Nexxt Level Rexx

Crafting more complex logic and functionality in Rexx

12 steps 3 hours

THE CHALLENGE

Now that you know the basics, it's time to go further. This program takes an example program, twists it to our purposes, and then open it up for crafting some subroutines and additional logic. We'll revisit one of our old challenges, the list of credit card information and get a Rexx version of it working, then flip the script to generate our own data.

BEFORE YOU BEGIN

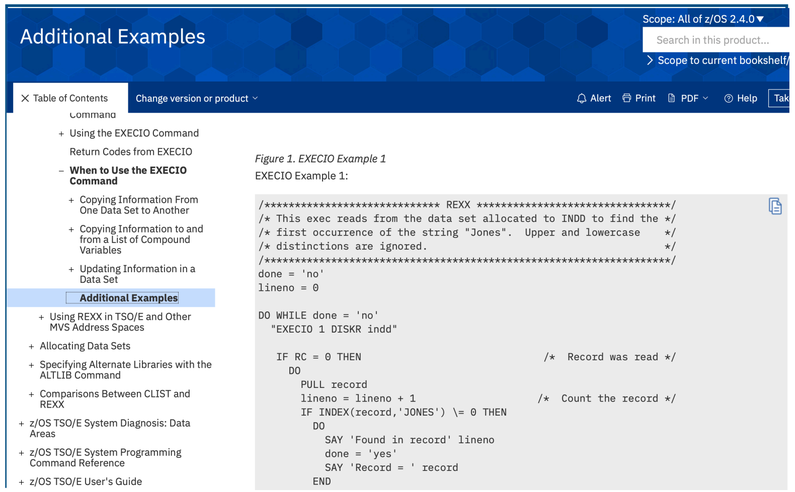
This shouldn't be your first time seeing Rexx, so we suggest running the first Rexx challenge. However, if Rexx is totally your thing, you should be able to jump right into this.

```
/***************************** REXX ***********************************/
/* This exec illustrates the use of "EXECIO 0 ..." to open, empty, */
/* or close a file. It reads records from file indd, allocated     */
/* to 'sams.input.dataset', and writes selected records to file    */
/* outdd, allocated to 'sams.output.dataset'. In this example, the */
/* data set 'smas.input.dataset' contains variable-length records  */
/* (RECFM = VB).                                                   */
/*******************************************************************/
"FREE FI(outdd)"
"FREE FI(indd)"
"ALLOC FI(indd) DA('MTM2020.PUBLIC.CUST16') SHR REUSE"
"ALLOC FI(outdd) DA(OUTPUT.CUSTOMER) SHR REUSE"
eofflag = 2                 /* Return code to indicate end-of-file */
return_code = 0                /* Initialize return code           */
in_ctr = 0                     /* Initialize # of lines read       */
out_ctr = 0                    /* Initialize # of lines written    */
```

1. LOAD THIS CODE

Copy CCVIEW from MTM2020.PUBLIC.SOURCE into your own SOURCE data set, and then open it up. This is a little more advanced from what you worked with in the last challenge, but should still be fairly understandable.

In short, we're going to open up an INDD, an OUTDD, and then iterate over the records, reading them from one into the other.



2. THE SOURCE OF THE SOURCE

99% of the code was taken directly from Example #5 on the Rexx Knowledge Center page if you'd like to see a few variations. These specific examples are based around using EXECIO. You'll typically use EXECIO when doing file-specific tasks on a mainframe system. This is one of the areas where Rexx on a PC differs from Rexx on Z.

We've also added a few pieces of code to help you in your upcoming tasks. You'll probably notice lines 40 and 41, which don't seem to do much at the moment.

```
DO WHILE (return_code \= eofflag) /* Loop while not end-of-file
  'EXECIO 1 DISKR indd'           /* Read 1 line to the data st
  return_code = rc                /* Save execio rc
  IF return_code = 0 THEN         /* Get a line ok?
    DO                            /* Yes
      in_ctr = in_ctr + 1         /* Increment input line ctr
      PARSE PULL line.1           /* Pull line just read from s
      IF LENGTH(line.1) > 10 then /* If line longer than 10 cha
        DO
          "EXECIO 1 DISKW outdd (STEM line." /* Write it to outc
          cc_digits = SUBSTR(line.1,3,19)
          /* call INSPECT */
          out_ctr = out_ctr + 1        /* Increment output line
        END
    END
END
```

3. DEEP IN DO DO DO

Check out lines 30-45. There are three nested DO statetments here. The outermost loop is just a check to make sure there are more lines to read, the middle loop iterates through those lines, and the innermost loop checks on a very specific attribute of that line. Don't be afraid to stack program logic like this, as long as you document it fully in comments so you can always tell which level you're working in. This is also where the formatting and aids provided by certain extensions help with indentation.

Want to talk? Join our Slack
ibm.biz/mtm_slack

Tweet about it!
#MasterTheMainframe

```
2   /* This exec illustrates the use of "EXECIO 0 ..." to open, empty, */
3   /* or close a file. It reads records from file indd, allocated     */
4   /* to 'sams.input.dataset', and writes selected records to file    */
5   /* outdd, allocated to 'sams.output.dataset'. In this example, the */
6   /* data set 'smas.input.dataset' contains variable-length records  */
7   /* (RECFM = VB).                                                   */
8   /*******************************************************************/
9   "FREE FI(outdd)"
10  "FREE FI(indd)"
11  "ALLOC FI(indd) DA('MTM2020.PUBLIC.CUST16') SHR REUSE"
12  "ALLOC FI(outdd) DA(OUTPUT.CUSTOMER) SHR REUSE"
13  eofflag = 2                 /* Return code to indicate end-of-file */
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: bash

Monorail:~ jbisti$ zowe tso issue command "exec 'Z99999.source(CCVIEW)'" --ssm
IKJ56247I FILE OUTDD NOT FREED, IS NOT ALLOCATED
IKJ56247I FILE INDD NOT FREED, IS NOT ALLOCATED
File outdd now contains  1500 lines.
READY
```

4. ENOUGH TALK. LET'S RUN

Yeah yeah yeah, code code code, let's get this thing to run. You'll obviously need to adjust those input/output sections to make it work, and will probably also want to pre-allocate a dataset for the output. Use the MTM2020.PUBLIC.CUST16 data set to start with and write the output somewhere in your own data sets.

Since this non-interactive Rexx, you can run it without starting up a TSO address space, and save yourself a few keystrokes.

Z99999.SOURCE(CCVIEW) | Z99999.OUTPUT.CUSTOMER

.vscode > extensions > zowe.vscode-extension-for-zowe-1.6.0 > resources > temp > _D_ > mtm2020 > Z99999.OU

```
1   %B000745927462394l467Graham      Noel        1505A00355.37201231106571234501?
2   %B000160603807751904 5Margarito  Madden      1701A00192.72201231940591234501?
3   %B000189011409142331 9Darren     Oconnor     1906A00548.66201231119321234501?
4   %B000569894989815325 3Neil       Reeves      1804A00179.20201230124401234599?
5   %B000828291336645882 2Waldo      Barnes      1403A00836.50201231033481234501?
6   %B000559893537061352 7James      Winters     1405A00462.80201230814421234599?
7   %B000610697595000033 4Hayden     Farley      1403A00859.19201230505361234599?
8   %B000461582786280237 3Reuben     Reed        1903A00535.31201231455501234501?
9   %B000215370120399203 1Rudy       Cantrell    1710A00817.20201230012341234599?
10  %B000101018159639320 1Jordon     Richardson  1805A00953.03201231557271234599?
11  %B000287252027446065 6Archie     Levy        1704A00591.08201231343541234599?
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: bash

Monorail:~ jbisti$ zowe tso issue command "exec 'Z99999.source(CCVIEW)'" --ssm
IKJ56247I FILE OUTDD NOT FREED, IS NOT ALLOCATED
IKJ56247I FILE INDD NOT FREED, IS NOT ALLOCATED
File outdd now contains  1500 lines.
READY

Monorail:~ jbisti$
```

5. CHECK THE OUTPUT

Output should look the same as input. If you uncomment line 41, you'll see a running tally of inspected lines in the output. At this point, we've got code that runs, and now we can build upon that foundation to make it do something else. It is suggested you grab a beverage at this time, as it's about to get interesting.

1. From the rightmost digit (excluding the check digit) and moving left, double the va located immediately left of the check digit. If the result of this doubling operation is final result can be found by subtracting 9 from that result (e.g., 16: 16 – 9 = 7, 18:
2. Take the sum of all the digits.
3. If the total modulo 10 is equal to 0 (if the total ends in zero) then the number is val

Assume an example of an account number "7992739871" that will have a check digit adc

Account number	7	9	9	2	7	3	9	8	7	1	**x**
Double every other	7	**18**	9	**4**	7	**6**	9	**16**	7	**2**	**x**
Sum digits	7	**9**	9	4	7	6	9	**7**	7	2	**x**

The sum of all the digits in the third row, the sum of the sum digits, is 67.

The check digit (x) is obtained by computing the sum of the sum digits then computing 9

1. Compute the sum of the sum digits (67).
2. Multiply by 9 (603).
3. 603 mod 10 is then 3, which is the check digit. Thus, **x=3**.

6. ENHANCE. ENHANCE.

Your first task is to add logic to this program that checks whether the credit card number being read in satisfies the Luhn algorithm. If you need a refresher, revisit ZOAU2.

There's a few ways of doing this, and you should use whatever approach you like. Check out steps 7-9 for some tips on how to start and approach this problem.

You may want to break apart the logic for checking and writing into two separate subroutines, or keep the whole thing in one subroutine. You wouldn't be wrong to use three or more, either.

When done, your output can be simple output that writes to the screen, a detailed text file, or anywhere in between. This is a utility you're writing for yourself, and you'll need it to work to move to the final task of this challenge.

HOW DOES REXX GET USED TODAY? IS IT A WIDELY-USED LANGUAGE?

These days, you won't find Rexx used outside of a mainframe unless it's a very particular use case. However, it fits a need for a simple scripting language that speaks mainframe, and for many many years, that has fit the need of countless Systems Programmers all around the world. Getting rid of it would be like asking a carpenter to give up their hammer.

With that said, there are a lot more scripting languages available on the platform today that you may be more comfortable with. Python is immensely popular, and is also getting good at "speaking mainframe", but if you want to impress the higher-ups, Rexx is something you definitely want a strong command of.

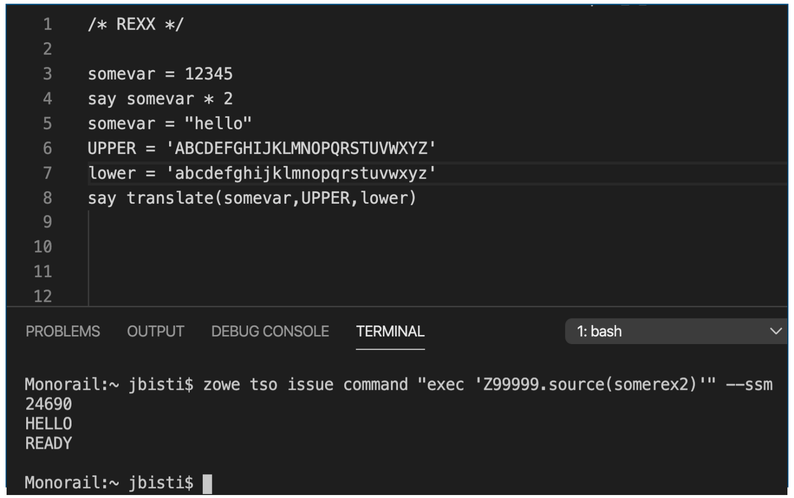
```
"FREE FI(outdd)"
EXIT

INSPECT:
  say 'inspecting' cc_digits
RETURN
```

7. SUBROUTINE LOGIC

Subroutines are simple, but may not work the way you expect them to, especially when coming from another language. The Knowledge Center has some useful information.

Be aware that subroutine code has access to all existing variables, so use that to your advantage.



8. STILL NOT MY TYPE

Rexx is what's known as a "Typeless" language. It looks at variables, input, and output as variable length strings. That means if variable X is a number, it will be able to multiply it by 2. If it's letters, it should be able to set those letters to uppercase. This makes some things a lot easier to pull off, especially when dealing with numerical data that you want to read in one digit at a time, HINT HINT HINT.

```
/* Again with:  Numeric digits 5 */
2**3        ->     8
2**-3       ->     0.125
1.7**8      ->    69.758
2%3         ->     0
2.1//3      ->     2.1
10%3        ->     3
10//3       ->     1
-10//3      ->    -1
10.2//1     ->     0.2
10//0.3     ->     0.1
3.6//1.3    ->     1.0
```

9. TOOLS FOR YOUR TOOLBOX

Just a few methods you may find useful in this task:

SUBSTR – Substring – returns just the characters at a specific location within a string. There's an example of this in action at line 40.

LENGTH – Returns the total length, in characters, of a string.

MATH OPERATORS – In particular, the // symbol, which returns the remainder after dividing by a number.

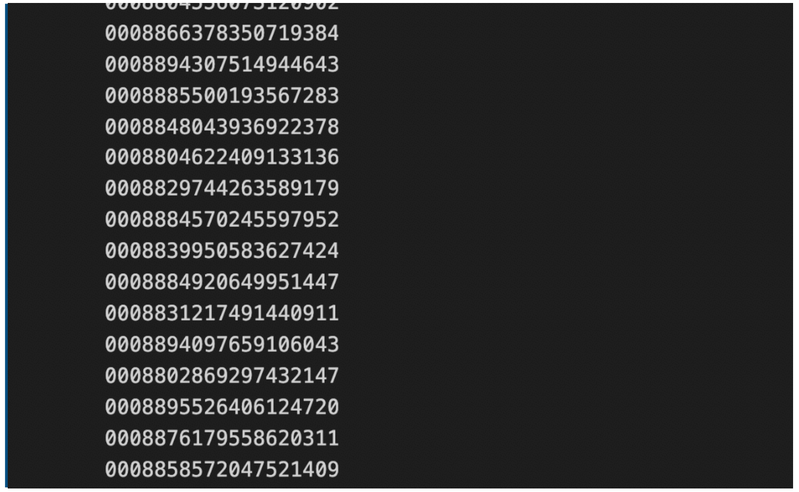
In your approach to build your checker, remember that you have a 1/10 chance of getting a valid number just by generating random numbers. It's that last digit, the check value, that needs to validate against the digits that came before it.

"MY LOGIC WORKS, BUT IT DOESN'T PASS THE TEST. WHAT GIVES?"

The internet is a magical place. You can find anything there, including differing opinions on facts.

We've found that there's a number of implementations of the Luhn Algorithm out there, and they don't all add up the same way... literally. The algorithm we're using to check your work is the same one we used back in ZOAU2. Make sure you start counting from the right digit, and from the right direction. This is one of those cases where writing a few extra lines of code to be clear is better than trying to be clever.

The long andshort of what we're saying here is: Don't count on internet examples unless you understand them enough to fix them.

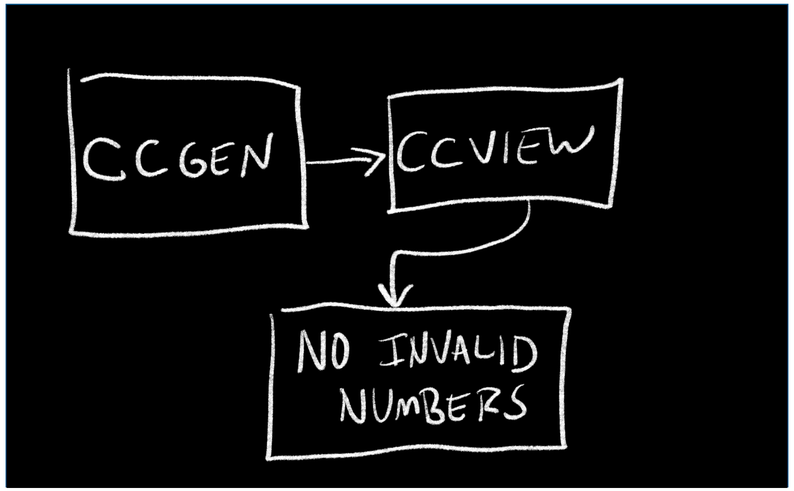


10. FLIP THAT AND REVERSE IT

Once you're happy with your code, finish this assignment by building a generator that outputs Luhn-algorithm compatible 16-digit numbers.

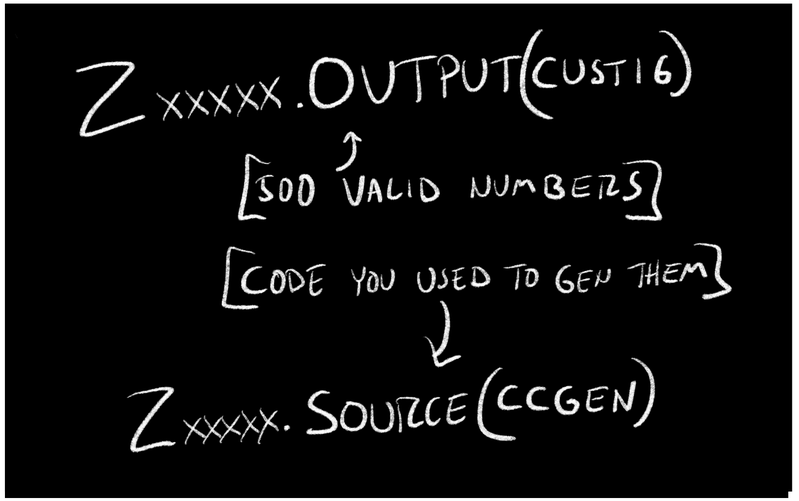
Put your working code in your SOURCE data set, with the name CCGEN. It should require no parameters, and output 500 unique numbers, one on each line, no need to generate names or other information, just the digits starting in the first column, one luhn algorithmn compatible entry per line, just like the above screenshot, please.

You should be able to re-use a lot of the code from the CCVIEW program, so we suggest starting by just copying that code over, instead of starting from scratch. Don't let us stand in the way of a good time, though, if you want to build something from the ground up, that could be fun too.



11. VALIDATING THE OUTPUT

In this case, your CCVIEW program will be a good test of your new program's output. If it's able to spot the invalid entries in the original set, it should be able to work on your new output once you adjust a few bits to make it look in the right spot. In fact, you may want to integrate the code from one into the other so it can generate, and self-validate. Again, do what you think works best, we're just throwing a few ideas out there.



12. MAKE IT COUNT

Put your output in your OUTPUT data set with the name CUST16. We'll be checking that, as well as the validity of your CCGEN code, so no dirty tricks, please.

This one's not rocket science, but when done, your Rexx skills will be up there with the best of 'em. Take your time, take breaks, and remember to comment your code.

Any code can be made better by including good comments. When done, submit **CHK3**.

NICE JOB! LET'S RECAP

If you want to make an experienced mainframe System Programmer angry, tell them Rexx is a useless language. While it is certainly not on mainstream Top 10 language lists, anything worth doing in a mainframe environment is worth rewriting into Rexx scripts. Said another, more simple way, is this: People Love Rexx.

Hopefully by the time you complete these tasks, you see just why it's so handy, and why so many simple automation tasks get tackled in Rexx.

NEXT UP...

Part 3 is really a strength-building series of challenges. If you're looking for more programming experience, jump over to COBOL. For those wanting to do more scripting, Zowe CLI ought to do the trick, and Ansible tackles more z/OS task automation. It's all good, though, just keep moving.