**The REXX Tutorial**

**Contents**

* [Introductory text](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#intro)
* [1. Creating a Rexx program](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#A)
* [2. Doing arithmetic](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#B)
* [3. Errors](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#C)
* [4. Untyped data](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#D)
* [5. More on variables](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#E)
* [6. Functions](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#F)
* [7. Conditionals](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#G)
* [8. Loops](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#H)
* [9. Parsing](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#I)
* [10. Interpret](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#J)
* [11. The stack](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#K)
* [12. Subroutines and functions](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#L)
* [13. Executing commands](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#M)
* [14. Signal](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#N)
* [15. Tracing](https://www.cs.ox.ac.uk/people/ian.collier/Docs/rexx_info/whole.html#O)

**Introductory text**

Note: some of the information in this file is system-specific, though

most of it pertains to every implementation of Rexx.

# Lines containing implementation-specific information are flagged with

# letters in column 1. The letter "I" denotes information for REXX/imc.

# I have also started adding information about OS/2 Rexx, which will be

# flagged with "O" in column 1. I hope it is not too confusing to see some

# sections written twice, once for each system. The file is designed so

# that if you run it through "egrep '^(x| |$)'|cut -c3-" (where x is the

# desired letter) then it should still make sense and cut out the flag

# letters. Doing this will select one of the following lines indicating

# which system was selected; the characters to the left of this paragraph

# will make sure it gets deleted when that happens.

I This file describes REXX/imc.

O This file describes OS/2 Classic Rexx (which is also pretty much

O compatible with the OS/2 Object Rexx interpreter).

I More advanced information can be found in rexx.summary (bare details

I of each command and builtin function, with a list of differences from

I standard Rexx) and rexx.ref (technical details of every aspect of this

I Rexx implementation).

O More information is available in the OS/2 help system. For example,

O typing "help rexx signal" will give the syntax of the Rexx "signal"

O instruction.

**1. Creating a Rexx program**

Many programmers start by writing a program which displays the message

"Hello world!". Here is how to do that in Rexx...

I Write a file called "hello.rexx" containing the following text. Use any

I text editor or simply `cat' the text into the file.

O Write a file called "hello.cmd" containing the following text. Use any

O text editor (for example, E).

Note that the text, as with all example text in this guide, starts at the

first indented line and ends at the last. The four spaces at the start of

each line of text should not be entered.

/\* This program says "Hello world!" \*/

say "Hello world!"

This program consists of a comment saying what the program does, and an

instruction which does it. "say" is the Rexx instruction which displays

data on the terminal.

The method of executing a Rexx program varies greatly between

implementations.

I Here is how you execute that file using REXX/imc:

I

I rexx hello

I

I Notes about this command: rexx is the name of the interpreter. In order

I for this command to work, the interpreter must lie on your current PATH.

I If you require more information about this then you should contact the

I person who installed REXX on your system.

I

I The word "hello" which comes after the command "rexx" is the name of your

I program. Ordinarily, the interpreter adds ".rexx" to the name you give in

I order to construct the name of the file to execute.

I

O Here is how you execute that file on OS/2:

O

O hello

O

When you execute your first Rexx program using the method detailed above,

you should see the message "Hello world!" appear on the screen.

**2. Doing arithmetic**

Rexx has a wide range of arithmetic operators, which can deal with very

high precision numbers. Here is an example of a program which does

arithmetic.

I Make a file called "arith.rexx" containing the following.

O Make a file called "arith.cmd" containing the following.

/\* This program inputs a number and does some calculations on it \*/

pull a

b=a\*a

c=1/a

d=3+a

e=2\*\*(a-1)

say 'Results are:' a b c d e

I Run it with "rexx arith" and type in a positive integer.

O Run it by typing "arith" and type in a positive integer.

Here is a sample run:

I rexx arith

O arith

5

Results are: 5 25 0.2 8 16

The results you see are the original number (5), its square (25), its

reciprocal (0.2), the number plus three (8) and two to the power of one

less than the number (16).

This example illustrates several things:

\* variables: in this example a, b, c, d and e are variables. You can

assign a value to a variable by typing its name, "=", and a value,

and you can use its value in an expression simply by typing its name.

\* input: by typing "pull a" you tell the interpreter to ask the user for

input and to store it in the variable a.

\* arithmetic: the usual symbols (+ - \* /) as well as \*\* (to-power) were

used to perform calculations. Parentheses (or "brackets") were used

to group together an expression, as they are in mathematics.

\* string expressions: the last line of the program displays the results by

saying the string expression

'Results are:' a b c d e

This has six components: the string constant 'Results are:' and the

five variables. These components are attached together with spaces

into one string expression, which the "say" command then displays on

the terminal. A string constant is any sequence of characters which

starts and ends with a quotation mark - that is, either " or ' (it makes

no difference which you use, as long as they are both the same).

If you supply the number 6 as input to the program, you should notice

that the number 1/6 is given to nine significant figures. You can easily

change this. Edit the program and insert before the second line:

numeric digits 25

If you run this new program you should see that 25 significant figures are

produced. In this way you can calculate numbers to whatever accuracy you

require, within the limits of the machine.

At this point it seems worth a mention that you can put more than one

instruction on a line in a Rexx program. You simply place a semicolon

between the instructions, like this:

/\* This program inputs a number and does some calculations on it \*/

pull a; b=a\*a; c=1/a; d=3+a; e=2\*\*(a-1); say 'Results are:' a b c d e

Needless to say, that example went a bit over the top...

**3. Errors**

Suppose you ignored the instructions of the previous section and typed a

non-integer such as 3.4 as input to the program. Then you would get an

error, because the \*\* (to-power) operator is only designed to work when

the second parameter (that is, the power number) is an integer. You might

see this, for example:

I rexx arith

I 3.4

I 6 +++ e=2\*\*(a-1)

I Error 26 running arith.rexx, line 6: Invalid whole number

O arith

O 3.4

O 6 +++ e = 2 \*\* ( a - 1 );

O REX0026: Error 26 running D:\ARITH.CMD, line 6: Invalid whole number

Or if you typed zero, you might see the following (because you cannot

divide by zero):

I rexx arith

I 0

I 4 +++ c=1/a

I Error 42 running arith.rexx, line 4: Arithmetic overflow or underflow

O arith

O 0

O 4 +++ c = 1 / a;

O REX0042: Error 42 running D:\ARITH.CMD, line 4: Arithmetic

O overflow/underflow

Perhaps most interestingly, if you type a sequence of characters which is

not a number, you might see this. It does not complain about the

characters you entered, but at the fact that the program tries to use it

as a number.

I rexx arith

I hello

I 3 +++ b=a\*a

I Error 41 running arith.rexx, line 3: Bad arithmetic conversion

O arith

O hello

O 3 +++ b = a \* a;

O REX0041: Error 41 running D:\ARITH.CMD, line 3: Bad arithmetic conversion

In each case, you have generated a "syntax error" (it is classified as a

syntax error, even though the problem was not directly related to the

program's syntax). What you see is a "traceback" starting with the line

which caused the error (no other lines are listed in this traceback,

because we have not yet considered any Rexx control structures), and a

description of the error. This information should help you to determine

where the error occurred and what caused it. More difficult errors can be

traced with the "trace" instruction (see later).

**4. Untyped data**

In the previous section, you found that you could input either a number or

a sequence of letters and store it in the variable a, although arithmetic

can only be performed on numbers. That is because data in Rexx are

untyped. In other words, the contents of a variable or the result of an

expression can be any sequence of characters. What those characters are

used for matters only to you, the programmer. However, if you try to

perform arithmetic on a random sequence of characters you will generate a

syntax error, as shown in the previous section.

You have seen that you can add strings together by placing a space in

between them. There are two other ways: the abuttal and the concatenation

operator. An abuttal is simply typing the two pieces of data next to each

other without a space. This is not always appropriate: for example you

cannot type an "a" next to a "b", because that would give "ab", which is

the name of another unrelated variable. Instead, it is safer to use the

concatenation operator, ||. Both these operations concatenate the strings

without a space in between. For example:

/\* demonstrates concatenation and the use of untyped data \*/

a='A string'

b='123'

c='456'

d=a":" (b||c)\*3

say d

The above program says "A string: 370368". This is because (b||c) is the

concatenation of strings b and c, which is "123456". That sequence of

characters happens to represent a number, and so it can be multiplied by 3

to give 370368. Finally, that is added with a space to the concatenation

of a with ":" and stored in d.

**5. More on variables**

The previous examples only used single-letter variable names. In fact

it is more useful to have whole words as variable names, and Rexx allows

this up to an implementation maximum (which should be suitably large, e.g.

250 characters). Moreover, not only letters but numbers and the three

characters "!", "?" and "\_" are allowed in variable names - or "symbols",

as they are generally called. These are valid symbols:

fred

Dan\_Yr\_0gof

HI!

The case of letters is unimportant, so that for example "Hello", "HELLO"

and "hellO" all mean the same.

If you use a symbol in an expression when it has not been previously given

a value, that does not cause an error (unless "signal on novalue" is set -

see later). Instead, it just results in its own name translated into

upper case.

/\* A demonstration of simple symbols \*/

foo=3

bar="Characters"

say foo bar':' hi!

This program says "3 Characters: HI!".

As well as "simple symbols", which are variables like the above, there

are arrays (strictly speaking, they are not called arrays in Rexx but

"stem variables", though equivalent things in other languages are called

"associative arrays"). Any ordinary variable name can also be used as the

name of an array:

/\* This program uses an array. \*/

pull a

array.1=a

array.2=a\*a

array.3=a\*\*3

say array.1 array.2 array.3 array.1+array.2+array.3

An element of an array is accessed by typing the array name, a dot, and

the element number. The array name and the dot are together known as

the "stem" of the array. The rest of the element's name is known as the

"tail" of the name. The stem and tail together are called "a compound

symbol". Note that an array does not have to be declared before it is

used. Also note that the simple variable "a" and the stem "a." refer

to completely separate variables.

In fact not only numbers, but strings and variable names may be used as

element names. Also, an element name can consist of two or more parts

separated by dots, so giving two or more dimensional arrays.

/\* This program uses an array with various elements \*/

book.1.author="M. F. Cowlishaw"

book.1.title="The REXX Language, a practical approach to programming"

book.1.pub="Englewood Cliffs 1985"

book.2.author="A. S. Rudd"

book.2.title="Practical Usage of REXX"

book.2.pub="Ellis Horwood 1990"

/\* insert lots more \*/

say "Input a book number"

pull i

say "Author: " book.i.author

say "Title: " book.i.title

say "Publisher:" book.i.pub

In the above program, a stem variable called "book" is created, containing

a number of records each with elements AUTHOR, TITLE and PUB. Notice that

these three uppercase names are produced by symbols "author", "title" and

"pub", because those symbols have not been given values. When a book

number i has been input, the elements of the (i)th record are printed out.

It is not an error to reference an undefined element of a stem variable.

If you type "3" into the above program, you will see this:

Input a book number

3

Author: BOOK.3.AUTHOR

Title: BOOK.3.TITLE

Publisher: BOOK.3.PUB

As before, if a compound symbol has not been given a value, then its name

is used instead.

There is a way to initialise every element of a stem variable: by

assigning a value to the stem itself. Edit the above program and insert

after the comment line:

book.="Undefined"

This gives every possible element of the stem variable the value

"Undefined", so that if you again type "3" you will see the following:

Input a book number

3

Author: Undefined

Title: Undefined

Publisher: Undefined

**6. Functions**

The Rexx Summary contains a list of the functions which are available in

Rexx. Each of these functions performs a specific operation upon the

parameters. For example,

/\* Invoke the date function with various parameters \*/

say date("W")',' date()

This might say, for example, "Friday, 22 May 1992".

A function is called by typing its name immediately followed by "(".

After that come the parameters, if any, and finally a closing ")". Note

that you should not type a space between the name and "(", because that

space would be a concatenation operator, as in section 4. In the above

example, the "date" function is called twice. The value of date("W") is

the name of the weekday, and the value of date() is the date in "default"

format.

**7. Conditionals**

It is time to use some Rexx control structures. The first of these will

be the conditional. Here is an example:

/\* Use a conditional to tell whether a number is less than 50 \*/

pull a

if a<50 then say a "is less than 50"

else say a "is not less than 50"

The program is executed in the manner in which it reads - so, if a is less

than 50 then the first instruction is executed, and otherwise the second

instruction is executed.

The "a<50" is a conditional expression. It is like an ordinary

expression (in fact conditional expressions and ordinary numeric

expressions are interchangeable), but it contains a comparison operator.

There are many comparison operators, as follows:

= (equal to) < (less than) > (greater than) <= (less or equal)

>= (greater or equal) <> (greater or less) \= (not equal)

\> (not greater) \< (not less)

All the above operators can compare numbers, deciding whether one is equal

to, less than, or greater than the other. They can also compare

non-numeric strings, first stripping off leading and trailing blanks.

There are analogous comparison operators to these for comparing strings

strictly. The main difference between them is that 0 is equal to 0.0

numerically speaking, but the two are different if compared as strings.

The other difference is that the strict operators do not strip blanks

before comparing. The strict operators are

== (equal to) << (less than) >> (greater than) <<= (less or equal)

>>= (greater or equal) \== (not equal) \>> (not greater) \<< (not less)

Conditional expressions may be combined with the boolean operators:

& (and), | (or) and && (xor). They may also be reversed with the \ (not)

operator.

/\* Decide what range a number is in \*/

pull a

if a>0 & a<10 then say "1-9"

if a\<10 & a<20 then say "10-19"

if \ (a<20 | a>=30) then say "20-29"

if a<=0 | a>=30 then say "Out of range"

As well as demonstrating the boolean and comparison operators, this

program shows that the "else" clause is not required to be present.

The above program may also be written using Rexx's other conditional

instruction, "select":

/\* Decide what range a number is in \*/

pull a

select

when a>0 & a<10 then say "1-9"

when a\<10 & a<20 then say "10-19"

when \ (a<20 | a>=30) then say "20-29"

otherwise say "Out of range"

end

The "select" instruction provides a means of selecting precisely one from

a list of conditional instructions, with the option of providing a list of

instructions to do when none of the above was true. The difference is

that if no part of a "select" instruction can be executed then a syntax

error results, whereas it is OK to miss out the "else" part of an "if"

instruction.

Only one instruction may be placed after the "then" or "else" of

a conditional instruction, but Rexx provides a way of bracketing

instructions together so that they can be treated as a single instruction.

To do this, place the instruction "do" before the list of instructions and

"end" after it.

/\* execute some instructions conditionally \*/

pull a

if a=50 then

do

say "Congratulations!"

say "You have typed the correct number."

end

else say "Wrong!"

If you wish for one of the conditional instructions to do "nothing", then

you must use the instruction "nop" (for "no operation"). Simply placing

no instructions after the "then", "else" or "when" will not work.

**8. Loops**

Rexx has a comprehensive set of instructions for making loops, using the

words "do" and "end" which you met briefly in the previous section.

a. Counted loops

The instructions within a counted loop are executed the specified number

of times:

/\* Say "Hello" ten times \*/

do 10

say "Hello"

end

A variation of the counted loop is one which executes forever:

/\* This program goes on forever until the user halts it \*/

do forever

nop

end

b. Control loops

Control loops are like counted loops, but they use a variable (called

the control variable) as a counter. The control variable may count

simply in steps of 1:

/\* Count to 20 \*/

do c=1 to 20

say c

end

or in steps of some other value:

/\* Print all multiples of 2.3 not more than 20 \*/

do m=0 to 20 by 2.3

say m

end

It may take a specific number of steps:

/\* Print the first six multiples of 5.7 \*/

do m=0 for 6 by 5.7

say m

end

or it may go on forever:

/\* Print all the natural numbers \*/

do n=0

say n

end n

The "n" at the end of this last example is optional. At the end of any

controlled loop, the name of the control variable may be placed after

the "end", where it will be checked to see if it matches the control

variable.

c. Conditional loops

A set of instructions may be executed repeatedly until some condition is

true. For example,

/\* I won't take no for an answer \*/

do until answer \= "NO"

pull answer

end

Alternatively, they may be executed as long as some condition is true:

/\* It's OK to repeat this as long as there is no error \*/

do while error=0

pull a

if a="ERROR" then error=1

else say a

end

Note that in this example, the variable "error" must be zero to

start with. If there is already an error to start with then the

set of instructions will not be executed at all. However in the

previous example the instructions will always be executed at least

once. That is, the expression after an "until" is evaluated at the

end of the loop, whereas the expression after a "while" is evaluated

at the start of the loop.

d. Controlled conditional loops

It is possible to combine forms a or b with form c mentioned above, like

this:

/\* I won't take no for an answer unless it is typed three times \*/

do 3 until answer \= "NO"

pull answer

end

or this:

/\* input ten answers, but stop when empty string is entered \*/

do n=1 to 10 until ans==""

pull ans

a.n=ans

end

The "iterate" and "leave" instructions allow you to continue with, or to

leave, a loop respectively. For example:

/\* input ten answers, but stop when empty string is entered \*/

do n=1 to 10

pull a.n

if a.n=="" then leave

end

/\* print all integers from 1-10 except 3 \*/

do n=1 to 10

if n=3 then iterate

say n

end

If a symbol is placed after the instructions "iterate" or "leave", then

you can iterate or leave the loop whose control variable is that symbol.

/\* Print pairs (i,j) where 1 <= i,j <= 5, except (2,j) if j>=3 \*/

do i=1 to 5

do j=1 to 5

if i=2 & j=3 then iterate i /\* or "leave j" would work,

or just "leave" \*/

say "("i","j")"

end

end

**9. Parsing**

The following program examines its arguments:

/\* Parse the arguments \*/

parse arg a.1 a.2 a.3 a.4

do i=1 to 4

say "Argument" i "was:" a.i

end

Execute it as usual, except this time type "alpha beta gamma delta" after

the program name on the command line, for example:

I rexx arguments "alpha beta gamma delta"

O argumnts alpha beta gamma delta

The program should print out:

Argument 1 was: alpha

Argument 2 was: beta

Argument 3 was: gamma

Argument 4 was: delta

The argument "alpha beta gamma delta" has been parsed into four

components. The components were split up at the spaces in the input.

If you experiment with the program you should see that if you do not

type four words as arguments then the last components printed out are

empty, and that if you type more than four words then the last component

contains all the extra data. Also, even if multiple spaces appear between

the words, only the last component contains spaces. This is known as

"tokenisation".

It is not only possible to parse the arguments, but also the input. In

the above program, replace "arg" by "pull". When you run this new program

you will have to type in some input to be tokenised.

Replace "parse arg" with "parse upper arg" in the program. Now, when

you supply input to be tokenised it will be uppercased. The instruction

"parse upper" is a variant of "parse" which always translates the data to

upper case.

"arg" and "pull" are, respectively, abbreviations for the instructions

"parse upper arg" and "parse upper pull". That explains why the "pull"

instruction appeared in previous examples, and why it was that input was

always uppercased if you typed letters in response to it.

Other pieces of data may be parsed as well. "parse source" parses

information about how the program was invoked, and what it is called,

and "parse version" parses information about the interpreter itself.

However, the two most useful uses of the parse instruction are

"parse var [variable]" and "parse value [expression] with". These

allow you to parse arbitrary data supplied by the program.

For example,

/\* Get information about the date and time \*/

d=date()

parse var d day month year

parse value time() with hour ':' min ':' sec

The last line above illustrates a different way to parse data. Instead of

tokenising the result of evaluating time(), we split it up at the

character ':'. Thus, for example, "17:44:11" is split into 17, 44 and 11.

Any search string may be specified in the "template" of a "parse"

instruction. The search string is simply placed in quotation marks,

for example:

parse arg first "beta" second "delta"

This line assigns to variable first anything which appears before

"beta", and to second anything which appears between "beta" and "delta".

If "beta" does not appear in the argument string, then the entire string

is assigned to first, and the empty string is assigned to "second". If

"beta" does appear, but "delta" does not, then everything after "beta"

will be assigned to second.

It is possible to tokenise the pieces of input appearing between search

strings. For example,

parse arg "alpha" first second "delta"

This tokenises everything between "alpha" and "delta" and places the

tokens in first and second.

Placing a dot instead of a variable name during tokenising causes that

token to be thrown away:

parse pull a . c . e

This keeps the first, third and last tokens, but throws away the second

and fourth. It is often a good idea to place a dot after the last

variable name, thus:

parse pull first second third .

Not only does this throw away the unused tokens, but it also ensures

that none of the tokens contains spaces (remember, only the last token

may contain spaces; this is the one we are throwing away).

Finally, it is possible to parse by numeric position instead of by

searching for a string. Numeric positions start at 1, for the first

character, and range upwards for further characters.

parse var a 6 piece1 +3 piece2 +5 piece3

The value of piece1 will be the three characters of a starting at

character 6; piece2 will be the next 5 characters, and piece3 will

be the rest of a.

**10. Interpret**

Suppose you have a variable "inst" which contains the string "a=a+1". You

can execute that string as an instruction, by typing:

interpret inst

The interpret instruction may be used to accept Rexx instructions from

the user, or to assign values to variables whose names are not known in

advance.

/\* Input the name of a variable, and set that variable to 42 \*/

parse pull var

interpret var "=" 42

**11. The stack**

Rexx has a data stack, which is accessed via the "push", "queue" and

"pull" instructions. The "pull" instruction (or in full, "parse pull")

inputs data from the user as we have seen before. However, if there is

some data on the stack then it will pull that instead.

/\* Access the Rexx stack \*/

queue "Hello!"

parse pull a /\* a contains "Hello!" \*/

parse pull b /\* b is input from the user \*/

push "67890"

push "12345"

parse pull c /\* c contains "12345" \*/

/\* there is one item left on the stack \*/

The difference between "push" and "queue" is that when the items are

pulled off the stack, the items which were queued appear in the same

order that they were queued (FIFO, or first in, first out), and the items

which were pushed appear in reverse order (LIFO, or last in, first out).

If the queue contains a mixture of items which were pushed and items which

were queued, then those which were pushed will always appear first.

The stack may be used to communicate data between REXX programs, or

between various subroutines within a REXX program (see the next section).

In certain circumstances, it may also be used to communicate data between

a REXX program and another program not written in REXX.

On some systems, if a program finishes and returns to the operating

system while there are still items on the stack, then the operating

system will read and execute those items as if they had been typed on the

terminal. However, on other systems the leftover items are just thrown

away. Sometimes the items are even saved until the next REXX program is

executed. It is good practice to ensure that the stack is empty before a

program returns control to the operating system (except, of course,

when the program has intentionally stacked a command for the system to

execute or a string for the next program to read).

**12. Subroutines and functions**

The following program defines an internal function, and calls it with

some data. "Internal" just means the function is written in the same

file as the rest of the program.

/\* Define a function \*/

say "The results are:" square(3) square(5) square(9)

exit

square: /\* function to square its argument \*/

parse arg in

return in\*in

The output from this program should be: "The results are: 9 25 81"

When Rexx finds the function call "square(3)", it searches the program

for a label called "square". It finds the label on line 5 - the name

followed by a colon. The interpreter executes the code which starts

at that line, until it finds the instruction "return". While that code

is being executed, the arguments to the function can be determined with

"parse arg" in the same way as the arguments to a program. When the

"return" instruction is reached, the value specified is evaluated and

used as the value of "square(3)".

The "exit" instruction in this program causes it to finish executing

instead of running into the function.

A function which takes multiple arguments may be defined, simply by

separating the arguments with a comma. That is, like this:

/\* Define a function with three arguments \*/

say "The results are:" conditional(5,"Yes","No") conditional(10,"X","Y")

exit

conditional: /\* if the first argument is less than 10 then return the

second, else return the third. \*/

parse arg c,x,y

if c<10 then return x

else return y

A subroutine is similar to a function, except that it need not give a

value after the "return" instruction. It is called with the "call"

instruction.

/\* Define a subroutine to print a string in a box, then call it \*/

call box "This is a sentence in a box"

call box "Is this a question in a box?"

exit

box: /\* Print the argument in a box \*/

parse arg text

say "+--------------------------------+"

say "|"centre(text,32)"|" /\* centre the text in the box \*/

say "+--------------------------------+"

return

It is possible to call a function, even a built-in function, as if it

were a subroutine. The result returned by the function is placed into

the variable called "result".

/\* print the date, using the "call" instruction \*/

call date "N"

say result

If a function or subroutine does not need to use the variables which the

caller is using, or if it uses variables which the caller does not need,

then you can start the function with the "procedure" instruction. This

clears all the existing variables away out of sight, and prepares for a

new set of variables. This new set will be destroyed when the function

finishes executing. The following program calculates the factorial of a

number recursively:

/\* Calculate factorial x, that is, 1\*2\*3\* ... \*x \*/

parse pull x .

say x"!="factorial(x)

exit

factorial: /\* calculate the factorial of the argument \*/

procedure

parse arg p

if p<3 then return p

else return factorial(p-1) \* p

The variable p which holds the argument to funtion factorial is unaffected

during the calculation of factorial(p-1), because it is hidden by the

"procedure" instruction.

If the subroutine or function needs access to just a few variables, then

you can use "procedure expose" followed by the list of variable names to

hide away all except those few variables.

You can write functions and subroutines which are not contained in the

same Rexx program. In order to do this, write the function and save it

into a file whose name will be recognised by the interpreter. This type

of function is called an "external" function, as opposed to an "internal"

function which can be found inside the currently running program.

I If you want to call your function or subroutine using "call foobar" or

I "foobar()", then you should save it in a file named "foobar.rexx" which

I can be found in the current directory or in your path.

O If you want to call your function or subroutine using "call foobar" or

O "foobar()", then you should save it in a file named "foobar.cmd" which

O can be found in the current directory or in your path.

The "procedure" instruction is automatically executed before running

your external function, and so it should not be placed at the start of

the function. It is not possible for an external function to access

any of its caller's variables, except by the passing of parameters.

For returning from external functions, as well as the "return" instruction

there is "exit". The "exit" instruction may be used to return any data to

the caller of the function in the same way as "return", but "exit" can be

used to return to the caller of the external function even when it is used

inside an internal function (which is in turn in the external function).

"exit" may be used to return from an ordinary Rexx program, as we have

seen. In this case, a number may be supplied after "exit", which will

be used as the exit code of the interpreter.

**13. Executing commands**

Rexx can be used as a control language for a variety of command-based

systems. The way that Rexx executes commands in these systems is

as follows. When Rexx encounters a program line which is nether

an instruction nor an assignment, it treats that line as a string

expression which is to be evaluated and passed to the environment.

For example:

/\* Execute a given command upon a set of files \*/

parse arg command

command "file1"

command "file2"

command "file3"

exit

Each of the three similar lines in this program is a string expression

which adds the name of a file (contained in the string constants) to the

name of a command (given as a parameter). The resulting string is passed

to the environment to be executed as a command. When the command has

finished, the variable "rc" is set to the exit code of the command.

The environment to which a command is given varies widely between systems,

but in most systems you can select from a set of possible environments by

using the "address" instruction.

**14. Signal**

Where other programming languages have the "goto" command, Rexx has

"signal". The instruction "signal label" makes the program jump to

the specified label (which is a name followed by a colon, the same as

is used to start an internal subroutine or function). However, once

this has been done, it is not possible to resume any "select" or "do"

control structures that have recently been entered. Thus the main use

of "signal" is to jump to an error handling routine when something goes

wrong, so that the program can clean up and exit.

There is a much more useful way of using "signal", however. That is

to trap certain kinds of error condition. The conditions which may

be trapped include: "syntax" (that is, any syntax error), "error" (any

environment command that results in a non-zero exit code), "halt" (when

the user interrrupts execution) and "novalue" (which is when a symbol is

used without having been given a value).

Error trapping for one of these conditions is turned on by

signal on <condition>

and is turned off by

signal off <condition>

When one of these conditions occurs, the program immediately signals to a

label whose name is the same as that of the condition. Trapping is turned

off, so another "signal on" will be required if you want to continue to

trap that condition.

If you wish for the program to signal to a label whose name is not the

same as that of the condition, then you may specify that label using the

"name" parameter of a "signal on" instruction, like this:

signal on syntax name my\_label

Whenever a "signal" occurs, the variable "sigl" is set to the line number

of the instruction which caused the jump. If the signal was due to an

error trap, then the variable "rc" will be set to an error code.

/\* This program goes on forever until someone stops it. \*/

I say "Press Control-C to halt"

O say "Press Control-C or Control-Break to halt"

signal on halt

do i=1

say i

do 10000

end

end

halt:

say "Ouch!"

say "Died at line" sigl

**15. Tracing**

I Full details of how to use Rexx's tracing facility are contained in

I rexx.ref.

O More details of how to use Rexx's tracing facility can be found in the

O OS/2 help system (help rexx trace).

If a program goes wrong and you need more information in order to work out

why, then Rexx provides you with the ability to trace all or part of your

program to see what is happening.

The most common form of tracing is turned on by

trace r

This causes each instruction to be listed before it is executed. Also, it

displays the results of each calculation after it has been found.

Another useful trace command is:

trace ?a

This makes the interpreter list the next instruction and then stop.

You can continue to the next instruction by pressing return, or you

can type in some Rexx to be interpreted, after which the interpreter

will pause again. You can use this to examine the program's variables

and so forth.

If you want the interpreter to stop only when passing a label, use:

trace ?l

This is useful for setting breakpoints in the program, or for making the

program stop at a function call.