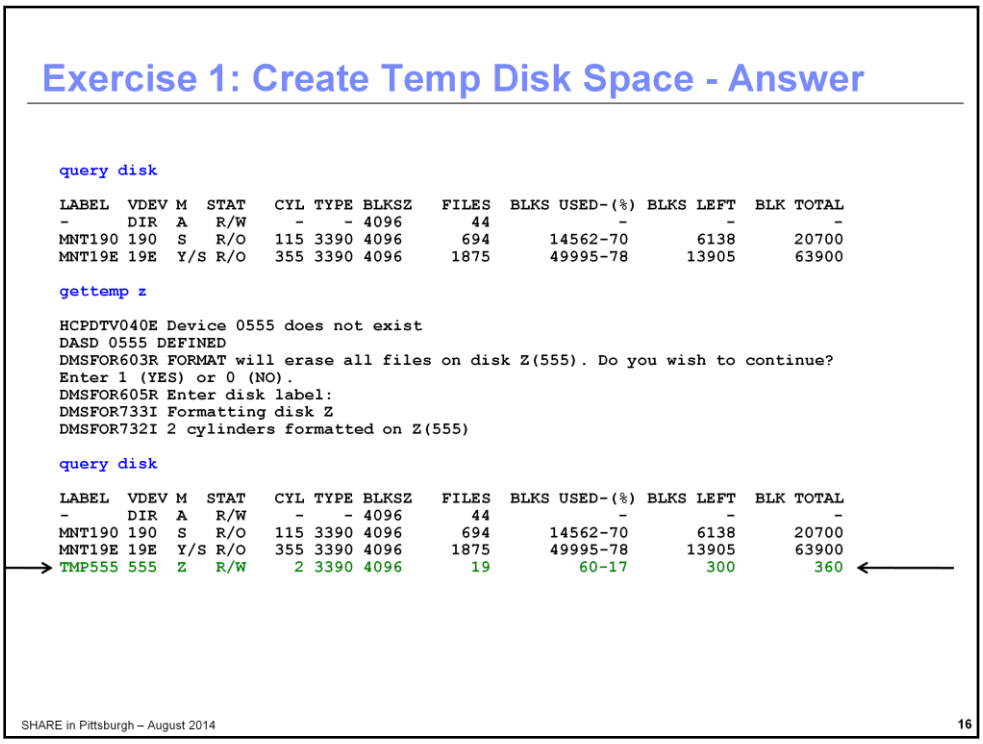
1. **LOGON** to your VM lab userid
2. Issue command **QUERY DISK** to see which disks are accessed
3. Run existing exec **GETTEMP *mode*** (*mode* is input parameter) to:
   - create a temporary disk at filemode *mode*
   - copy existing EXEC programs from a-disk to new temp disk
     - Note: – ***mode*** can be a letter from ***b - z*** representing an unused disk mode
4. Issue **QUERY DISK** again – notice new disk at *mode*
5. Issue command **FILELIST * * *mode***
6. Run **GETTEMP** again with mode ***a***
7. Issue **QUERY DISK** again – notice new disk at mode ***a***
8. **LOGOFF**

Now it is time for our first Exercise. This exercise will demonstrate how to run a Rexx program and observe the changes that result.

You will not be writing a Rexx program (yet), but you will running a program that is already on your A-disk. This will also familiarize you with the z/VM environment that you will be using throughout this lab.

Follow the steps listed above. ***mode*** described in Step 3 is a filemode, as was discussed on previous charts. You specify ***mode*** as input to the Rexx program named GETTEMP. The first time you run GETTEMP, use a mode other than **a**. The second time, you will specify mode **a** and observe the differences in the output.

When you complete this exercise, please LOGOFF. When you log back on, your userid's configuration will be reset for the rest of the lab exercises.



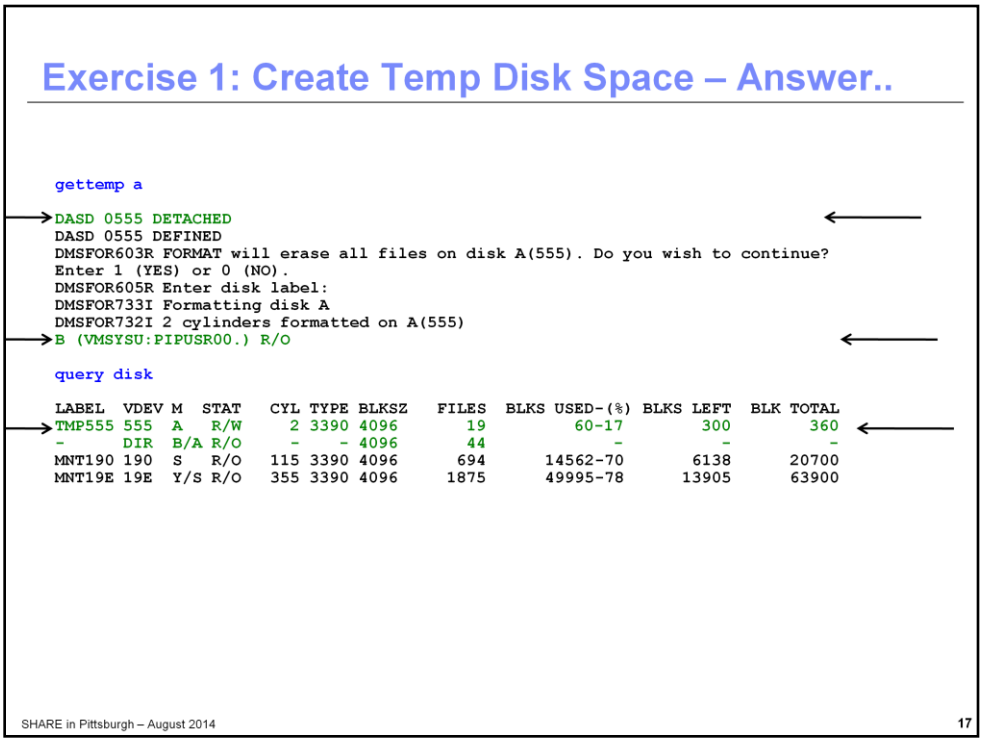
This is the result from Steps 2-4:

• Issue command **QUERY DISK** on the command line. Here you see the disks that your userid has accessed (your actual result may be a little bit different).

• Issue **GETTEMP Z** to run program GETTEMP with an input mode of **Z**. What you see are messages indicating the results of the commands that were issued from the program. A new disk must be formatted when it is created; the **FORMAT** command issued by the program prompts for input; the responses to the prompts are included in the program (see the actual program in Appendix A).

•After running GETTEMP, you see new disk **TMP555** at file mode **Z**. Remember, **Z** is what was specified as the input mode for the new disk.

We don't show the output for step 5, but if you issued the **FILELIST * * Z** command (or whatever mode you specified to GETTEMP), you saw a group of files with filetype EXEC on the new temporary disk that was created.

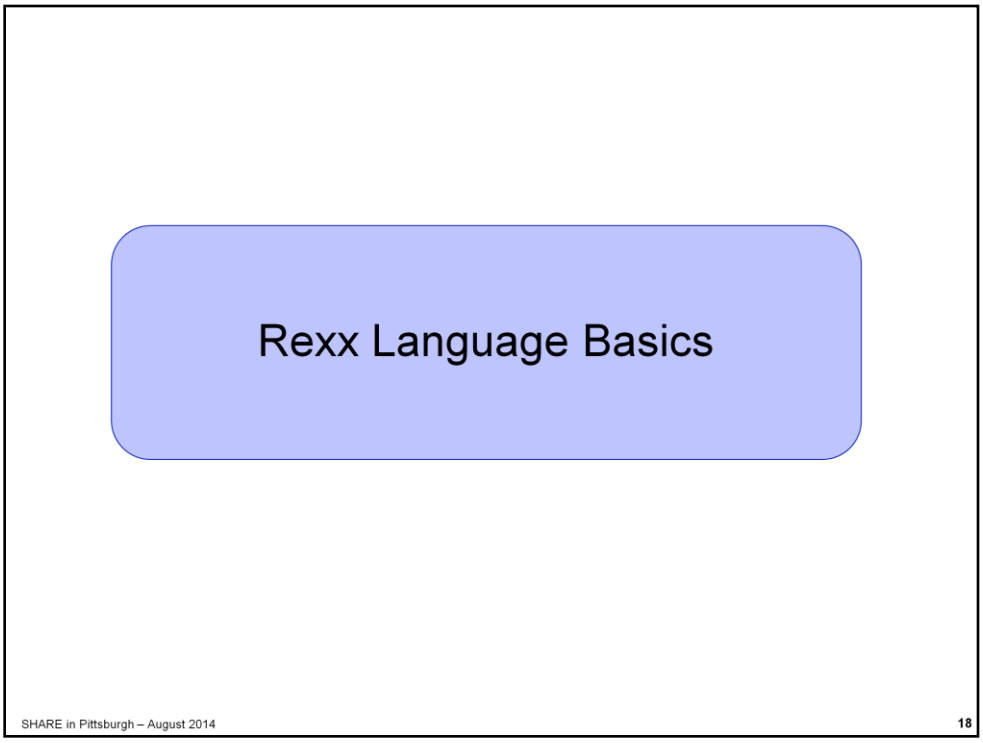


This is the output from Steps 6-8:

• When GETTEMP is re-issued with mode **A**, it checks to see if there is already a disk defined at the address of the new disk. If so, it DETACHes (removes) the disk from the userid's configuration and defines a new disk. We also see the current A-disk moved to mode **B**. Even though we are creating a new disk at mode **A**, we still need the files that are on the current A-disk so we preserve access to them at mode **B**.

• After running GETTEMP, **QUERY DISK** shows us the new disk defined at mode **A**, and the disk that was previously at mode **A** re-accessed at mode **B**.

Step 8 is to **LOGOFF**. This will remove the new temporary disk that we created with GETTEMP as well as the files that were copied to it. When you log back on for the next exercise, your A-disk will be back at mode **A** and will still have all of the files that it previously had.



This next section explains about Rexx syntax – about strings, operators, numbers, variables, and the keywords for input and output.

Rexx Language Syntax

- Case **In**sensitivity
  Pittsburgh is the same as **p**ittsburgh
  - specific support for upper and lower case is provided
  - cases in quoted strings are respected
- **All** Rexx programs must begin with a comment

```
/* This is a comment */
```

- Long lines are common
  - Continuation with commas

```
say 'This text is continued ',
       'on the next line'
```

  - May wrap as a long single line *(but don't do this)*

```
say 'This text is continued
       on the next line'
```

SHARE in Pittsburgh – August 2014 19

Rexx was developed to be a very "forgiving" language. In contrast to Assembly Language, where the case and placement of every character is very precise, Rexx's format is easy. The biggest requirement, in fact, is that the program begin with a comment string. An example of a comment is shown in the middle of this slide, beginning with "**/***" and ending with "***/**".

Names of constants and variables in Rexx are not case sensitive. As shown above, variable *Budapest* is logically equivalent to variable *budapest*. Case is automatically respected when text is inside quoted strings "**..**" or '**..**' – although this setting can be changed.

Long lines are common in Rexx. If a line is going to go past the end of your terminal window, use the comma '**,**' to continue the line of code on the next line of your text editor.

Rexx Strings

- Literal strings: Groups of characters inside single or double quotation marks

```
"Try a game of blackjack",'and beat the odds!'
```

- Two " or ' indicates a " or ' in the string

```
'Guess the dealer''s top card'
"The dealer""s card is an Ace"
```

- Hexadecimal strings: Hex digits (0-9,a-f,A-F) grouped in pairs:

`'123 45'x` is the same as `'01 23 45'x`

- Binary strings: Binary digits (0 or 1) grouped in quads:

`'10000 10101010'b` is the same as `'0001 0000 1010 1010'b`

Strings in Rexx may be contained in either single quotes '..' or double-quotes ".." – the only rule is that you must end a string with the same quote type as that with which you began the string. A string opened with " and closed with ' will produce an error.

If you want to use quote characters inside a string opened and closed by that same quote character, use two of that character in a row to do so.

Hexadecimal and binary strings are automatically grouped in Rexx. This allows for some forgiveness in data entry.

Operators & Expressions

- String Expressions

(blank) "Three" "Rivers" --> "Three Rivers"
|| 'Pitts'||'burgh' --> 'Pittsburgh'

(abuttal) **abc** = 'Alle'
abc'gheny' --> 'Allegheny'

- Arithmetic Expressions

\+ - * / % (int division) // (remainder)
** (power) Prefix - Prefix+

SHARE in Pittsburgh – August 2014 21

Strings may be combined in multiple ways. Two strings next to each other, separated only by a space character, are logically equivalent to a single string with a space character in the middle of it.

The concatenation operator || can be used to combine two or more strings, as shown in the second example above.

This applies when using literal strings as well as variables. In the third example, we see a variable *abc* defined with the first half of the name of a famous Budapest avenue. We see that this is then combined with the literal string that completes its name.

Rexx follows the Order of Operations when processing math, and the operator symbols it uses are constant with most other high-order languages. Distinct to Rexx is using **%** for integer division, // for remainders, and two asterisks ****** for exponents.

Input and Output

- **say [expression]**
  - writes output to the user's terminal

```
say 'Five Euros equals ' ,
      5 * 1.35 'USD'
```

- **pull**
  - prompts for input from the user

```
pull rate
say 'Five Euros equals' 5 * rate 'USD'
```

- **parse arg**
  - collects arguments passed to a Rexx Program
    - Invoke program: `EXAMP input1 dataX moreData`

```
parse arg A1 A2 A3
say A1 A2 A3
```

    - Result:

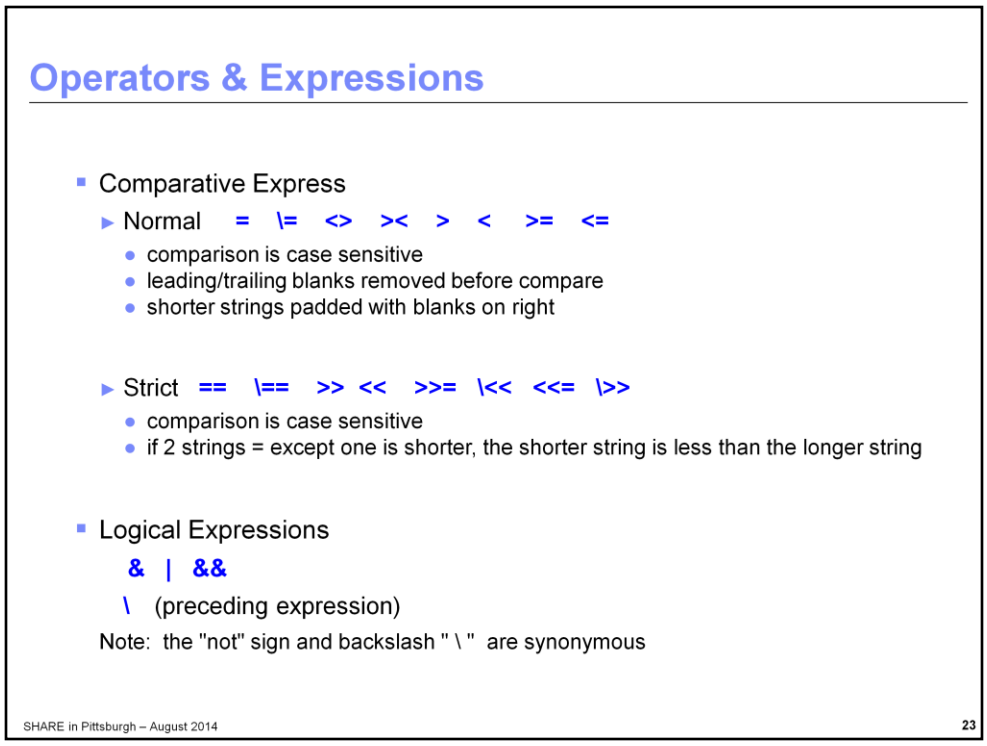
```
input1 dataX moreData
```

SHARE in Pittsburgh – August 2014 22

Output in Rexx is easy. The **say** keyword will print the rest of that line of code onto the terminal screen. **Say** can be used with literal strings, numbers, variables, or even mathematical operations.

There are two ways to collect input to a program. The first is the keyword **pull**. This keyword stops your Rexx program and waits for a user at a keyboard to enter data.

The other means is via the command line. You used this in Exercise 1 with GETTEMP. **Parse arg** is a pair of keywords which, when used together, will collect information from the command-line invocation of the program. In the example above, we provide three strings after we type in a program named EXAMP. These are assigned the values starting with *A1* via the **parse arg** keywords.



Comparisons between numbers, variables, or literal strings can be done either in a "normal" or "strict" format. In both instances, the comparison is case-sensitive. With a normal compare, leading and trailing blanks are removed, and shorter strings are padded on the right for the comparison only. For a strict compare, the data is compared precisely as presented, with no padding or removal of blanks.

There are four logical expressions in Rexx: the ampersand character **&** represents AND (**a & b**), the single vertical bar **|** represents OR (**a | b**), and two ampersands **&&** represent XOR (**a && b**). A NOT or negation character can be represented by either the backslash \ or the "not" sign ¬

Numbers

- A Rexx character string that includes 1 or more decimal digits with an optional decimal point
  - May have leading and trailing blanks
  - Optional sign + or -
  - An "E" specifies exponential notation
    - Be careful with device addresses such as 1E00 (use quotes)
- Precision in calculations may be controlled by the NUMERIC DIGITS instruction
  - Default is 9 digits
- Examples (could also be enclosed in quotes):

  12 -17.9 + 7.9E5

Because Rexx does not do data typing, all numbers are automatically strings. Rexx thinks a string is a number when that number starts with a digit or + or – sign. It may or may not contain a decimal point. If using exponential notation, the uppercase E represents this. This can cause confusion with certain z/VM device addresses (for example 1E00), so exercise caution in manipulating fields like that.

Math precision in Rexx is controlled by a **NUMERIC DIGITS** instruction. By default, mathematical operations are calculated out to 9 decimal places. There is no hard coded upper bound; it is constrained only by the size of your virtual machine or address space.

Variables

- Data known by a unique name whose value may change
- Variable names
  - **NOT** case sensitive
  - **Cannot** begin with a digit 0-9
- Defined by assignment (give it a value)

  population = 184627
- Variables with no assigned value will have the uppercase variable name as its initial value
- Special variables: **rc, result, sigl**
  - may be set automatically during program execution

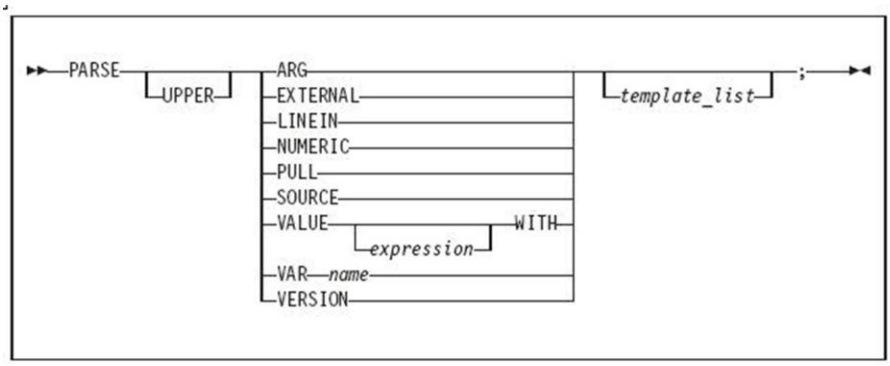
A variable in Rexx is the same as a variable in other high-order programming languages: it is a unique name and storage space for data that may change over time. Variable names are NOT case sensitive, and they cannot begin with a number.

Variable typing in Rexx is done by assignment; there is no need to declare a variable as an integer or character.

If you use a variable in Rexx before you have actually assigned a value to it, its default value will be its own name uppercased. So in the slide above, the variable *population*, before assigning it a value of 184627, would have a value of "POPULATION".

There are a few special variables in Rexx: *rc*, *result*, *sigl*. These variables are set based upon programmatic operations and events inside your Rexx program or your virtual machine. This means that any data you have stored in these variables may be overwritten. Exercise caution if using these special variable names in your program.

Parsing Strings

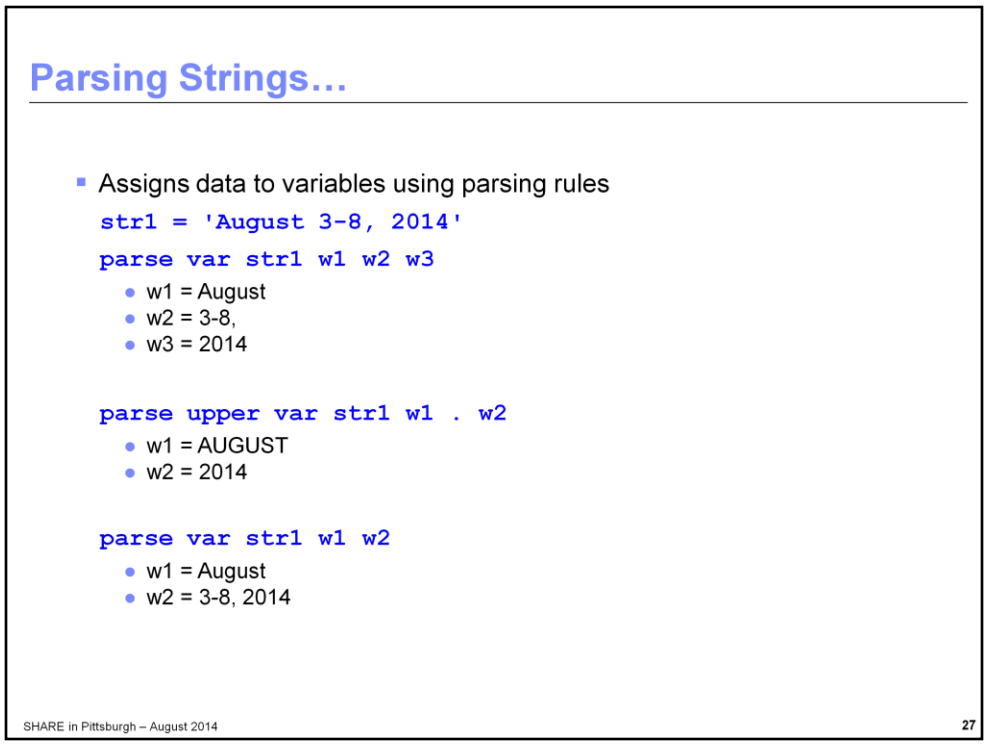


- Parse Arg – takes data passed into exec or internal routine
  - (see example on "Input and Output" chart)
- Parse Var – parses variable into other variable(s)

We previously saw the **PARSE** instruction as part of **parse arg**, our keywords for pulling input from EXEC command-line invocation. **PARSE** has a few different keywords, in fact, as we see on this slide. **Parse Pull**, for example, is the expanded form of the keyword **pull** – they mean the same thing. **Parse Var** will parse an existing variable into one or more variables – this is useful for manipulating longer strings of data. We will see examples of this on the next few slides.

Parse Value takes an expression and stores it into one or more variables. We will see examples of this near the end of the presentation.

Finally, any **PARSE** keyword can also be a **PARSE UPPER** – for example, **parse upper arg**. This will automatically convert inputted text into upper-case value.

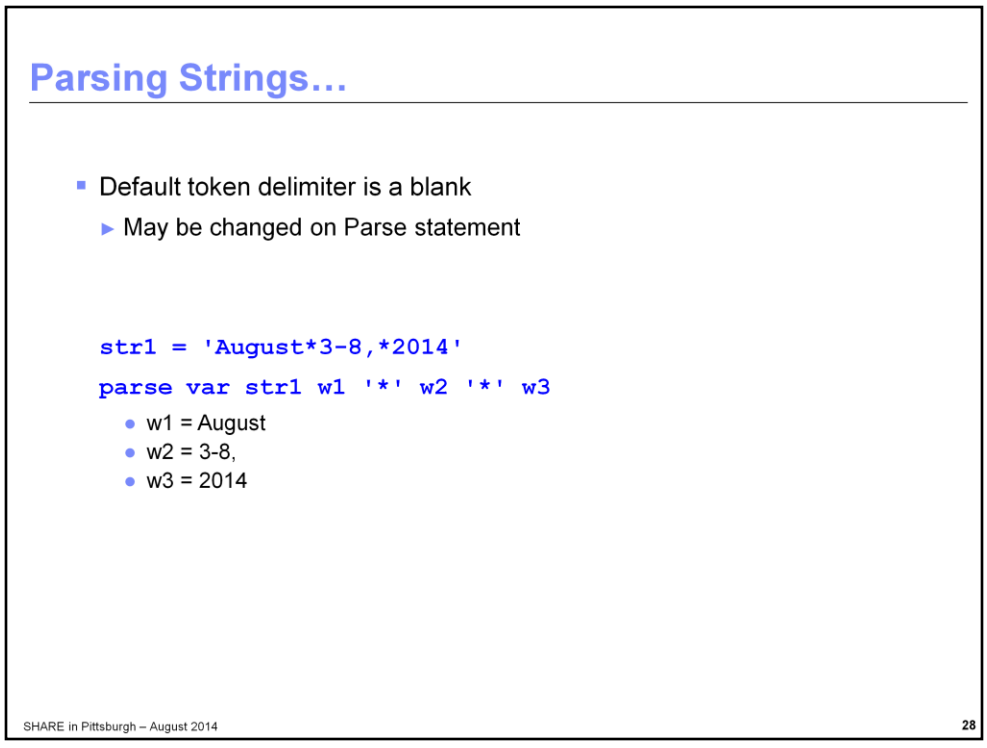


Here are three examples of parsing a single string, called *str1*.

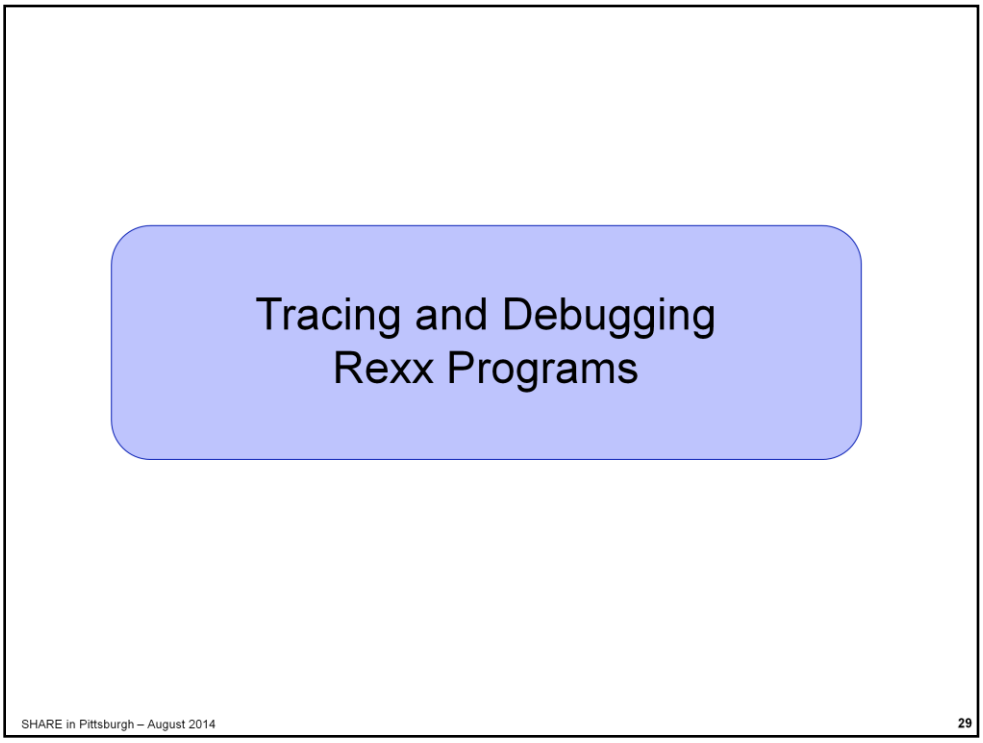
In the first example, we use **PARSE VAR** to break a longer string into three smaller strings, and we place the results in three variables: *w1, w2,* and *w3*. Because our processing is blank-delimited, we take all text in a token and insert it into one of the target variables. This includes the comma in *w2*.

In the second example, we uppercase the text (see *w1*), and we provide a period or full stop character **"."** in place of one of our target variables. This tells Rexx to "skip" the next token. As a result, *w2* now contains "2014" inside of the "3-8," token.

In the third example, we see that only two target variables have been provided – *w1* and *w2*. The first token is inserted into *w1*, as we would expect. Our *w2* variable now contains every token that is left inside the original string.



Of course, your records may not be delimited by a blank or space character " ". This example demonstrates how to use **parse var** to parse a variable using other tokens, such as an asterisk **'*'**.



This section covers tracing and debugging functions that are built into Rexx and can be very helpful when testing and debugging Rexx programs.

Tracing

- **Trace All** - clauses before execution
- **Trace Commands** - commands before execution. If the command has an error, then also displays the return code
- **Trace Error** - any command resulting in an error after execution and the return code
- **Trace Failure/Normal** – default setting, any command with a negative return code after execution, and the return code
- **Trace Intermediates** – Trace All, plus intermediate results during evaluation of expressions and substituted names
- **Trace Labels** - only labels passed during execution
- **Trace Off** - traces nothing and resets options
- **Trace Results** – Trace All, plus results of an evaluated expression and values assigned during PULL, ARG, and PARSE instructions
- **Trace Scan** – Trace All, but without the clauses being processed

The **TRACE** statement has several options to trace different aspects of a Rexx program and provide various levels of detail. **TRACE** statements may be placed anywhere in a Rexx program; you can trace an entire program or any part of it.

Tracing (cont.)

- output identifier tags:
  - *-* source of a single clause
  - >>> result of expression
  - >.> value assigned to place holder
  - +++ error messages

- prefixes if TRACE Intermediates in effect:
  - >C> data is compound variable
  - >F> data is result of function call
  - >L> data is a literal
  - >O> data is result of operation on 2 terms
  - >P> data is result of prefix op
  - >V> data is contents of variable

Symbols are provided on each line of trace output. These symbols indicate what kind of trace data is being shown.

Tracing (cont.)

- Prefix Options **!** and **?** modify tracing and execution
  - **?** controls interactive debugging
    TRACE ?Results
  - **!** inhibits host command execution
    TRACE !C causes command to be traced but not processed
- CMS command **SET EXECTRAC ON** allows you to switch tracing on without modifying the program
- **TS** and **TE** immed commands turn tracing on/off asynchronously

There are some special options that can be used to modify tracing and program execution. You can trace interactively by specifying **?** before the trace option, causing the program to stop at every trace step. This is very powerful; Rexx statements and commands may be issued while the program is stopped, allowing checking and even setting of data values and variables.

! can be used to trace commands that are issued in the Rexx program without the command actually being processed.

If you have long-running Rexx program, you can use asynchronous commands to turn tracing on and off while the Rexx program is running.

Tracing - Example

- Program

```
/* Trace Sample Program */
Trace Intermediates
number = 1/7
say number
```

- Output

```
 3  *-* number = 1/7
>L>   "1"
>L>   "7"
>O>   "0.142857143"
4  *-* say number
>V>   "0.142857143"
0.142857143
```

SHARE in Pittsburgh – August 2014 33

This example shows "**Intermediates**" tracing of a very short Rexx program:

- The first line of the trace output shows the entire statement on Line #3
- The next 2 lines show the literals "1" and "7"
- Next, we see the result of the division operation on the literals "1" and "7"
- Line 4 of the program is a **SAY** statement to display variable *number*
- The trace shows the value contained in variable *number*
- Finally, we see the output of the program (also the contents of variable *number*)

Exercise 2: Say, Pull, & Passing Parameters

- Assume a card deck with suits of Hearts, Diamonds, Clubs, and Spades
- Write a Rexx program to:
  - **pass in** 1 of the 4 suits as an argument
  - **prompt** for a number from 2-10
  - **display** the number and the suit in the format:

```
'Your card is a 10 of Hearts'
```

- **Run** the program with different suits and numbers

Here is Exercise 2. Spend some time working on this exercise; when you feel comfortable with your answer, you may move on to the next section. This exercise should use multiple concepts you learned in this section, but this is not a long program – perhaps 5 lines. There is no need to insert additional error-checking into the logic of your program.

As a reminder, the answer to this exercise is in Appendix A. Because this lab is not graded, you are welcome to study the answer at any time.

Exercise 3: Tracing and Debugging

The following Rexx Programs are on your VM A-disk:

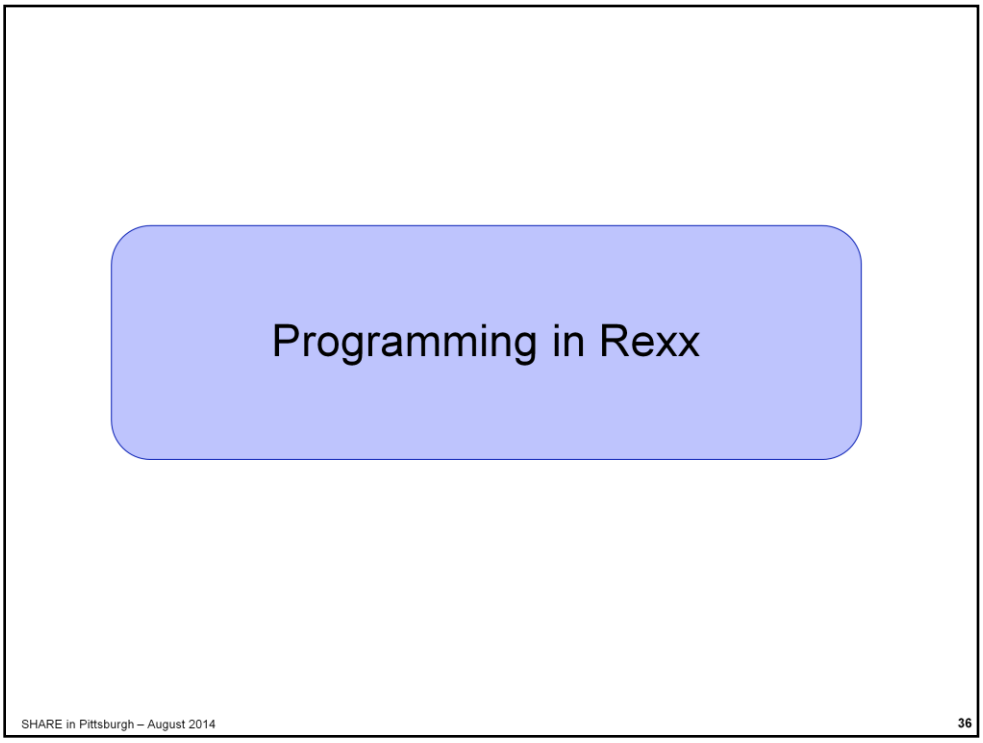
- REXXEX3A.EXEC
- REXXEX3B.EXEC

There is something wrong with each program

- Using the TRACE instruction, debug each problem
- Fix the code so that it functions properly

In this exercise, you will use Tracing to debug 2 Rexx programs that are already on your A-disk. Once you figure out where the bugs are, fix the programs so they run correctly.

As usual, if you have any questions let us know and we will be there to help.



The next section deals with more advanced Rexx programming concepts – compound variables, loops, if-then statements, and other logical constructs. It will also cover subroutines, the Rexx built-in functions, Pipelines, and VM-specific Rexx programming techniques.

Symbols and Stems

- Constant symbol starts with a digit (0-9) or period:

  77 .123 12E5

- Simple symbol does not start with a digit and does not contain periods:

  ABC ?3

- Compound symbol contains at least one period, and at least 2 other characters
  - **Stem** (up to 1st period), followed by **tail**

    ABC.3 Array.i Total.$name x.y.z

We have already covered numbers and simple variables in Rexx programming. However, there is another form of variable, called a "compound variable." It consists of two parts, a "stem" and a "tail." Some programmers think of compound variables as being Arrays, and they are correct. Other programmers think of them as being similar to records or C++ object classes, and they are also correct. Compound variables are very flexible and very powerful. The next slide will cover examples on how to use them.

Symbols and Stems…

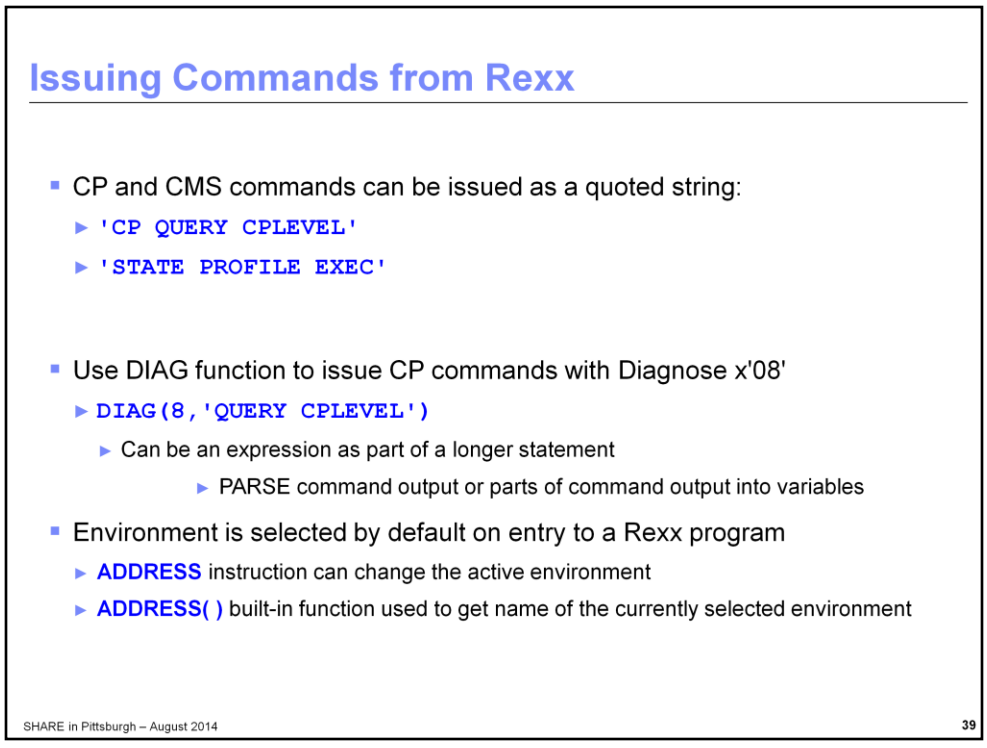
```
/* Stems as arrays */
  do i=1 to 50 by 1
    array.i = i+5
  end
  say array.25          /* Output: "30" */
  say array.51          /* Output: "ARRAY.51" */


/* Stems as records */
  If attendee.payment == "LATE" then
  do
    say attendee.$fullname
    say attendee.$email
    say attendee.$company.telephone
  end
```


In the first example, we see a compound variable being used as an array. It sits in a **DO** loop, and is assigned a value for *Array.1* through *Array.50*. When we output the value of *Array.25*, we will see its value ("30"). When we output *Array.51*, we see it has not been initialized … and therefore the value is its own name, capitalized: "ARRAY.51".

By convention or tradition, Rexx programmers use the *Array.0* element to store the size of that array. However, this value is not automatically included unless you are using Pipelines. (More on this in a later slide.)

As we saw on the previous slide, compound variables can even be used to form multidimensional arrays – for example, *array.i.j.k*

The second example demonstrates compound variables as records. In this case, we check to see if a conference attendee's payment status is "LATE". If so, we output more information about this attendee. As you can see, compound variables can be used to organize data in more than just a single array.



One of the primary benefits of using Rexx for VM is that we can use it to automate CP and CMS commands inside the VM operating environment. This allows us to reduce hard operations or repeated tasks to a single program. In the examples above, we see that our CP or CMS commands can be enclosed inside of single quotes. We will show how to use these examples on the next side.

Rexx/VM provides a built-in function called **DIAG()**. This function allows a Rexx programmer access to the Diagnose Instructions – APIs into the hypervisor layer itself. In this example, we're issuing Diagnose x'08', which passes a CP command to the hypervisor … in this case, **QUERY CPLEVEL**. *Note* that the DIAG function IS case-sensitive and requires its input in UPPER case. The **parse var** or **parse value** keywords can be used to pull the results into your Rexx program.

Finally, Rexx can change your "expected environment" so that you can send commands to CP, to CMS, to XEDIT, to TSO … all without making complicated changes to your program.

Let's explore these concepts in more detail, shall we?

Issuing Commands – z/VM Example

```
Address CMS           /* send cmds to CMS */
 'STATE PROFILE EXEC'


If RC=0 Then          /* file found */
    'COPY PROFILE EXEC A TEMP = ='


             /* Save command output in variable */
Parse Value diag(8,'QUERY CPLEVEL') With queryout
say queryout
```

```
z/VM Version 6 Release 2.0, service level 1101 (64-bit)
Generated at 05/09/12 19:47:52 EDT
IPL at 06/03/12 16:29:17 EDT
```


In the first example, we change the address of our program to CMS, and we send the CMS command directly to CMS as a result. We then check our special variable *rc* … Rexx automatically inserts the return code from our CMS or CP command into variable *rc*. Useful! So if the command passes (*rc*= 0) then we can issue another command. What the example does is check to see if a certain file exists …. And if it does exist, we copy that file into a new file with a new filename.

The second example displays the **diag()** function. Here we are parsing the value of **QUERY CPLEVEL** into a single variable called *queryout*. When we output *queryout*, we are given three lines full of text – **QUERY CPLEVEL** is very generous with its information! But we have already covered how to parse bigger variables into smaller, more usable pieces of information on previous slides. Perhaps that will be useful in an upcoming exercise …?

Issuing Commands – TSO

```
"CONSOLE ACTIVATE"
...

ADDRESS CONSOLE  /* change environment to CONSOLE for all commands */
"mvs_cmd"
...
"mvs_cmd"

ADDRESS TSO tso_cmd    /* change environment to TSO for one command */
...
"mvs_cmd"

ADDRESS TSO    /* change environment to TSO for all commands */
"tso_cmd"
...

"CONSOLE DEACTIVATE"
```


If using TSO, similar concepts apply – remember, Rexx is available on more platforms than just z/VM! Here we are sending mvs commands alternately to the **CONSOLE** (in blocks), a single TSO command in the middle of the example, and then we change the **ADDRESS** to TSO toward the end for tso-specific commands.

Using Pipelines with Rexx

- PIPE is a command that accepts *stage* commands as operands
  - Stages separated by a character called a *stage separator*
    - Default char is vertical bar **|** **(x'4F')**
- Allows you to combine programs so the output of one serves as input to the next
  - Like pipes used for plumbing: data flows through programs like water through pipes!
- User-written stages are Rexx programs
  - Reads in data, works on it, places it back into pipe

PIPE is a Rexx keyword; it processes "Pipelines." These are staged commands that process data very rapidly. The output of one stage becomes the input to the next stage, allowing the data to flow like water. PIPE can be combined with Rexx programs to handle rapid data management. It is complex, but very powerful!

Using Pipelines with Rexx – Examples

- Invoking from CMS command line:

```
pipe < profile exec | count lines | console
```

- Invoking from an Exec:

```
/*  Count number of lines in exec */
'PIPE < profile exec | count lines| console'

/*  or … on multiple lines  */
  'PIPE < profile exec',
        '| count lines',
        '| console'
```

SHARE in Pittsburgh – August 2014 43

Here is an example of a simple Pipeline. The goal is to take our PROFILE EXEC file, count the number of lines in it, and then post that output to the console. This would take several lines in Rexx, but in these examples we see it can be done in a single line. It can be executed outside of a Rexx program, right on the z/VM CMS command line (in the first example), or issued inside a Rexx EXEC. While they can be written on one line, Pipelines can extend for many stages, so tradition or common practice is to write them onto multiple lines as shown in the last example. The **PIPE** command must be enclosed in quotes (it is a CMS command), and the comma **","** must be used as a continuation character (since it is all meant to be one line, no matter how many stages).

Using Pipelines with Rexx – Examples

- Invoking commands and parsing output into a stem:

```
/* Pipe example #2 */
'pipe',
'CMS LISTFILE * EXEC A',  /* issue cmd        */
  '| SPECS w1 1',          /* parse first word */
  '| STEM response.'       /* save in stem     */


do i = 1 to response.0
   say response.i    /* display file names */
end
```

SHARE in Pittsburgh – August 2014 44

Here is another example of a **PIPE** – we issue a **LISTFILE** to gather the list of Rexx EXECs on our A disk, taking only the first word (the file name), and then placing the resultant list into a compound variable called response.

We then issue a **DO** loop from 1 to *response.0* – our **PIPE** stage has automatically put the number of items from our **LISTFILE** into the .0 element, making it easy for us to program this loop. We display the file names in turn, and then our program is completed.

Control Constructs – DO…END

DO … END can be used to create a code block

```
if wins > losses then
   do
     say 'Congratulations!'
     say 'You have won!'
   end
else say 'Sorry, you have lost'
```

SHARE in Pittsburgh – August 2014 45

We have mentioned **DO** loops and **IF-THEN** briefly. We will now cover them in more detail.

DO..END are pairs of keywords which indicate a block of code. They are often used in loops, or to indicate code that will be executed after a comparison is done. In the case above, we have an **IF-THEN** statement where we compare two variables. **IF** wins > losses, then we issue a congratulations message. The keyword **ELSE** allows us to output another set of data if the original statement was not true.

Control Constructs - Selection

```
if wins > losses then say 'you have won'
                 else say 'you have lost'


select
  when wins > losses then say 'winner'
  when losses > wins then say 'loser'
  otherwise say 'even'
end


select
  when wins > losses then say 'winner'
  when losses > wins then say 'loser'
  otherwise NOP
end
```


Sometimes **IF-THEN** is not enough, and multiple options are required. We can use a **SELECT** in these cases. They are similar to SELECT and CASE statements in other programming languages. The keyword **WHEN** pairs with **THEN** to indicate a single case inside a **SELECT**. **OTHERWISE** covers all remaining options. When using **SELECT**, an **OTHERWISE** statement must always be programmed, even if it is to say '**otherwise** NOP' (otherwise, no operations occur). If an **OTHERWISE** statement is not provided, Rexx will return an error.

Control Constructs – DO Loops

```
do forever
  say 'You will get tired of this'
end


do 3
  say "Roll, Roll, Roll the dice"
end


do i=1 to 50 by 1
  say i
end
```

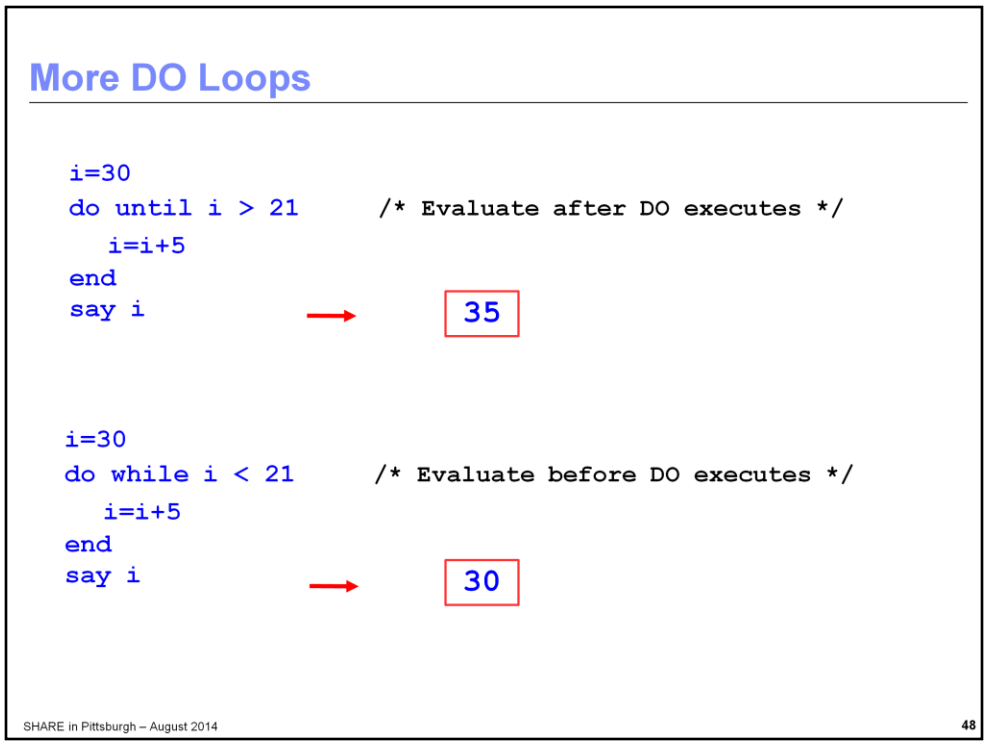
SHARE in Pittsburgh – August 2014 47

The **DO..END** can also be used to build loops.

DO FOREVER executes its contents forever.

DO 3 executes its contents precisely 3 times.

Finally, **DO** loops can be programmed with a counter variable (in this case, *i*) to process a variable number of iterations. We have previously seen examples of this when dealing with compound variables as arrays.



DO loops can also be executed conditionally, in either the **DO WHILE** or **DO UNTIL** format. They are different; we shall explain what is meant by each.

A **DO UNTIL** checks its conditional statement (above, "i > 21") only after the first pass through the loop. The loop will always execute at least once, no matter what. It is similar to the Pascal language's "REPEAT..UNTIL" keywords. So in our example, *i* is already larger than 21, but the loop executes once anyway, so our output is 35.

In a **DO WHILE** loop, the conditional statement (above, "i < 21") is checked before the first pass through the loop. So in this case, *i* is never less than 21 … the loop never executes, and our output is 30.

Iterate, Leave, and Exit

- **Iterate** causes a branch to end of control construct

```
do i=1 to 4
  if i=2 then iterate
  say i
end
```

```
1, 3, 4
```

- **Leave** exits the control construct and continues the REXX program

```
do i=1 to 4
   say i
   if i=3 then leave
end
say 'I''m free!'
```

```
1, 2, 3
I'm free!
```

- **Exit** exits the REXX program unconditionally

```
i=1
do forever
   say i
   if i=3 then exit
   i=i+1
end
say 'I''m free!'
```

```
1, 2, 3
```

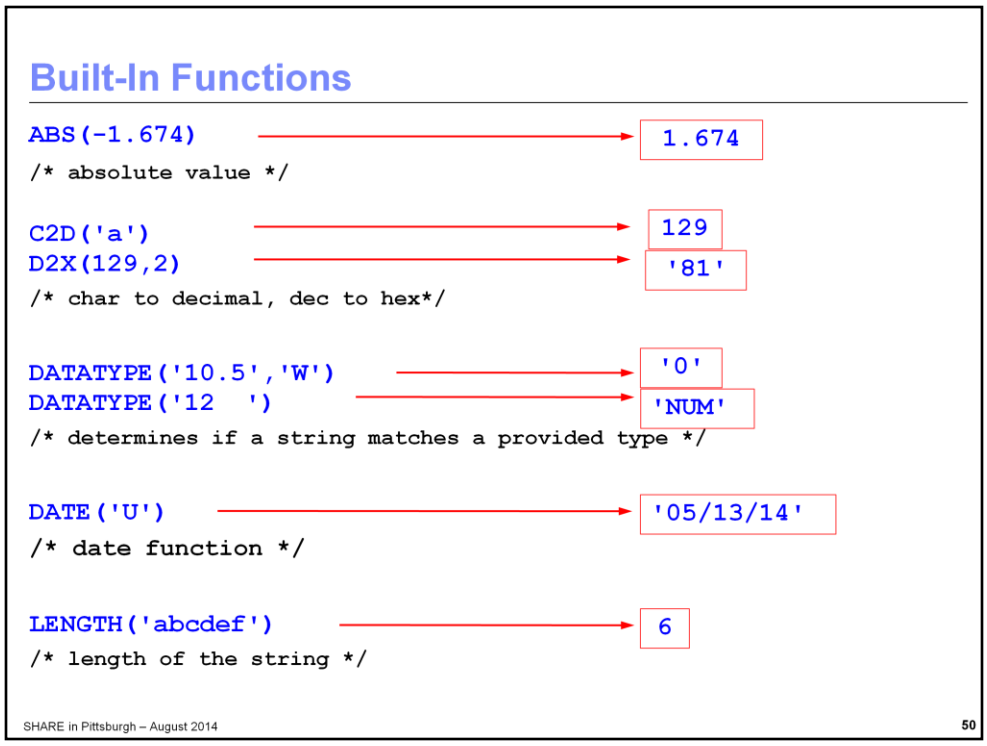
SHARE in Pittsburgh – August 2014 49

Here are three keywords which are useful to have when writing loops:

Iterate branches to the end of your construct or loop. In the first example, we **iterate** if *i* =2 … which means we skip the **say** statement that provides output.

Leave exits the control construct altogether. So in our middle example, if *i*=3, we exit our loop and provide additional output. Because we left the loop, we never **say** the number "4".

The **Exit** keyword terminates the Rexx program altogether. In our last example, we **EXIT** if *i* =3 … which means the output after the loop is never processed.



Rexx provides approximately 75 "built-in" functions. These can be used by programmers without having to write their own interfaces or subroutines, they come with the Rexx language. This slide and the next slide provide examples of these built-in functions.

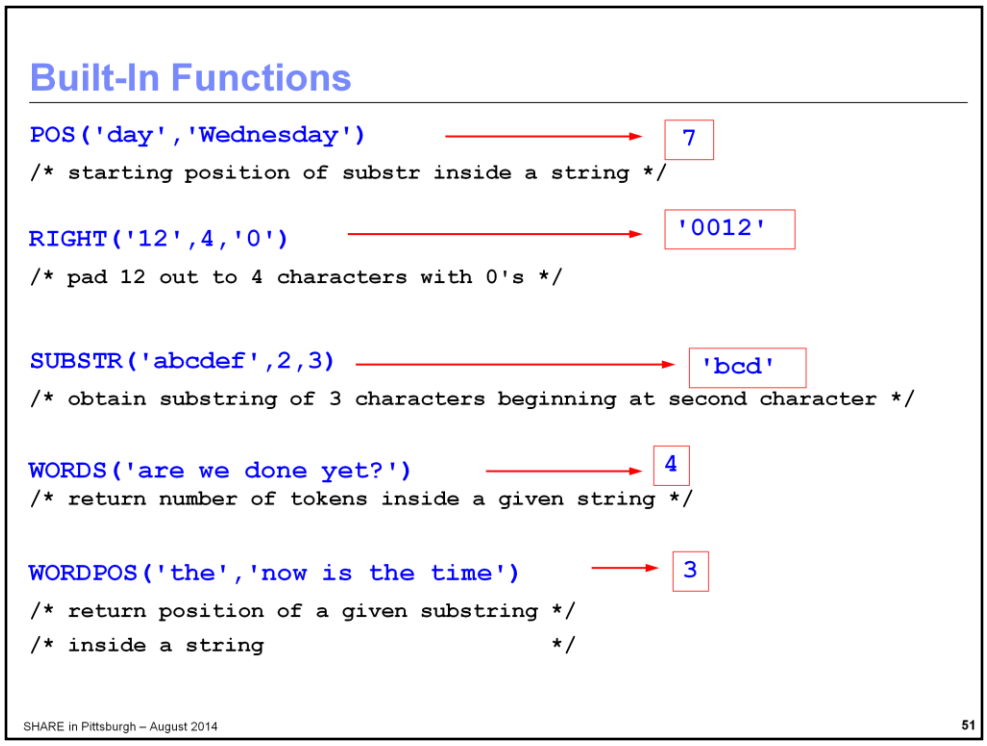
ABS() returns the absolute value of a number.

C2D() and D2X() are conversion functions for translating Characters to Decimal or Decimal to Hex. Similar functions exist for other conversions.

DATATYPE() returns information about a variable or number.

DATE() will return the date in one of any number of formats.

LENGTH() returns the length of a string, whether it is a variable or a literal string.



POS() returns the starting position of a substring within a larger string.

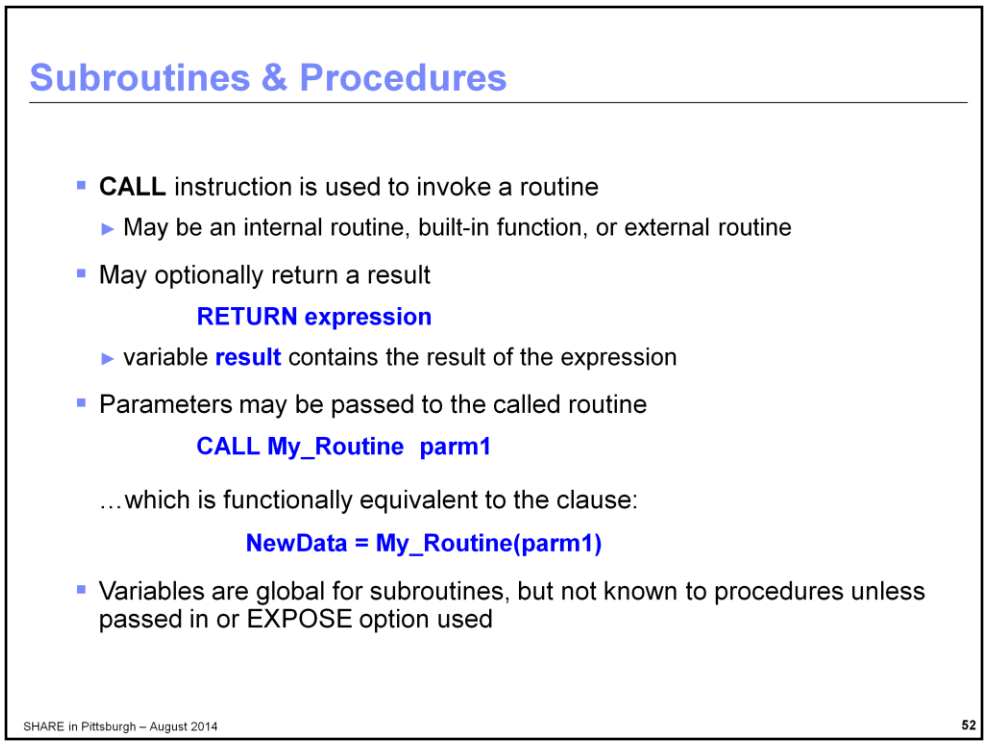
RIGHT() will right-align your data and pad a variable with 0's on the left-hand side.

SUBSTR() can be used to obtain a substring of a larger string.

WORDS() counts the number of tokens in a literal string or a variable

WORDPOS() returns the position of a particular token inside a string.

There are a lot more Rexx functions! Check out the Rexx Reference on your workstation for more information.



Of course, if the built-in functions do not suffice, you can write your own functions.

The **CALL** keyword can be used to execute a function, be it internal to your Rexx EXEC, a built-in function, or external (a separate Rexx EXEC file). Alternately, a function can be called by routine name and have parentheses to note input parameters.

A function may or may not **RETURN** a result. The **RETURN** keyword will store a value inside special variable *result* if it is used.

By default, all variables within a single Rexx EXEC are global and can be modified. Special keywords **procedure** and **EXPOSE** can be used to control variables, or make them local-only for a function.

Subroutine Example: Returning a Value

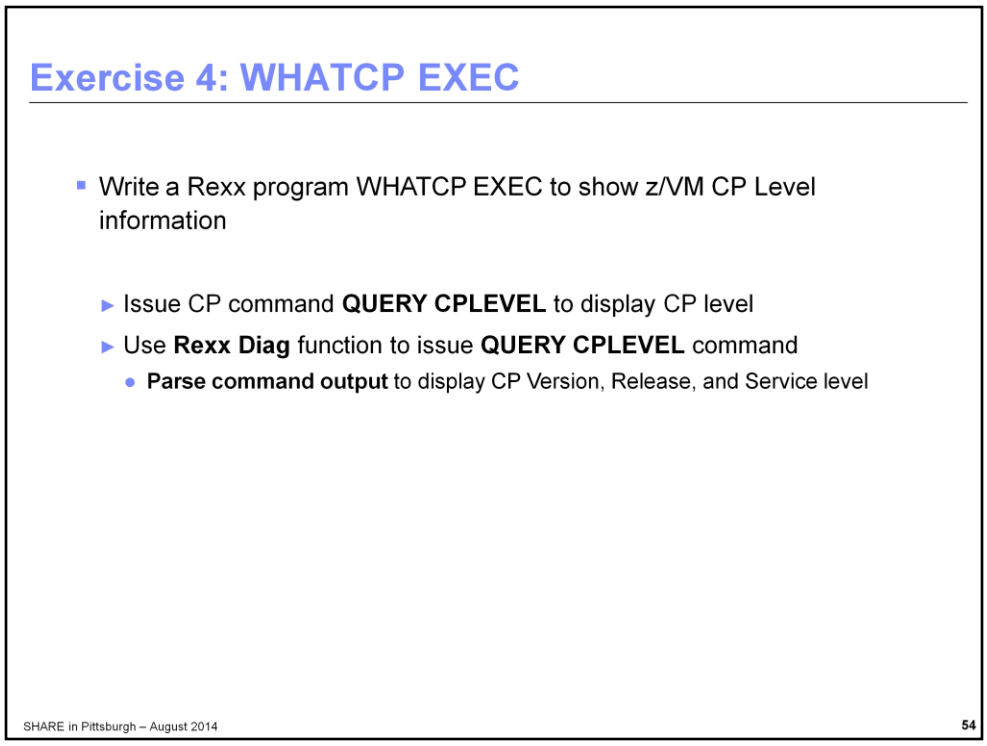
```
/* subroutine call example */
x = 5
y = 10
Call Calc x y                /* call subroutine Calc */
If result > 50 Then
  say "Perimeter is larger than 50"
Else
  say "Perimeter is smaller than 50"
exit

Calc:                        /* begin subroutine     */
Parse Arg len width          /* input args           */
return 2*len + 2*width       /* calculate perimeter  */
                             /* ...and return it     */
```


Here is an example of a subroutine call. In this example, we use the **CALL** keyword to execute subroutine *Calc*:. *Calc*: is located near the bottom of the example, and the label name has a colon **":"** at the end of it. That colon is required for function label names.

Calc parses its arguments (*x* and *y*) with **PARSE ARG**, and then returns a mathematical calculation – specifically, it calculates the perimeter of a rectangle. That information is stored in variable *result*.

In the main program, we see *result* is part of an **IF-THEN** statement; we use it to determine our output.

Finally, at the end of the program, we have the **EXIT** keyword. This is very important – if the **EXIT** keyword were not used, the *Calc*: subroutine would be executed a second time! So make sure to end your program with **EXIT** when using internal subroutines like this.



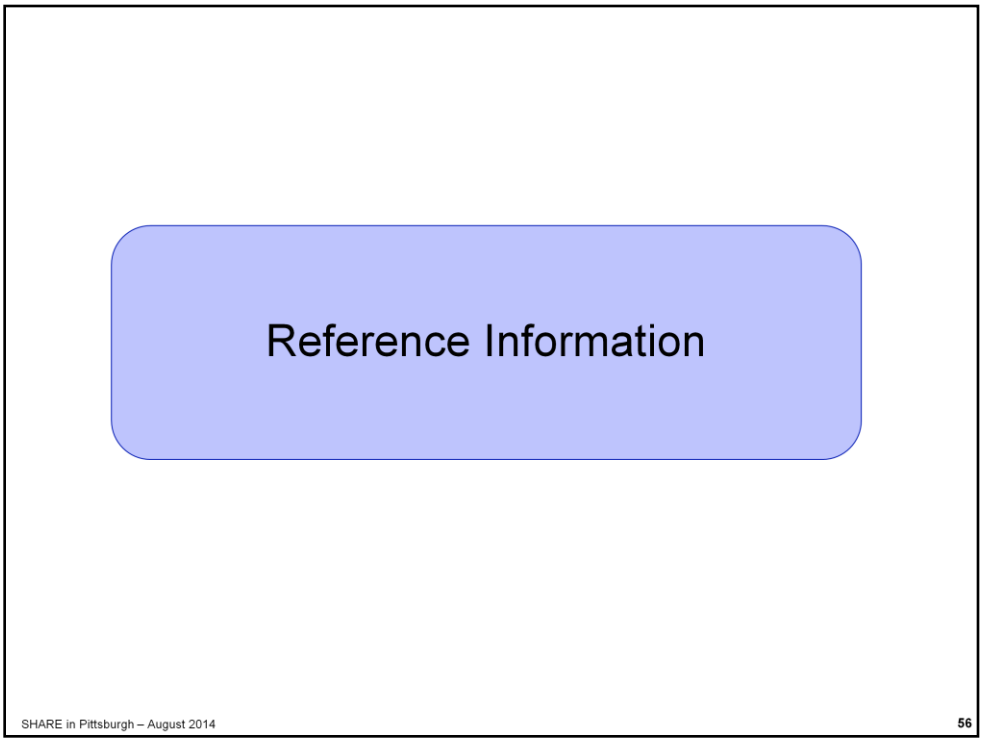
The last two exercises are provided together at the end of our lab. Work on them with the time you have available. Remember, the answers are available in Appendix A.

Exercise 4 asks you to write a new Rexx program called WHATCP EXEC that shows z/VM CP Level information. It will do this twice – first, issue the **QUERY CPLEVEL** command inside your **EXEC** to display the CP Level. Second, inside that same EXEC, use the Rexx **Diag()** function to issue **QUERY CPLEVEL**. Remember, this gives us three lines of output, and we only want a small piece of it – the CP Version, Release, and Service level. Use **PARSE** to collect the data you need from this longer string.

Exercise 5: MYDISKS EXEC

- Write a Rexx program to show which disks your userid has accessed
  - **Call a subroutine** that
    - Uses a PIPE to **issue** CMS command **QUERY DISK** and **save** response
    - **Determine** the number of disks accessed
    - **Return** the value to the main routine
  - **Display** the returned number of disks accessed
  - **Display** each of the disks that are accessed
  - **Issue** the CMS command **QUERY DISK** without using a PIPE
  - **Verify** that output from Steps 3 and 4 match

Exercise 5 asks you to write a Rexx program to show you the disks a userid has accessed. You will use nearly every concept we have covered in this lab as part of this exercise, including Pipes and subroutines.



The next slide contains places where you can learn more about the Rexx programming language.

For More Information…

- **Websites:**
  - http://www.ibm.com/software/awdtools/rexx/ — Rexx webpage
  - http://www.ibm.com/software/awdtools/netrexx/ — Netrexx
  - http://www-01.ibm.com/software/awdtools/rexx/opensource.html — Object Rexx
  - http://regina-rexx.sourceforge.net/ — Regina Rexx
- **z/VM publications:**
  - Rexx/VM Reference - SC24-6113
  - Rexx/VM User's Guide - SC24-6114
  - website for library downloads: http://www.vm.ibm.com/library/
- **z/OS publications:**
  - TSO/E Rexx User's Guide - SC28-1974
  - TSO/E Rexx Reference - SC28-1975
  - website for library downloads: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/IKJOSE10?filter=rexx
- **Rexx Compiler**
  - Products ordered separately from z/VM:
    - REXX/370 Compiler, 5695-013
    - REXX/370 Library, 5695-014
- **Other books:**
  - The Rexx Language — ISBN 0-13-780651-5
  - The Netrexx Language — ISBN 0-13-806332-X
- **List servers:**
  - http://listserv.uark.edu/scripts/wa.exe?A0=ibmvm

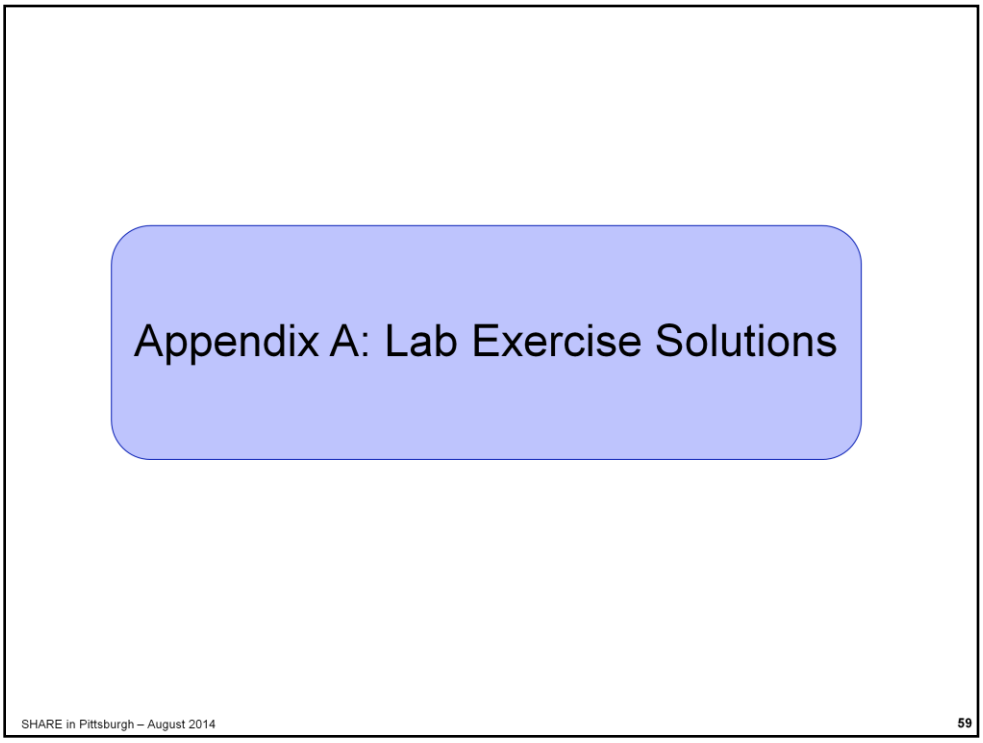
There is a lot of information on the Rexx Programming language available on the internet, be it via the IBM libraries or from other publications. The IBMVM list server is another good source for Rexx – this is a customer-run mailing list, and the people there are very helpful and friendly toward anyone learning VM or Rexx, so don't be afraid to ask questions!

Thanks!

John Franciscovich
IBM
z/VM Design and Development
Endicott, NY

francisj@us.ibm.com

Session 15735


Appendix A: Lab Exercise Solutions
SHARE in Pittsburgh – August 2014
59

The answers!

Exercise 1: Create Temp Disk Space

```
/* Get Temporary disk space */

/* File mode of temporary disk is input argument */

 parse upper arg fmode rest
 If (fmode = '') | (rest ¬= '') then
  Do
    say ''
    say 'ERROR:  Input parm is FILEMODE.'
    say ''
    exit 4
  End

 'CP DETACH 555'             /* Get rid of old disk */
 'CP DEFINE T3390 555 2'     /* Define 2 cylinders of temp space */

 queue 1                     /* Answer YES to FORMAT prompt   */
 queue TMP555                /* Disk label is TMP555          */
 'FORMAT 555 'fmode          /* Format the disk for CMS files */

 If (fmode = 'A') Then       /* If input mode is "A" move A disk to B */
   Do
     Parse Value Diag(8,'QUERY 'UserId()) With thisuser .
     'access VMSYSU:'thisuser'. b/a'
     frommode = 'b'
   End
 Else frommode = 'a'

'COPYFILE * exec ' frommode '= =' fmode /* COPY existing EXEC files
                                           to new temp disk          */
 exit 0
```

SHARE in Pittsburgh – August 2014 60

This is the Rexx Program which creates a temporary disk at a file mode that is provided as input. The **PARSE** statement converts the input argument *mode* to **UPPER** case, and then stores it in variable *fmode*. If there is any extra input it is placed in variable *rest*. Next, the program checks to make sure valid input was provided. If not, it issues a message and terminates the program.

The next we see are system commands issued by program **GETTEMP**. These commands remove an existing temporary disk from if it exists, and then defines the new temporary disk. Before files can be placed on a new disk, it must be formatted. The **FORMAT** command does this; you see input to the prompts issued by the **FORMAT** command provided on the **QUEUE** statements. Also, we specify the input *mode* on the **FORMAT** command so the disk will be accessed at that mode when the format is complete.

Once the disk is formatted, it is ready to accept files. But first, we check if the input mode was "A"; if so, we move the disk that is currently at **A** to **B**; then we use the **COPYFILE** command to copy existing files with type EXEC from the existing disk to the new temporary disk.

Exercise 2: Say, Pull, & Passing Parameters - Answer

```
/*  */
parse arg suit
say 'Enter a number from 2-10:'
pull num
say 'Your card is a 'num' of ' suit
```


As we indicated, this is a short program. All Rexx programs begin with a comment, even if the comment is blank. We expect the suit to be entered on the command line (for example "CARDLAB DIAMONDS" or "CARDLAB SPADES") … it is stored in variable *suit*. Output from the **say** statement prompts someone sitting at the terminal to enter a number. That input is collected by the **pull** statement and is stored in variable *num*. We then provide output based upon your input.

Exercise 3: Tracing and Debugging – Answer A

Trace Intermediate output:

```
 6 *-* string1 = "Rexx" 'Lab'
       >L>   "Rexx"
       >L>   "Lab"
       >O>   "Rexx Lab"
     7 *-* say string11
       >L>   "STRING11"
STRING11
     9 +++ string2 = "Exerc"||"ise'say string2
Error 6 running REXXTR3A EXEC, line 9: Unmatched "/*" or quote
```


When **Trace Intermediates** is used, we see the incorrect output "STRING11", instead of expected output "Rexx Lab" in variable *string1***.**

There is also an error on the statement that attempts to set variable *string2* with the value of concatenated string "Exercise".

A closer look at the program reveals that *string11* was specified on the **SAY** statement instead of *string1*. Since *string11* has not been initialized, its contents are set to its name in uppercase: "STRING11".

We can also see that there are mismatching quotes delimiting string *"ise'*.

Exercise 3: Tracing and Debugging – Answer A

Corrected Rexx Program:

```
Trace I

string1 = "Rexx" 'Lab'
say string1                 /* Was: say string11             */

string2 = "Exerc"||"ise"    /* Was: string2 = "Exerc"||"ise' */
say string2
```

Result:

```
 6 *-* string1 = "Rexx" 'Lab'
       >L>   "Rexx"
       >L>   "Lab"
       >O>   "Rexx Lab"
     7 *-* say string1
       >V>   "Rexx Lab"
Rexx Lab
     9 *-* string2 = "Exerc"||"ise"
       >L>   "Exerc"
       >L>   "ise"
       >O>   "Exercise"
    10 *-* say string2
       >V>   "Exercise"
Exercise
```


The bugs are fixed as follows:

- The **SAY** statement is corrected to specify *string1* instead of *string11*
- String "ise" is corrected with both delimiters now double quotes

With these corrections, the program correctly displays the strings "Rexx Lab" and "Exercise".

Exercise 3: Tracing and Debugging – Answer B

Trace Intermediate output:

```
      7 *-* Nums = "25 35 71"
        >L>   "25 35 71"
      9 *-* parse arg w1 . w2 w3
        >>>   ""
        >.>   ""
        >>>   ""
        >>>   ""
     11 *-* $average = (w1 + w2 + w3) // 3
        >V>   ""
        >V>   ""
     11 +++ $average = (w1 + w2 + w3) // 3
DMSREX476E Error 41 running REXXTR3B EXEC, line 11: Bad arithmetic conversion
```


In this program, **Trace Intermediates** shows that the values in variables *w1*, *w2*, and *w3* that are used to compute *$average* are nulls. This causes an error when the computation is attempted.

This happens because the **PARSE** statement is incorrect. The values to be used to compute the average are in variable *Nums;* they were not provided as input arguments.

In addition, on the "$average =" statement, // will provide the remainder rather than the quotient for the division operation. This will give an incorrect value for *$average.*

Exercise 3: Tracing and Debugging – Answer B

Corrected Rexx Program:

```
Trace I
Nums = "25 35 71"

parse var Nums w1 w2 w3            /* Was: parse arg w1 . w2 w3   */

$average = (w1 + w2 + w3) / 3      /* Was: (w1 + w2 + w3) // 3    */
say "The average value of these numbers is" $average "."
```


To correct this program:

• The **PARSE** statement was changed from **parse arg** *nums* to **parse var** *nums* . This causes the values in variable *Nums* to be parsed into *w1*, *w2*, and *w3.*

• // on the '$average =' statement was changed to /. This corrects the division operation to compute the correct average value.

Exercise 3: Tracing and Debugging – Answer B

Result:

```
      7 *-* Nums = "25 35 71"
        >L>   "25 35 71"
      9 *-* parse var Nums w1 w2 w3
        >>>   "25"
        >>>   "35"
        >>>   "71"
     11 *-* $average = (w1 + w2 + w3) / 3
        >V>   "25"
        >V>   "35"
        >O>   "60"
        >V>   "71"
        >O>   "131"
        >L>   "3"
        >O>   "43.6666667"
     12 *-* say "The average value of these numbers is" $average "."
        >L>   "The average value of these numbers is"
        >V>   "43.6666667"
        >O>   "The average value of these numbers is 43.6666667"
        >L>   "."
        >O>   "The average value of these numbers is 43.6666667 ."
The average value of these numbers is 43.6666667 .
```


The trace of the corrected program looks much different than when we originally traced it. Now we see the correct values being used to compute *$average*, as well as the addition and division operations being done correctly. The resulting correct average is now displayed.

Exercise 4: WHATCP – Answer

```
/* Display CP Level information for the z/VM system */

'CP QUERY CPLEVEL'

Parse value diag(8,'QUERY CPLEVEL') with ,
                  . . version . release . ',' . . servicelvl .

say 'z/VM Version = ' version
say 'z/VM Release = ' release
say 'Service Level = ' servicelvl
```


The first half of this exercise is simple – issue the CP command by putting it into quotes. Rexx does the rest.

The second part is tougher. We use **PARSE VALUE** to gather the information from the **DIAG()** function. We could just place this information into a single variable like our *queryout* variable from the lecture slide, and then parse that into smaller variables … but why add the extra step? Instead, we use the full stop (our "skip this token" character) so that we only pull three tokens worth of data from the command. The final full stop skips everything else that's left.

Exercise 5: MYDISKS EXEC – Answer #1

```
/* Find Number of disks accessed and list them */
Call GetDisks
Say 'This user has' NumDisks 'disks accessed.'
Say '  '

Do i = 1 to Numdisks
   Say DiskList.i
End

Say '  '
ADDRESS CMS
'QUERY DISK'
Exit

/* Subroutine: Get list of disks and return number of disks accessed*/
GetDisks:
   'PIPE',
     'CMS QUERY DISK',
     '| Drop 1',
     '| STEM DiskList.'
    NumDisks = DiskList.0
Return NumDisks
```

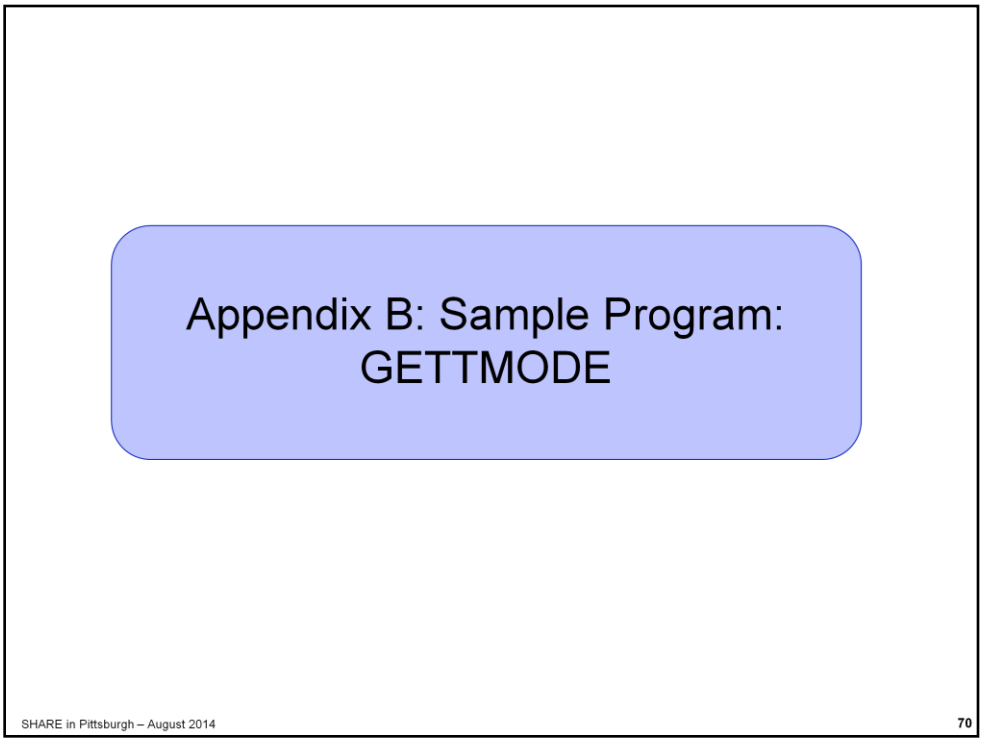

There were two ways to write this exec, but the main program is the same no matter what. We call our subroutine, output the number of disks, and then we use a **DO** loop to output the number of disks. Then we end with a **QUERY DISK** to provide comparative output. Finally, we **EXIT** the program.

Our subroutine in this case is a **PIPE** which issues a **CMS QUERY DISK** command, drops the first line (the header of the output), and then takes the remaining input and places it in a stem variable *DiskList*. Remember, our **PIPE** automatically tells us how many disks were involved by putting it into *DiskList.0* … we place that data in variable *NumDisks* and return it.

Exercise 5: MYDISKS EXEC – Answer #2

```
/* Find Number of disks accessed and list them */
Call GetDisks
Say 'This user has' NumDisks 'disks accessed.'
Say '  '

Do i = 1 to Numdisks
   Say DiskList.i
End

Say '  '
ADDRESS CMS
'QUERY DISK'
Exit

/*Subroutine: Get list of disks and return number of disks accessed*/
GetDisks:
   'PIPE',
     'CMS QUERY DISK',
     '| Drop 1',
     '| STEM DiskList.',
     '| count lines',
     '| var NumDisks'
Return NumDisks
```


In the second example, our subroutine adds in another two extra lines. Rather than assigning a value to *NumDisks* based on *DiskList.0*, we continue the **PIPE**. The third Pipeline stage, "count lines," takes our *DiskList* variable as input, so it can tell us how many elements are in our array. We then pass that information into a final stage, "var *NumDisks*," which stores that value into the variable we plan to return.



This Appendix shows a sample Rexx program which could be used as a utility on a z/VM system.

Sample Program: GETTMODE EXEC

- Rexx program GETTMODE locates the first unused file mode (A-Z) and creates a temporary disk at that file mode
  - Illustrates usage of many Rexx features covered in this lab
    - Subroutine
    - Issuing commands
    - Building and parsing strings
    - Built-in functions
    - Stems
    - Pipelines
    - Displaying output

GETTMODE builds on the GETTEMP program that we saw in the first lab exercise. Instead of providing a filemode as input, GETTMODE dynamically locates an unused filemode and creates a temporary disk at that filemode.

GETTMODE uses most of the language constructs that were covered in the lab material (plus a few Rexx programming tricks!).

Sample Program: GETTMODE EXEC

- Logic:
  - **Calls subroutine** that:
    - Uses a PIPE to **issue** CMS command **QUERY SEARCH** to obtain the used modes (file mode is 3rd word of response); **saves it in a stem**
    - **Builds a string** of used modes from the **output stem** of the PIPE
    - **Creates a string** of possible file modes (A-Z)
    - **Builds a stem** containing the possible file modes
    - **Marks** the used file modes "unavailable" in the list of possible modes
    - **Locates** the first available mode and **returns** it to the main program
  - **If a file mode is returned:**
    - **Issues commands** to define and format a temporary disk at the returned mode

This chart shows the logic of GETTMODE.

Sample Program: GETTMODE EXEC (1 of 3)

```
/* Get temporary disk space and access it at an available file mode  */

'CP DETACH 555'             /* Get rid of old disk */

/* Call subroutine Findmode to locate the first available file mode.   */
/* Once found, define a temporary disk and format and access it at     */
/* the returned file mode.                                             */

Call Findmode

If rtnmode <> 0 Then
  Say 'Temp disk will be accessed at mode' rtnmode
Else
  Do
    Say 'No Filemodes available for temp disk'
    Exit 8
  End

'CP DEFINE T3390 555 2'     /* Define 2 cylinders of temp space */

queue 1                     /* Answer YES to FORMAT prompt   */
queue TMP555                /* Disk label is TMP555          */
'FORMAT 555 'rtnmode        /* Format the disk for CMS files */

Exit rc
```


This is the main path code for GETTMODE. As you can see it is rather short, and very similar to the logic in GETTEMP. The difference is that since the mode for the new disk is an unused mode, we do not have to include the test to determine if we are replacing the A-disk.

Most of the work is done in subroutine *Findmode:*, which is called at the beginning of the program. The result of *Findmode:* will be used by the main path to define and format the new temporary disk.

Sample Program: GETTMODE EXEC (2 of 3)

```
/* Subroutine Findmode will locate the first available (A-Z) file mode.*/
/* and return it in variable rtnmode. If no file modes are available,  */
/* rtnmode will be set to zero.                                        */
Findmode:
   'PIPE',
     'CMS QUERY SEARCH',
     '| SPEC WORDS 3 1',
     '| STEM usedmode.'

/* Build string of accessed file modes                                */
acc_modes = ''
Do I = 1 TO usedmode.0
   acc_modes = acc_modes || SUBSTR(usedmode.I,1,1)
END

/* Build stem containing all possible file modes                       */
possible_modes = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
Do i = 1 TO 26
   modelist.i = SUBSTR(possible_modes,i,1)
End

/* Remove all accessed file modes from possible file mode list         */
mlength = LENGTH(acc_modes)
Do n = 1 TO mlength
      Do i = 1 TO 26
         If (SUBSTR(acc_modes,n,1) = modelist.i) Then
           Do
              modelist.i = ' '
              Leave
           End
      End
End
```


The most complex part of GETTMODE is determining which disk modes are available for assignment of the temporary disk. This is done in subroutine *Findmode:*.

Findmode: starts with a CMS Pipeline. The pipe issues a **QUERY SEARCH** command, which provides output listing all of the currently accessed disks. The third token, or word, in each line of the **QUERY SEARCH** output is the filemode of that particular disk. The modes from each line of the **QUERY SEARCH** output are saved in stem variable *usedmode*. Each filemode will be in a separate entry in *usedmode*.

Next, the **DO** loop extracts each entry from stem variable *usedmode* and concatenates them to a string variable called *acc_modes*. At the end of this loop, *acc_modes* will contain all of the currently used filemodes.

The second **DO** loop builds a list of all possible modes into stem *modelist*.

Next, there is a nested loop which compares each character (representing a filemode) in string *acc_modes* to its equivalent entry in stem *modelist*. When they are equal, the filemode represented by that character is already being used. The entry in *modelist* is cleared, indicating that this mode is not available for the new disk.

Sample Program: GETTMODE EXEC (3 of 3)

```
/* Locate the first possible file mode that is "available" and        */
/* return it                                                         */

foundmd = 'NO'
Do i = 1 TO 26
   If modelist.i ¬= ' ' Then
     Do
       rtnmode = modelist.i
       foundmd = 'YES'
       Leave
     End
End

/* If no file modes available, return zero                           */

If foundmd = 'NO' Then
   rtnmode = 0

Return
```


Once all of the used filemodes have been erased from stem *modelist*, GETTMODE searches for the first non-blank filemode that remains in the *modelist* stem. This indicates the first available filemode in the search order, and will be returned to the main program to be used when creating the temporary disk.

If no filemodes are available, **0** is returned to the main program and no new disk is created.

Sample Program: GETTMODE EXEC – Pipelines Only

```
FINDMODE:  procedure

'Pipe',
'  literal A B C D E F G H I J K L M N O P Q R S T U V W X Y Z',
'| Split ',
'| Spec 1.1 13',
'| Append CMS Q disk *',
'| Nlocate 8.4 /VDEV/',
'| Spec 13.1',
'| Sort ',
'| Unique Single ',
'| Take 1',
'| Var freefm'
```


This program uses CMS Pipelines to accomplish the same thing as the GETTMODE program on the previous pages. You can see how much shorter the program is when Pipelines is used.