

2 Introduction to the REXX programming language

2.1 WHAT IS REXX?

The IBM brochure **TSO-E REXX Reference**, form number SA22-7790 contains the following short description:

The REstructured extended eXecutor (REXX) language is particularly suitable

- Command procedures
- Application front ends
- User-defined macros (such as editor subcommands)
- Prototyping
- Personal computing.

Individual users can write programs for their own needs.

REXX is a general-purpose programming language like PL/I. REXX has the usual structured-programming instructions – IF, SELECT, DO WHILE, LEAVE, and so on – and a number of useful built-in functions.

The language imposes no restrictions on program format. There can be more than one clause on a line, or a single clause can occupy more than one line. Indentation is allowed. You can, therefore, code programs in a format that emphasizes their structure, making them easier to read.

There is no limit to the length of the values of variables, as long as all variables fit into the storage available.

So far the short description contained in the official publication concerning the REXX language under z/OS.

The following excerpt contained in the preface of Michael Cowlishaw's book explains his basic ideas for developing REXX. See section 1.3.1.2 The REXX Language on page 2.

The REXX programming language has been designed with just one objective. It has been designed to make programming easier than it was before, in the belief that the best way to foster high quality programs is to make writing them as simple and as enjoyable as possible. Each part of the language has been devised with this in mind; getting the design right for people to use is more important than providing for easy implementation.

A programming language is a complex structure, typically characterized by its most visible aspect—its syntax. Of equal importance is its semantics, the meaning behind the instructions. But perhaps most important of all is the philosophy behind the language—the guiding principles that governed the decisions made as the language was designed.



I met Michael Cowlishaw at the 25th anniversary of REXX and he told me that during the development of REXX in the following sentence hung on the wall his office:



Mike Cowlishaw gave us with REXX a language that really of a minimal vocabulary consists. The programs developed with REXX are indeed small.

2.2 OVERVIEW OF REXX UNDER TSO

REXX is a procedural language. This means that you need an interpreter to execute a REXX procedure. The TSO contains a REXX procedure interpreter. That is, REXX programs runs in TSO like a TSO command. Before the REXX procedure language interpreter was integrated in the TSO, there was already the CLIST interpreter available. Therefore, a method had to be introduced where the TSO can be see that a program contains REXX code and it is not a CLIST.

2.2.1 Recognizing a REXX procedure by the TSO

The TSO Command Processor interprets a procedure as a REXX program if the first line of the program contains the text REXX or rexx. When the first line misses one of the words REXX or rexx, then the TSO command processor assumes that it is a procedure of type CLIST and passes the procedure to the CLIST procedure processor. This requirement means that the first line of a REXX program executed in the z/OS TSO must always be a comment line containing the text REXX or rexx. As an example, here the program head of one of my REXX procedures:

2.2.2 Running REXX procedures in the TSO

Call and execute REXX procedures as follows in the TSO:

1. Explicitly by the TSO EXEC command.

When using the TSO command EXEC to execute a REXX program member, it is determined directly in the EXEC command operands. This means that a member that contains a REXX procedure is executable from each PDS.

Example

TSO EXEC 'dsn(member)'

The EXEC command still knows a number of options that are only interesting for special cases. See the command description in the manual TSO-E Reference in chapter EXEC Command.

2. Implicitly by calling up as a TSO command.

In this case, there are several possibilities to call up a REXX program:

- Only the name of the REXX procedure as a TSO command is entered. E.g.: TSO pgm. In this case all allocated system and user load libraries are searched for this program name. If the name is not found there, then the SYSEXEC and SYSPROC libraries are seanned.
- To call a REXX program you can alternatively use the command TSO
 %name. In this case only the SYSEXEC and SYSPROC libraries are scanned for the program.
- When a REXX program to be executed is not found in any of the existing allocated libraries, you can use the command ALTLIB to allocate the required library temporarily and then use one of the two methods mentioned above for calling. See https://doi.org/10.1007/j.chm/dis/normal/ ALTLIB Dynamic linking of EXEC libraries on page 126.

3. Implicitly by calling up as an ISPF command in a DSLIST display panel.

To execute this type of call see the following panel:

Screen 2.1: Example of a command call in a DSLIST display panel

Menu	Options	View	Utilities	Compilers	Help	
DSLIST - Data Sets MatchingPROX.PACKTIME.*						Row 1 of 6
Command	>				Scro	11> CSR
Command	- Enter	•/• to	select act	ion.	H essage	Volume
	PROX.PAG	KTIME.	DB2P			POXYZ6
compdat	e / april	KTIME	.DGP1			POXYZ6
	PROX.PAG	KTIME.	DGP2			POXYZ6
	PROX.PAG	KTIME.	DGP3			POXYZ6
******	******	*****	**** End of	Data Set 1	.1st ********	***********

In this case, the following will happen

The program compdate is searched in all allocated LOAD and EXEC libraries. If it is found, it is executed and the DSN standing in this line and the text april are passed as parameters to the program. For this type of call, a slash (/) represents the DSN standing in this line. Depending on the position where the program expects the DSN in the takeover of the parameters, the slash is to be in accordance with the position.

4. Implicitly invoked using the ISPF SELECT service

The following example shows the execution of a small REXX program using the

1

mem = 'compdate'
pp = " PROX.PACKTIME.DGP1 april'
"ALTLIB ACTIVATE APPLICATION(EXEC) DDNAME(##DD)'
"ISPEXEC SELECT CMO('mem strip(pp)')'
"ALTLIB DEACTIVATE APPLICATION(EXEC)'

This is the same execution of the program compdate as shown in Screen 2.1 on page 10.

For information about ISPF SELECT service, refer to the IBM ISPF Services
Guide. Information on the use of ALTLIB command, see section 7.5 ALTLIB –
Dynamic linking of EXEC libraries on page 126. The ALTLIB command also
has a display option. I once had the above program executed without the ALTLIB
DEACTIVATE command and then ran the command ALTLIB DISPLAY
manually. I received the following display:

2.3 COMPILE REXX PROCEDURES

In some z/OS systems, a REXX compiler is available. Therefore, you can compile REXX procedures before executing them. The REXX compiler is also able to produce genuine z/OS load modules. These load modules can be executed in a batch job using the z/CL statement EXEC PGM=NAME. Since the focus of this book is the ISPF and the REXX, compiler topics would go beyond the scope of this book too far, unfortunately I have to refrain from treating the REXX compiler. However, you can find in the SMART ISPF utilities procedures to generate cexee and load modules using the REXX compiler. See Section: 15.5 Programming aids on page 273 and the subsequent pages.

2.4 PERFORMANCE OF REXX PROCEDURES

When running REXX procedures, each statement must be interpreted by the system. This, of course, requires computing time, which is added to the time of execution of the REXX commands. The currently available host processors are so fast that in practice hardly an execution time difference between an interpreted REXX procedure and a compiled program are detectable. If there is concern about the performance of REXX procedures, you should first take a closer look at the greatest performance brakes. The following operations normally slow down the execution of a REXX program the most:

- I/O operation
- Loading of external procedures and load modules
- Calling external system functions

Therefore, it is advisable to always start first at these points with actions for performance improvement. You should not expect too much of compiled REXX procedures because of the following reasons:

Performance tips

1. Compiled programs:

The time needed to perform the called ISPF and TSO functions is usually much higher than the execution time required for the pure REXX code. Thus, it is usually not advisable to use compiled REXX procedures. I have gained this knowledge from working in a very large project with hundreds of compiled REXX programs.

2. Functions and subroutines

Functions and subroutines that do not belong to the scope of functions of REXX are loaded every time when they are called from the external data set. It is obvious that this can be a significant performance brake. I strongly recommend investigating your programs for such performance brakes and if necessary, copying the source code of these programs into the main program. The fastest way to detect such a speed brake is the execution of this program in a batch job. After the program execution, you can check the SDSF as to how many EXCPs the program has consumed. If the EXCP count is exorbitantly high, then it is obvious that a function or subprogram was continuously loaded.

2.5 THE SYNTAX OF THE REXX LANGUAGE

The syntax of the REXX language is very simple and strongly influenced by

- A statement is normally always in a row. A semicolon normally terminates it.
 However, it does not have to be explicitly completed!
- Multiple statements on one line must each be terminated with a semicolon except the last one. The last statement can of course also terminated with a semicolon.
- If a statement is continued on the next line, then the preceding line must end with a comma (.). This also applies to literals. The continuation comma itself is always replaced by a blank. If this blank should be avoided, then the two literal parts must be concatenated with the concatenation operator (II).



- Comments always begin with /* and end always with */. All texts between these two limits belong to the comment even if they span multiple lines and contain REXX commands. Comments can be wherever a blank is allowed be placed within a statement and they can be nested. Note: When the pairing of the comments is interrupted, it may happen that the rest of the program remains as a comment.
- Literals are put in paired quotation marks or apostrophes. There are three types of literals:
- Text literals. These literals can contain any text, and it can be defined over several lines. One literal must not exceed 250 characters.
- Hexadecimal literals. They can only contain hexadecimal digits and must be completed by the letter x or X. The individual bytes can be, separated by blanks, entered.
- Binary literals. These can only contain 0 and 1 and must be completed by the letter b or B respectively. The individual bytes and half-bytes can be separated by blanks.

Examples of correct REXX statements:

 $a=5;\;B="FF\;FF\;FF\;FF^*x;\;L="1111\;0000\;1111\;0001\;1111\;0010"b\;X=1\;/^*$ Set initial value for X */ do k=1 to 50; m.k = 0; /* Set initial values to zero */ end k;

Here you see an example of continuation lines in the definition of text literals. Keep in mind that the continuation character in the resulting text appears as blank. Therefore, you should always define continuation lines of literals so that the resulting text contains a blank at this point anyway.

if sysdem(newdem) <> "CK" them do
 zedimeg = "The COPY data set does not exist."
"Please enter the name of an existing data set"
"ISPEXECT SEA another function."
"ISPEXED TO MED (ISECOL)"
meglv11 = "Error at COPY"
iterate ipanel

Rules for inserting comments and blanks in REXX statements.

After defining the rules, you will see a sample program that shows the applications of the rules. In the description of each rule there is always reference to the program line to which this rule applies.

- In general, comments that are inserted in a statement will be removed before executing the statement. Lines 06, 07, 08.
- If the statement after the removal of the comments is executable, it is executed.
- $-\,$ If multiple blanks between the elements of a statement are present, they will be removed down to a single blank. Line 05.
- For functions that contain parameters in a pair of parentheses, a blank is never allowed between the function name and the opening parenthesis. Line 10: The blank in column 30 that remains when the comment is removed, is not allowed.

Program 2.1: Example for handling blanks and comments in REXX programs

If this program is running, the following output is displayed:

First word Second word
First wordSecond word
First word Second word
First word /**/ Second word
26 May 201516:44:17
10 +++ say date/* date */()/**/time /* time */()
Error running TEST3, line 10: Unexpected *, * or *)"

2.6 VARIABLES IN REXX

Rules

- REXX variables cannot be explicitly defined before use
- The type of a variable can only be CHAR or NUM. It is always defined by the content of the variable. The type is determined after each assignment of a value. This means that, REXX variable can continuously change their data type between CHAR and NUM depending on which data was assigned last.
- The length of the variable name can have a maximum of 250 characters.

1

- A variable that is not the type NUM by functions or external input assigned must be explicitly assigned a numerical value before it can be used in an arithmetic operation.
- The length of a character string variable must not exceed 16MB.

One of the most elegant facilities in REXX is that variables do not need to be defined prior to use. I remember that using other programming languages, I very often got the error message Undefined variable during program execution Sometimes I needed several test runs until I found all the missing variables.

Example:

When you perform the statement:

say hugo is my boy friend

The display is:

hugo is my boy friend

When execute the following statement.

hugo_is_my_boy_friend = "fred"

and you perform the above say statement again

say hugo_is_my_boy_friend

The display is now: fred

2.7 DATA TYPES OF REXX VARIABLES

As mentioned above in REXX knows only two data types of variables. These are NUM for a numerical content and CHAR for all other contents As explained above, the data type is determined internally on each assignment of a value to a variable. It cannot be explicitly specified in the program, but the function DATATYPE(var) can used to query the data type of a variable. Here is a short list showing the data types of the variable HUGO after an appropriate assignment of values:

Assignment to hugo	Content/Value of hugo	Datatype
hugo = "250"	250	NUM
hugo = 250	250	NUM
hugo = "25 0"	"25 0"	CHAR
hugo = date()	"14 May 2004	CHAR
hugo = date("S")	20040514	NUM
hugo = "F0 F1"x	01	NUM
hugo = "C1 F1"x	"A1"	CHAR
hugo = "1111 0001 1111 1001"b	19	NUM



The resulting data type is only NUM when an admissible number in the sense of REXX is assigned to it. Permissible blanks in HEX and BIN strings are removed by the REXX assignment command.

Null strings:

A special type of data is the null string. This type of data is often assigned to variables if no result of an executed command is present. The null string is determined as a literal by two consecutive apostrophes. Null strings are of data type CHAR and have a data length of 0 (zero). Examples of definition and query of null strings:

2.8 OPERATORS OF THE REXX LANGUAGE

The operators define the operations which with variables and constants are to be performed. Types of operators:

- String operators
- Arithmetic operators
- Logical operators

2.8.1 String operators

Here we meet one of the greatest strengths of REXX. To merge text, you simply write the various constants and variables in one statement. The following rules apply:



- If in a statement between the constants and variables there is more than one blank, the result is at this point only a single blank.
- If you want to prevent the automatic insertion of blanks, elements with the concatenation operator (||) need to be connected. Blanks that stand on the left or right of the concatenation operator are ignored in this c5ase
- If you want to insert multiple blanks, they must be defined as a literal in
- The concatenation operator is not necessary when individual elements of the statements can be clearly recognized by the interpreter.

This program example shows how texts can be assembled.

```
say "1' date() time()
say "2' date() time()
say "3' date()
say "4' date() || time()
say "4' date() || time()
say "6' date(),
time()
say "6' date(),
time()
say "7 The date of the present day is: 'date()
say "8 The date of the present day is: '||,
date()
```

When executed, the program produces the following list output:

```
1 24 Mar 2015 18:29:24

2 24 Mar 201518:29:24

3 24 Mar 201518:29:24

4 24 Mar 201518:29:24

5 24 Mar 2015

6 24 Mar 2015

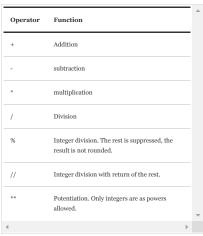
7 The date of the present day is:24 Mar 2015

8 The date of the present day is:24 Mar 2015
```

2.8.2 Arithmetic Operators

The following table lists the arithmetic operators:

Table 2.1: Arithmetic REXX operators



The + and - signs can be set as a sign in front of variables and constants

```
a = 3
b = a * 3 % 5
c = -a
d = a * 3 // 5
say a b c d c+d a*100/10*%"
```

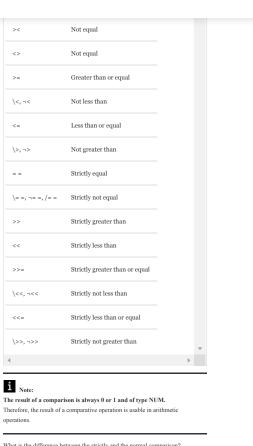
When executed, the program produces the following list output: 3 1 -3 4 1 30%

2.8.3 Compare operators

Table 2.2: Compare operators







What is the difference between the strictly and the normal comparison?

- If at least one of the compared values is of type CHAR then both leading and trailing blanks are removed. Then the shorter value is padded with blanks until it has the length of the other value. Then the comparison is performed.
- If both comparative values are of type NUM, one value is subtracted from the other one. The result is then greater than zero, equal to zero or less than zero.

In strictly comparison:

- No padding of CHAR type values take place.
- Values of type NUM are compared character by character from left to right rather than with a subtraction.
- Values of different lengths are always NOT equal.

The following example shows how to use comparison operators in arithmetic instructions. This subroutine calculates the "Julian Day" within a year in a

Screentext 2.1: Example for using combinations of arithmetic and comparative

```
year of date 2004.12.31 is: "jd2("2004 12 31")
say "Day in the year of date 2004.12.31 is: "jd2("2004 12 31")
say "Day in the year of date 2005.12.31 is: "jd2("2005 12 31")
say "Day in the year of date 2002.03.01 is: "jd2("2006 03 01")
say "Day in the year of date 2014.02.29 is: "jd2("2016 02 29")
exit
procedure
arg yryy mm dd
crim="2" jd2 ("2016 02 29")
crim="1" jd2 ("2016 02 29")
crim="1"
```

This program produces the following printout:

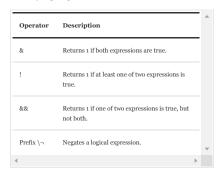
Day in the year of date 2004.12.31 is: 366 Day in the year of date 2005.12.31 is: 365 Day in the year of date 1999.12.31 is: 365

See also Program 15.6: Subroutine JULDATE on page 282.

2.8.4 Logical operators

The logical operators are usually only used in logical statements, such as IF, WHEN etc. The following table shows the logical operators:

Table 2.3: Logical operators



2.9 STEMS IN REXX

Stem are something similar to dimensioned arrays in other programming languages. STEMS start with a variable name followed by a dot. The values behind the point are something similar indices. These indices can be **numbers** and any text.

In the REXX literature, stems are also referred to as Compound Variables. This means that a variable hugo.1 as long as the content HUGO.1 has until it is be assigned a different value. In simpler terms: Each stem variable per se is a normal variable; it is only written differently. In addition, you can work based on composition by the details behind the points easier with a whole set of variables each with an own name.

My experience is for understanding the stems, beginners in REXX are faced with greater difficulties. The following small examples show how this technology works

Example program for using stems:

Task:

The dataset user001.flight.data(flights) contains the flight log of the performed flights. I wanted to know how many flights I have on the individual aircraft types carried out and how many hours with each type. I did not know beforehand how many aircraft types were used in total. During the program run, the types must be successively detected and stored

Program 2.2: Program FLIGHTS to explain the using stems and EXECIO

```
/* DOC. PLIGHTS MEXIMATE
/* DOC. Calculating the number of flights per aircraft type and the '/
/* DOC. Calculating the number of flights per aircraft type and the '/
/* DOC. Calculating the number of flights are calculated to the '/
/* DOC. Calculating to the calculated the '/
* The Calculated the '/
* Alloc Sdilini' the '/
* The Calculated the '/
* The Structure of the '/
* The Structure of the '/
* The Structure of '/
* The Structure
```

See the output of the program:

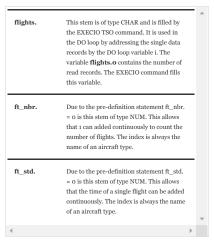
```
C152 336 96.02
C172 140 85.66
TB20 80 74.31
TB09 16 8.18
TB10 26 28.15
PA28 28 43.87
C182 56 51.23
C310 4 5.70
```



```
PN68 17 11.42
C210 200 209.44
BB36 16 16.73
PA44 8 11.83
PA34 6 6.55
C303 19 16.56
C340 28 25.80
C421 28 30.93
SR20 29 34.49
DA42 52 73.57
```

Explanation:

There are three stems used in this program:



This example also shows how stems are used for data set processing!

2.9.1 Initialize stems with null string

The following program example illustrates the effect of stems null string assignments:

The program prints the following lines:

```
>10< >111<
>< >222<
>20< >JANE.3<
>< >JANE.4<
>< >JANE.5<
```

As you can see, not all explicitly value assigned positions of the stem **hugo.** are as null strings printed.

2.9.2 File processing in connection with stems

In the Program FLIGHTS to explain the using stems and EXECIO on page 20, shows an example how stems are used in the file processing. I will now explain this tonic in principle.

When processing data sets, REXX can read all records of a file with a single command in a stem or write all elements of a stem to a data set. Let us look at an example of data set processing:

```
address "TgO" 'alloo dd(save) dsn("savedsn') shr reuse' address "TgO" 'execto ' diskr save (stem save, finis' address "TgO" 'free dd(save)'
```

The EXECIO command reads all the records of the file in the stem save.. You will now ask: Where is the information about how many records are read?



All functions that populate a REXX stem from the outside with a variable number of records, write the number of records in the index 0 (zero) of the stem In the above example, after the EXECIO command has been executed contains save 0 the number of records read.



several ones. The assignments look like this: name = "MEYER"
HUGO.name.city = "New York"
HUGO.name.street = "Hudson Road"
say HUGO.MEYER.city
say HUGO.MEYER.street The SAY output is then: New York Hudson Road If you are using texts in indices of stems, of course, you must be sure that you address these variables properly. As an example, see Program 2.2: Program FLIGHTS to explain the using stems and EXECIO on page 20. As for all variables in REXX, also applies for STEMS the rule, that a stem variable as long their own name in uppercase letters as content has until it gets explicitly another content. This rule applies only when the stem was by a general null string assignment not predefined. For clarification, we look at the following hugo.1.1 = 25 hugo.1.2 = "XXXX" hugo.2.1 = "Mill arrived at "date()" hugo.2.3 = "finished" do i = 1 to 2 do k = 1 to 3; say ">"hugo.i.k"c"; end k ed 1 The result: >25< >XXXX< >HUGO.1.3< > Will arrived at 8 Apr 2004< >HUGO.2.2< >finished< The gray stem variables were not assigned to a value, so their names appear as the content. When I initialize hugo, with null string the result looks like this: The result is then: >25< >XXXX< >< > Will arrived at 8 Apr 2004< >< >finished< See also the Remark in section 2.6 Variables in REXX on page 14

You can also create stems, which are not only divided by one point, but also by

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out





