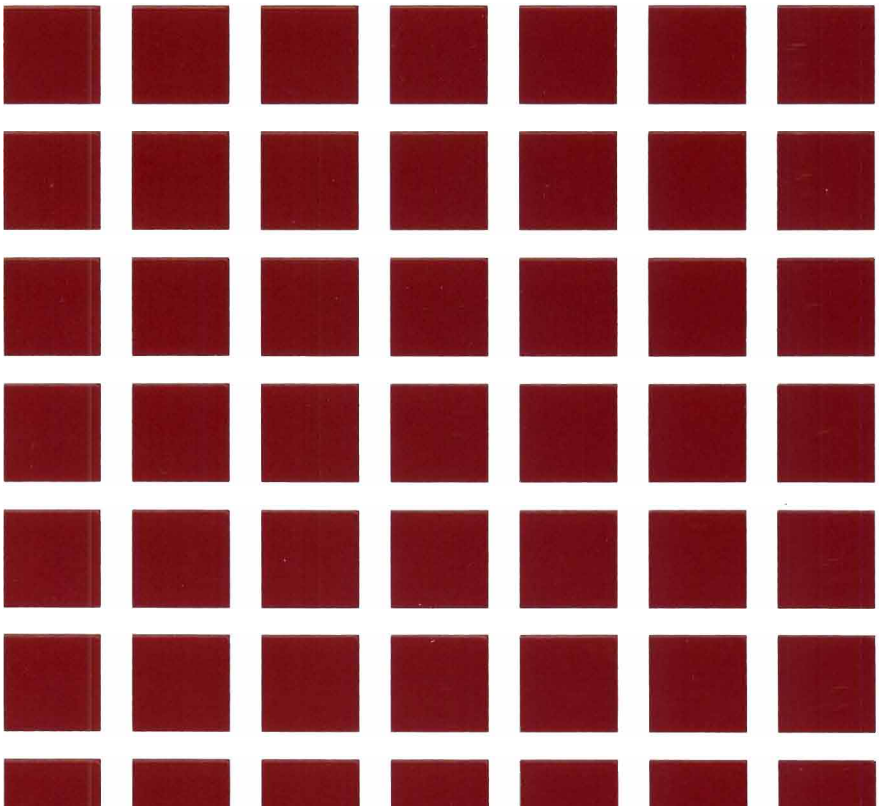
Writing Simple Programs with REXX

VM/Integrated System
Release 5

SC24-5357-00

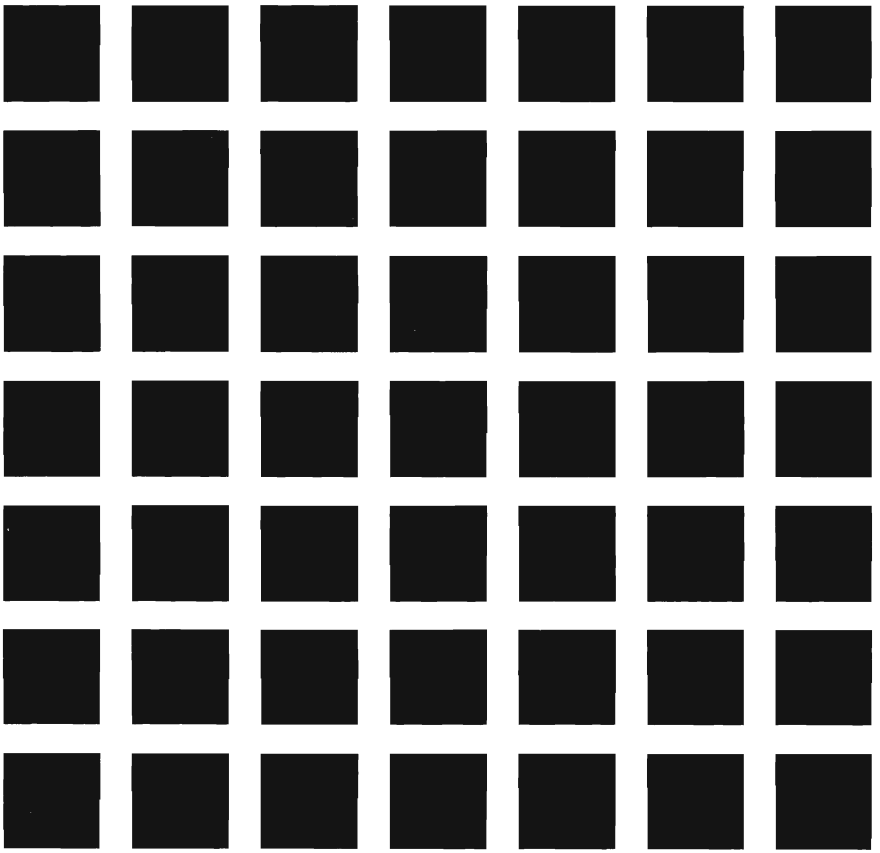


IBM

Writing Simple Programs with REXX

VM/Integrated System
Release 5

SC24-5357-00



IBM

First Edition (April 1987)

This edition, SC24-5357-0, applies to the VM/Integrated System that is based on Release 5 of VM/Integrated System BASE (Program 5664-301) This edition applies to all subsequent releases until otherwise indicated in new editions. Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

In this manual are illustrations in which names are used. These names are fanciful and fictitious, created by the author; they are used solely for illustrative purposes and not for identification of any person or company

Ordering Publications

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Dept. G60, P.O. Box 6, Endicott, NY, U.S.A. 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you

About This Book

When you hear that someone is a computer programmer, do you think of a wizard performing magic through a terminal keyboard? Do you think that only a scientist or a mathematician can unravel the complexities of computers?

Programming is not so mysterious or complicated. Even if you are just learning about computers, you can be a programmer.

This book introduces you to programming through the Restructured Extended Executor language, called REXX for short. Learning REXX doesn't require a special skill. All you need is a terminal connected to the VM/IS system, and a willingness to learn. By learning the essentials of REXX, you can write programs to simplify your daily tasks.

It's easy—that's the magic of REXX.

Who Should Read This Book

You should read this book if you are interested in learning a computer programming language. You do not need any previous programming experience.

What This Book Contains

This book contains three parts and an appendix.

Part 1. Getting to Know REXX includes two chapters. The chapters contain information on what REXX programming is, what you need to get started, and some basic rules and concepts.

Part 2. Writing Your Own Programs includes four chapters. The chapters explain how to write your first three programs, how to correct errors, how to work with arithmetic and true and false, and more. You can test yourself through some exercises.

Part 3. Improving Your Skills includes two chapters. These chapters discuss writing more advanced programs, and where to find additional information on REXX.

The **Appendix** contains the answers to all the exercises throughout the book.

The back cover of this book has a wrap-around flap that you can use as a bookmark. The flap contains a summary of the REXX instructions used in this book.

Conventions Used In This Book

Convention	Meaning
`sample`	Information that you type is shown like this.
`sample`	Information that appears on the screen is shown like this.
sample	Words shown in bold italics appear in the glossary.
sample	Bold type is used for emphasis.

Related Information

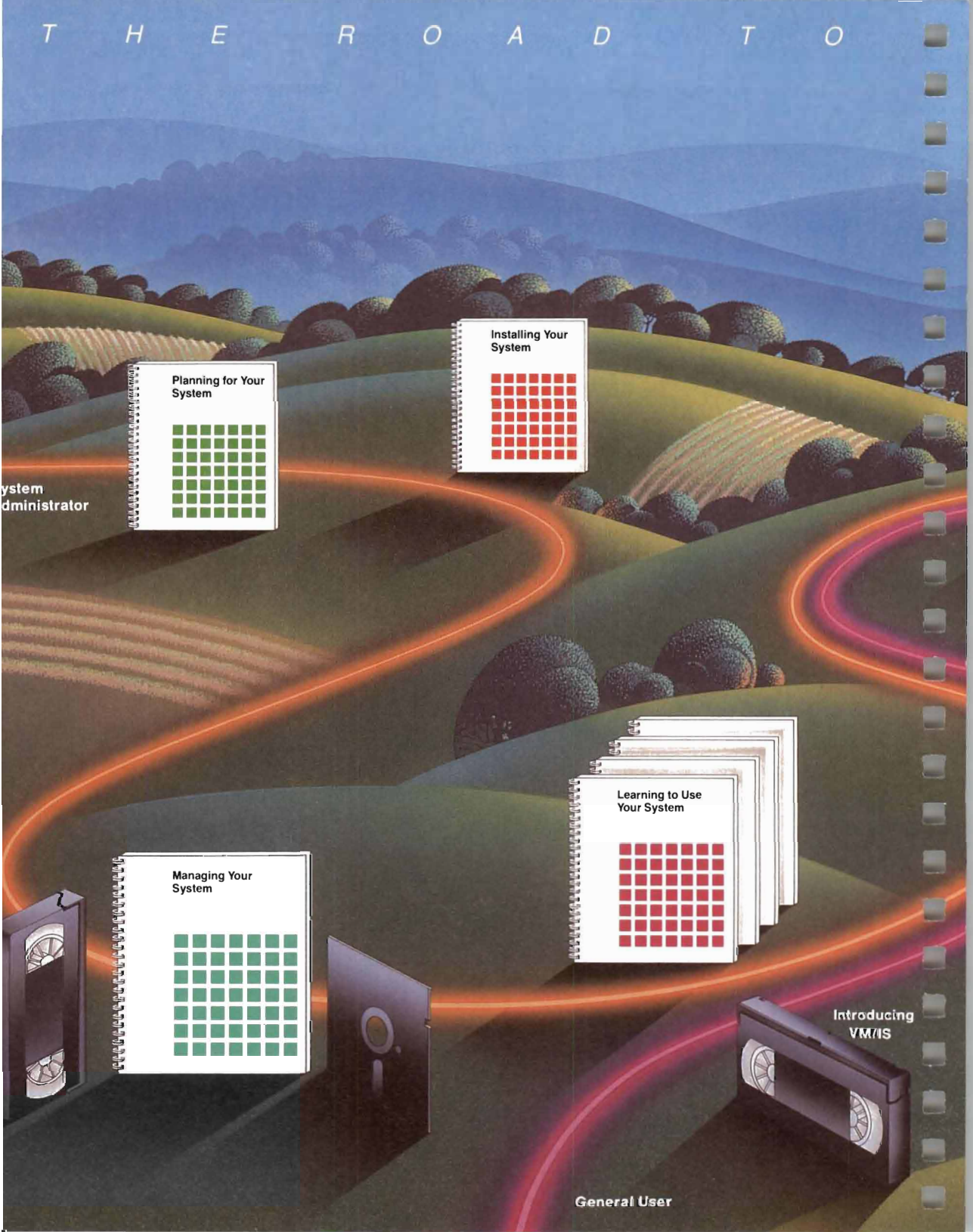
As a user of VM and REXX, you may find the following books helpful:

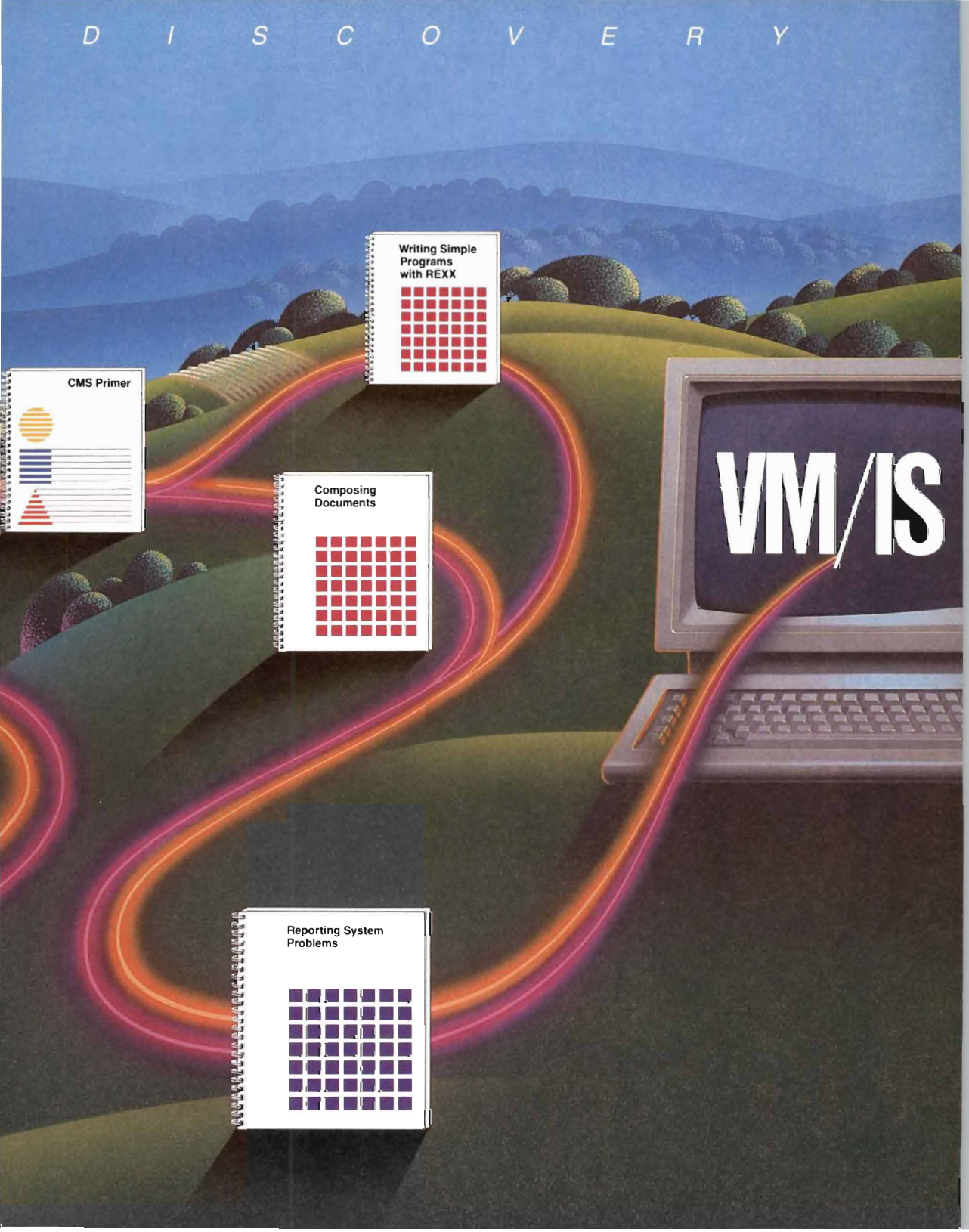
The *VM/IS Learning to Use Your System: Getting Started* book shows you how to use the VM/IS system.

The *Virtual Machine/System Product CMS Primer* shows you how to create and edit files.

The *Virtual Machine/System Product System Product Interpreter User's Guide* explains more about REXX through three reading levels: from beginner to advanced user.

The *Virtual Machine/System Product System Product Interpreter Reference* contains reference material for the experienced programmer, particularly those who have used other programming languages.


T H E R O A D T O
Installing Your System
Planning for Your System
ystem
dministrator
Learning to Use Your System
Managing Your System
Introducing VM/IS
General User


D I S C O V E R Y
Writing Simple Programs with REXX
CMS Primer
Composing Documents
VM/IS
Reporting System Problems

The following information is illustrated in "VM/IS The Road to Discovery":

VM/Integrated System:

Planning for Your System SC24-5337
Installing Your System SC24-5341
Managing Your System SC24-5338
Reporting System Problems SC24-5339
Learning to Use Your System:
Getting Started SC24-5343
Using FORTRAN and Advanced Graphics SC24-5344
Using IBM BASIC and Pascal/VS SC24-5345
Performing Office Tasks SC24-5346
Using APL2 SC24-5347
Using a Data Base SC24-5348
Using Your IBM Personal Computer as a Display Station SC24-5349
Communicating with Other VM Systems SC24-5350
Error and Information Messages SC24-5351
A Day in the Life of an Engineering Firm SC24-5352

VM/System Product CMS Primer ST00-1992
Writing Simple Programs with REXX SC24-5357
Composing Documents with the Generalized Markup Language S544-3421

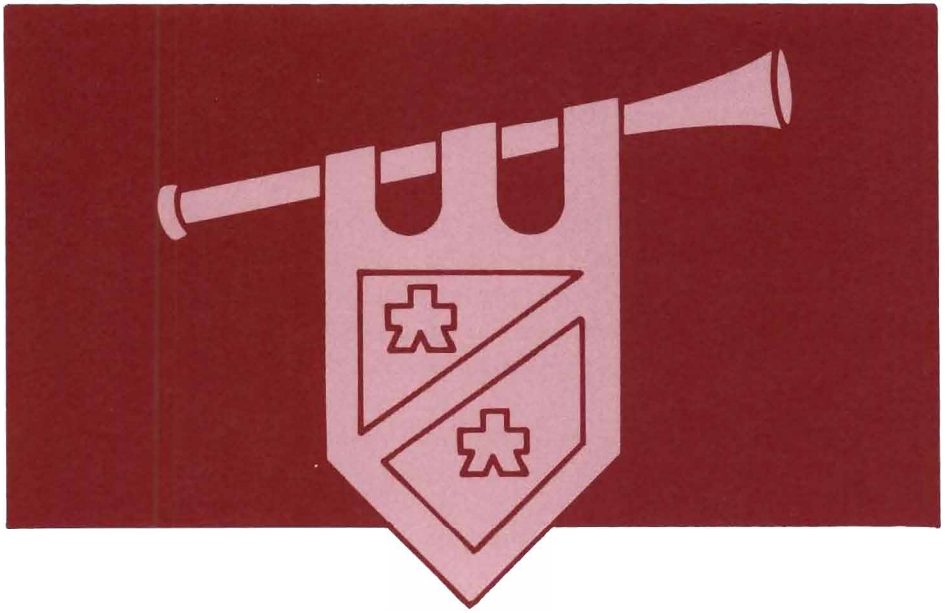
Managing Your System: The Practice Diskette SV21-5273
Managing Your System (videotape 3/4″ U-Matic) SV26-1012
Managing Your System (videotape 1/2″ VHS) SV26-1013
Introducing VM/IS (videotape 3/4″ U-Matic) SV26-1016
Introducing VM/IS (videotape 1/2″ VHS) SV26-1017

Contents

part 1

Getting to Know REXX

Chapter 1. Introduction



Learning a new programming language is like meeting someone for the first time: the first step is to get acquainted. In this chapter, you will be introduced to the Restructured Extended Executor language—REXX. Once you feel comfortable with REXX, you can begin to learn how to use it.

What Can REXX Do for You?

Programs are lists of instructions to a computer. For example, someone has to write a program to tell the computer how to add and subtract numbers to produce your bank balance. Someone has to put codes and prices into the computer at the grocery store so the price appears when the code is given.

Programming has many uses in your everyday life. It can be fun and educational. It can make your work easier.

Several languages are used to write programs. BASIC, a language widely used in home computing, has very few rules. However, writing an intricate program in BASIC usually involves writing a great many lines. Languages like PL/I, APL, and PASCAL have more rules but allow you to do more in fewer lines.

REXX is a programming language that combines the simplicity of a language such as BASIC with the ability to write fewer lines as in a more powerful language. It is easy to learn because it uses familiar words and concepts. REXX allows you to do simple tasks, yet has the ability to handle complex tasks.

A REXX program is processed by the System Product Interpreter. All you do to run a REXX program is type in the program's name.

As you become more experienced with REXX, you can include VM commands to enhance a program's capability.

Getting Started

To begin using REXX, you should have access to a terminal connected to the VM/Integrated System (VM/IS). Once you are set up with a terminal, here's what you need to do:

- Ask your system administrator for a system userid and password if you don't have them. The system administrator also can answer any questions you may have about the computer.

- Familiarize yourself with VM. You need to know what a CMS file is, and how to create a file using an editor. If you need help, or want to brush up on your computer skills, see the *CMS Primer* or the *Learning to Use Your System: Getting Started* book.

- This book assumes you are NOT working with the menus and panels of the VM/IS Productivity Facility. If you are in the PRIMARY MENU or one of the panels, press the PF3 key to leave. You may have to press PF3 more than once. When you reach the CONFIRM END panel, follow the instructions to leave it, too.

- When you feel comfortable working with files, start the next section of this book, "Conversing with Your Computer."

Conversing with Your Computer

Now let's look at a REXX program. First, study the explanation of the sample program to see what the program contains. Later, you can try it out.

The sample program is called the HELLO EXEC. HELLO is the filename, and EXEC is the filetype. REXX programs in this book have a filetype of EXEC. Sometimes people use the word "exec" interchangeably with the word "program." The program looks like this:

```
/* An introduction to REXX */
say "Hello! I am REXX."
say "What's your name?"
pull who
 if who = ""
  then
  say "Hello stranger"
 else
  say "Hello" who
exit
```

Figure 1. HELLO EXEC, a sample REXX program

A brief description of each part of the sample program follows:

/* An introduction to REXX */

This **comment** explains what the program is about. A comment starts with a /* and ends with a */. All REXX programs must start with a comment.

say "Hello! I am REXX."
say "What's your name?"

These **instructions** display the words between the quotes on the screen.

pull who

The PULL **instruction** reads the response entered from the keyboard and puts it into the computer's memory. `Who` is the **name** of the place in memory where the user's response is put. Any name can be used with the PULL instruction.

if who = ""

The IF **instruction** tests a condition. The **test** in this example determines if `who` is empty. It is empty if the user types in a blank space and presses the ENTER key, or just presses the ENTER key.

then

This **direction** executes the instruction that follows, if the tested condition is true.

say "Hello stranger"

This **instruction** displays `Hello stranger` on the screen.

else

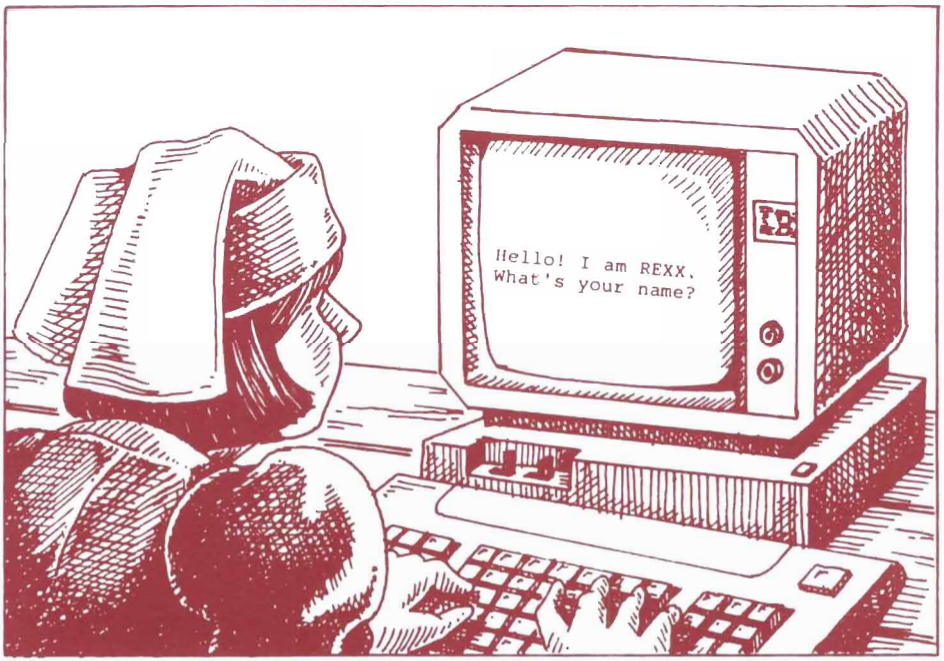
If the tested condition is not true, this **direction** executes the instruction that follows.

say "Hello" who

This **instruction** displays `Hello` on the screen, followed by whatever is in `who`.

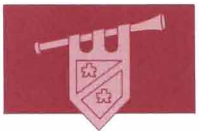
exit

This **instruction** causes the program to stop at this point.



Here's what happened when Mike tried the HELLO program:

```
Ready;
hello
Hello! I am REXX.
What's your name?
Mike
Hello MIKE
Ready;
```



If Mike did not enter his name, but entered a blank space, this is what would have happened:

```
Ready;
hello
Hello! I am REXX.
What's your name?

Hello stranger
Ready;
```

Running Your First Program

Not all REXX programs are this small, of course, but many are just as easy. To try this program, first type on the command line:

```
xedit hello exec
```

and press the ENTER key.

If you already have a file with the filename of HELLO and the filetype of EXEC, just use another name for the file (for example, HELLO1 EXEC). Keep this in mind whenever you write a program or try one of the sample programs.

This creates a file using the System Product Editor (XEDIT). You can also use an "x" for the XEDIT command. For example:

```
x hello exec
```

Next, type in the REXX program exactly as it appeared in Figure 1 on page 6 and file it by typing on the terminal:

```
file
```

To run the program, enter the name of your program, or use

```
hello
```

When the program requests it, you can enter your name, or you can press the ENTER key to see the other response.

What's Next?

If you tried the HELLO EXEC and it ran without any errors, you now have run your first program. Congratulations! If it didn't work, compare your program with the sample program and see if you typed something incorrectly.

The next chapter includes some rules for creating your own programs. To learn the concepts quickly, follow as we write some programs together. To reinforce what you have learned, try the exercises throughout the book. Don't worry about making mistakes because you will be guided through the steps.

Chapter 2. Learning a Few Rules

When you are trying a new sport or card game, it is usually easier to understand the plays if someone explains the rules. Learning is a step-by-step process. If you know the rules in the beginning, you can follow when the activity or action gets more complicated. We won't show you any complicated moves here, just the easiest way to write programs that will execute correctly.

The System Product Interpreter works on your REXX program, line by line, word by word, doing what you have written. This chapter includes a few rules to get you started.

Comments

From the first chapter, you may remember that all REXX programs must begin with a comment.

A ***Comment*** is a group of words that tell what the program is for, what kind of input it can handle, and what kind of output it produces. Comments help you understand the program when you read it over later, perhaps to improve it.

With VM/IS, there are three languages for writing EXECs: REXX, EXEC 2, and CMS EXEC. The system distinguishes a REXX program from the other types because it contains a REXX comment on the first line.

The symbols used for comments are:

/* to mark the start of a comment

***/** to mark the end of a comment.

When the interpreter finds a /*, it stops interpreting; when it encounters a */, it begins interpreting again with the information following the symbol. The comment can be a few words or several lines, as in the following examples:

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

or,

```
say "Hello!" /* This comment is on the same line as the instruction */
```

or,

```
/* Comments can
also occupy
more than one line. */
```

The /* */ is sufficient to start a program, but it is better to put a brief description of the program in the space.

Strings

A ***string*** is any group of characters inside single quotes or double quotes. Single and double quotes are interchangeable but the beginning and the ending must match. The interpreter stops interpreting when it sees a quote and looks for the matching quote. The characters inside the quotes remain as they are typed, with uppercase and lowercase letters. For example:

```
'number'
"Live and let live."
```

are both strings.

If you want to use a quotation mark or an apostrophe within a string, you should use different quotation marks around the whole string. For example:

```
"Don't cross the bridge until you come to it."
```

You can also use a pair of quotes (the same as those used to mark the string):

```
say "Look out! ""He's"" here."
```

This is interpreted by REXX as:

```
Look out! "He's" here.
```

What's in a REXX Program

In addition to comments, a REXX program can contain the following items:

Instructions
Assignments
Labels
Commands.

It's best to use one line for each item. If you want an item (for example, an instruction) to span more than one line, you must put a comma at the end of the line to indicate that the instruction continues on the next line. If you want to put more than one item on a line, you must use a semicolon to separate the items.

The following program contains seven instructions. Can you find them?

```
/*Find the items and cheer! */
say "Everybody cheer!"
say "2"; say "4"; say "6"; say "8";
say "Who do we",
"appreciate?"
exit
```

Figure 2. RAH EXEC, identifying items in a program

Did you remember that `exit` is an instruction?

As you read this book, you will encounter instructions, assignments, labels, and commands in more detail. In this chapter, they are explained briefly so you can see how they contribute to a REXX program.

Instructions

An ***instruction*** tells the computer to do something:

```
say "Time is of the essence"
```

The computer displays `Time is of the essence` on your screen. Instructions can contain one or more assignments, labels or commands and usually start on a new line. Some instructions that you will use often in your programs follow.

The SAY Instruction

The example above uses the SAY instruction. The format is:

```
say expression
```

The expression can be something you want displayed on the screen or something to be computed, such as an equation:

```
say 4 + 3 "= seven"
```

This will display `7 = seven` on the screen. With the SAY instruction, anything not in quotes gets changed to uppercase. If you want something to appear "as is," use quotes.

The PULL and PARSE PULL Instructions

In a program, the usual sequence of instructions is to use SAY to ask a question and use PULL to receive the answer. The response typed in by the user is put in the computer's memory. The following program would not work correctly if the PULL instruction came before you requested information with the SAY instruction.

What do you think happens when the following program is run?

```
/*Using the PULL Instruction */
say "Enter your name"
pull name          /* Puts response from user into memory */
say "Hello" name
exit
```

Figure 3. NAME EXEC puts a name in memory

Let's examine how this program works. You type `name` on the terminal, and `Enter your name` appears on the screen. You enter your name, and the program responds with a `Hello` to you.

If you were curious and tried the NAME program, you probably noticed that your name was CHANGED TO UPPERCASE. If you want to keep the characters as you typed them, you can use the PARSE PULL instruction.

Here's an example using PARSE PULL.

```
/* Using the PARSE PULL Instruction */
say "Hello! Are you still there?"
say "I forgot your name. What is it?"
parse pull name
say name", are you going to the movies?"
pull answer
if answer = "YES"
 then
 say "Good. See you there!"
exit
```

Figure 4. CHITCHAT EXEC, how PULL and PARSE PULL work

The PARSE PULL instruction reads everything from the keyboard "as is" in uppercase or lowercase. In this program, the name is repeated just as the user (you) typed it. However, "answer" is changed to uppercase characters because the PULL instruction was used. This ensures that if "yes," "Yes," or "YES" is typed, the same action is taken.

In "Parsing Words" on page 85, you can find more information about what "PARSE" does.

The EXIT Instruction

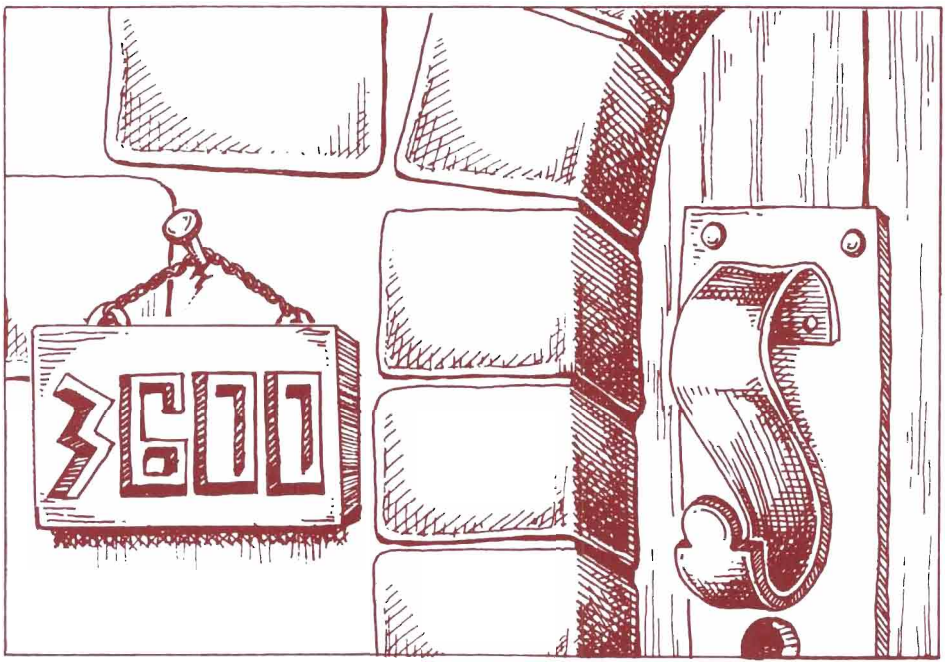
By putting an EXIT in a program, you tell the program to end. The EXIT instruction should be used in a program that contains subroutines (subroutines are explained in "Using Subroutines" on page 102), or if an error occurs. Although the EXIT instruction is optional in some programs, it is good programming practice to use it at the end of every program.

Assignments

An ***assignment*** says that the string should be put in a special place in the computer's memory. In the example:

```
Home = '3600 Castle Way'
```

the string `3600 Castle Way` is put in `Home`. Because `Home` can have different values (be reassigned to mean different things) in different parts of the program, it is called a ***variable***. Variables will be discussed in "Using Variables" on page 38.



Labels

Any word followed by a colon (with no space between the word and the colon) and not in quotes, is treated as a ***label.*** For example:

```
MYNAME:
```

A label marks the start of the subroutine. The following example shows one use of a label (called error) within a program:

```
   .
   .
   .
if problem = 'yes' then call error
   .
   .
   .
error:
   say 'Problem in your data'
   exit
```

Figure 5. This part of a program shows how labels identify subroutines

For more information on labels, see "The CALL Instruction" on page 104.

Commands

A ***command*** is a word, phrase, or abbreviation that tells the system to do something. In REXX, anything that is not a REXX instruction, assignment, or label is considered a command. For example, you can use the CMS commands of COPYFILE, QUERY, PRINT, or TYPE in your programs.

The CMS command `type` may appear in a REXX program like this:

```
/* Issuing commands in REXX */
type hello exec
exit
```

Figure 6. COMMAND EXEC, using commands in programs

Although a variety of commands can be used in REXX, only CMS commands will be discussed in this book. In "Issuing Commands from an EXEC" on page 107, you can find more information on issuing commands from programs.

Writing in Mixed Case

You can use mixed case in REXX programs. Mixed case letters helps you distinguish instructions from variables, and helps you to find errors quickly.

Here's what the HELLO program looks like using mixed case:

```
/* An introduction to REXX */
SAY "Hello! I am REXX."
SAY "What's your name?"
PULL who
 IF who = ""
  THEN
  SAY "Hello stranger"
 ELSE
  SAY "Hello" who
EXIT
```

Figure 7. New HELLO EXEC, using mixed case

Using Quotes for Spacing

If you put several spaces between words in your program, the interpreter keeps only one space between words. If you want more space, you should use quotes, as in this program:

```
/* Example of cases and spaces */
say Brevity   is   the   soul of wit. /*One space between words*/
say "Brevity    is    the    soul"
say of"           "wit.
exit
```

Figure 8. QUOTES EXEC, putting spaces between words

When run, the program looks like this on the screen:

```
Ready;
quotes
BREVITY IS THE SOUL OF WIT.
Brevity    is    the    soul
OF           WIT.
Ready;
```

Adding Blank Lines

To leave a blank line between lines displayed in the output, you can use the SAY instruction with nothing following it. For example:

```
/* Examples of cases and spaces */
say Brevity is the soul of wit.
say                                    /*This displays a blank line.*/
say "Brevity    is    the    soul"
say of"           "wit
exit
```

Figure 9. QUOTES EXEC, adding blank lines

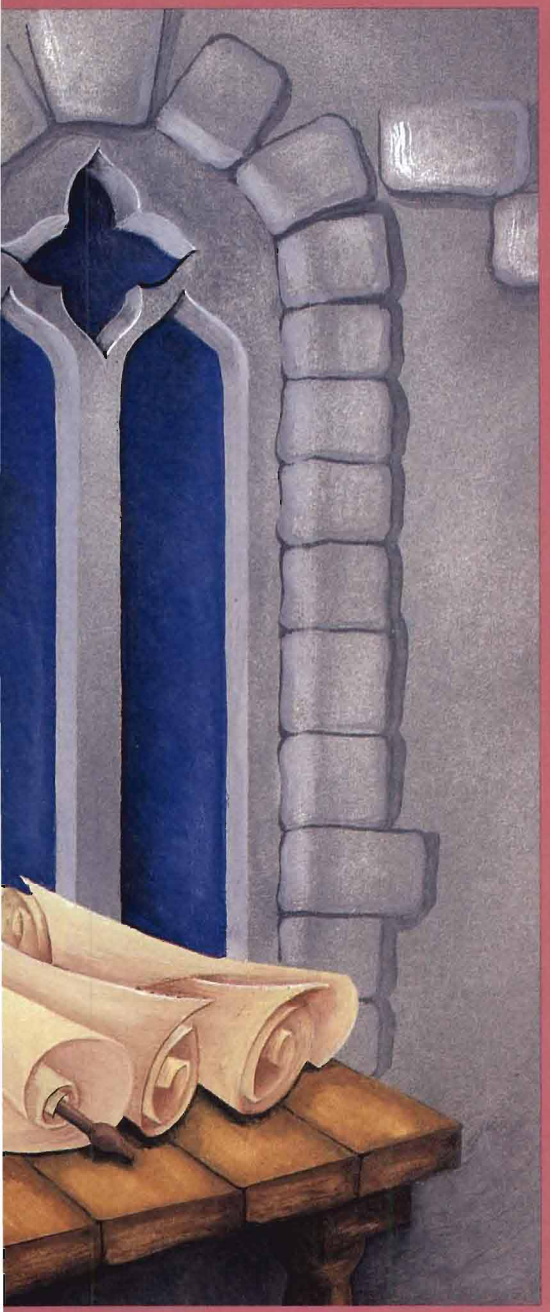
It looks like this on the screen:

```
Ready;
quotes
BREVITY IS THE SOUL OF WIT.

Brevity    is    the    soul
OF           WIT.
Ready;
```

Summary

You learned the following in this chapter:

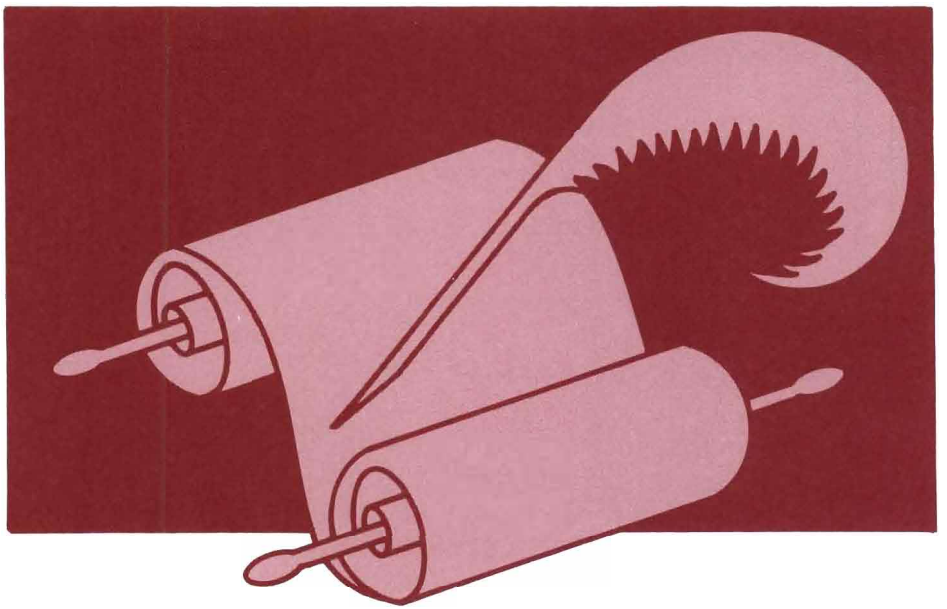
Term/Concept	Description	Page
Comment	Explains what the program will do. All REXX programs start with a comment. Identified by /* ... */.	12
String	Groups of characters within matching single or double quotes.	13
Instruction	Tells the interpreter to do something.	15
SAY	Tells the interpreter to display words, or something computed, on the screen.	15
PULL	Puts an answer in memory. Response read from keyboard is put in uppercase.	16
PARSE PULL	Puts an answer in memory. Reads everything from the keyboard "as is" in uppercase or lowercase so the response appears as you typed the input.	16
EXIT	Tells the interpreter to leave the program.	17
Assignment	Gives a value to a variable.	18
Label	A name followed by a colon. Used in subroutines.	19
Command	Anything not identified as a REXX instruction, assignment or label.	20
Mixed Case	Not changed to uppercase. Use quotes around strings to keep upper- and lowercase letters and words.	20
Quotes for Blanks	To maintain blank spaces in what's displayed when the program is run.	21
Adding Blank Lines	Use the SAY instruction with nothing following it.	22



part 2

Writing Your Own Programs

Chapter 3. Writing Your First Program



Do you remember the first time you drove a car? "Firsts" often make us feel uncertain. But writing your first program can be easier than you think.

Putting the Pieces Together

Let's try writing a simple program to give you a list of things to do today, and call it the NOTEPAD EXEC. You will receive hints along the way to help you write the program.

The basic steps to writing a program are:

1. Identify the problem to solve and translate it into a step-by-step procedure

2. Write the REXX instructions to solve each step

3. Run the program to see if the results meet the requirements

4. Revise the program to correct errors.

The steps outlined above will help you understand the process in writing a program. As a first hint, you can use two files for the NOTEPAD program: one for the EXEC and one to contain your list. Using a separate file for the list allows you to update the "things to do" every day without changing the EXEC.

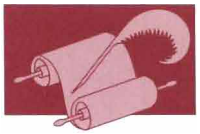
Writing the EXEC

To help you get started, here are the requirements for the NOTEPAD EXEC. The items are in the proper sequence for the program.

The NOTEPAD EXEC should do the following:

1. Identify and describe the REXX program

2. Print a heading for your list of things to do



3. Print on the screen the file where you maintain the list

   Another hint: use the CMS command TYPE for this step. The TYPE command has this format: TYPE fn ft. For the filename (fn), name the file for your list anything you want, or use NOTEPAD. For filetype (ft), use LIST.

4. Tell the interpreter to leave the program.

Now, read each item and see if you can write an instruction or command for it.

If you feel comfortable with the four lines you have written, type the EXEC and file it. Continue with the section, "Creating the File for Your List."

If you had trouble writing the program, continue with the following discussion of how to put it together.

1. First, which of the following identifies a REXX program? Select one and write a statement for it.

   Choices: Assignment, String, Comment, Instruction

2. Which instruction should you use to print a heading or message on the screen? Select one and write a statement for it.

   Choices: PULL, SAY, EXIT, PARSE PULL

3. Next, how can you use the TYPE command to print the file with your list? Select a sequence. Insert your filename and the filetype "list" in the appropriate place.

Choices:

say type fn ft
pull fn ft type
parse pull type ft fn
type fn ft

4. For the last step, which of the following instructions tells the interpreter to leave a program? Select one and write a statement for it.

   Choices: EXIT, END, LEAVE, STOP

Now you can create a file for the EXEC. Enter:

```
xedit notepad exec
```

In input mode, type the program and file it. Congratulations! You have written your first REXX program.

Creating the File for Your List

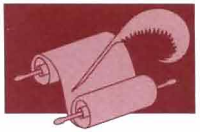
Now you need to create a file with things to do. The file for your list can contain anything you want. If you are not sure what to write, use the following:

```
1. Answer mail.
2. Call George about trip.
3. Order book.
```

Create the file for your list. Use the name you have used in your NOTEPAD EXEC for the name of the file.

```
xedit notepad list
```

Enter your list or our sample list, and file it. Now you are ready to try running your first program.



Running the Program

To run the NOTEPAD EXEC, enter:

```
notepad
```

Did you see a list of "things to do" on your screen? If not, take a look at our sample program and compare it to the one you wrote. Remember, there are many ways to write the same program. Your program may be correct, too!

This is the program:

```
/* A reminder of things to do. */
say 'Things to do today:'
type notepad list
exit
```

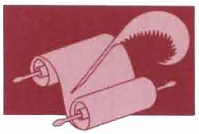
Figure 10. NOTEPAD EXEC, listing things to do

If you ran our program, you would see this displayed on your terminal:

```
Ready;
notepad
Things to do today:

1. Answer mail.
2. Call George about trip.
3. Order book.

Ready;
```

Fixing Errors

If you had trouble running your program, you may have typed something wrong which caused the interpreter to stop running the program. Most errors are simply syntax errors. (The ***syntax*** of a language is the way in which words are put together to form phrases, instructions, or sentences. The rules discussed in this book are part of the syntax for interpreting REXX.)



Suppose you typed as the NOTEPAD EXEC:

```
/* A reminder of things to do. */
say 'Things to do today:'
type notepad list               /*Types out your list
exit
```

Figure 11. NOTERR EXEC, finding the error

The program has a syntax error: the missing */ to end the second comment. When you run the program, this appears on the screen:

```
Ready;
noterr
Things to do today:
     3 +++ type notepad list              /* Types out your listexit
Error 6 running NOTERR EXEC, line 3: Unmatched "/*" or quote
R(20006);
```

This screen shows that the error occurred on line 3 of the program. The line that caused the error is printed on your screen:

```
3 +++ type notepad list           /*Types out your listexit
```

The next line contains an error message and number:

```
Error 6 running NOTERR EXEC, line 3: Unmatched "/*" or quote
```

You can find all REXX error numbers and messages in the *VM/SP System Product Interpreter Reference* if you need more information about the error. To correct the program, add the end */ to the comment. The line now looks like this:

```
type notepad list           /* Types out your list */
```

A syntax error also occurs if you leave off a quote in an instruction. This program would cause a similar error:

```
/* A program with a syntax error. */
say 'This program does absolutely nothing
exit
```

Figure 12. ERROR EXEC, missing an end quote

To correct the SAY instruction, add the end quote or remove the first quote.

Exercises

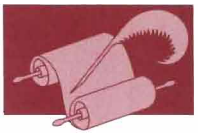
You can find the answers to these exercises in the Appendix.

1. Read the following program, MADAM EXEC, carefully. On a piece of paper, write down what each word is (for example, an instruction), and what the interpreter will do with it.

   MADAM EXEC:

```
/* A polite enquiry  */
Jane = "Mrs. Doe"
say "How" is jane ?
exit
```

   Now, use XEDIT to create a file called MADAM EXEC and try out the program. Did everything happen as you expected it? If not, read Chapter 2 again and study the explanation in the Appendix.



2. The next program, TROUBLE EXEC, has an error in it. Type the program in and run it. Note the error number and write down what you think caused the error.

   TROUBLE EXEC:

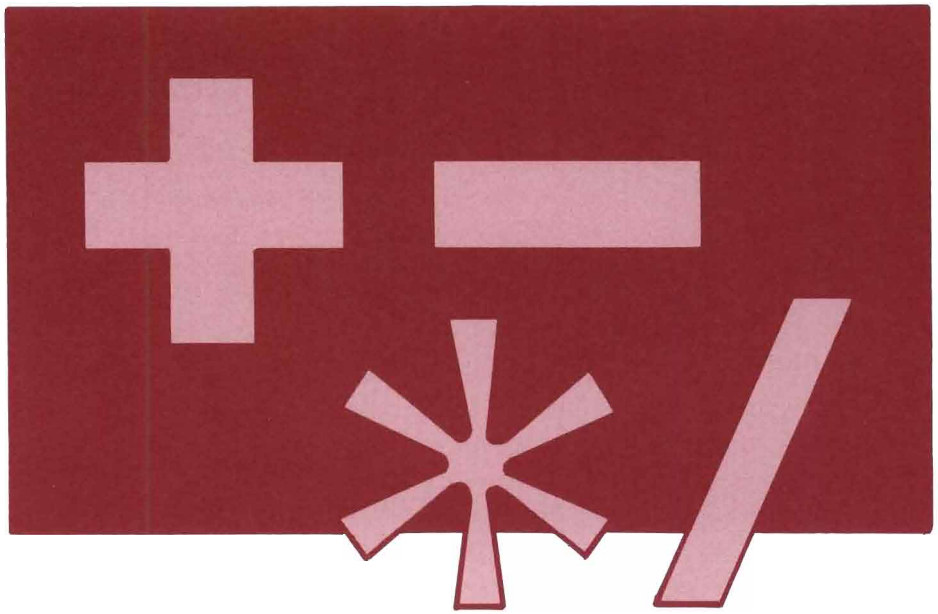
```
/* Finding errors */
say Look closely, and find the error here
exit
```

Summary

You learned the following in this chapter:

Term/Concept	Description/Solution	Page
Writing Programs	Follow the basic steps: Identify the problem; write the instructions; run the program; revise the program.	28
Running the Program	Type the program name and press the ENTER key.	31
Fixing Errors	Correct the syntax of the program.	32

Chapter 4. Working with Variables and Arithmetic



By now we hope you are feeling comfortable with REXX programming. In this chapter you will write another program that you can use repeatedly. Before you write the program, you will learn how to use variables and arithmetic. You also will learn how to add comments throughout a program to describe how it works.

Using Variables

A ***variable*** is a particular piece of data, used by a program in a particular way, but whose value may vary. Within a program, each variable is known by its own unique name and is always referred to by that name.

Choosing Names for Variables

When you choose a name for a variable, you must follow these rules:

- The first character must be one of:

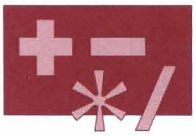
  A B C...Z @ # $ ¢ ! ? _

  Lowercase letters such as a b c...z are also allowed. The interpreter changes them to uppercase.

- The rest of the characters can be any of the above, plus these:

  0-9

  A period can also be used as a special kind of variable. You will not learn how to use a period for variables in this book.



Assigning Values

A variable's **value** may vary, but not its name. When you name a variable and give it a value, it is an ***assignment*** . For example, any statement of the form:

```
symbol = expression
```

is an assignment statement. You are telling the interpreter to compute what the `expression` is and put the result into a variable called `symbol`. It is the same as saying:

"Let `symbol` be made equal to the result of `expression`."

The idea of assigning a value to a variable is the same as the difference between a post office box and the contents of the box. The box number does not change, but the contents of the box may be changed at any time. Another example of an assignment is:

```
num1 = 10
```

One way to give the variable `num1` a new value is by adding to the old value in the assignment:

```
num1 = num1 + 3
```

The PULL instruction can also be used to assign a variable. Do you remember the YOURNAME EXEC? The `pull name` says to give `name` whatever value the user types.

A special concept in REXX is that any variable which has not received a value will have the uppercase version of the variable as its initial value. For example, if you wrote in a program,

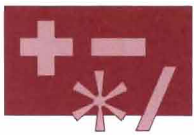
```
list = 2 20 40
say list
```

you would see this on the screen:

```
2 20 40
```

As you can see, `list` receives the values it is assigned. But if you wrote:

```
say list
```



you would see this:

```
LIST
```

Let's look at some ways of assigning values to variables. Here is a simple program:

```
/* Assigning a value to a variable */
a = 'abc'
say a
b = 'def'
say a b
exit
```

Figure 13. VARIABLE EXEC assigns values

When you run the program, it looks like this:

```
Ready;
variable
abc
abc def
Ready;
```

Assigning values is easy, but you have to make sure a variable is not used unintentionally, as in this example:

```
/* Unintentional interpretation of a variable */
a = 'abc'
say Today would be much nicer if it were a Friday
exit
```

Figure 14. VAROOPS EXEC, assigning a variable unintentionally

What do you think you will see when the program is run? It looks like this:

```
Ready;
varoops
TODAY WOULD BE MUCH NICER IF IT WERE abc FRIDAY
Ready;
```

To avoid unintentionally substituting a variable for the word, all you have to do is put the sentence in quotes.

```
/* No interpretation of a variable */
a = 'abc'
say 'Today would be much nicer if it were a Friday'
exit
```

Figure 15. New VAROOPS EXEC, assigning a variable correctly

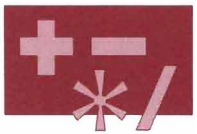
Working with Arithmetic

Your REXX programs may need to include arithmetic operations of addition, subtraction, multiplication, and division.

For example, you may want to assign a numeric value to two variables and add the variables together.

Arithmetic operations are performed the usual way. You can use whole numbers and decimal fractions. A **whole number** is a number that has a zero (or no) decimal part (for example, 1 or 29). A **decimal fraction** contains a decimal point (for example, 1.5 or 0.5).

Before you see how these four operations are handled in a program, we will explain what the operations look like and the symbols used.



These are just a few of the arithmetic operations used in REXX. The examples contain a blank space between numbers and operators so you can see the equations better, but the blank is optional.

Addition

The symbol to add numbers is the plus sign (+). An instruction to add two numbers is:

```
say 4 + 2
```

The answer you see on the screen is: 6.

Subtraction

The symbol to subtract numbers is the minus sign (-). An instruction to subtract two numbers is:

```
say 8 - 3
```

The answer on the screen is: 5.

Multiplication

The symbol to multiply numbers is the asterisk (*). An instruction to multiply two numbers is:

```
say 2 * 2
```

The answer on the screen is: 4.

Division

With division, there are several operators you can use, depending on whether or not you want the answer expressed as a whole number. For example:

- To divide, the symbol is one slash (/). An instruction is:

  ```
  say 7 / 2
  ```

  The answer on the screen is 3.5.

- To divide, and return just a remainder, the symbol is two slashes (//). An instruction is:

  ```
  say 7 // 2
  ```

  The answer on the screen is 1.

- To divide, and return only the whole number portion of an answer and no remainder, the symbol is the percent sign (%). An instruction is:

  ```
  say 7 % 2
  ```

  The answer on the screen is: 3.

This sample program shows you how to perform four arithmetic operations on variables:

```
/* Performing arithmetic on variables */
a = 4
b = 2
c = a + b
say 'The result of' a '+' b 'is' c
say
c = a * b
say 'The result of' a '*' b 'is' c
say
c = a - b
say 'The result of' a '-' b 'is' c
say
c = a / b
say 'The result of' a '/' b 'is' c
exit
```

Figure 16. MATH EXEC, using arithmetic

On the screen, you see this:

```
Ready;
math
The result of 4 + 2 is 6

The result of 4 * 2 is 8

The result of 4 - 2 is 2

The result of 4 / 2 is 2
Ready;
```

Operators

The symbols used for arithmetic (+, -, *, /) are also called ***operators*** because they "operate" on the adjacent ***terms***. In the following example, the operators act on the numbers (terms) 4 and 2:

```
say 4 + 2          /* says '6'          */
say 4 * 2          /* says '8'          */
say 4 / 2          /* says '2'          */
```

When you are attempting to do arithmetic from data you enter from the keyboard (in response to a prompt to enter numbers, for example), you should check that the data is valid. You can do this using the DATATYPE() function. This function and how to use other built-in functions is explained in "Using Functions" on page 98.

Evaluating Expressions

In REXX, expressions are normally evaluated left to right. An equation helps to illustrate this point. Until now, you have seen equations with only one operator and two terms, such as: 4 + 2. Suppose you had this equation:

```
9 - 5 + 4
```

The 9 - 5 would be computed first. The answer, 4, would be added to 4 for a final value: 8. However, some operations are given priority over others. In general, the rules of algebra apply to equations. In this equation, the division is handled before the addition:

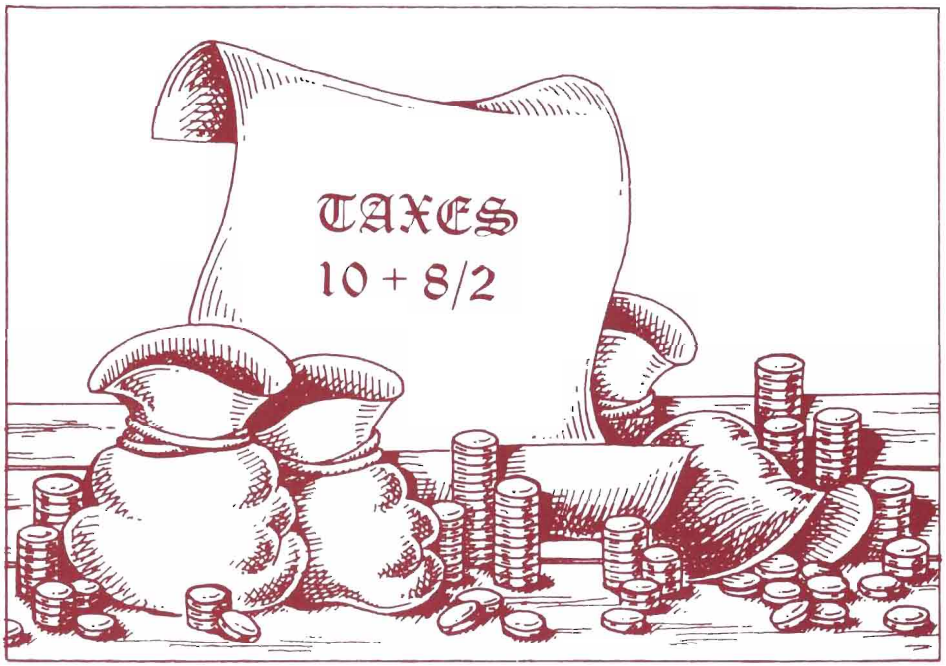
```
10 + 8 / 2
```

the value is: 14.

If you use parentheses in an equation, the interpreter evaluates what's in the parentheses first. For example:

```
(10 + 8) / 2
```

The value is: 9.



Creating a Program

For your second program, let's put some of the rules you have learned into a program that you might use every day. You have just learned how REXX handles arithmetic. How about writing a program to add two numbers? Let's call it the ADD EXEC.

Keeping in mind the basic steps to writing a program, here is a list of what you need to do in this program:

1. Identify and describe the REXX program
2. Tell the user to enter numbers
3. Read the numbers entered from the keyboard and put the numbers into the computer's memory
4. Add the two numbers and display the answer on the screen
5. Tell the interpreter to leave the program.

Writing the Program

As you know, there are many ways to write programs to accomplish the same task. See if you can write instructions for each step. To make it easier, ask the user for each number separately, then add the numbers together.

If you have written some instructions and want to try them, create a file for the ADD EXEC. Type your lines in and file it. Now run the program. Did it work?

If your program worked, skip to the section on "Running the Program" to see how the sample program. If you had trouble writing

the program, continue with the following discussion of how to put the program together.

1. First, what identifies a REXX program? If you thought of a comment, you were right. Take a minute to write a brief comment now.

2. Next, you need to ask the user to enter numbers. What instruction displays a message on the screen? Did you think of the SAY instruction? Write an instruction to ask for the `first number`.

3. If the number is entered, it needs to be put in the computer's memory. Do you know the instruction that collects an answer and puts it in memory? If you guessed the PULL instruction, you are correct. Write your next instruction.

4. Now, write an instruction to ask for a `second number`.

5. Write another instruction to put the second number in memory.

6. The next instruction is similar to one in the MATH program. In one statement, you can tell the interpreter to add the two values kept in memory, and display the sum on the screen. Hint: this is one instruction. It contains a string and the addition operation.

7. Finally, write the instruction to finish the program.

Now you can type the program and file it. Congratulations! You have written another REXX program.

Running the Program

To test the ADD EXEC, enter `add` and try some numbers. Were the numbers added together and the sum displayed on the screen? If not, try to correct your program by looking at the error messages.

Take a look at the following sample program to see the ADD EXEC:

```
/* This program adds two numbers */
say 'Enter the first number.'
pull num1
say 'Enter the second number.'
pull num2
say 'The sum of the two numbers is' num1 + num2
exit
```

Figure 17. ADD EXEC, adds two numbers and displays the sum

Here's what happened when Mike tried it:

```
Ready;
add
Enter the first number.
3
Enter the second number.
12
The sum of the two numbers is 15
Ready;
```

Using Comments

When you write programs, keep in mind that other people may want to modify them. It's a good idea to add comments to the instructions so that anyone can understand each step. If you don't use a program for a while, you might be glad to have the reminder too. The ADD EXEC could have these comments:

```
/* This program adds two numbers */
say 'Enter the first number.'        /* Enter a number           */
pull num1                            /* Store number             */
say 'Enter the second number.'       /* Enter another number     */
pull num2                            /* Store second number      */
                                     /* Add numbers and display */
say 'The sum of the two numbers is' num1 + num2
exit
```

Figure 18. Comments in the ADD EXEC

Longer programs may require a block of comments to explain a group of instructions. For example:

```
/*                                                              */
/* Subroutine starts here to repeat shout three times.          */
/* The first argument is displayed on the screen three          */
/* times, with punctuation.                                     */
/*                                                              */
```

In general, if you explain your program well, everyone will understand it.

Exercises

You can find the answer to these exercises in the Appendix.

1. Which of the following could be used as the name of a REXX variable?

   a. Scroll

   b. KP

   c. old_world

   d. 2B

   e. #1

2. The following program, ASSIGN EXEC, assigns a value to a variable. However, something is missing in the program. Write down what you think the error is and how you would correct it.

   ASSIGN EXEC:

```
/* This program has something missing! */
say input
exit
```

   Now, type the program in with your correction. Does it work?

3. What will this program display on the screen?

   FAMILY EXEC

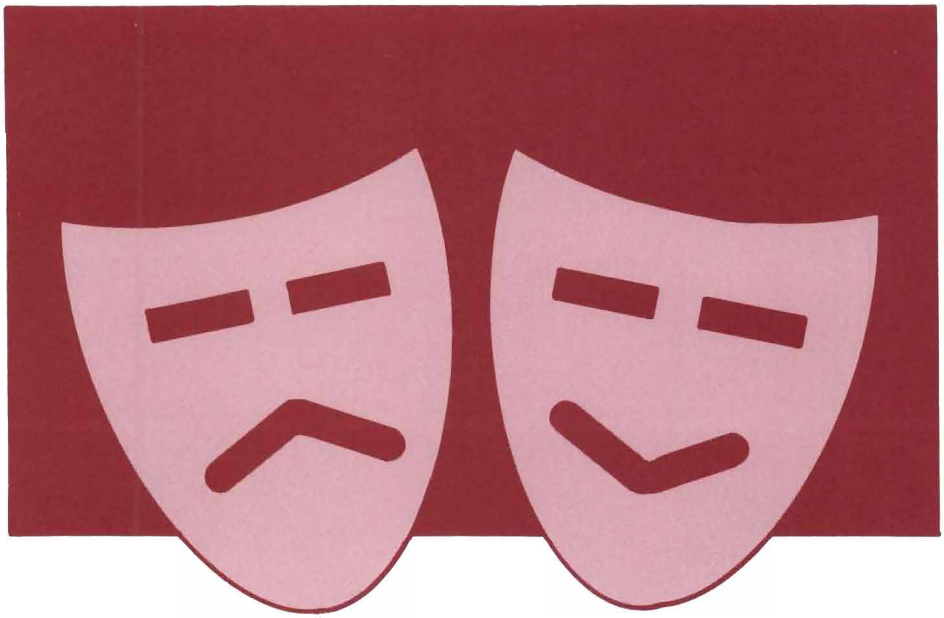
```
/* Simple arithmetic using variables */
pa = 1
ma = 1
kids = 3
say "There are " pa+ma+kids " people in this family"
```

Summary

You learned the following in this chapter:

Term/Concept	Description	Page
Variable	A piece of data given a unique name.	38
Value	What a variable contains.	39
Addition	+ operator	43
Subtraction	- operator	43
Multiplication	* operator	43
Division	/, //, % operators	44
Arithmetic Expressions	Follow the rules of algebra.	46

Chapter 5. More about Expressions



So far you have seen several REXX programs and you have created two of your own programs. You know how to communicate with the computer in simple terms, just as you might learn to talk in brief sentences to someone who speaks another language.

In this chapter, you will see how to take advantage of some features of REXX that can help you write more involved programs. You will learn how to have a program make decisions by testing a value with the IF instruction. You will see how to compare values and

determine if an expression is true or false. All these features help you communicate more fluently with REXX.

Making Decisions

All the programs you have seen or written so far have executed "sequentially." This means that each instruction is processed in the order it is written, beginning with the first line of the program. An important feature of programming is that you can control the order in which statements are run.

Two instructions that let you make decisions in your programs are the IF and SELECT Instructions. The IF instruction lets you control whether the next instruction should be run or skipped. The SELECT

instruction lets you choose one instruction to run from a group of instructions.

The IF Instruction

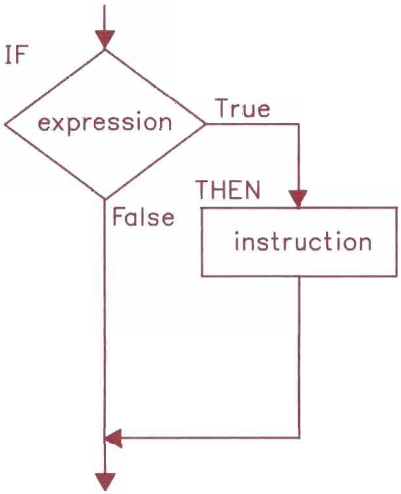
The IF instruction is used with a THEN keyword to make a decision. The interpreter executes the instruction if the expression is true. For example:

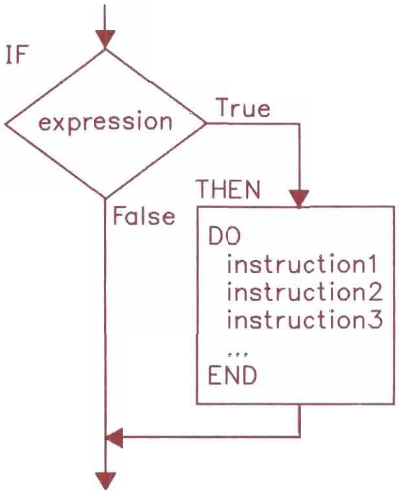
```
if answer = "YES"
 then
 say "OK!"
```

In the above example, the SAY instruction is run only if `answer` has the value of `YES`.

Notice, too, that the instructions above are indented. Use indentation, especially for longer programs, because it groups related lines and makes your program easier to read. The number of spaces to indent the instructions is up to you.

The following diagram illustrates making a decision with the IF-THEN format.

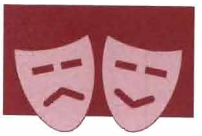




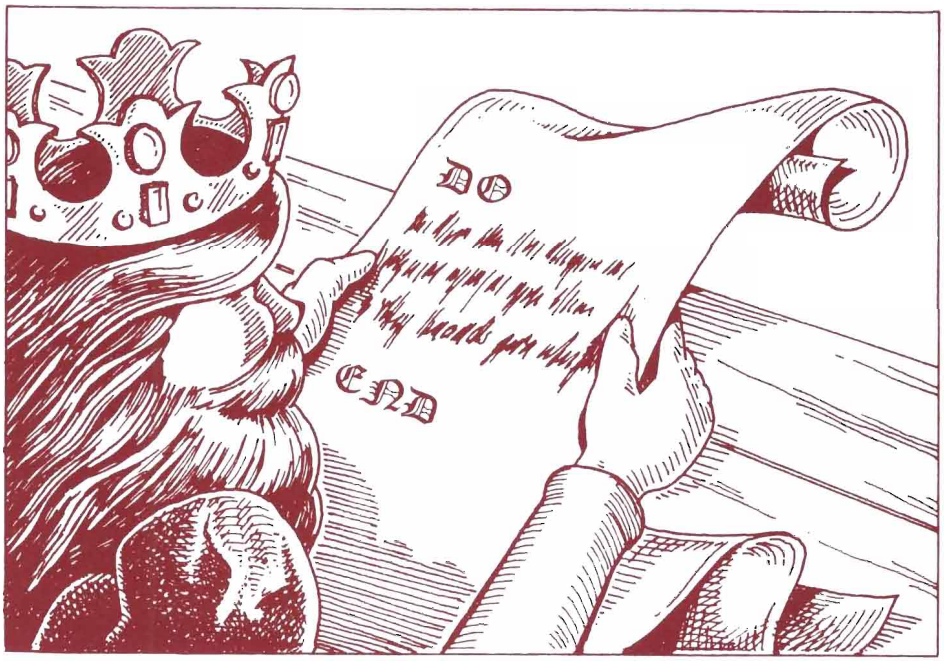
Grouping Instructions

To tell the interpreter to execute a list of instructions following the THEN keyword, use:

```
DO
   instruction1
   instruction2
   instruction3
END
```



The DO instruction and its END keyword tell the interpreter to treat any enclosed instructions as a single instruction.



The ELSE Keyword

To tell the interpreter to select from one of *two* possible instructions, use:

```
if expression
 then instruction1
else instruction2
```

You could include the IF-THEN-ELSE format in a program like this:

```
if answer = 'YES'
 then say 'OK!'
else say 'Why not?'
```

Try this example to see how it works:

```
/* Using IF-THEN-ELSE */
say "Do you like me?"
pull answer
if answer = "YES"
 then
 say "I like you too."
else
 say "I don't like you either."
exit
```

Figure 19. LIKEME EXEC, choosing between two instructions

When Mike tried the program, he saw:

```
Ready;
likeme
Do you like me?
yes
I like you too.
Ready;
```

The SELECT Instruction

The SELECT Instruction tells the interpreter to select one of a number of instructions. It is used only with the keywords WHEN, THEN, END and sometimes, OTHERWISE. The END statement marks the end of every SELECT group.

The SELECT instruction looks like this:

```
SELECT
   WHEN expression1
      THEN instruction1
   WHEN expression2
      THEN instruction2
   WHEN expression3
      THEN instruction3
   ...
   OTHERWISE
      instruction
      instruction
      instruction
END
```

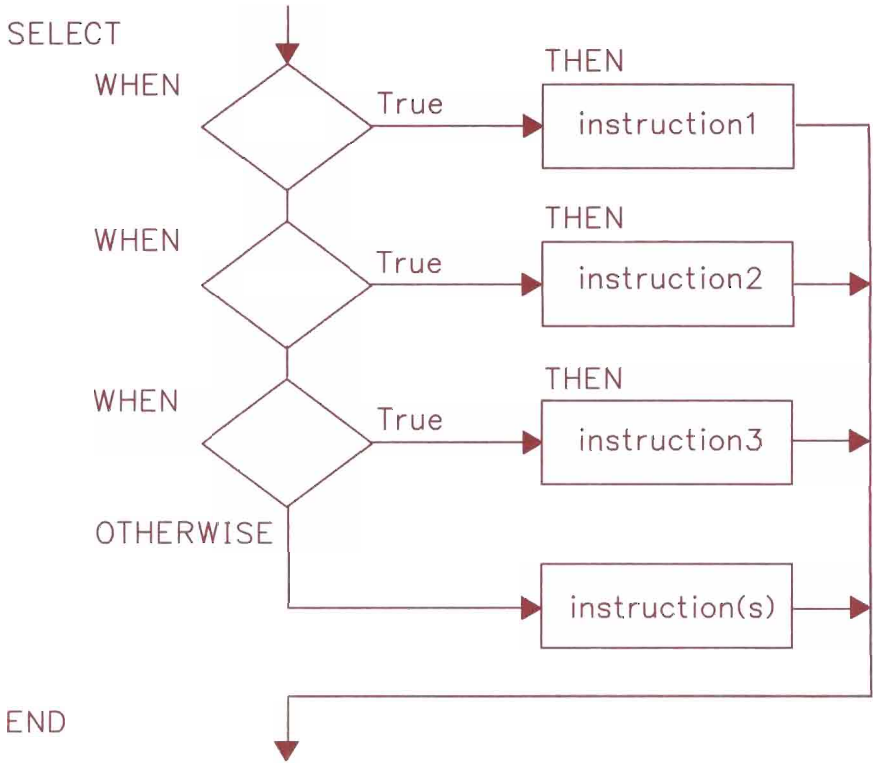
Note that an IF-THEN instruction cannot be used with a SELECT instruction, unless it follows a WHEN or OTHERWISE instruction. You can read this format as follows:

- If `expression1` is true, `instruction1` is run. After this, processing continues with the instruction following the `END`. The END keyword signals the end of the SELECT instruction.

- If `expression1` is false, `expression2` is tested. Then, if `expression2` is true, `instruction2` is run and processing continues with the instruction following the `END`.

- If, and only if, all of `expression1`, `expression2`, etc., are false, then processing continues with the instruction following the `OTHERWISE`.

This diagram shows the SELECT instruction:



A DO-END could be included inside a SELECT instruction like this:

```
SELECT
   WHEN expression1 THEN
      DO
        instruction1
        instruction2
        instruction3
      END
  .
  .
  .
```

You can use the SELECT instruction when you are looking at one variable that can have several different values associated with it. With each different value, you can set a different condition.

How does the SELECT instruction fit a program you might use? Suppose you wanted a reminder of weekday activities. For the variable "day", you can have a value of Monday through Friday. Depending on the day of the week (the value of the variable), you can list a different activity (instruction). You could use a program like this:

```
/* Selecting weekday activities */
say 'What day is it today?'
pull day
select
   when day = 'MONDAY'
      then
      say 'Exercise class today'
   when day = 'TUESDAY'
      then
      say 'Do the laundry'
   when day = 'WEDNESDAY'
      then
      say 'Exercise class again'
   when day = 'THURSDAY'
      then
      say 'Clean the house'
   when day = 'FRIDAY'
      then
      say 'Happy Hour!'
otherwise
   say "It's the weekend!"
end
exit
```

Figure 20. SELECT EXEC, choosing from several instructions

Mike tried this program too. Here's his reminder:

```
Ready;
select
What day is it today?
Thursday
Clean the house
Ready;
```

The NOP Instruction

In a program, a THEN or ELSE keyword must be followed by an instruction. If you are using an instruction such as SELECT, and you intend that nothing should happen for one expression, you can use the NOP (No Operation) instruction.

This example uses the previous SELECT program:

```
/* Selecting weekday activities */
say 'What day is it today?'
pull day
select
   when day = 'MONDAY'
      then
      say 'Exercise class today'
   when day = 'TUESDAY'
      then nop                    /* Nothing happens here */
   when day = 'WEDNESDAY'
      then
      say 'Exercise class again'
   when day = 'THURSDAY'
      then
      say 'Clean the house'
   when day = 'FRIDAY'
      then
      say 'Happy Hour!'
otherwise
   say "It's the weekend!"
end
exit
```

gure 21. SELECT EXEC with NOP

True and False Operators

Determining if an expression is true or false is useful in your programs. If an expression is true, the computed result is "1". If an expression is false, the computed result is "0". The following sections show several ways to check for true or false.

Comparisons

Some operators you can use for comparisons are:

>	Greater than
<	Less than
=	Equal to

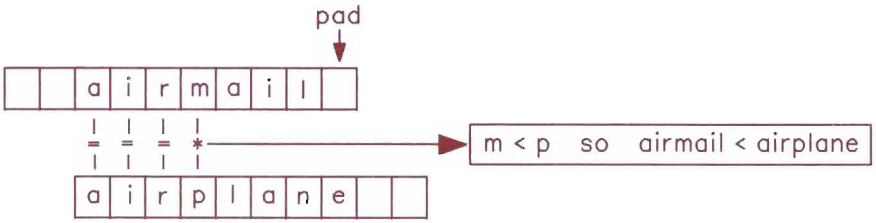
Comparisons can be made with numbers, or they can be character by character. Some numeric comparisons are:

The value of 5 > 3 is 1 This result is true.

The value of 2.0 = 002 is 1 This result is true.

The value of 332 < 299 is 0 This result is false.

If the terms being compared are not numbers, the interpreter compares characters. For example, the two words (strings) "airmail" and "airplane" are compared as follows:



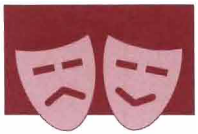
If this seems confusing, think of the REXX interpreter comparing the words like this:

- Leading and trailing blanks are ignored. These are the blank spaces before or after the word.
- The shorter word (airmail) is padded on the right with blanks.
- The words are compared from left to right, character by character.
- If the strings are not equal, the first pair of characters that do not match are used to determine the result.

A character is less than another character according to this sequence of lowest to highest value:

```
Lowest ──────────────────────────────────────────────► Highest

       blank
       special characters (for example: &, !, $, %, ?, #, @)
       a   z
       A   Z
       0   9

Highest
```

When "airmail" and "airplane" are compared, the first character that is different is the "m" of airmail and the "p" of airplane. The "m" is less than the "p," so "airmail" is less than "airplane."



Equal

An equal sign (=) can have two meanings in REXX, depending on its position. For example:

```
amount = 5              /* This is an assignment */
```

says the variable `amount` gets the value of 5, as discussed in "Assigning Values" on page 39. If an equal sign is in a statement other than as an assignment, it means the statement is a comparison. For example:

```
say amount = 5          /* This is a comparison */
```

compares the value of `amount` with 5. If they are the same, a "1" is displayed, otherwise, a "0" is displayed.

Using Comparisons

This program uses comparisons and an equal expression to determine if numeric expressions are true or false.

```
/* Determining if expression is true or false */
/* 1 is true; 0 is false                      */
a = 4
b = 2
c = a > b
say 'The result of' a '>' b 'is' c
c = a < b
say 'The result of' a '<' b 'is' c
c = a = b
say 'The result of' a '=' b 'is' c
exit
```

Figure 22. TF EXEC, checking for true or false

When you run the program, it looks like this:

```
Ready;
tf
The result of 4 > 2 is 1
The result of 4 < 2 is 0
The result of 4 = 2 is 0
Ready;
```

The Logical NOT Operator

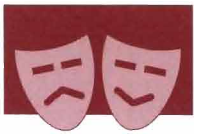
Logical operators can only return the values of 1 or 0. The ***NOT*** operator (¬) in front of a term reverses its value from true to false, or from false to true.

```
say ¬ 0          /* says '1'                  */
say ¬ 1          /* says '0'                  */
say ¬ (4 = 4)    /* says '0'                  */
say ¬ 2          /* gives a syntax error      */
```

The Logical AND Operator

The ***AND*** operator (&) between two terms gives a value of true only if both terms are true.

```
say (3 = 3) & (5 = 5)   /* says '1'           */
say (3 = 4) & (5 = 5)   /* says '0'           */
say (3 = 3) & (4 = 5)   /* says '0'           */
say (3 = 4) & (4 = 5)   /* says '0'           */
```



The following program shows the AND operator:

```
/* Using the AND (&) Operator  */
/* 0 is false; 1 is true       */
a = 4
b = 2
c = 5
d = (a > b) & (b > c)
say 'The result of (a > b) & (b > c) is' d
d = (a > b) & (b < c)
say 'The result of (a > b) & (b < c) is' d
exit
```

Figure 23. AND EXEC, checking for two true statements

On the screen, the program looks like this:

```
R;
and
The result of (a > b) & (b > c) is 0
The result of (a > b) & (b < c) is 1
R;
```

The Logical OR Operator

The ***OR*** operator (|) between two terms gives a value true, unless both terms are false.

```
say (3 = 3) | (5 = 5)  /* says '1'      */
say (3 = 4) | (5 = 5)  /* says '1'      */
say (3 = 3) | (4 = 5)  /* says '1'      */
say (3 = 4) | (4 = 5)  /* says '0'      */
```

The following program shows the OR operator:

```
/* Using the OR (|) Operator */
/* 0 is false; 1 is true     */
a = 4
b = 2
c = 5
d = (a > b) | (b > c)
say 'The result of (a > b) & (b > c) is' d
d = (a > b) | (b < c)
say 'The result of (a > b) & (b < c) is' d
exit
```

Figure 24. OR EXEC, statement true unless both values false

On the screen, the program looks like this:

```
Ready;
or
The result of (a > b) & (b > c) is 1
The result of (a > b) & (b < c) is 1
Ready;
```

Exercises

You can find the answer to these exercises in the appendix.

1. Read the following program, MEASURES EXEC. Write down what you think will appear on the screen when the program is run.

MEASURES EXEC:

```
/* Comparing numbers */
dozen = 12
score = 20
say score = dozen + 8
say
/* Using the AND operator */
say dozen = 12 & score = 21
exit
```

2. What is the value of each of the following expressions?

   "5" > "five"
   "Kilogram" > "kilogram"
   "a" > "#"
   "q" > "?"
   "9a" > "9"
   "?" > " "

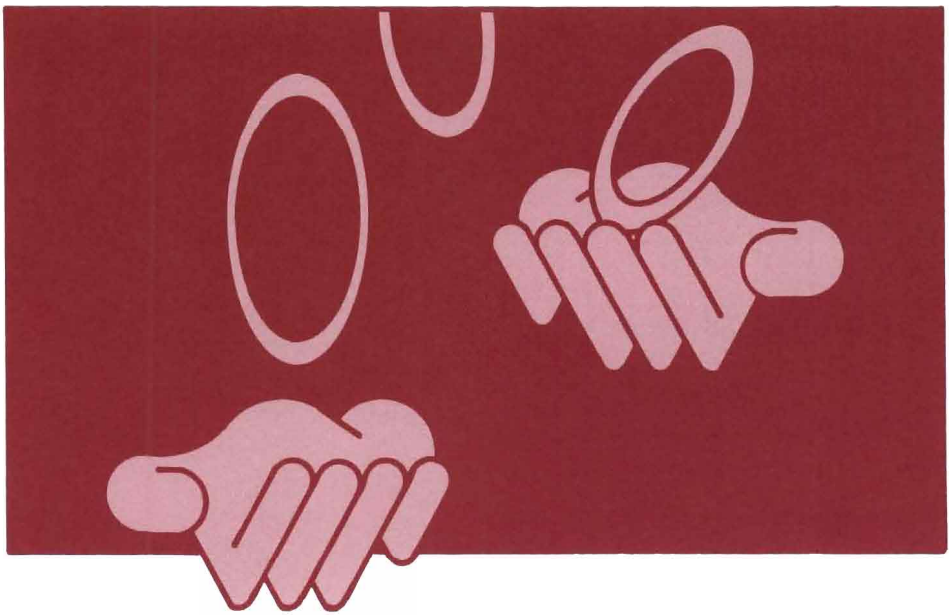
Summary

You learned the following in this chapter:

Term/Concept	Description	Page
IF	Used with THEN. Checks if the expression is true. Makes a decision about a single instruction.	57
THEN	Identifies the instruction to be executed if the expression is true.	57
ELSE	Used with the IF instruction. Tells the interpreter to select one of two instructions.	59

Term/Concept	Description	Page
DO-END	Indicates that a group of instructions should be executed.	58
SELECT	Tells the interpreter to select one of a number of instructions.	61
WHEN	Used with SELECT. Identifies an expression to be tested.	61
OTHERWISE	Used with SELECT. Indicates the instructions to be executed if expressions tested are false.	61
NOP	Used with an instruction when you want nothing to happen for one expression.	64
Comparisons > < =	Greater than, less than, equal to	65
NOT Operator ¬	Changes the value of a term from true to false, or from false to true.	68
AND Operator &	Gives the value of true if both terms are true.	68
OR Operator |	Gives the value of true unless both terms are false.	69

Chapter 6. Automating Repetitive Tasks



Do you often do one task over and over again?

Within a program, you can automatically repeat a task by using loops. Through loops, you can keep adding or subtracting numbers until you want to stop. You can define how many times you want a program to handle an operation.

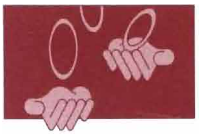
In this chapter, you will learn how to use some simple loops. And, you will write your third program.

Using Loops

If you want to repeat several instructions in a program, you can use a ***loop***. Loops are often used in programming because they condense many lines of instructions into a group that can be executed more than once. Loops make your programs more concise. And, with a loop, you can keep asking for input from a user until the correct answer is given.

The two types of loops you may find useful are repetitive loops and conditional loops. Loops begin with a DO statement and end with the END statement.

Simple repetitive loops can be executed a number of times. You can specify the number of repetitions for the loop, or use a variable that



has a changing value. ***Conditional*** loops are executed when a true or false condition is met.

Repetitive Loops

To repeat a loop a fixed number of times you can use the following simple loop:

```
DO num
   instruction1
   instruction2
   instruction3
   ...
END
```

The *num* is a whole number, which is the number of times the loop is to be executed.

Here is an example of a simple repetitive loop:

```
/* A simple loop */
do 5
   say 'Ho'
end
exit
```

Figure 25. LOOP EXEC, an example of a simple loop

When you run the LOOP EXEC, you will see this on your screen:

```
Ready;
loop
Ho
Ho
Ho
Ho
Ho
Ready;
```

Another format is:

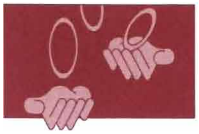
```
DO I = 1 to 10
```

This format numbers each pass through the loop so you can use it as a variable. The value of `I` is changed (increased by 1) each time you pass through the loop. The "1" (or some number) gives the value you want the variable to have the first time through the loop. The "10" (or some number) gives the value you want the variable to have the last time through the loop.

An example is:

```
/* Another loop */
sum = 0
Do I = 1 to 10
   say 'Enter value' I
   pull value
   sum = sum + value
end
say 'The total is' sum
exit
```

Figure 26. NEWLOOP EXEC, an example of another loop



Here's the results when Mike tried this program:

```
Ready;
newloop
Enter value 1
2
Enter value 2
4
Enter value 3
6
Enter value 4
8
Enter value 5
10
Enter value 6
12
Enter value 7
14
Enter value 8
16
Enter value 9
18
Enter value 10
20
The total is 110
Ready;
```

When a loop ends, the program continues with the next instruction following the end of the loop, identified by the END keyword.

Conditional Loops

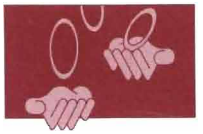
Conditional loops are executed as long as a condition is met. The following sections describe some instructions used for conditional loops.

The DO WHILE and DO UNTIL Instructions

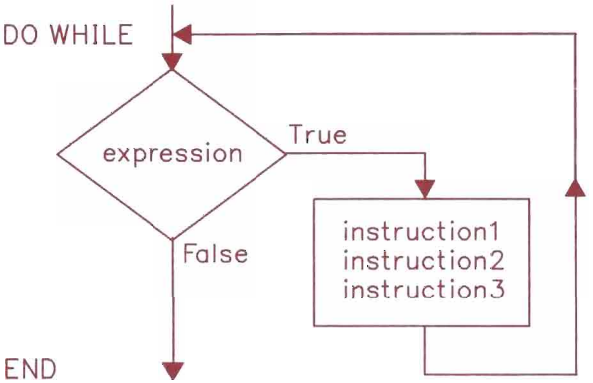
The DO WHILE and DO UNTIL instructions are run "while" or "until" some condition is met. A DO WHILE loop is:

```
DO WHILE expression
   instruction1
   instruction2
   instruction3
END
```

With the DO WHILE instruction, the program evaluates if the expression is true *before* processing the instructions that follow. If the expression is true, the instructions are executed. If the expression is false, the loop ends and moves to the instruction following the END instruction. The DO WHILE instruction tests for a true or false condition at the top of the loop.



This diagram shows the DO WHILE instruction:



A program using a DO WHILE loop is:

```
/* Using a DO WHILE loop */
say 'Enter the amount of money available'
pull salary
spent = 0
do while spent < salary
   say 'Type in cost of item'
   pull cost
   spent = spent + cost
end
say 'Empty pockets.'
exit
```

Figure 27. DOWHILE EXEC tests for true or false at top of loop

After running the DOWHILE program, it looks like this:

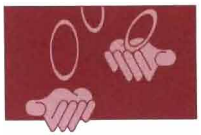
```
Ready;
dowhile
Enter the amount of money available
100
Type in cost of item
57
Type in cost of item
24
Type in cost of item
33
Empty pockets.
Ready;
```

A DO UNTIL instruction differs from the DO WHILE because it processes the body of instructions first, then evaluates the expression. If the expression is false, the instructions are repeated (a loop). If the expression is true, the program ends or moves to the next step outside the loop.

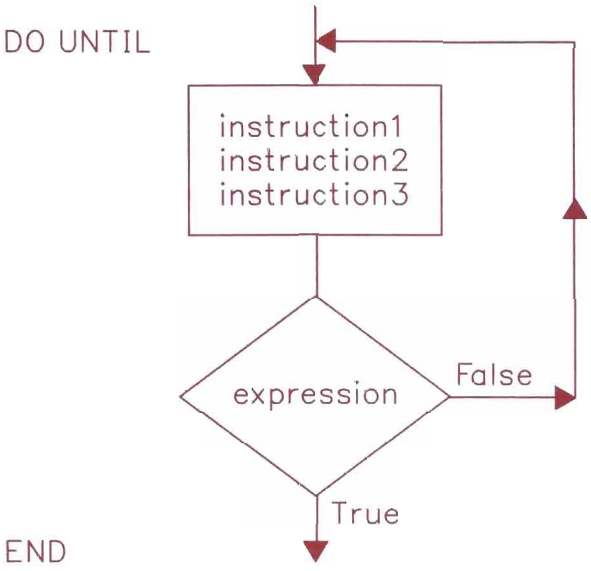
The DO UNTIL instruction tests at the bottom of the loop and, therefore, the instructions within the DO loop are executed at least once.

A DO UNTIL loop looks like this:

```
DO UNTIL expression
   instruction1
   instruction2
   instruction3
   ...
END
```



This diagram shows the DO UNTIL instruction:



The DOWHILE program can be changed to process a DO UNTIL loop like this:

```
/* Using a DO UNTIL loop */
say 'Enter the amount of money available'
pull salary
spent = 10          /* Sets spent to a value of 10 */
do until spent > salary
   say 'Type in cost of item'
   pull cost
   spent = spent + cost
end
say 'Empty pockets.'
exit
```

Figure 28. DOUNTIL EXEC tests for true or false at bottom of loop

It looks like this on your screen:

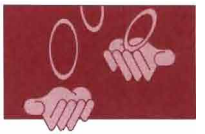
```
Ready;
dountil
Enter the amount of money available
50
Type in cost of item
37
Type in cost of item
14
Empty pockets.
Ready;
```

The LEAVE Instruction

In the above example, you may want to end the loop before the ending conditions are met (before you run out of money). You can accomplish this with the LEAVE instruction. This instruction ends the loop and and continues processing with the instruction following the END.

```
/* Using the LEAVE instruction in a loop */
say 'Enter the amount of money available'
pull salary
spent = 10          /* Sets spent to a value of 10 */
do until spent > salary
   say 'Type in cost of item or END to quit'
   pull cost
      if cost = 'END'
      then
      leave
   spent = spent + cost
end
say 'Empty pockets.'
exit
```

Figure 29. LEAVE EXEC causes the interpreter to end the loop



The DO FOREVER Instruction

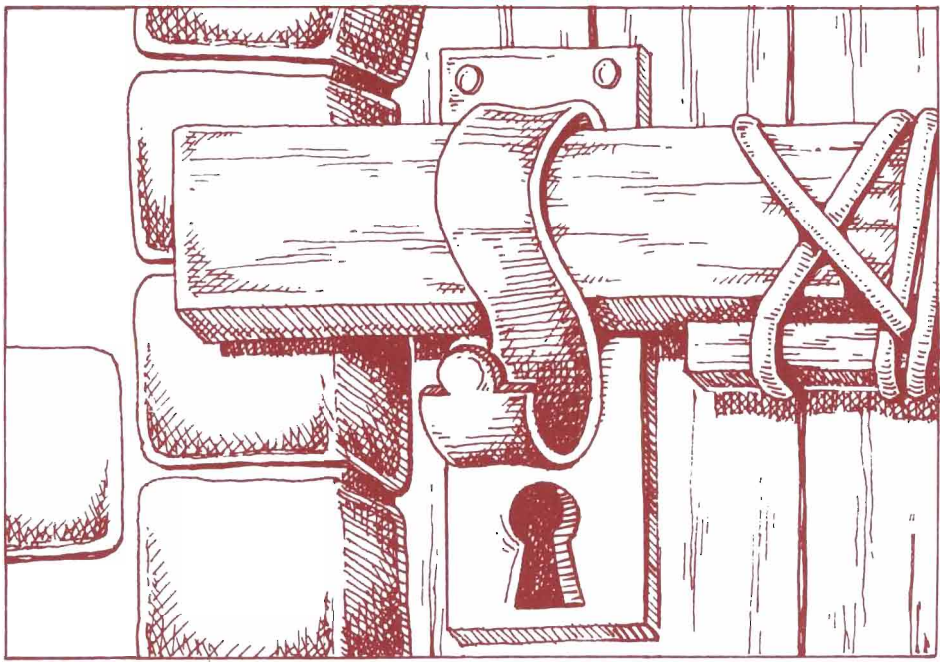
There may be situations when you don't know how many times to repeat a loop. For example, you may want a user to enter specific numeric data (numbers to add together), and you want the loop to perform the calculation until the user says to quit. For this program, you can use the DO FOREVER instruction with the LEAVE instruction.

A simple use of a DO FOREVER loop is:

```
/* Using a DO FOREVER loop to add numbers */
sum = 0
do forever
   say 'Enter number or END to quit'
   pull value
   if value = 'END'
      then
      leave            /* Program quits when user enters 'end' */
   sum = sum + value
end
say 'The sum is' sum
exit
```

Figure 30. FOREVER EXEC ends when the user quits

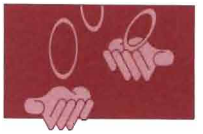
Getting Out of Loops



To stop most programs, you can enter the command:

```
HI
```

However, if you tried a program like the one below, the HI command will not work.



```
/* Guess the secret password ! */
do until answer = "I quit!"
   say "What is your answer?"
   pull answer
end
exit
```

Figure 31. SECRET EXEC, a program that may never end

If you were not familiar with the program, you would not know the correct response to end it. Typing `HI` will not stop the program. The `HI` just gets compared with `I quit!`, and because they are not equal, entering `HI` will never work.

You can recognize this situation because, whatever you do, the words `VM READ` continue to appear in the bottom right hand corner of your screen. If you don't know the answer, the simplest way out is to enter:

```
#cp ipl cms
```

For more information on this command, see the *VM/SP CMS Primer*.

Parsing Words

In "The PULL and PARSE PULL Instructions" on page 16 you learned that the PULL instruction collects a response and puts it in the computer's memory as a variable. PULL can also be used to fetch several words and put each word into a different variable. In REXX, this is called parsing. The variable names used in the next example are: FIRST, SECOND, THIRD, and REST.

```
say 'Please enter three or more words:'
pull first second third rest
 ...
```

Suppose you entered this as your response:

```
garbage in garbage      out
```

When you pressed the ENTER key, the program would continue. However, the variables would be assigned as follows:

The variable FIRST is given the value "GARBAGE"
The variable SECOND is given the value "IN"
The variable THIRD is given the value "GARBAGE"
The variable REST is given the value " OUT"

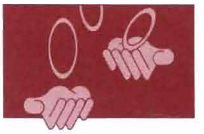
In general, each variable gets a word, without blanks, and the last variable gets the rest of the input, if any, with blanks. If there are more variables than words, the extra variables are assigned the null, or empty, value.

Creating Another Program

The third program you will create combines several concepts from this part of the book. It's something you can use every day: a simple desk calculator. The CALC EXEC will handle the four arithmetic operations of addition, subtraction, multiplication, and division. Let's also incorporate a loop in the program so you can enter calculations without restarting the program.

Think about the requirements for this program. Cover up the next section with a piece of paper, and try writing a list of what the program should do line by line.

Did you have trouble writing a list? Now look at the requirements below and see if you understand them.



In the program, you will:

1. Identify and describe the program.

2. Tell the user to enter numbers and operators (like an equation).

3. Put the numbers and operators in the computer's memory.

4. Tell the interpreter to select one of the arithmetic operations to perform and process the arithmetic operator.

5. Display the answer on the screen.

6. Allow the user to keep entering calculations without starting the program again, or allow the user to end the program.

Writing the Program

You probably know several instructions to write for this program. Write down as many statements as you can for the steps. If you can't complete an entire instruction or statement, write as much as you can. Think about the program from beginning to end, and the proper sequence of the steps.

If you think you have a complete program, type it in and run it. If you need more explanation, continue with the following section.

1. Write a statement to identify the REXX program. You should feel confident in completing this required first step.

2. The next step requires asking for input. With a piece of paper, cover the paragraphs that follow so you can think through this step before you see our suggestions. Remember, the user should enter numbers and arithmetic operators. Write instructions to tell users to enter numbers and an operator, what

form they should use, and what symbols represent the operators. These instructions should be informative so the user knows how to use the program.

Hint: Use a block of SAY instructions to display all the information on the screen.

Write down your ideas. When you are finished, look at our suggestions below. If your instructions are somewhat different, remember, they may work too!

```
/* This program is your desk calculator.  */
say 'This program acts as a desk calculator.'
say 'Enter calculations in the form of:'
say     'number  operation  number'
say 'where operation is the symbol:  + - * /'
say 'for ADD, SUBTRACT, MULTIPLY, or DIVIDE'
say
say 'Enter your calculation (or press ENTER to quit)'
```

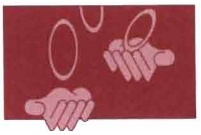
3. Now the equation (numbers and an operator) needs to be put in the computer's memory. Write an instruction to do this.

4. Next, if valid numbers and an operator are given, you want the interpreter to "select" the appropriate instruction to perform. What instruction would you use?

   Hint: Write a separate statement for each operation. Include an instruction to display the answer on the screen. Look back at the MATH EXEC and the ADD EXEC for ideas.

5. To keep entering calculations without having the program stop requires repetition of a group of instructions. If you thought of using a loop, you're right! Write down the instruction for a DO FOREVER loop.

6. Suppose the user doesn't enter a number. Let's provide a way of exiting the program if no numbers are given.



Hint: You can use the IF instruction with a THEN keyword.

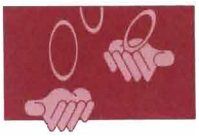
7. You are almost finished. Write the instruction to end the program.

You now have worked through the steps for the CALC EXEC. It's important, however, to put the instructions in the proper sequence. Think about the placement of the loop. Remember to end the loop and the SELECT instruction. When you have finished putting the CALC EXEC together, type the program in and file it. Congratulations again on writing a program!

When the instructions discussed above are put together, the program looks like this:

```
/* This program is your desk calculator.  */
say 'This program acts as a desk calculator.'
say 'Enter calculations in the form of:'
say 'number  operation  number'
say 'where operation is the symbol:  + - * /'
say 'for ADD, SUBTRACT, MULTIPLY, or DIVIDE'
say
do forever
say 'Enter your calculation (or press ENTER to quit)'
pull num1 op num2      /* Gets numbers and operator          */
if num1 = ''           /* If user presses ENTER key or space */
   then                /* program ends.                      */
   leave
select
when op = '+'
   then
   say num1 op num2 'is' num1 + num2
when op = '-'
   then
   say num1 op num2 'is' num1 - num2
when op = '*'
   then
   say num1 op num2 'is' num1 * num2
when op = '/'
   then
   say num1 op num2 'is' num1 / num2
otherwise
   say num1 op num2 'is not a valid calculation, Try again!'
end
end
exit
```

Figure 32. CALC EXEC uses the DO FOREVER and SELECT instructions



Running the Program

To test the CALC program, enter `calc` on the command line. Mike tried some numbers and here's the result:

```
Ready;
calc
This program acts as a desk calculator.
Enter calculations in the form of:
number  operation  number
where operation is the symbol:  + - * /
for ADD, SUBTRACT, MULTIPLY, or DIVIDE

Enter your calculation (or press ENTER to quit)
56 / 9
56 / 9 is 6.22222222
Enter your calculation (or press ENTER to quit)
12 * 12
12 * 12 is 144
Enter your calculation (or press ENTER to quit)
17 - 6
17 - 6 is 11
Enter your calculation (or press ENTER to quit)

Ready;
```

Exercises

You can find the answer to this exercise in the appendix.

Write a program to say the days of the week repeatedly. Use:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Monday
...
```

To stop the program, use the CMS command HI (Halt Interpretation).

Summary

You learned the following in this chapter:

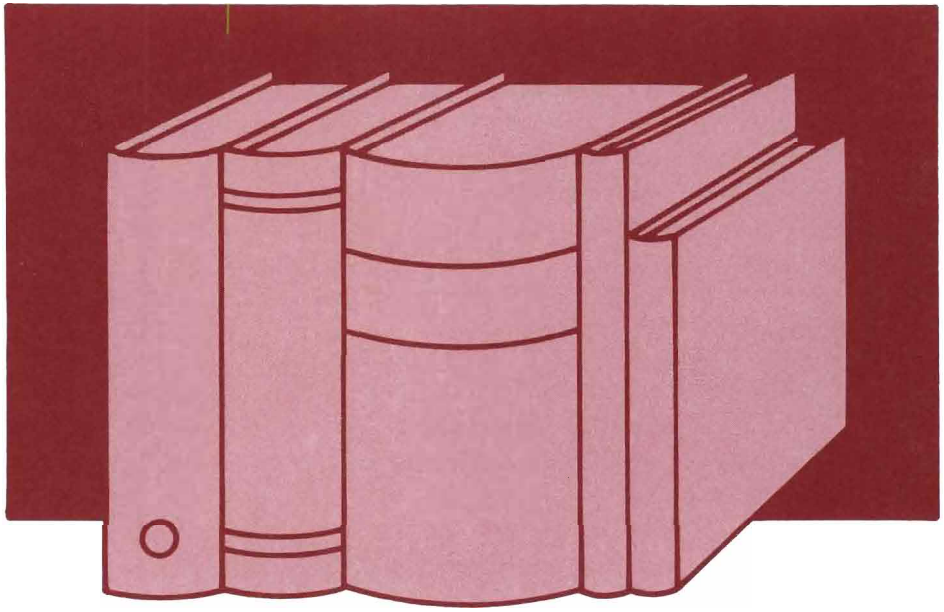
Term/Concept	Description	Page
DO num Loop	Repeats loop a fixed number of times.	75
DO I = 1 to 10 Loop	Each pass through the loop is numbered. Sets a starting and ending value for variable.	75
DO WHILE	Tests for true or false at top of loop. Repeats loop on true. On false, continues processing after END.	78

Term/Concept	Description	Page
DO UNTIL	Tests for true or false at bottom of loop. Repeats loop on false. On true, continues processing after END.	78
LEAVE	Causes interpreter to exit a loop.	82
DO FOREVER	Repeats instructions until the user says to quit.	83
Getting Out of Loops	Use commands: HI or #cp ipl cms	84
Parsing Words	Assigns a different variable to each word in a group.	85

part 3

Improving Your Skills

Chapter 7. Using More Advanced Features



As you become more skilled at programming, you will want to create programs that do more and run more efficiently. Sometimes this means adding a special function to your program or calling a subroutine.

This chapter explains how these tools will help you build a better foundation in REXX.

Using Functions

In REXX, a ***function call*** can be written anywhere in an expression. The function performs the requested computation and returns a result. REXX then uses the result in the expression in place of the function call.

Think of a function like this: You are trying to find someone's telephone number (complete a function). You call the telephone operator and ask him to look up the number. He gives you a number, and you call the person.

Generally, if the interpreter finds this in an expression:

```
name(expression)
```

it assumes that `name` is the name of a function and that this is a call to the function `name()`. There is *no space* between the end of the name and the left parenthesis. If you leave out the right parenthesis it is an error.

The expressions inside the parentheses are the ***arguments***. The argument can itself be an expression; the interpreter computes the value of this expression before passing it to the function. If a function requires more than one argument, use commas to separate each argument.

Built-in Functions

More than fifty functions are "built-in" to REXX. In this book, only a few will be introduced. You will find a dictionary of built-in functions in the *VM/SP System Product Interpreter Reference*.

Let's continue the discussion of functions by looking at the built-in function to obtain the greatest number of a set of numbers:

```
MAX(number,number, ... )
```

For example:

```
MAX(2,4,8,6) = 8
```

MAX(2,4+5,6) = 9

Note that in the second example, the 4+5 is an expression. A function call, like any other expression, usually appears in a clause as part of an assignment or instruction.

The DATATYPE() Function

When attempting to do arithmetic on data entered from the keyboard, you can use the DATATYPE() function to check that data is valid.

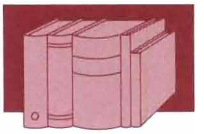
This function has several forms. The simplest form returns the word, NUM, if the argument (the expression inside the parenthesis) would be accepted by the interpreter as a number that could be used in arithmetical operations. Otherwise, returns the word, CHAR. For example:

The value of DATATYPE(56) is NUM
The value of DATATYPE(6.2) is NUM
The value of DATATYPE($5.50) is CHAR

This program asks the user to keep entering a valid number until he succeeds:

```
/* Using the DATATYPE() Function  */
do until datatype(howmuch) = 'NUM'
 say 'Enter a number'
 pull howmuch
  if datatype(howmuch) = 'CHAR'
   then
   say 'That was not a number. Try again!'
end
say 'The number you entered was' howmuch
exit
```

Figure 33. DATATYPE EXEC, using a REXX built-in function



If you want the user to enter only whole numbers, you could use another form of the DATATYPE() function:

```
DATATYPE(number,whole)
```

The arguments for this form are:

1. `number` refers to the data to be tested.

2. `whole` refers to the type of data to be tested. In this example, the data must be a whole number.

This form returns a "1" if `number` is a whole number, or a "0" otherwise.

The SUBSTR() Function

The value of any REXX variable can be a string of characters. To select a part of a string, you can use the SUBSTR() function. SUBSTR is an abbreviation for substring. The first three arguments are:

1. The string from which a part will be taken

2. The position of the first character that is to appear in the result (characters are numbered 1, 2, 3 ...in the string)

3. The length of the result.

For example:

```
S = 'reveal'
say substr(S,2,3)       /* Says 'eve'. Beginning with the second   */
                        /* character, takes three characters.      */
say substr(S,3,4)       /* Says 'veal'. Beginning with the third   */
                        /* character, takes four characters.       */
```

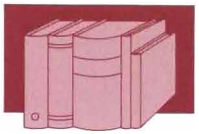
User-Written Functions

You can also write your own functions in REXX. You can use functions written by others in your organization. As a new programmer, you should learn more about the built-in functions, before attempting to write your own.

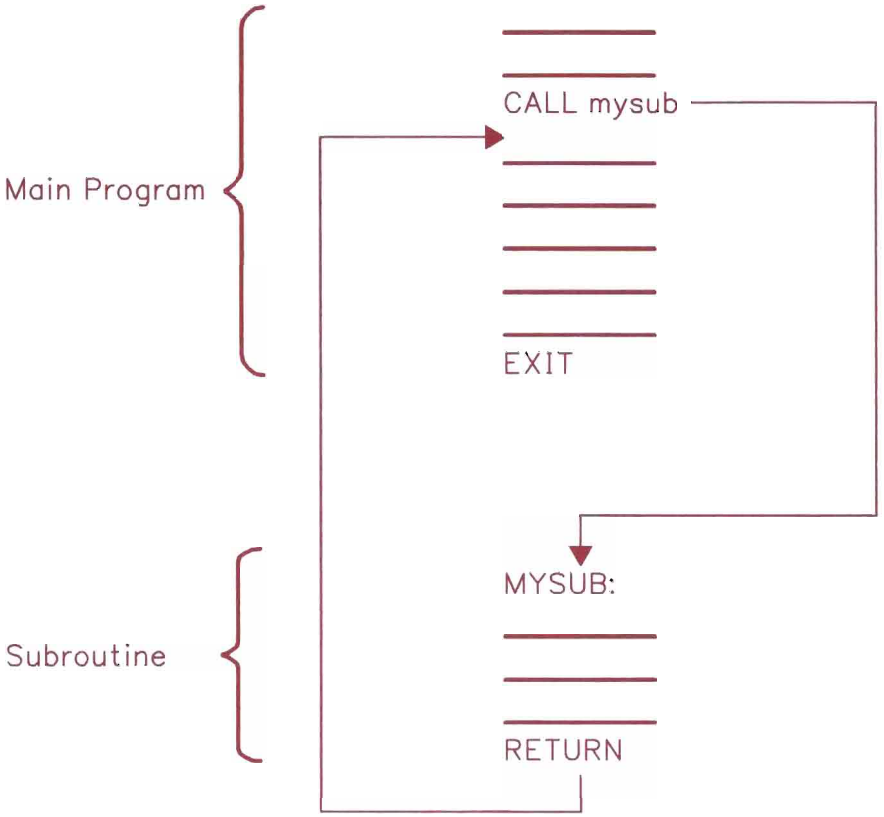
For more information, see user-written functions in the *VM/SP System Product Interpreter User's Guide.*

Using Subroutines

A ***subroutine*** is a group of statements that can be called from more than one place in your main program. Subroutines can be in the same file as the main program, or they can be in a separate EXEC file. The advantage of using a subroutine is that you can reuse blocks of instructions in several places in a program.



The following diagram shows a subroutine that is included in the same EXEC file as the main program.



The CALL Instruction

The CALL instruction causes the interpreter to look through your program until it finds a label that marks the start of the subroutine. Remember, a label (word) is a symbol followed by a colon (:), as shown in the diagram above. Processing continues from there until the interpreter finds a RETURN or an EXIT instruction.

A subroutine can be CALLed from more than one place in a program. When the subroutine is finished, the interpreter always RETURNs to the instruction following the CALL instruction from which it came.

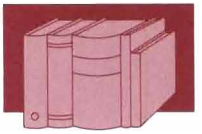
Often each CALL instruction supplies data, called arguments, that the subroutine is to use. In the subroutine, you can find out what data has been supplied by using the ARG instruction.

The CALL instruction appears in this form:

```
CALL name (argument1, argument2 ...)
```

For the name, the interpreter looks for the corresponding label (name) in your program. If no label is found, the interpreter looks for a built-in function or an EXEC file of that name.

The arguments are expressions. You can have up to ten arguments in a CALL instruction. An example of a program that calls a subroutine follows. Note that the EXIT instruction causes a return to CMS. It stops the main program from running on into the subroutine.



For example, this is your program:

```
/* Calling a subroutine from a program */
do 3
 call triple "R"
 call triple "E"
 call triple "X"
 call triple "X"
 say
end
say "R...!"
say "E...!"
say "X...!"
say "X...!"
say ''
say "REXX!"
exit             /* This ends the main program. */
/*                                                    */
/* Subroutine starts here to repeat shout three times. */
/* The first argument is displayed on the screen three */
/* times, with punctuation.                           */
/*                                                    */
TRIPLE:
say arg(1)",  "arg(1)",  "arg(1)"!"
return           /* This ends the subroutine. */
```

Figure 34. CHEER EXEC calls a subroutine from a main program

On the screen, the CHEER EXEC looks like this:

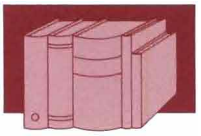
```
Ready;
cheer
R,   R,   R!
E,   E,   E!
X,   X,   X!
X,   X,   X!

R,   R,   R!
E,   E,   E!
X,   X,   X!
X,   X,   X!

R,   R,   R!
E,   E,   E!
X,   X,   X!
X,   X,   X!

R...!
E...!
X...!
X...!

REXX!
Ready;
```

The ARG Instruction

To assign the arguments to variables (and make the program easier to read), you can use the PARSE ARG instruction or the PARSE UPPER ARG instruction.

For example, if you have the following CALL instruction:

```
CALL DINNER 'apple', 'coffee', 'steak', 'pie'
```



and you want the results of the four expressions in this instruction to be assigned to appetizer, drink, main_course, and dessert, you can write:

```
PARSE ARG appetizer, drink, main_course, dessert
```

The other form of the instruction, PARSE UPPER ARG, can be shortened to ARG. Use this instruction if you want the four arguments changed to uppercase. For example:

```
ARG appetizer, drink, main_course, dessert
```

Notice that, just as there are commas between the expressions in the CALL instruction, so there are commas between the symbols in the PARSE ARG or ARG instruction when used this way.

The RETURN Instruction

As previously noted, the RETURN instruction takes you back to the main part of the program. Processing continues with the instruction following the CALL. The form of the instruction is simply:

```
RETURN
```

Issuing Commands from an EXEC

REXX works in a number of environments (for example, CMS or XEDIT). To keep things simple, only CMS commands will be discussed in the examples.

As discussed in "What's in a REXX Program" on page 14, anything not recognized as an instruction, assignment, or a label, is considered a command. The statement recognized as a command is

treated as an expression. The expression is evaluated first, then the result is passed to CMS.

The following example, COPYLIST EXEC, shows how a command is treated as an expression. Note how the special character (*) is put in quotes.

```
/* Issuing a command from a program. This example copies  */
/* all files that have a filetype of LIST from your       */
/* A-disk to your B-disk.                                 */
say
copy '*' list a '= =' b    /* This statement is treated as */
                           /* an expression.               */
                           /* The result is passed to CMS. */
exit
```

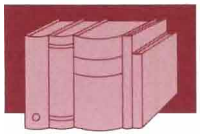
Figure 35. COPYLIST EXEC copies files from your A-disk to your B-disk

In the above example, if the asterisk (*) were not in quotes, the interpreter would attempt to multiply `copy` by `list`.

Working with Return Codes

When you write commands in your programs, you should consider what would happen if the command failed to execute correctly. For example, a COPYFILE command might fail because the user's disk was full. After this failure, you should at least EXIT from the program.

Here's how you discover such a failure. When commands have finished executing, they always provide a return code. A return code of zero nearly always means "all's well." Any other number usually means that something is wrong. You can see these codes on your screen when you issue commands from the CMS command line.



If the command worked normally (the return code was 0), you will see the "Ready" message, like this:

```
Ready;
```

If the command did not work, you will see the "Ready" message with a return code, like this:

```
Ready(00028);
```

Any command that would be valid on the command line is valid in a REXX program. The interpreter treats the command statement like any other expression, substituting the values of variables, and so on. (The rules are the same as for commands on the command line. For more information, see "The CMS Environment" in the *VM/SP System Product Interpreter Reference*.)

When the interpreter has issued a command and CMS or CP has finished running it, the interpreter gets the return code and stores it in the REXX special variable RC. In your program, you should test this variable to see what happened when the command was executed.

The following example shows a few lines from a program where the return code is tested:

```
copyfile '*' list a '= =' b
if rc = 0     /* RC contains the return code from COPYFILE command */
then
 say 'All "* list" files copied'
else
 say 'Error occurred copying files'
```

Exercises

You can find the answers to these exercises in the Appendix.

1. Suppose someone wrote a function called HALF(). The function returns half of a number. If a number is not even, the result returned is rounded high.

   What is the value of:

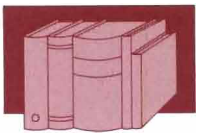
   a.HALF(100)
   b.HALF (100)
   c.HALF(19)
   d.HALF(HALF(26) + HALF(3+3))

2. The RANDOM() function can be used for games and for statistical models. For example, to obtain a number, chosen at random from the range 1 through 6, you could write:

   ```
   random(1,6)
   ```

   Write a program called TOSS that will display either the word "Heads" or (just as likely) the word "Tails". (Hint: The range is from 1 to 2). Run your program a number of times to see if the results are the same as if tossing a coin.

3. This program prints a list of items ordered for an office. Write the subroutine ASTERISK to print a line of asterisks between each item.



ASTERISK EXEC:

```
/* This EXEC calls a subroutine to print a line of asterisks */
call asterisk
say 'Item 1     Notebook'
call asterisk
say 'Item 2     Black Pens'
call asterisk
say 'Item 3     Calendar'
call asterisk
say 'Item 4     Staples'
call asterisk
exit
```

Copy the program with your subroutine into an EXEC file and test the program.

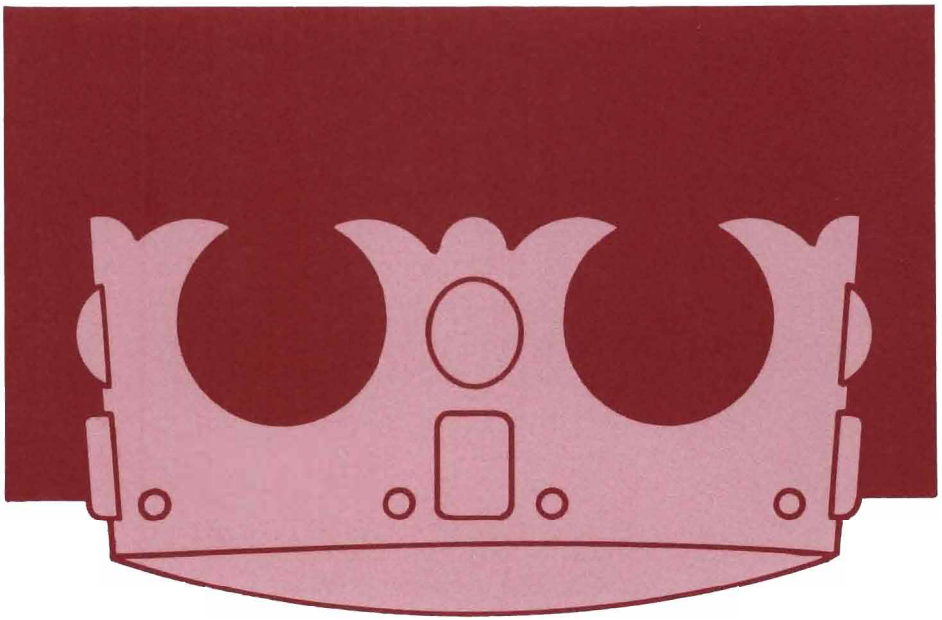
Summary

You learned the following in this chapter:

Term/Concept	Description	Page
Functions	Perform a computation and returns a result.	98
DATATYPE()	Built-in function: verifies if the data is a specific type.	99
SUBSTR()	Built-in function: selects part of a string.	101
Subroutines	Sequenced group of statements that can be called from more than one place in a program.	102
CALL	Causes the program to look for a subroutine label and begin running the instructions following the label.	104
ARG	Assigns arguments to variables.	106

Term/Concept	Description	Page
RETURN	Ends a subroutine; causes the interpreter to go to the next line following the CALL.	107
Issuing Commands from An EXEC	Commands are treated like expressions.	107
Return Codes	Tells you if the command executed correctly. A zero return code means "all's well."	108

Chapter 8. Learning More About REXX



You have just completed a basic introduction to REXX, but your education doesn't have to stop here. In this chapter, you will see how to add to previously developed programs, and put ideas to work in brand-new programs. You'll find some suggestions for getting help, too. Just as you may continue special studies beyond graduation from school, your knowledge of REXX can grow even after you close the pages of this book.

Enhancing Your Programs

The key to a successful program is that it can be used frequently. As your needs change, you may want to add more function to a program. Or, you may want to modify a program so it handles a different task. In the following examples, we will show you some ways to enhance two programs you wrote: the NOTEPAD EXEC and CALC EXEC.

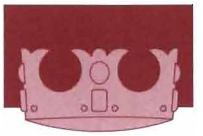
Modifying the NOTEPAD Program

The NOTEPAD program can be modified several ways. For a simple change, you can have the program tell what day it is when the reminders are listed. To do this, you can use the DATE() function. The original program contained these lines:

```
say 'Things to do today:'
type notepad script
exit
```

Adding the date requires a new assignment of a variable with the DATE() function, and the modification of the SAY instruction. The DATE() function has these options:

- Basedate
- Century
- Days
- European
- Julian-OS
- Month
- Ordered
- Sorted
- USA
- Weekday.



For more information on the options, see the *VM/SP System Product Interpreter Reference*.

You can assign the variable `today` a weekday like this:

```
today = date(weekday)
```

Remember, to indicate a REXX function, there should be no space between the name of the function (DATE) and the left parenthesis. The SAY instruction also needs to be changed so that `today` is treated as a variable.

```
say 'Things to do' today
```

The last line of the program remains the same:

```
'type notepad script'
```

The new program is:

```
/* A reminder of things to do */
today = date(weekday)              /* Assigns a weekday */
say 'Things to do' today
type notepad script
exit
```

Figure 36. NEWNOTE EXEC, modifying a program

When Mike tried the new program, it looked like this:

```
Ready;
newnote
Things to do Wednesday

1. Answer mail.
2. Call George about trip.
3. Order book.

Ready;
```

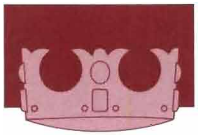
Modifying the CALC Program

Let's look at two changes you can make to the CALC program. The first change allows you to enter a character or the symbol for an arithmetic operator. The original program contained these lines:

```
/* This program is your desk calculator.  */
say 'This program acts as a desk calculator.'
say 'Enter calculations in the form of:'
say 'number  operation  number'
say 'where operation is the symbol:  + - * /'
say 'for ADD, SUBTRACT, MULTIPLY, or DIVIDE'
say
do forever
say 'Enter your calculation (or press ENTER to quit)'
pull num1 op num2      /* Gets numbers and operator          */
if num1 = ''           /* If user presses ENTER key or space */
   then                /* program ends.                      */
   leave
select
when op = '+'
   then
   say num1 op num2 'is' num1 + num2
when op = '-'
   then
   say num1 op num2 'is' num1 - num2
when op = '*'
   then
   say num1 op num2 'is' num1 * num2
when op = '/'
   then
   say num1 op num2 'is' num1 / num2
otherwise
   say num1 op num2 'is not a valid calculation, Try again!'
   end
end
exit
```

You can add this SAY instruction to the program:

```
say 'or enter the first character: A, S, M, D.'
```



To make this change work, you also have to modify the SELECT instruction as follows:

```
select
   when op='A' | op= '+'
      then say num1 '+' num2 'is' num1 + num2
   when op='S' | op= '-'
      then say num1 '-' num2 'is' num1 - num2
   when op='M' | op= '*'
      then say num1 '*' num2 'is' num1 * num2
   when op='D' | op= '/'
      then say num1 '/' num2 'is' num1 / num2
   otherwise
      say num1 op num2 'is not a valid calculation, try again!'
end
```

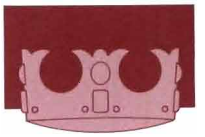
The second change lets you check the input from the user to see if numbers are entered for the variables `num1` and `num2`. To do this, you can use the DATATYPE() function with the IF instruction, as shown:

```
if datatype(num1) ¬= 'NUM' | datatype(num2) ¬= 'NUM'
  then say num1 op num2 'is not a valid calculation, try again!'
```

With the changes added to the CALC EXEC the new program is:

```
/* This program is your desk calculator.  */
say 'This program acts as a desk calculator.'
say
say 'Enter calculations in the form of:'
say
say '    number  operation  number'
say
say 'where operation is the symbol:  + - * /'
say 'for ADD, SUBTRACT, MULTIPLY, or DIVIDE'
say
say 'or enter the first character:  A, S, M, D.'
say
say
do forever
  say 'Enter your calculation (or press ENTER to quit)'
  pull num1 op num2
  if num1 = '' then leave
/*                                              */
/* Checking that numbers were entered           */
/*                                              */
  if datatype(num1) ¬= 'NUM' | datatype(num2) ¬= 'NUM'
    then say num1 op num2 'is not a valid calculation, try again!'
    else
      select
         when op='A' | op= '+'
            then say num1 '+' num2 'is' num1 + num2
         when op='S' | op= '-'
            then say num1 '-' num2 'is' num1 - num2
         when op='M' | op= '*'
            then say num1 '*' num2 'is' num1 * num2
         when op='D' | op= '/'
            then say num1 '/' num2 'is' num1 / num2
         otherwise
         say num1 op num2 'is not a valid calculation, try again!'
      end
end
exit
```

Figure 37. NEWCALC EXEC, enhancing the calculator program



When Mike tried the new program, it looked like this:

```
Ready;
newcalc
This program acts as a desk calculator.

Enter calculations in the form of:

    number  operation  number

where operation is the symbol:  + - * /
for ADD, SUBTRACT, MULTIPLY, or DIVIDE

or enter the first character:  A, S, M, D.


Enter your calculation (or press ENTER to quit)
7 a 5
7 + 5 is 12
Enter your calculation (or press ENTER to quit)
30 / 3
30 / 3 is 10
Enter your calculation (or press ENTER to quit)
16 m 4
16 * 4 is 64
Enter your calculation (or press ENTER to quit)

R;
```

Designing New Programs

As you progressed through this book, you may have thought of other programs you would like to write. Experienced programmers use several techniques to develop a program, such as flowcharts and block diagrams. Remember, though, that as a beginning programmer, you can design simple REXX programs by following these steps:

1. Identify the problem to solve, and translate it into a step-by-step procedure.

2. Write REXX instructions to solve each step.

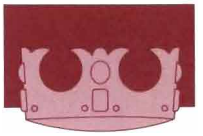
3. Test the program to see if the results meet the requirements.

4. Revise the program to correct errors.

Sometimes a simple problem requires many instructions to solve it. A program with many instructions is not necessarily more difficult than one with few instructions. The simple program may require more instructions because steps cannot be combined. A more difficult program may use built-in functions or subroutines to handle complex steps easily.

The QTIME EXEC

The following program provides a solution to the question, "How can I check what time it is through my computer?" It contains some familiar instructions and functions, as well as two or three new items. Although this program may look complicated, you have already learned many of the steps in this book. Putting the instructions together just requires some practice. You may want to try this program and keep it for your everyday use.



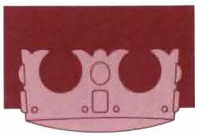
This is the QTIME EXEC:

```
/*-------------------------------------------------------------*/
/*  QTIME EXEC                                                 */
/*                                                             */
/*   Displays the current time in words and numbers.           */
/*-------------------------------------------------------------*/
/*     Assign words to compound variables.                     */
near.0=''                       /* Exactly.                    */
near.1='till'                   /* Till the hour.              */
near.2='after'                  /* After the hour.             */
now=time()                      /* Get the current system time */
/*                                                             */
/*Split the hours, minutes, and seconds into separate variables.*/
/*                                                             */
parse var now hour':'min':'sec
/*                                                             */
/*     If seconds are 30 or more, round up to the next minute. */
/*                                                             */
if sec > 29
 then min=min+1
/*                                                             */
/*     Adjust the hour and minutes if we are past the half hour.*/
/*                                                             */
select
  when min > 30 then            /* If we are past the half     */
                  do            /* hour, set the hour          */
                   hour=hour+1  /* variable to the next hour,  */
                   min=60-min   /* subtract current minutes    */
                                /* from 60 to get minutes      */
                                /* "till" the next hour.       */
                   if min=0     /* If minutes turn out to be   */
                    then mod=0  /* 0, then it is on the hour   */
                                /* (not exactly, but close     */
                                /* enough).  If minutes are    */
                    else mod=1  /* not 0, then prepare to      */
                  end           /* select the "till" compound  */
                                /* variable.                   */
  when min > 0 then mod=2       /* If we are not past the      */
                                /* half hour, but not exactly  */
                                /* on the hour, it's "after"   */
```

Figure 38 (Part 1 of 3). QTIME EXEC displays the time

```
 otherwise
  mod=0                          /* Otherwise, it's exactly on    */
end                              /* the hour.                     */
if min¬=00                    /* If the minutes are not        */
  then min=strip(min,'L',0)      /* zero, remove leading zeros    */
                                 /* (if any) from the minutes.    */
                                 /* Strip is a built-in           */
                                 /* function.                     */
/*     Check for special case of noon or midnight.               */
/*                                                               */
if hour//12=0 & min=00           /* Check if it's noon or         */
 then                            /* midnight.                     */
  do                             /* If so, which one?             */
   if hour=12                    /* If the hour equals 12,        */
    then
     say 'It''s 12 Noon.'        /* It's noon.                    */
    else
     say 'It''s 12 Midnight.'    /* Otherwise, it's midnight.     */
   exit                          /* We are finished now.          */
  end
 else                            /* If not noon or midnight,      */
  nop                            /* continue processing.          */
/*                                                               */
/* Otherwise, start building the appropriate response string.    */
/*                                                               */
select
  when min=1 then minute='minute' /* One minute after or till.  */
  when min>1 then minute='minutes'/* More than one minute.      */
  when min=00 then                /* Zero minutes, we need not  */
                do                /* display the minutes.       */
                 minute=''        /* Make the min and minute    */
                 min=''           /* variables null so they do  */
                end               /* not display.               */
 otherwise
  nop
end
```

Figure 38 (Part 2 of 3). QTIME EXEC displays the time



```
out='It''s' min minute near.mod    /* Compound variable, variable*/
                                   /* used as special character. */
                                   /* Build the output string as */
                                   /* it stands so far.          */
if hour > 12
 then hour = hour - 12             /* Get rid of 24-hour clock   */
 else if hour=0
       then hour = 12              /* ...and allow for midnight  */

hour = strip(hour,'L',0)           /* Remove any leading zeros   */
                                   /* from the hour, if any.     */
out=out hour                       /* Attach the hour to the     */
                                   /* output string.             */
if min=''
 then out=out 'o''clock'           /* Add o'clock if exactly on  */
                                   /* the hour.                  */
out=space(out,1)                   /* Make sure there is only    */
                                   /* one space between each     */
                                   /* word.                      */
                                   /* Space is a built-in        */
                                   /* function.                  */
say out'.'                         /* Display the final result.  */
exit                               /* Now we're done.            */
```

Figure 38 (Part 3 of 3). QTIME EXEC displays the time

Finding More Information

When learning something new, most of us make mistakes or have a problem now and then. Sometimes it is difficult to ask for help. If you have trouble with REXX programming, see if you can talk to someone about it. Sources to consider are:

- Your system administrator
- A programmer working in your organization

- An instructor in the computer science department at a local high school or college.

Conclusion

We hope this book took the mystery out of computer programming in REXX for you. You may have already thought of new ways REXX can help you with your tasks. If you tried the exercises in the book, and wrote a program, you can be pleased with your accomplishment.

Appendix. Answers to Exercises

Chapter 3 Answers

1. The syntax of the MADAM EXEC is:

   `/* Polite inquiry */` This is a comment.

   `Jane = "Mrs. Doe"` This is an assignment. The variable Jane gets the value of Mrs. Doe.

   `say` This is an instruction. The rest of the line is interpreted and the result is displayed.

   `How` This is a string.

   `is` This is changed to uppercase because it is not in quotes.

   `jane` This is the name of a variable. The value of Mrs. Doe is substituted.

   `?` This is the name of a variable.

   Here's what appears on the screen:

```
madam
How IS Mrs. Doe ?
```

2. There is a syntax error in the TROUBLE EXEC. It was caused by line 2, which contained an unexpected comma. To correct the program, remove the comma or add quotes around the comma.

Chapter 4 Answers

1. REXX variables are:

   a. Yes
   b. Yes
   c. Yes. It's the same as OLD_WORLD.
   d. No, because the first character is a number.
   e. Yes

2. The ASSIGN EXEC is missing the assignment of a value to the variable "input." We assigned input a value of 10 as follows:

```
/* This program has something missing! */
input = 10
say input
exit
```

3. The FAMILY program displays:

```
There are 5 people in this family
```

Chapter 5 Answers

1. For MEASURES EXEC, this is displayed when the program is run:

```
measures
1

0
Ready;
```

2. The values of the expressions are "1" (true).

Chapter 6 Answers

- A simple solution to say the days of the week repeatedly is:

  DAYS EXEC:

```
/* This program says days of the week indefinitely */
do forever
 say "Sunday"
 say "Monday"
 say "Tuesday"
 say "Wednesday"
 say "Thursday"
 say "Friday"
 say "Saturday"
end
exit
```

Chapter 7 Answers

1. The values using the HALF() function are:

   a. 50
   b. HALF 100. This is not a function because there is a space between the HALF and the left parenthesis.
   c. 10. The result (9) gets the remainder (1).
   d. 8.

2. A simple solution for the TOSS EXEC, using the RANDOM() function, is:

```
/* This program simulates tossing a coin */
if random(1,2) = 1
 then
 say "Heads"
else
 say "Tails"
exit
```

3. A possible solution for the ASTERISK subroutine is:

```
/* This EXEC calls a subroutine to print a line of asterisk */
call asterisk
say 'Item 1     Notebook'
call asterisk
say 'Item 2     Black Pens'
call asterisk
say 'Item 3     Calendar'
call asterisk
say 'Item 4     Staples'
call asterisk
exit
asterisk:
/* This prints a line of asterisk */
say '**************************'
return
```

Glossary

argument. In a function, the expression inside the parenthesis.

assignment. Putting a string in a special place in the computer's memory.

command. A command is a word, phrase or abbreviation that tells the system to do something. In REXX, anything that is not identified as a REXX instruction, assignment, or label is considered a command.

comment. In a REXX program, words that tell what a program is for, what kind of input it can handle, and what kind of output it produces.

decimal fraction. A number with a decimal point (for example, 1.5)

function call. A procedure that performs a requested computation and returns a result.

instruction. Tells the REXX interpreter to do something.

label. A name followed by a colon. Used with subroutines.

loop. A group of instructions that can be executed more than once. A `simple repetitive` loop can be executed a fixed number of times. A `conditional` loop is executed when a true or false condition is met.

program. A list of instructions to a computer.

string. In a REXX program, a group of characters inside single or double quotes.

subroutine. A sequenced group of statements that can be called from more than one place in a main program.

syntax. The way in which words are put together to form phrases or sentences.

value. What a variable is assigned.

variable. A particular piece of data, used by a program in a particular way, but whose value may vary.

whole number. A number with a zero (or no) decimal part.

Index

International Business
Machines Corporation
P.O. Box 6
Endicott, New York 13760

File No. S370/4300-40
Printed in U.S.A.

SC24-5357-00

VM/IS Writing Simple Programs with REXX
Order No. SC24-5357-00

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___ YES ___ NO

Please print your name, company name, and address:

_____

_____

_____

_____

IBM Branch Office serving you: _____

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

24-5357-00

ld and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

ld and tape Please Do Not Staple Fold and tape

IBM®

VM/IS Writing Simple Programs with REXX
Order No. SC24-5357-00

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___ YES ___ NO

Please print your name, company name, and address:

_____

_____

_____

_____

IBM Branch Office serving you: _____

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

C24-5357-00

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape Please Do Not Staple Fold and tape

IBM®

International Business
Machines Corporation
P.O. Box 6
Endicott, New York 13760

File No. S370/4300-40
Printed in U.S.A.

SC24-5357-00

SC24-5357-00