Learn to Program in Rexx
Hands-On Lab Part 1 of 2

Session 7485
August 5, 2010

Thomas Conley
Pinnacle Consulting Group, Inc. (PCG)
59 Applewood Drive
Rochester, NY 14612-3501
P: (585)720-0012
F: (585)723-3713
pinncons@rochester.rr.com
http://home.roadrunner.com/~pinncons/

These files allow you to review and practice the fundamentals of Rexx.

Before we get into the example files, let's see how Rexx processes your program statements:

-- Rexx is “free form” - write it so it looks good to you and makes sense to the person reading your code

- Start and end in any column
- Use spaces between Rexx terms on a line for readability
- Insert blank lines between blocks of code for readability
- Use upper-, lower-, and mixed-case as you prefer
- Indent subordinate statements as much as you wish
- Use as many or as few comments as you feel necessary

-- The readability of your program is up to you. It won’t bother Rexx if it’s poor, but it will surely slow you down.

-- There are five kinds of Rexx statements. The Rexx interpreter inspects each statement in your program in turn, determines which of the five types of Rexx statement it is, and then processes it accordingly. Each of the five types is checked in the following order:

1. NULL - a blank line or a line containing nothing but a comment. Used to document the code or to improve readability.
2. LABEL - a symbol immediately followed by colon (:). The label marks a point in the program which may be used to pass control to.
3. ASSIGNMENT - a statement of the form "symbol = expression". The expression is evaluated (see Expressions, below) and the resulting string value is assigned to the variable represented by the symbol.
4. KEYWORD - a statement which begins with one of the specific Rexx keywords, such as IF, SAY, DO, PARSE, etc.
5. COMMAND - anything else. The statement is treated as an expression and evaluated (see Expressions, below). The resulting string value is then passed to the outside environment (operating system, package, etc.) for handling. When the environment returns control to the Rexx program, any value returned is placed in the special variable "RC".

As you can see, by virtue of treating anything it doesn't understand as a command (which it very likely is) there is no such thing as an invalid Rexx statement! It may not have been what you meant to tell Rexx to do, but it will do something with any statement you code.

-- Expressions are combinations of constants, variables, operators, and function invocations which, when valid, result in a character string. What happens to that string depends on when the expression was evaluated: If it was an assignment statement, that string is assigned to the variable

represented by the symbol to the left of the "=". If it followed a Rexx keyword, such as "Say", it depends on what the keyword does. In the case of "Say", the string is written to the console. If the expression was the only thing on the line, it is treated as a command and the string is sent to the outside environment for processing.

-- Constants must be enclosed in quotation marks, unless they are numbers.

- Must be enclosed in a pair of single (‘) or double (“) quotes
- Quotes not necessary if constant is a number (e.g. 42, -6.02 ’42’)
- *Constants are case-sensitive* (e.g. **‘String’**, **“strIng”**, and **‘sTrInG’** are all different constants)
- Rexx is case-INsensitive about the way you write the program, but very case-sensitive about the data!

-- Variables are represented by names, which are symbols that may consist of up to 250 characters, using only a-z (upper- or lower-case, it does not matter), 0-9 (as long as it isn't the first character), and the special characters "_", "!", "?", and "." A period in a variable makes it a "compound variable", so you should avoid using that character in your variable names for the time being. There are no "reserved words" in Rexx. Any valid combination of the above characters may be used as a variable name.

- *symbol* can be any length you wish (TSO is an exception, where symbols are limited to 250 characters in length)
- *symbol* must consist of only: **A-Z a-z 0-9 _ ! ? .**
  - *symbol* must not start with a numeral or the period
- No “reserved words” - any *symbol* meeting the above rules is valid
- Case doesn’t matter, any reference to *symbol* is uppercased before access (e.g. **my_var**, **My_Var**, and **MY_VAR** are all the same variable)
- The value of the variable replaces *symbol* and expression evaluation continues
- If you try to use a variable that has not yet been assigned a value, its initial value is its own name in uppercase (e.g. the initial value of my_var is “MY_VAR”)

-- Operators represent verbs that do something to one or two values, such as "multiply", and are each represented by a special character("*" in the case of multiplication). You are probably familiar with many of them, but Rexx has quite a few operators because there are many things you can do to two values.

- Unless grouped by parentheses, operations will be performed from left to right in the following order of precedence:

- Prefix (plus, minus, not) **+ - \ -3 +y \bittwo**

- Exponentiation (power) **** **4.2 ** 3 ➔ 74.088**
- Multiply and divide
  - multiplication ***** **4.2 * 3 ➔ 12.6**
  - floating point divide **/** **13 / 4 ➔ 3.25**
  - integer divide **%** **13 % 4 ➔ 3**
  - remainder **//** **13 // 4 ➔ 1**
- Add and subtract
  - addition **+** **4.2 + 3 ➔ 7.2**
  - subtraction **-** **4.2 - 3 ➔ 1.2**
- Concatenation (a='bird' ; b='song' ; pct = 42)
  - variable **||** **b || a ➔ 'songbird'**
  - blank **b a ➔ 'song bird'**
  - abuttal **'+'pct'%' ➔ '+42%'**
- Comparison (true=1, false=0)
  - These comparisons ignore blanks around strings (case does matter), and zeros in numbers that do not change the arithmetic value of the number:
    - equal **=** **a = b ➔ 0**
    - not equal **\=** **a \= b ➔ 1**
    - greater than **>** **pct > 50 ➔ 0**
    - less than **<** **b < a ➔ 0**
    - GT or equal to **>=** **pct >= 40 ➔ 1**
    - LT or equal to **<=** **pct <= 42 ➔ 1**
  - **These comparisons do strict comparison, so everything must match:**
    - **strictly equal** **==** **a == b ➔ 0**
    - **not strictly equal** **\==** **a \== b ➔ 1**
    - **strictly greater than** **>>** **pct >> 50 ➔ 0**
    - **strictly less than** **<<** **b << 'song ' ➔ 1**
- **AND (logical)** **&** **0 & 1 ➔ 0**
- **OR (logical)**
  - **Inclusive** **|** **0 | 1 ➔ 1**
  - **Exclusive** **&&** **0 && 1 ➔ 1**

-- Functions are separate pieces of executable code which may be within your program or not. They can be thought of as "black boxes" which, when you supply the proper inputs, return a useful output. How it works internally is not your concern as long as you know the rules for its operation. For example, there is a function named "Word" which, when you pass it a string of multiple blank-delimited words and a number, will return the "number-th" word in the string.

- A routine that accepts 0 or more *args* and returns a string value (TSO has a limitation of 20 arguments)
- The value of the function replaces the function invocation and expression evaluation continues (much as it does for a variable)

- Multiple *args* are positional and must be separated by commas
- If an argument is omitted, its comma must remain to indicate this
- Many useful functions are part of Rexx (the built-in functions) and more are provided in external function packages

Example:

say "It is now" time() "on" date("Weekday")

Result:

It is now 15:15:36 on Thursday

-- To begin the lab:

- Login to WindowsXp with password: sharebosa
  - Double-click the desktop "Command Prompt" icon and enter: **cd \S7485**
- To modify your Rexx execs, you can use one of the following methods:
  - Notepad
  - Edit from a Windows Command box
  - Right-click the file in Windows File Manager and select "Edit"
- For each exercise in your Intro handout:
  - Read the commentary and make sure you understand what the code is supposed to do
  - To test it, enter **rexx *progname*** (it is not necessary to enter the ".CMD", but it won't hurt if you do)
- A manual of Rexx command syntax is available at:

http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/DOWNLOAD/ikj4a370.pdf?DN=SA22-7790-07&DT=20060626210253

- If you have any questions, raise your hand and we'll come to you
- Before you leave:
  - If you want a copy of your work, you must have a diskette or USB drive
    - Copy what you want from the C:\S7485 directory
  - Logoff Windows – Do NOT power off!
  - Fill out a session evaluation sheet and leave it at the front of the lab
  - Pick up an answer set at the front of the lab

The files are identified below.

FIRST.CMD	simplest Rexx program – how to do output
NEXT.CMD	simple but better Rexx program – output again
PROMPT1.CMD	getting fancier – how to get input and
PROMPT2.CMD	variations on input and output
PROMPT3.CMD	

```
  GREET1.CMD         simplest loop
  GREET2.CMD         more useful looping

  NUMBERS.CMD        how numbers are treated in Rexx

  DATATYPE.CMD       determining what kind of string this is - validating
  DATATYP2.CMD         input

  CONTINUE.CMD       continuation of lines in a program
  CONTINU2.CMD

  PRECISE.CMD        Rexx's decimal arithmetic

  FUNCS.CMD          using functions - and some useful functions

  ADDR1.CMD          interfacing with the external environment, also
  ADDR2.CMD            illustrating some potential pitfalls in Rexx and
  ADDR3.CMD            how to avoid them

  MAGICNO.CMD        a self-documenting Rexx program, though all good
                       programmers document their code
```

```
/* FIRST.CMD */
/*   My first REXX program                          */
/* say is the Rexx command to echo output to the terminal */
say Hello world
```

```
/* NEXT.CMD */
/*   My next REXX program                          */
say 'Hello world'
```

```
/* PROMPT1.CMD */
/*   Prompt sequence example                  */
/* pull will take input from the terminal and assign it to a variable */
say 'Please type your name'
pull reply
say 'Hello' reply
```

```
/* PROMPT2.CMD */
/*  Case-sensitive prompt sequence example                */
say 'Please type your name'
parse pull reply
say 'Hello' reply
```

```
/* PROMPT3.CMD */
/*  "Prettier" and fancier prompt sequence example                    */
say ''
say 'Please type your name'
say ''
parse pull name
say ''
say 'Hello' name
say 'Welcome to the Intro REXX Tutorial!'


/* GREET1.CMD */
/*  Greetings        */
do 3
   say ''
   say 'Hi there'
   end


/* GREET2.CMD */
/*  Polite greetings        */
/* do is the Rexx loop construct.  The variable i will hold the value */
/* 1 and be incremented by 1 until howmany, after which the loop ends */
say ''
say 'How many times would you like me to greet you?'   /* ask user */
pull /* get /* why is this comment here? */ from terminal */  howmany
do i = 1 to howmany
   say ''
   say 'Hi there'
   end


/* NUMBERS.CMD
 * Numbers in REXX - just strings until you need to do arithmetic - then
 *                   they are treated as numbers
 */
x = 256
y = 8
say ''
say 'x is' x
/* left is a builtin function that left justifies a string */
say 'Leftmost character of x is' left(x, 1)
say ''
say 'y is' y
/* length is a builtin function that returns the length of the string */
say 'Length of y is' length(y)
say ''
say 'x divided by y is' x/y
```

```
/* DATATYPE.CMD
 * Even though everything is a string in REXX, it is sometimes
 * useful to know what sort of string we have - particularly if
 * we want to attempt to validate user response to a prompt
 */
say ''
say 'Introducing "DATATYPE()"'
say ''
say 'Type a number or a letter'
say ''
pull reply
say ''
if datatype(reply) = 'NUM' then say 'You entered a number'
else say 'You entered a letter'


/* DATATYP2.CMD
 * Even though everything is a string in REXX, it is sometimes
 * useful to know what sort of string we have - particularly if
 * we want to attempt to validate user response to a prompt
 */
say ''
say 'Introducing "DATATYPE()"'
say ''
say 'Type a number'
say ''
pull reply
say ''
if \datatype(reply, 'w') then say 'That was not a whole number'
   else say 'That was a whole number'


/* CONTINUE.CMD
 * Continuation is REXX is easy
 *
 * Suppose we want to create a really long string but we want to
 * see all everything in our editor without having to do
 * left/right scrolling
 */
string = 'January February March April May June July August',
         'September October November December'
say ''
say string


/* CONTINU2.CMD
 * Continuation is REXX is easy
 *
```

```
 * You can continue any clause
 *
 */
say ''
say 'Please type a number between 1 and 10'
say ''
pull reply
if \datatype(reply, 'w') | reply < 1,
   | reply > 10
 then say "You didn't follow my instructions"
 else say "Congratulations!  You're paying attention"


/* PRECISE.CMD */
/* Illustration of numeric precision                                    */

arg precision

if precision \= '' then numeric digits precision

say 4/7


/* FUNCS.CMD */
/*
 * Some very useful functions
 */
say ''
say 'Various date formats'
say ''
say '"Normal":    ' date() '  normal in the US, at least'
say 'Ordered:     ' date('o') '     good when you need to sort by date'
say 'Sorted:      ' date('s') '     possibly even better for sorting'
say 'Base date:   ' date('b') '       necessary for date calculations'
say 'Weekday:     ' date('w') '     self explanatory'
say 'Month:       ' date('m')
say ''
say ''
say 'Time formats'
say ''
say '"Normal":    ' time() '   hours, minutes, and seconds'
say 'Civil:       ' time('c') '     if you want am or pm'
say ''
say 'And there are options to get the hours, minutes, or seconds since'
say '  midnight, as well as an "elapsed time clock" which may be'
say '  reset within the program for internal timings'


/* ADDR1.CMD
```

```
 * REXX interacts well and easily with its environment
 * If an expression does not evaluate to a valid REXX clause, the
 *    result is automatically send to the external environment to
 *    be handled
 */
dir


/* ADDR2.CMD
 * REXX interacts well and easily with its environment
 * If an expression does not evaluate to a valid REXX clause, the
 *    result is automatically send to the external environment to
 *    be handled
 *
 * But some care is in order.  If uninitialized, "dir" evaluates to
 *    "DIR".  This is not a REXX clause so is sent to the OS to handle.
 *
 * What if, somewhere else in your program, you tried to use dir as
 *    a variable name?
 */
say ''
dir = 'C:\RXTUTOR\SHARE\INTRO2'
say ''
say 'Some code gets executed here, and then .....'
say ''
dir


/* ADDR3.CMD
 * REXX interacts well and easily with its environment
 * If an expression does not evaluate to a valid REXX clause, the
 *    result is automatically send to the external environment to
 *    be handled
 *
 * But some care is in order.  If uninitialized, "dir" evaluates to
 *    "DIR".  This is not a REXX clause so is sent to the OS to handle.
 *
 * What if, somewhere else in your program, you tried to use dir as
 *    a variable name?
 *
 * So it's always safe ... and generally considered good programming
 *    practice to enclose commands for the external environment in
 *    quotes.  This ensures they are literal strings and not evaluated
 *    by REXX before execution.  It also allows for the use of characters
 *    that would be considered by REXX to be an operator.
 */
say ''
dir = 'C:\RXTUTOR\SHARE\INTRO2'
say ''
```

```
say 'Some code gets executed here, and then .....'
say ''
"dir /w"


/* MAGICNO.CMD */
/* Self-documenting program      */
secret_number = random(1, 10)
guessed_it = 0
say "I'm thinking of a number between 1 and 10"
say "Let's see if you can guess it"
do until guessed_it
   say ''
   say 'Pick a number or type "Q" if you give up'
   pull guess
   if guess = "Q" then leave
   if guess = secret_number then guessed_it = 1
      else say "Sorry, that's not it"
   end
say ''
if guessed_it then say "Hey, that's right! The number I was thinking of
was" secret_number
say 'Thanks for playing my game'
```

==========

Now you will have to write some Rexx code yourself. We will illustrate a feature of Rexx, then assign you a problem to solve as an exercise.

If you get stuck, ask your instructor or lab assistant for help.

==========

```
/*
 *  Strings  --  Learning about string manipulation in Rexx
 */

file = 'CATALOG'                    /* some variables to play with */
literal = 'Filename is'

/* Illustrations of various string concatenation methods */

myfile1 = file||'.DAT'              /* concatenation operator */
myfile2 = file'.DAT'                /* abuttal concatenation */

say 'Filename is' || ' ' || myfile1  /* concatenation operators again */
say 'Filename is'||' '||myfile1      /* exactly the same thing */

say 'Filename is' myfile1           /* blank concatenation */
```

```
say literal myfile1                  /* do it all with variables */

say literal file||'.DAT'             /* mix and match */
say 'Filename is' file'.DAT'         /* and mix and match */

say 'Filename extension is' right(myfile1,3)  /* functions work, too */

/*
 * Student Exercise 1
 *
 * Write the code necessary to
 *
 *    - prompt the user for his/her name
 *    - greet the user with the phrase
 *        "Hello, <user>.  Today is <day of week>, <today's date>."
 *
 * Remember that there is a Rexx function that will return today's date
 * and the day of the week.
 *
 */

================================================================================

/* Strings2  -  More string manipulation */

string = 'Beam me up Scotty'
string2 = date('s')

say 'STRING IS:   ' string
say '   How many tokens in the string?'

howmany = words(string)         /* words() function counts tokens */
say center(howmany,40)          /* center the output within 40 characters
*/

say '   What is the second token?'

word_two = word(string, 2)      /* word() function gets specified token */
say center(word_two,40)

/* But not all strings are neatly arranged as individual tokens */

say ''
say 'STRING2 IS:  ' string2
say '   How long is the string?'

str_len = length(string2)
say center(str_len, 40)
```

```
say '   Year part is leftmost 4 characters'

year = left(string2, 4)          /* left() function returns chars from
left */
say center(year, 40)

say '   Day part is rightmost 2 characters'

day = right(string2, 2)          /* right() returns characters from right
*/
say center(day, 40)

say '   Month is the middle bit'

month = substr(string2, 5, 2)   /* substr() gets things from the middle */
say center(month, 40)

/* And functions can be nested */

say ''
say ''
say 'The first character of the third token in our first string'

char = left(word(string, 3), 1) /* leftmost 1 char of word 3 of string */
say center(char, 40)


/*
 * Student Exercise 2
 *
 * There are 2 employees in the department with the following information:
 *
 *   Jane Doe       --  phone: 555-1234   DOB: 14-Jan-60
 *   John Q. Smith  --  phone: 555-9876   DOB: 12-Jul-55
 *
 * Write the code necessary to display, for each employee,
 *
 *   'First name is' <firstname>
 *   'Last name is' <lastname>
 *
 * There are at least 2 different ways to write code that can be used for
 * both employees even though one has a middle initial and the other does
not.
 *
 * Then for one of the employees (you pick which), display
 *
 *   'Phone extension is' <last 4 digits of phone number>
 *   'Month of birth is' <month part of DOB>'
```

```
 *
 * To help you out, some variables and literal values have already been
 * set up  --  all you have to do to use them is uncomment them.  Stems
 * are used in the variables to allow you to process the variables in a
 * DO loop.
 *
 */

/*
emp.1 = 'Jane Doe'
emp.2 = 'John Q. Smith'
phone.1 = '555-1234'
phone.2 = '555-9876'
dob.1 = '14-Jan-60'
dob.2 = '12-Jul-55'
*/

============================================================================

/*   parse  -  the PARSE instruction         */

string = 'Make it so, Number One'

say 'String is:  ' string
say ''

/*
 * Each token assigned to an individual variable
 */
parse var string first second third fourth fifth
say 'First is:   ' first
say 'Second is:  ' second
say 'Third is:   ' third
say 'Fourth is:  ' fourth
say 'Fifth is:   ' fifth
say ''

/*
 * First 3 tokens into individual variables; whatever is left is assigned
 * to the last variable in the template
 */
parse var string first second third rest
say 'First is:   ' first
say 'Second is:  ' second
say 'Third is:   ' third
say 'Rest is:    ' rest
say ''

/*
```

```
 * First 2 tokens into individual variables; third token into bit bucket
 * specified by placeholder symbol "."; whatever is left goes into the
last
 * variable in the template
 */
parse var string first second . rest
say 'Start is: ' first
say 'Next is:  ' second
say '  Something skipped here'
say 'Tail is:  ' rest
say ''

/*
 * Everything up to the literal "," assigned to the first variable in the
 * template; literal "," is discarded; everything following the literal
 * assigned to the last variable in the template
 */
parse var string command ',' who   /* separate at location of literal
string */
say 'Command is:  ' command
say 'Who is:      ' who

/*
 * Student Exercise 3
 *
 * Parse the phone number into area code, exchange, and number
 * (exchange is the bit before the '-' and number is the bit that
follows).
 * Display the phone number and its component parts.
 *
 */

phone_num = '(800) 555-4040'

============================================================================

/* select - illustrates the SELECT instruction                    */

today = date('w')              /* get the day of the week */
select
   when today = 'Monday' then say 'Sigh'
   when today = 'Friday' then say 'Party!!'
   otherwise say 'Just another weekday'   /* default if no condition
matched */
   end

============================================================================

/* iforsel  --  a valid, but awkward, use of if */
```

```
arg op num1 num2     /* get command line arguments into variables */

if op = 'ADD' then say num1 'plus' num2 '=' num1+num2
if op = 'SUB' then say num1 'minus' num2 '=' num1-num2
if op = 'MULT' then say num1 'times' num2 '=' num1*num2
if op = 'DIV' then say num1 'divided by' num2 '=' num1/num2

/*
 * Student Exercise 4
 *
 * Improve this code by providing some default action if the operator
 * specified by the user is not in the list as, for example, if the
 * user typed
 *                   2 power 3
 *
 * to try to compute the value of 2 raised to the third power.
 */
```

```
=============================================================================
```

```
/*  ezdo - illustrates simple "DO" usage  */

say 'SIMPLE DO LOOP'
do 3                       /* simplest do loop */
   say "Hello"
   end
say ''
say 'Press Enter to continue'
pull reply                 /* pause to look at output */

say ''
say 'DO LOOP USING INDEX VARIABLE'
do i = 1 to 3              /* do loop with an index variable */
   say 'Line' i
   end
```

```
=============================================================================
```

```
/*  loops - illustrates more complex "DO" loops  */

say ''
say 'DO WHILE -- tests a condition at the beginning of the loop'

zoo = 'LION TIGER BEAR ZEBRA'                /* list of items */
say 'Pick an animal'                         /* prompt for choice */
pull this_animal                             /* get user's choice */
do while wordpos(this_animal, zoo) = 0       /* do while choice not in
list */
```

```
/*
 *   If user's first choice is in the list, loop is never entered and
 *   the following code is never executed
 */
   say 'Sorry, that animal is not in our zoo'
   say 'Pick a different animal'             /* reprompt */
   pull this_animal                          /* new choice */
   end
say 'Yes, there is a' this_animal 'in our zoo'  /* condition met - left
loop */
say ''
say 'Press Enter to continue'
pull .             /* pause to look at output (we don't care what is
entered)*/

say ''
say 'DO UNTIL  --  test a condition at the end of the loop'

this_animal = ''                             /* initialize the "choice" var
*/
zoo = 'LION TIGER BEAR ZEBRA'                /* list of items */
do until wordpos(this_animal, zoo) \= 0      /* enter loop */
/*
 * Loop is always executed at least once
 */
   say 'Pick an animal'                      /* prompt user */
   pull this_animal                          /* get user's choice */
   end                                       /* now test the condition */
say 'Yes, there is a' this_animal 'in our zoo'
say ''
say 'Press Enter to continue'
pull .             /* pause to look at output (we don't care what is
entered)*/

say ''
say 'DO FOREVER  --  test condition inside the loop'
say 'LEAVE       --  exiting the loop
zoo = ''
do forever
   say 'Pick an animal to put in the zoo or type "Q" to quit'
   pull this_animal
   if this_animal = 'Q' then leave  /* be sure to create a way out of loop
*/
   zoo = zoo this_animal            /* build up list from user input */
   say 'The zoo now has:' zoo
   end
say ''
say 'Press Enter to continue'
pull reply               /* pause to look at output */
```

```
say ''
say 'BYPASSING PART OF THE LOOP - the "iterate" instruction'

zoo = ''
do forever
   say 'Pick an animal to put in the zoo or type "Q" to quit'
   pull this_animal
   if this_animal = 'Q' then leave                /* user wants to stop
now */
/*
 * If user input is a duplicate of something we already have, bypass the
 * code that adds it to the list and return to the top of the loop
 */
   if wordpos(this_animal, zoo) \= 0 then iterate
/*
 * This code executed only when user input doesn't dupicate previous input
 */
   zoo = zoo this_animal
   say 'The zoo now has:' zoo
   end

================================================================================

/*  subproc.cmd - writing & using your own functions & subroutines */

say "Return first and last letters of a word"
say 'For example: "borborygmus" returns' Ends("borborygmus")

do forever
   say "What's the word?"           /* prompt user */
   pull the_word .                  /* get a word */
   if the_word = 'QUIT' then leave  /* we're outta here */

   call Ends the_word

   f_let = word(Result, 1)
   l_let = word(Result, 2)
   say "The letters are" f_let "and" l_let
   end
say ''

exit       /* must have! if not, will fall thru into "Ends" code */

Ends:                   /* label; identifies start of function */
arg argwd .             /* put argument into our variable */
first = left(argwd,1)   /* get the first letter of argument word */
last = right(argwd,1)   /* get the last letter of argument word */
return first last       /* return both letters separated by a blank */
```

```
/*
 * Student Exercise 5
 *
 * Write a subprocedure to return the number of words and number of
 * characters in any string supplied to it.  In a loop, prompt the user
 * for a sentence, invoke the subprocedure, and report what it returns.
 * Use a function or subroutine call, your choice.  Be sure to leave
 * the user a way out and don't accidentally execute your subprocedure
 * at the end.
 * For extra credit, tell if the sentence is a question or exclamation.
 */

================================================================================

/*  stemz - how compound variables work   */

list.1 = 'January'       /* element "1" of stem has value "January" */
a = 1
say 'List.1 is' list.1
say 'List.a is' list.a   /* tail is evaluated first, so list.a == list.1
*/
say 'List.2 is' list.2   /* element with no value assigned */
say ''
say 'Press Enter to continue'
pull .

say ''
zoo. = 'Unknown animal'  /* assigning a default value to the stem */
zoo.1 = 'Lion'
zoo.2 = 'Tiger'
zoo.3 = 'Bear'
a = 2
say 'Zoo.a =' zoo.a     /* tail evaluated first; zoo.a == zoo.2 */
say 'Zoo.b =' zoo.b     /* tail evaluated; b == B; zoo.B takes default
value */
say ''
say 'Press Enter to continue'
pull .

/* Non-numeric tails   */

special_months = April June September November
days. = 31                                  /* default value */
days.february = '28 (most of the time)'     /* special value for February
*/
do while special_months \= ''               /* set values for 30-day
months */
   parse var special_months next special_months  /* get next month/tail */
```

```
   days.next = 30                              /* assign value */
   end

say 'Enter a month'                            /* prompt user */
/*
 * Get user choice; be sure it's in uppercase, because string values
 * of tails only match in uppercase
 */
pull month .

say 'There are' days.month 'days in' month  /* display result */
```

==========

```
/*
 * Final Class Exercise
 *
 * Process the employee roster to send out birthday cards.
 * Birthday cards are send on the last day of the month preceding the
 * employee's birth month.  For example, if your birthday is April 10,
 * HR will send your card to you on March 31 just to be sure it gets
 * there on time.
 *
 * The roster is all set up for you along with several variables that
 * you might find useful  --  don't worry if you don't use any of those
 * variables or if you use only a few.  There's no single right way to
 * do this.  The function from the previous sample code that returns
 * the number of days in a month is also provided if you wish to use it.
 *
 * Display for the HR director a list of birthday cards that must be
 * sent for the coming year in the format
 *
 *   name, birthday, date to send card, room number to send it to
 *
 *
 * A couple of hints:
 *
 *   - the translate() function converts a string to upper case
 *
 *   - parse upper var ....   converts values to upper case before
 *     assigning them to the variables in the template
 *
 *   - arg <variable name> (rather than parse arg <variable name>)
 *     gives you the argument string in upper case
 *
 *   - remember that the wordpos() function gives you the token number
 *     of a specific string within a list of blank-delimited tokens
 *
 *   - remember that the strip() function can remove leading and trailing
```

```
 *     blanks from a string
 *
 *   - remember that functions may be nested
 *
 *
 *
 * Extra credit:
 *
 *   - align the output neatly in columns with a nice header instead of
 *     the comma-separated form shown above;  hints:
 *
 *        - the max() function returns the maximum value in a
 *             comma-separated list of values
 *
 *        - the left() function will pad a string on the right with
 *             blanks if you ask for more characters than exist in
 *             the string; e.g., left('the', 8) returns "the     "
 *
 *        - compound variables can have more than one tail
 *
 *
 *   - change one of the birthdays to "Jan" and deal with it appropriately
 *
 */

longest = 0
number = 6
months = 'JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC'

Employee. = ''
Employee.0 = 6
Employee.1 = 'Barker     Philip    24-Jul-55   555-4040   1424C'
Employee.2 = 'Caldwell   Helen     14-Jan-60   555-1210   1506'
Employee.3 = 'Davidson   Ed        12-Sep-70   555-7032   1424B'
Employee.4 = 'Ferguson   Pat       10-Mar-65   555-7048   1207'
Employee.5 = 'Miller     Chris     09-May-74   555-1814   1508'
Employee.6 = 'Stone      Eleanor   20-Aug-58   555-1410   1208'

lastday:
/*
 * Function to return last day of a given month
 */

parse upper arg mon .
special_months = 'APR JUN SEP NOV'
days. = 31                                  /* default value */
days.FEB = 28                              /* ignoring leap year */
do while special_months \= ''               /* set values for 30-day
months */
```

```
   parse var special_months next special_months  /* get next month/tail */
   days.next = 30                               /* assign value */
   end

return days.mon
```