Rexx Programmer's Reference, 2nd Edition

Howard Fosdick

Rexx Programmer's Reference, 2nd Edition

Published by Rexx Language Association
8525 Pinefield Road,
Apex, NC 27523-9601, USA

The Rexx Language Association is a Non-Profit Corporation registered in North Carolina, USA, incorporated in 1998. Visit RexxLA.org.

ISBN 978-9-40374-552-7

© 2025 2nd Edition by H. Fosdick

This is a fully revised 2nd edition of the book with the same title published in 2005 by Wiley Publishing, Inc.

This updated and revised reprint is published under authority of clause 18 of the publishing contract dated July 8, 2004 between Wiley Publishing Inc., and Fosdick Consulting Inc., and subsequent letter dated December 14, 2015, affirming right to reprint, from Fosdick Consulting Inc. to Wiley Publishing Inc., as pursuant to that contract.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER AND AUTHOR ARE NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT WAS READ.

TRADEMARKS: All trademarks are the property of their respective owners. The Rexx Language Association is not associated with any vendor mentioned in this book.

About the Author

Howard Fosdick is an independent IT consultant who supports databases and operating systems. He's written seven books and over 500 articles and frequently speaks at conferences. He co-founded the International DB2 Users Group, the Midwest Database Users Group, and CAMP IT, and invented concepts like hype curves and open consulting. While he's coded in many programming languages, Rexx remains his favorite.

RexxInfo.org for everything Rexx.

Download free guides --

How to Build a Free Computer -- from Cast-offs

How to Fix Computer Hardware

Privacy in a Digital World

… more ...

from **RexxInfo.org/guides**

Credits for 1st Edition

Senior Acquisitions Editor
Debra Williams Cauley

Development Editor
Eileen Bien Calabro

Production Editor
Felicia Robinson

Technical Reviewer
Mark Hessling

Copy Editor
Publication Services

Editorial Manager
Mary Beth Wakefield

Vice President & Executive Group Publisher
Richard Swadley

Vice President and Publisher
Joseph B. Wikert

Project Coordinator
Erin Smith

Graphics and Production Specialists
Jonelle Burns
Carrie Foster
Lauren Goddard
Denny Hager
Joyce Haughey
Jennifer Heleine

Quality Control Technicians
John Greenough
Leeann Harney
Jessica Kramer
Carl William
Pierce

Proofreading and Indexing
TECHBOOKS Production Services

Credits for 2nd Edition

Content, Layout, Typesetting, Cover Designs: Howard Fosdick

Technical Reviewers: Many. Please see the Acknowledgments.

To Pippy, Vandy, Josie, Phoebe Jane, and Schnappsi.

Foreword

Rexx is a very underrated programming language; elegant in design, simple syntax, easy to learn, use and maintain, yet as powerful as any other scripting language available today.

In 1979, Mike Cowlishaw, IBM fellow, designed a "human-centric" programming language, Rexx. Cowlishaw's premise was that the programmer should not have to tell the interpreter what the language syntax was in each program they wrote; that was the job of the interpreter. So unlike most other programming languages, Rexx does not suffer from superfluous, meaningless punctuation characters throughout the code.

Since the release of Rexx outside of IBM, Rexx has been ported to virtually all operating systems and was formally standardised with the publishing of the ANSI Standard for Rexx in 1996. In late 2004, IBM transferred their implementation of Object REXX to the Rexx Language Association under an Open Source license. This event signalled a new era in the history of Rexx.

This book provides a comprehensive reference and programming guide to the Rexx programming language. It shows how to use the most popular implementations of Rexx and Rexx external function packages and is suited to both the programmer learning Rexx for the first time as well as the seasoned Rexx developer requiring a single, comprehensive reference manual.

Rexx has had a major influence on my life for the past 20 years since I wrote my first XEDIT macro in Rexx. In the last 10 years I have maintained the Regina Rexx interpreter, ably assisted by Florian Große-Coosmann, and in my remaining spare time have developed several Rexx external function packages (and my XEDIT-like text editor, THE). However, like many developers of open source products, I have never quite documented the products as completely as they deserve.

This is the book I would have liked to write if I had had the time. I'm glad Howard had the time!

Mark Hessling

Author of Rexx/SQL, Rexx/gd, Rexx/DW,
Rexx/CURL, Rexx/Curses, Rexx/Wrapper,
Rexx/Trans,
The Hessling Editor (THE), Maintainer of Regina,
Rexx/Tk, PDCurses, `http://www.rexx.org/`

Foreword to the 2nd Edition

The release of the 2nd edition of *The Rexx Programmer's Reference* is a testament to the passion that Howard and the majority of users have for Rexx and the longevity of the Rexx programming language itself.

As software technologies change over the years, Rexx easily incorporates those new technologies due to its exceptional design. This flexibility to change does not come at the expense of breakage of existing code as is so common in most other programming languages. As a result, new program development using Rexx is as viable now as it was when Rexx first appeared, and existing code still runs unchanged as reliably now as when it was first developed.

This edition includes new content and changes that reflect the current state of Rexx.

I hope I'm around for the 3rd edition of *The Rexx Programmer's Reference!*

Mark Hessling

Author of Rexx/SQL, Rexx/gd, Rexx/DW, Rexx/CURL, Rexx/Curses, Rexx/Wrapper, Rexx/Trans, Rexx/JSON, The Hessling Editor (THE), and Maintainer of Regina, Rexx/Tk, and PDCurses.

www.Rexx.org

Acknowledgments for the 1st Edition

Special thanks are due to Mark Hessling, who writes and maintains Regina Rexx and a wide variety of open source Rexx tools and interfaces. As the technical reviewer for this book, Mark was an invaluable source of recommendations for improvement as well as (oops!) corrections. His expertise and helpfulness were critical to improving this book.

Special gratitude is also due to the inventor of Rexx, Michael Cowlishaw. His advice and feedback were very much appreciated.

In the process of developing this book, I wrote inquiries to many people without any prior introduction. Each and every one of them responded helpfully. It was a great pleasure to meet people with such an interest in Rexx, who so kindly answered questions and who greatly improved this book with their suggestions.

I would like to give heartfelt thanks to María Teresa Alonso y Albado, W. David Ashley, Gil Barmwater, Dr. Dennis Beckley, Alex Brodsky, Frank Clarke, Steve Coalbran, Ian Collier, Les Cottrell, Michael Cowlishaw, Chip Davis, Prof. Rony Flatscher, Jeff Glatt, Etienne Gloux, Bernard Golden, Bob Hamilton, Henri Henault , Stéphane Henault, Mark Hessling, Jack Hicks, IBM Corporation, René Vincent Jansen, Jaxo Inc., Kåre Johansson, Kilowatt Software, Les Koehler, Laboratorios Bagó S.A., Joseph A. Latone, Henri LeFebure, Michael Lueck, Antoni Levato, Dave Martin, Rob McNair, Patrick TJ McPhee, Dr. Laura Murray, Walter u. Christel Pachl, Lee Peedin, Priscilla Polk, the Rexx Language Association, Pierre G. Richard, Peggy Robinson, Morris Rosenbaum, Dr. Elizabeth Rovelli, David Ruggles, Roger E. Sanders, Thomas Schneider, Theresa Stewart, UniForum Chicago, Vasilis Vlachoudis, Stan Wakefield, Keith Watts, Dr. Sandra Wittstein, and Claudio Zomparelli.

Beyond those who provided technical advice and input for this book, I wish to thank my editors at John Wiley and Sons, Inc. Eileen Bien Calabro greatly improved the readability of this book through her writing recommendations. Debra Williams Cauley provided insightful perspective and guidance on the preparation and organization of the book. Finally, I thank Richard Swadley. I appreciate his confidence and hope this book fulfills its promise both in the quality of its material and in its sales and distribution.

Special thank you to the following developers for permission to reprint or refer to their code (most of these items fall under various open source licenses):

W. David Ashley—IBM Corporation, project leader of the Mod_Rexx project for scripts appearing in the chapter on Apache programming with Mod_Rexx

Les Cottrell and the Stanford Linear Accelerator Laboratory—Authors of Rexx/CGI library for a script illustrating their Rexx/CGI library

Henri Henault & Sons—Authors of the Internet/REXX HHNS WorkBench for a script and screen shot illustrating the Internet/REXX HHNS WorkBench.

Mark Hessling—Developer/maintainer of Regina Rexx and many open source Rexx tools
for material on Rexx/gd and the reference tables of Rexx/Tk functions

Acknowledgments

Patrick TJ McPhee—Developer of RexxXML for the example program appearing in the chapter on RexxXML

Pierre G. Richard, Joseph A. Latone, and Jaxo Inc.—Developers of Rexx for Palm OS for example scripts appearing in the chapter on Rexx for Palm OS

Acknowledgments for the 2nd Edition

Thank you to René Jansen, President of the Rexx Language Association, for his advice and guidance in putting together this updated edition.

Thank you also to all the members of the RexxLA and others who helped by giving advice or reviewing this update: Grant Ward Able, Bill Backs, Janet Backs, Volker Bandke, Gil Barmwater, Michael Beer, Wayne V. Bickerdike, Josep Maria Blasco, Frank Clarke, Michael Cowlishaw, Chip Davis, Marcel Dür, Lionel B. Dyck, Tony Dycks, Dr Rony Flatscher, Mark L. Gaubatz, Eva Gerger, Sam Golob, Thomas Grundmann-Kahr, Rob Hamilton, Jeffrey Hennick, Mark Hessling, Peter Jacob, Willy Jensen, Dave Jones, Per Olov Jonsson, Mark McDonald, Rick McGuire, Shmuel (Seymour J.) Metz, Jeremy Nicoll, Walter Pachl, Ross Patterson, Lee Peedin, Jochem Peelen, Priscilla Polk, Julian Reindorf, Marc Remes, Pierre G. Richard, Larry Schacher, Martin Scheffler, Bruce Skelly, David Spiegel, Hobart Spitz, Bob Stark, Erich Steinböck, Theresa Stewart, J. Leslie Turriff, Vasilis Vlachoudis, James Warren, and Jon Wolfers.

Thank you to those who contributed code for this new edition: Frank Clarke, Marcel Dür, Lionel B. Dyck, Tony Dycks, Dr Rony Flatscher, Eva Gerger, Thomas Grundmann-Kahr, Mark Hessling, IBM Corporation, Willy Jensen, Julian Reindorf, Pierre G Richard, and Vasilis Vlachoudis.

Also thank you to Marcel Dür, Eva Gerger, Thomas Grundmann-Kahr, and Julian Reindorf for the very useful theses they wrote on running Rexx on Android while at the Vienna University of Economics and Business. Congratulations to Dr Rony Flatscher for his success nourishing such a talented crop of budding developers and software engineers.

Finally, thank you to Mark Hessling for his help and advice on both editions of this work, and for writing the Forewords. Mark is the developer behind Regina Rexx and many of the excellent free tools that add key functionality to Rexx. Several chapters in this book cover products he develops and supports.

And, of course, special thanks to the inventor of Rexx and NetRexx, Michael Cowlishaw.

An Open Source, Not-for-Profit Book

The author has directed that all monies made by this book be donated directly to the Rexx Language Association, a non-profit entity incorporated in North Carolina, USA.

This book is published under the *Creative Commons license BY-ND*. You can freely use and distribute this work, but with these restrictions:

- ❑ BY -- You can not change the authorship
- ❑ ND -- You must distribute this work in its entirety without altering its contents

Thank you for respecting my effort in writing this book by your cooperation.

Contents

Table of Contents

Introduction

Of all the free scripting languages, why should you learn Rexx? Rexx is unique in that it combines *power* with *ease of use*. Long the dominant scripting language on mainframes, it is definitely a "power" language, yet it is also so easy to use that its popularity has expanded to every conceivable platform. Today Rexx developers use the language on Windows, Linux, Unix, BSD, Macs, mainframes and many dozens of other systems . . . and, there are many free and open source Rexx interpreters available.

Here's the Rexx story in a nutshell:

- ❑ Rexx runs on *every* platform under nearly every operating system.

  So, your skills apply anywhere . . . and your code runs everywhere.

- ❑ Rexx enjoys a strong international standard that applies to every Rexx interpreter . . .

  from cell phones and handhelds to PCs to servers to mainframes.

- ❑ Rexx is as easy as JavaScript or PHP, yet about as powerful as Java or Perl.

- ❑ Rexx's large user community means:

  - ❑ Many free interpreters optimized for different needs and environments
  - ❑ A vast array of free interfaces and tools
  - ❑ Good support

- ❑ Rexx comes in object-oriented versions as well as versions that are Java-compatible (one even runs on the Java virtual machine!)

You may be wondering why ease of use is so important in a programming language. A truly easy language is easy to use, learn, remember, and maintain. The benefits to beginners are obvious. With Rexx, you can start coding almost immediately. There are no syntax tricks or language details to memorize before you begin. And, since Rexx is also a power language, you can rest assured that you won't run out of capability as you learn and grow with it. Read the first few chapters in this book, and you'll be scripting right away. Continue reading, and you'll mature into advanced scripting before you finish.

If you are an experienced developer, Rexx offers more subtle benefits. You will be more productive, of course, as you free yourself from the shackles of syntax-driven code. More important is this: **simplicity yields reliability**. Your error rate will decline, and you'll develop more reliable programs. This benefit is greatest for the largest systems and the most complicated scripts. Your scripts will also live longer because others will be able to understand, maintain, and enhance them. Your clever scriptlets and application masterpieces won't die of neglect when someone else takes over and maintains your code.

Few easy languages are also *powerful*. Now, how does Rexx do *that*?

Rexx surrounds its small instruction set with an extensive function library. Scripts leverage operating system commands, external interfaces, programs, and functions. Rexx is a "glue" language that ties it all together. Yet the language has few rules. Syntax is simple, minimal, flexible. Rexx doesn't care about uppercase or lowercase or formatting or spacing. Rexx scripts don't use special symbols and contain no punctuation.

Power does not require coding complexity!

If you've worked in the shell languages, you'll breathe a sigh of relief that you've found a powerful language in which you can program now and then without trying to recall arcane language rules. If you've struggled with the syntax of languages such as Bash, Korn, Awk, Perl, C++, or the C-shell, you'll enjoy focusing on your programming problem instead of linguistic peculiarities. And if you've ever had to maintain someone else's code written in one of those languages, well . . . you might *really* be thankful for Rexx!

This book contains everything you need to know to get started with Rexx. How to freely download and install an appropriate interpreter. How to program in standard "classic" Rexx and object-oriented Rexx. How to program Windows, Linux, Unix, Macs, Androids, and mainframes. How to program in the Java environment with object-oriented Rexx, or a Rexx-based language called NetRexx. How to script operating system commands, control Web servers and databases and graphical user interfaces (GUIs) and Extensible Markup Language (XML) and Apache and . . . you name it.

Everything you need is in this one book. It's a Rexx encyclopedia. It teaches standard Rexx so that your skills apply to any platform—from cell phones to desktops and laptops to midrange servers to mainframes. Yet it goes beyond the basics to cover interface programming and advanced techniques. The book starts out easy, based on coding examples throughout to make learning fast, simple, and fun. But it's comprehensive enough to cover advanced scripting as well. You can freely download all the Rexx interpreters, tools, and interfaces it covers. Welcome to the world of free Rexx !

Who This Book Is For

This book is for anyone who wants to learn Rexx, or who already works with Rexx and wants to expand his or her knowledge of the language, its versions, interfaces, and tools. How you use this book depends on your previous programming or scripting knowledge and experience:

- ❑ If you are a complete beginner, you'll find Rexx easy to learn. This book will tell you everything you need to know. It's a progressive tutorial based on coding examples, so you won't get lost. You'll be able to handle almost any programming problem by the end of the book.
- ❑ If you are an experienced programmer, you can quickly learn Rexx by reading the tutorial chapters. You'll be programming immediately. As the book progresses into tutorials on interfaces to databases, Web servers, GUIs, and the like, you'll learn to program Rexx in the context of the larger environment.

- If you are a systems administrator or support person, you'll learn a language that applies to a wide variety of situations and can be a great tool. This book covers the interfaces, tools, and varieties and implementations of Rexx you'll need to know. It doesn't stop with the basics. It explores the advanced features you'll want to use.
- If you already script Rexx, you will be able to expand your knowledge. You'll learn about free Rexx interfaces and tools with which you may not be familiar. You'll learn Rexx programming in new environments, such as object-oriented scripting, scripting in the Java environment… and even how to program Rexx in mainframe emulation on your personal computer. You'll find this complete reference the only Rexx book you'll ever need.

What This Book Covers

This book teaches standard Rexx, quickly and simply. It emphasizes coding examples. It teaches you what you need to know to work with Rexx on any platform. You'll know a language that runs everywhere and applies to almost any programming problem.

Beyond the Rexx language proper, this book covers all the major interfaces into Web servers, SQL databases, GUIs, XML, JSON, graphics processing, and the like. It describes many of the free tools that are available to make scripting with Rexx easier and more productive.

The book covers a number of free Rexx interpreters. Most meet the international standards for Rexx, yet each adds its own special features and extensions. The book tells where to download each interpreter, shows how to install it, and demonstrates how to leverage its particular strengths.

All the Rexx interpreters, tools, and interfaces this book covers are free or open source. One exception is IBM mainframe Rexx, which comes bundled with IBM's operating systems.

Ultimately, this book covers not only Rexx scripting, but the whole world of Rexx programming across all environments and interfaces, and with all Rexx interpreters.

How This Book Is Structured

Take a quick look at the table of contents, and you will see that this book is broken down into three broad sections:

- The book begins with a progressive tutorial that covers the basics of the Rexx language. These eventually lead into more advanced scripting topics, such as how to write portable code and using optimal coding style. The last chapters of this section (Chapters 15 through 18) cover the most common Rexx interfaces and tools. These introduce and demonstrate how to code scripts that interface to operating systems, SQL databases, Web servers, GUIs, XML, and other tools.

- The second section of the book describes the different Rexx interpreters and the unique advantages of each. These chapters apply Rexx to different environments and include tutorials on object-oriented Rexx, cell phone scripting, programming in Java environments, and many other topics. All include complete example programs.
- Finally, the book contains a comprehensive reference section in its appendices. You won't need any other book to script Rexx.

What You Need to Use This Book

You need nothing beyond this book. All its examples were all run using freely downloadable Rexx interpreters, tools, and interfaces. The chapters tell you where to download any interpreters, tools, and interfaces the book demonstrates, as well as how to set up and install them. Most of the examples in this book were run under Windows and/or Linux, but you can work with this book with Rexx running any operating system you like.

Download the Example Code

We recommend that you download the source code for all of the examples in this book from `www.RexxInfo.org`. All our downloads are free and require no registration or email address.

Conventions

We've used a number of conventions throughout the book.

> **Boxes like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.**

Concerning styles in the text:

- We *italicize* important words when we introduce them.
- We show keyboard strokes like this: Ctrl-A.
- We show filenames, URLs, variable names, and code within the text like this: `my_file.txt`.
- We present code in two different ways:

```
In code examples we highlight new and important code with a gray background.
```

```
The gray highlighting is not used for code that's less important in the present
context, or that has been shown before.
```

The Rexx language is not case-sensitive, so its instructions and functions can be encoded in uppercase, lowercase, or mixed case. For example, the `wordlength` function can be encoded as `wordlength`, `WordLength`, or `WORDLENGTH`. This book uses capitalization typical to the platforms for which its sample scripts were written, but you can use any case you prefer.

Due to the typesetting software used in preparing this book, in a few places in the text, quotation marks might appear as forward-leaning or backward-leaning. Please consider the leftmost examples as equivalent to the vertical quotation marks that Rexx requires:

`Hello` ‘Hello’ `Hello’ “Hello” Correct: 'Hello' "Hello"

Only vertical single or double quotation marks will run correctly when coded in a Rexx program.

Reporting Errors

Thousands of programmers used the first edition of this book and reported no errors beyond a few typos. Our goal is to match this level of quality in this 2nd edition. This new edition has been reviewed by many experts from the Rexx Language Association.

However, updating for this new edition required working off the 1st edition's PDF files -- an error-prone process, especially when working with code in text boxes and frequent font changes. So, try as we might to catch them, errors introduced by typesetting could slip through.

If you find an error, please contact the author via his website at `www.RexxInfo.org` and report it. We'll correct the digital edition immediately, and the print edition as soon as the next printing. Thank you for your help in making this the best possible resource for Rexx programmers.

Online Resources

The website `www.RexxInfo.org` is a one-stop shop for all things Rexx. It offers free downloads for all Rexx interpreters and tools, articles, sample code, information, links, and other resources. Its reference section offers quick online lookup for all Rexx language instructions, functions, classes, and methods, and also a full set of free manuals for all Rexx products, including all IBM manuals.

Appendix A on "Resources" describes many other good sources of Rexx information.

What’s New in the 2nd Edition?

This is a fully revised 2nd edition of a book originally published in 2005.

Completely new content in this edition includes: the Introduction; chapters 1, 19, 22, 24, 25, and 27; and, appendices A, I, L, N, O, P, Q, R, S, T, U, V, W, X, Y, and Z. Of course, all other content has been updated as appropriate.

Introduction

Much new material has been added in this edition including:

- ❑ Java integration
- ❑ Using JSON
- ❑ Array I/O
- ❑ Advanced parsing
- ❑ Android programming
- ❑ Rexx on single board computers
- ❑ SQLite programming
- ❑ Equivalence charts for Rexx <--> Bash
- ❑ Equivalence charts for Rexx <--> Python
- ❑ Sample job interview questions

Readers have also asked for more coverage of mainframe Rexx, so we've added:

- ❑ Using EXECIO
- ❑ How to write ISPF edit macros
- ❑ How to run Rexx in batch
- ❑ Rexx on personal computers that emulate mainframes
- ❑ Equivalence charts for IBM mainframe REXX <--> ANSI-1996 standard Rexx
- ❑ Equivalence charts for Rexx <--> Clist

Much new content has been provided in new appendices, rather than chapters, due to typesetting constraints.

We've retained coverage of a few topics that are less important than previously. This includes the chapters on the out-of-support interpreters Reginald, r4, and roo!, and the brief discussion of Windows CE. We've kept this content because some readers requested it. It's clearly marked in the text. If it is not of interest to you, please just skip it.

Among Rexx's great benefits are its strong language standards and remarkable stability. Thus it hasn't been necessary to modify or update the book's code examples. The proof is in the product -- you can rely on Rexx to run mature code without difficulty or disruption.

Note that this edition does not always reflect the latest names of a few rebranded products. For example, it may refer to the Db2 database under its older name of "DB2".

And remember that website addresses do change. If one this book suggests becomes obsolete, just use your search engine to find the material you seek.

Part I

1

Introduction to Scripting and Rexx

Overview

Before learning the Rexx language, you need to consider the larger picture. What are scripting languages? When and why are they used? What are Rexx's unique strengths as a scripting language, and what kinds of programming problems does it address? Are there any situations where Rexx would *not* be the best language choice?

This chapter places Rexx within the larger context of programming technologies. The goal is to give you the background you need to understand how you can use Rexx to solve the programming problems you face.

Following this background, the chapter shows you how to download and install the most popular free Rexx interpreter on your Windows, Linux, Unix, BSD, or Apple computer. Called *Regina*, this open-source interpreter provides a basis for your experiments with Rexx as you progress in the language tutorial of subsequent chapters.

Note that you can use *any* standard Rexx interpreter to learn Rexx. So, if you have some other Rexx interpreter available, you are welcome to use it. We show how to download and install Regina for readers who do not already have a Rexx interpreter installed, or for those who would like to install an open-source Rexx on their PC.

Why Scripting?

Rexx is a *scripting language*. What's that? While most developers would claim to "know one when they see it," a precise definition is elusive. Scripting is not a crisply defined discipline but rather a directional trend in software development. Scripting languages tend to be:

- *High level* — Each line of code in a script produces more executable instructions — it does more — than an equivalent line encoded in a lower-level or "traditional" language.

- *Glue languages* — Scripting languages stitch different components together — operating system commands, graphical user interface (GUI) widgets, objects, functions, or service routines. Some call scripting languages *glue languages*. They leverage existing code for higher productivity.
- *Interpreted* — Scripting languages do not translate or *compile* source code into the computer's machine code prior to execution. No compile step means quicker program development.
- *Interactive debugging* — Interpreted languages integrate interactive debugging. This gives developers quick feedback about errors and makes them more productive.
- *Variable management* — Higher-level scripting languages automatically manage variables. Rexx programmers do not have to define or "declare" variables prior to use, nor do they need to assign maximum lengths for character strings or worry about the maximum number of elements tables will hold. The scripting language handles all these programming details.
- *Typeless variables* — Powerful scripting languages like Rexx even relieve the programmer of the burden of declaring data types, defining the kind of data that variables contain. Rexx understands data by usage. It automatically converts data as necessary to perform arithmetic operations or comparisons. Much of the housekeeping work programmers perform in traditional programming languages is automated. This shifts the burden of programming from the developer to the machine.

Figure 1-1 contrasts scripting languages and more traditional programming languages.

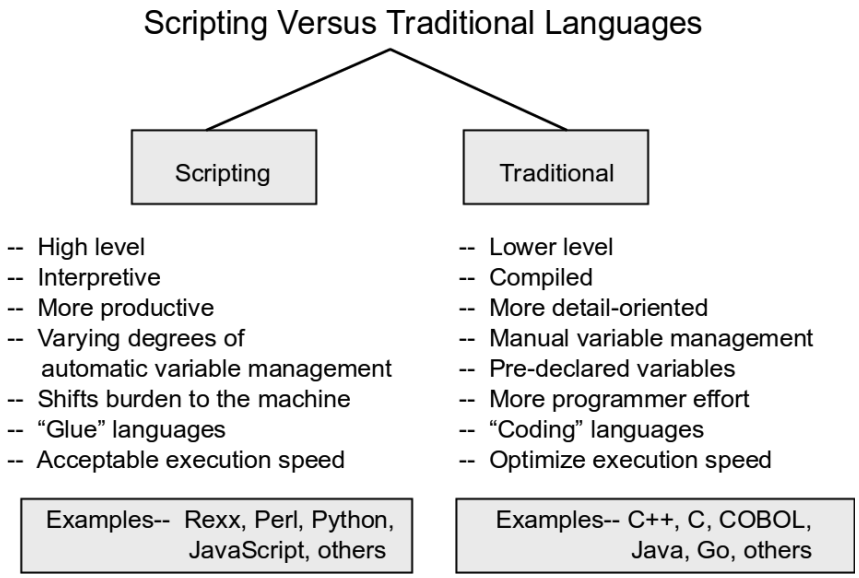

Figure 1-1

On the downside, scripting requires greater machine resources than hand-coded programs in traditional, compiled languages. But in an era where machine resources are less expensive than ever and continue to decline in price, trading off expensive developer time for cheaper hardware makes sense.

Hardware performance increases geometrically, while the performance differential between scripting and compiled languages remains constant.

Here's how hardware addresses scripting performance. The original IBM PC ran an 8088 processor at 4.77 MHz. It executes less than a hundred *clauses* or statements of a Rexx script every second. Current personal computers execute millions of Rexx clauses per second.

Just for fun, this table shows how much faster a standard Rexx benchmark script runs on typical PCs at various times. Later in this chapter, we'll show you how to benchmark your own computer against the numbers in this table:

Year	Make	Processor	Speed	Memory	Operating System	Rexx	Clauses per Second
1982	IBM PC	8088	4.77 Mhz	320 KB	DOS 6.2	Mansfield	70
	Zenith	8088-2	8 Mhz	640 KB	DOS 6.2	Mansfield	95
1988	Clone	386/DX	25 Mhz	2 MB	DOS 6.2	BRexx	3,600
1993	Clone	486/SX	25 Mhz	8 MB	Windows 3.1	BRexx	6,000
1998	Gateway	Pentium II	266 Mhz	512 MB	Red Hat 8	Regina	180,000
2005	Clone	Celeron	2.6 Ghz	1 GB	Windows XP	Regina	1,100,000
2010	Lenovo	Core2Duo	2*3.0Ghz	4 GB	Windows 7	Regina	4,545,455
2014	Dell	i5-3350p	4*3.1Ghz	8 GB	Windows 8.1	ooRexx	3,030,303
2017	HP 8570p	i7-3520m	8*2.9Ghz	8 GB	Win 7 Ent.	ooRexx	7,363,893
2019	Dell Vostro	i7-3770	8*3.4Ghz	8 GB	Linux Mint 19	BRexx	8,498,672
2022	Gigabyte	amd3900x	3.8 Ghz	32 GB	Win 10 Pro	Regina	9,250,694
2023	Acer	amd6800u	2.7 Ghz	16 GB	Windows 11	ooRexx	13,636,364
2023	Apple	M1	3.2 Ghz	16 GB	Ventura 13.3.1	Regina	22,233,751

Source- author's hands-on tests and contributions by members of the Rexx Language Association.

The bottom line is that the program that consumes over an hour on a decades-old 8088 runs in a split second on a modern desktop. While the table ignores subtle factors that affect performance, the trend is clear. For most programming projects, trading machine cycles for labor costs makes sense. Why not use a more productive tool that shifts the burden to the machine?

Labor-saving benefits extend beyond program development to maintenance and enhancement. Experts like T. Capers Jones estimate that up to 75 percent of IT labor costs are devoted to program maintenance. An easy-to-read, easy-to-maintain scripting language like Rexx saves a great deal of money.

Sometimes, you'll hear the claim that scripting languages don't support the development of large, robust, "production-grade" applications. Years ago, scripting languages were primitive and this charge rang true. But no longer. IT organizations routinely develop and run large applications written in Rexx and other scripting languages. The author has run across many large production business applications consisting of tens of thousands of lines of code. You can run an entire enterprise on scripts.

Why Rexx?

The distinguishing feature of Rexx is that it combines *ease of use* with *power*. Its goal is to make scripting as easy, fast, reliable, and error-free as possible. Many programming languages are designed for compatibility with older languages, the personal tastes of their inventors, the convenience of compiler-writers, or machine optimization. Rexx ignores extraneous objectives. It was designed from day one to be powerful yet easy to use.

One person invented Rexx and guided its development: Michael Cowlishaw of IBM's UK laboratories. Cowlishaw gave the language the coherent vision and guiding hand that ambitious software projects require to succeed. Anticipating how the Internet community would cooperate years later, he posted Rexx on the internet of its day, IBM's VNET, a network of tens of thousands of users. Cowlishaw solicited and responded to thousands of emailed suggestions and recommendations on how people actually used early Rexx. The feedback enabled Cowlishaw to adapt Rexx to typical human behavior, making Rexx a truly easy-to-use language.

Ease of use is critical — even to experienced developers — because it leads to these benefits:

- *Low error rate* — An easy-to-use language results in fewer bugs per program. Languages that rely on arcane syntax, special characters and symbols, and default variables cause more errors.
- *Reliability* — Programs are more reliable due to the lower error rate.
- *Longer-lived code* — Maintenance costs dictate the usable life span of code. Rexx scripts are much easier to maintain than scripts written in languages that rely on special characters and complex syntax.
- *Reduced cost* — Fast program development, coupled with a low error rate and high reliability, lead to reduced costs. Ease of maintenance is critical because up to three-quarters of IT professionals engage in maintenance activities. Code written by others is easier to understand and maintain if it is written in Rexx instead of syntax-driven languages like the Linux shell languages or Perl or Awk or C++. This reduces labor costs.
- *Higher productivity* — Developer productivity soars when the language is easy to work with. Scripting in Rexx is more productive than coding in either lower-level compiled languages or syntax-based shell languages.
- *Quicker testing* — Interpretive scripting languages lend themselves to interactive testing. Programmers get quick feedback and can easily trace program execution. Combined with the low error rate of an easy-to-use language, this means that less test time is required.
- *Easy to learn* — An easy-to-use language is easier to learn. If you have programmed in *any* other programming or scripting language, you can pick up Rexx very quickly.

- *Easy to remember* — If you write only the occasional program, you'll find Rexx easy to remember. Languages with special characters and quirky syntax force you to review their rules if you only script now and then.
- *Transfer skills* — Since Rexx is easy to work with, developers find it easy to adapt to platform differences or the requirements of different interfaces. Rexx has a strong platform-independent standard. As well, many Rexx interfaces and tools are themselves cross-platform products.

Power and Flexibility

That Rexx is easy to learn and use does *not* mean that it has limited features or is some sort of "beginner's language." Rexx competes, feature for feature, with any of the other major scripting languages. If it didn't, it certainly would not be the primary scripting language for mainframes, nor would it have attained the widespread use it enjoys today on so many other platforms. Nor would there be thousands of Rexx users distributed around the world.

Ease of use and power traditionally force language trade-offs. It is easy to get one without the other, but difficult to achieve both. Rexx is specifically designed to combine the two. It achieves this goal through these principles:

- *Simple syntax* — Some very powerful languages rely extensively on special symbols, nonobvious default behaviors, default variables, and other programming shortcuts. But there is no rule that power can only be achieved in this manner. Rexx eschews complex "syntax programming" and encourages simpler, more readable programming based on English-language keyword instructions and functions.
- *Small command set, with functions providing the power* — Rexx has a small core of only two dozen instructions. This simplicity is surrounded by the power of some 70 built-in functions. A well-defined, standard interface permits Rexx to call upon external function libraries. This allows you to extend the language yourself, and it means that many open-source extensions or libraries of routines are freely available. Rexx scripts also wield the full power of the operating system because they easily issue operating system commands.
- *Free-form language* — Rexx is not case-sensitive. It is a *free-form language* and is about as forgiving concerning placement of its source text as a programming language can be. This permits programmers to self-describe programs by techniques such as indentation, readable comments, case variations, and the like. Rexx relieves programmers from concern about syntax and placement, and lets them concentrate on the programming problem they face.
- *Consistent, reliable behavior* — Rexx behaves "as one would assume" at every opportunity. Its early user community provided feedback to one "master developer" who altered the language to conform to typical human behavior. As the inventor states in his book defining Rexx: *"The language user is usually right."* Rexx was designed to encourage good programming practice and then enhanced by user feedback to conform to human expectations.
- *Modularity and structured programming* — Rexx encourages and supports modularity and structured programming. Breaking up large programming problems into discrete pieces and restricting program flow to a small set of language constructs contributes greatly to ease of use and a low error rate when developing large applications. These principles yield simplicity without compromising power. (Chapters 3 and 8 explore them.)

- *Fewer rules* — Put the preceding points together, and you'll conclude that Rexx has fewer rules than many programming languages. Developers concentrate on their programming problem, not on language trivia.
- *Standardization* — While there are many free Rexx interpreters, nearly all adhere to the Rexx standards. This makes your scripts portable and your skills transferable. A standardized language is easier to use than one with numerous variants. Rexx has two strong, nearly identical standards. One is defined in the book *The Rexx Language*, or *TRL-2*, by Michael Cowlishaw (Prentice-Hall, 1990, second edition). The other is the 1996 standard from the American National Standards Institute, commonly referred to as *ANSI-1996*. The ANSI-1996 specification is a true superset of TRL-2 that adds just a few important language features.

Free and Open Source

As we'll describe below, you can select from a good number of different Rexx interpreters. They come in both open source and commercial versions. If you prefer a free and open source (or FOSS) product, there is always a Rexx interpreter available for your platform.

The vast majority of Rexx tools are free and open source, too. Hundreds of them cover every conceivable need, from graphical user interfaces to databases, from telecommunications to development environments, to almost every other category you can imagine.

Universality

Rexx is a *universal language*. It runs on every platform, from cell phones and handhelds, to laptops and desktops, to servers of all kinds, all the way up to the largest mainframes and supercomputers.

Here are the many platforms on which free Rexx interpreters run:

- All versions of Windows, Linux, Unix, BSD, Apple operating systems and macOS, and all IBM operating systems.
- Cell phones and tablets running Android and other various kinds of handhelds
- Single board computers and embedded systems applications
- IBM servers (i-series, pSeries, POWER/PowerPC, Linux for Z, OpenEdition/Unix System Services), Amiga-derived systems (AmigaOS 4 or AOS4, MorphOS, AROS, aeROS or AEROS), DOS-family systems (MS-DOS, PC-DOS, FreeDOS, all others), OS/2-derived systems (eCS and ArcaOS), plus OpenVMS, QNX, and others
- Many other lesser-used platforms too numerous to list

The benefits of a universal language are several. Among them:

- Your skills apply to any platform
- Your scripts run on any platform

Here's an example. A site that downsizes its mainframes to Linux servers could install free Rexx on the Linux machines. Rexx becomes the vehicle to transfer personnel skills, while providing a base for migrating scripts.

As another example, an organization migrating from procedural to object-oriented programming (OOP) could use free Rexx as its cross-platform entry point into OOP. Standard, procedural Rexx is a true subset of object-oriented Rexx.

A final example: a company runs a data center with mainframes and Unix servers, uses Windows on the desktop, and programs Android tablets for field agents. Rexx runs on all these platforms, making developers immediately productive across the whole range of company equipment. Rexx enables a mainframer to program a handheld, or Windows developer to script under Unix.

A standardized scripting language that is freely available across a wide range of systems yields unparalleled skills applicability and code portability.

Typical Rexx Applications

Rexx is a general-purpose language. It is designed to handle diverse programming needs. Its power gives it the flexibility to address almost any kind of programming problem. Here are examples.

- *As a "glue" language* — Rexx has long been used as a high-productivity "glue" language for stitching together existing commands, programs, and components. Rexx offers a higher-level interface to underlying system commands and facilities. It leverages services, functions, objects, widgets, programs, and controls.

- *Automating repetitive tasks* — Rexx scripts automate repetitive tasks. You can quickly put together little scripts to tailor the environment or make your job easier. Rexx makes it easy to issue commands to the operating system (or other environments or programs) and react to their return codes and outputs.

- *Systems administration* — Rexx is a high-level, easy-to-read, and easy-to-maintain way to script system administration tasks. By its nature, systems administration can be complex. Automating it with an easily understood language raises system administration to a higher, more abstract, and more manageable level. If you ever have to enhance or maintain systems administration scripts, you'll be thankful if they're written in Rexx instead of some of the alternatives!

- *Extending the operating system* — You typically run Rexx scripts simply by typing their name at the operating system's command prompt. In writing scripts, you create new operating system "commands" that extend or customize the operating system or programming environment.

- *Application interfaces* — Rexx scripts can create flexible user interfaces to applications programmed in lower-level or compiled languages.
- *Portable applications* — Rexx's standardization and extensive cross-platform support make it a good choice for applications that must be ported across a range of systems. Its readability and ease of maintenance make it easy to implement whatever cross-platform enhancements may be desired. For example, while Rexx is the same across platforms, interfaces often vary. Standardizing the scripting language isolates changes to the interfaces.
- *Prototyping and exploratory programming* — Since Rexx supports quick development, it is ideal for developing prototypes, whether those prototypes are throw-aways or revisable. Rexx is also especially suitable for exploratory programming or other development projects apt to require major revision.
- *Personal programming* — An easy-to-use scripting language offers the simplicity and the speedy development essential to personal programming. PCs and handheld devices often require personal programming.
- *Text processing* — Rexx provides outstanding text processing. It's a good choice for text processing applications such as dynamically building commands for programmable interfaces, reformatting reports, text analysis, and the like.
- *Handheld devices* — Small devices require compact interpreters that are easy to program. Rexx is useful for handheld devices of various kinds, including mobile and smart phones.
- *Migration vehicle* — Given its cross-platform strengths, Rexx can be used as a migration vehicle to transfer personnel skills and migrate legacy code to new platforms.
- *Macro programming* — Rexx provides a single macro language for the tools of the programming environment: editors, text processors, applications, and other languages. Rexx's strengths in string processing play to this requirement, as does the fact it can easily be invoked as a set of utility functions through its standardized application programming interface, or API.
- *Embeddable language* — ANSI Rexx is defined as a library which can be invoked from outside applications by its standard API. Rexx is thus a function library that can be employed as an embeddable utility from other languages or systems.
- *Mainframe support* – Rexx is the default scripting language for several operating systems. Mainframe systems like z/OS, z/VM, and VSE/n are examples. Not only is Rexx the most popular scripting language on these systems, it is also the **only** high-level language that links into many subsystems essential to managing and administering them.
- *Mathematical applications* — Rexx performs computations internally in decimal arithmetic, rather than in the binary or floating-point arithmetic of most programming languages. The result is that Rexx always computes the same result regardless of the underlying platform. And, it gives precision to 999999 decimal places! But Rexx is not suitable for all mathematical applications. Advanced math functions are external add-ins rather than built-in functions for most Rexx interpreters, and Rexx performs calculations slowly compared to compiled languages optimized for these tasks, such as Fortran or Julia.

What Rexx Doesn't Do

There are a few situations where Rexx may not be the best choice.

Rexx is not a *systems programming language*. If you need to code on the machine level, for example, to write a device driver or other operating system component, Rexx is probably not a good choice. While there are versions of Rexx that permit direct memory access and other low-level tasks, languages like C/C++ or assembler are more suitable. Standard Rexx does not manipulate direct or relative addresses, change specific memory locations, or call PC interrupt vectors or UEFI/BIOS service routines.

Rexx is a great tool to develop clear, readable code. But it cannot force you to do so; it cannot save you from yourself. Chapter 12 discusses "Rexx with style" and presents simple recommendations for writing clear, reliable code.

Scripting languages consume more processor cycles and memory than traditional compiled languages. This affects a few projects. An example is a heavily used transaction in a high-performance online transaction processing (OLTP) system. The constant execution of the same transaction might make it worth the labor cost to develop it in a lower-level compiled language to optimize machine efficiency.

Another example is a heavily computational program in scientific research. Continual numeric calculation might make it worthwhile to optimize processor cycles through a computationally oriented compiler.

A final example might be where you're writing part of an operating system that is in constant use. In this case it might be worth investing the extra effort to develop the code in a low-level language like an assembler, or at least a fast compiled language (like C), to gain maximum efficiency and the quickest possible execution time.

In most other situations, for most other applications, our profession has reached the consensus that scripting languages are plenty fast enough. And they are so much more productive! This is why the shift to scripting has been one of the biggest software trends since the turn of the century.

If you're interested in reading further about the evolution towards scripting, chapter 19 explores this in some detail.

Figure 1-2 summarizes the kinds of programming problems to which Rexx is best suited as well as those for which it may not be the best choice.

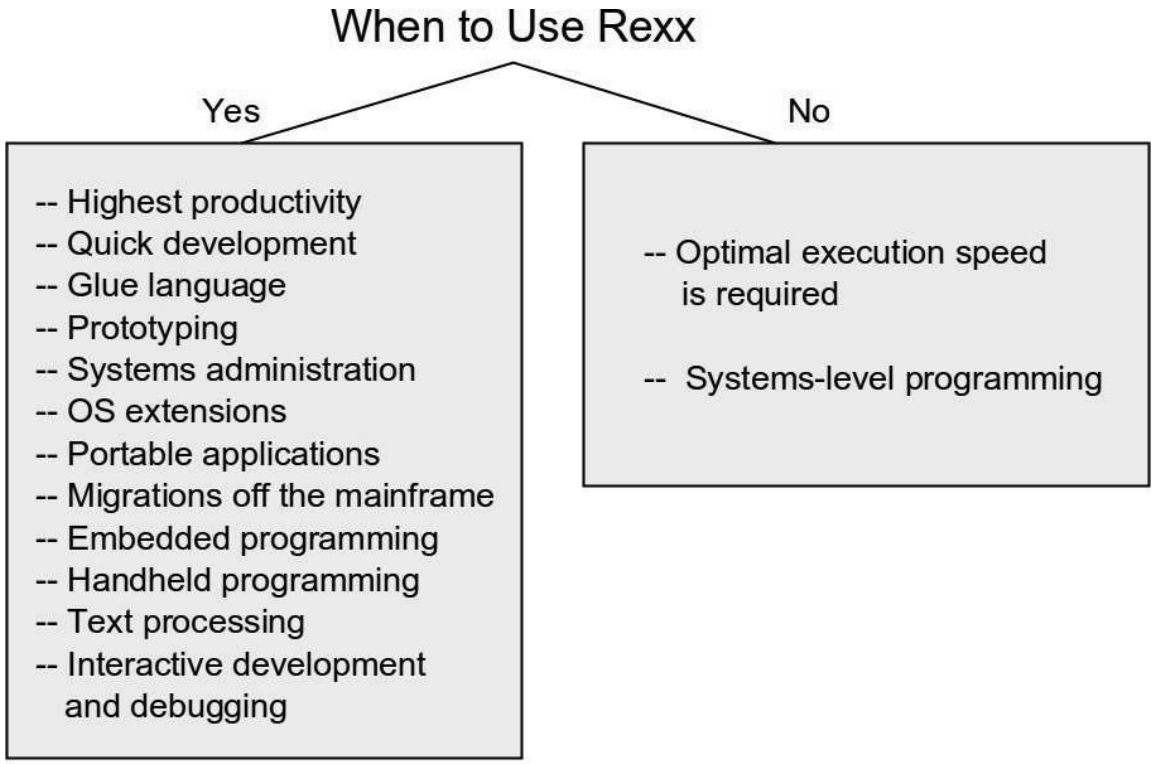

Figure 1-2

Which Rexx?

There are many free implementations of what we refer to as standard or *classic Rexx*. This is Rexx as defined by the TRL-2 or ANSI standards mentioned earlier. There are also two object-oriented supersets of classic procedural Rexx. And, there is NetRexx, the free Rexx-like language that runs in a Java Virtual Machine and presents a complementary or an alternative to Java for developing applications. Which Rexx should you use?

The first half of this book teaches classic Rexx. *It applies to any standard Rexx interpreter on any platform.* Once you know standard Rexx you can easily pick up the extensions unique to any Rexx interpreter. You can also easily learn interface programming, how to use Rexx tools and packages, object-oriented Rexx, NetRexx, or any Rexx variant. After all, the whole point of Rexx is ease of learning!

This table summarizes the major Rexx interpreters. All are free and available at no cost (except for IBM's Rexx Compiler). Most are open source, while a few are proprietary and come bundled with an operating system.

They are distributed either as easy-to-install packages, directly executable binaries, or source code.

Interpreter	Platforms	Cost & Licensing
Regina	Nearly everywhere (except mainframes)	Free, open source, GNU Library or LGPLv2
Open Object Rexx (aka "ooRexx")	Windows, Linux, Unix, BSD, macOS	Free, open source, GPLv2, CPL 1.0
Open Object Rexx for Android (aka "ooRexx for Android")	Android	Free, open source, Apache 2.0
BRexx	Linux, Unix, BSD, macOS, Android, Windows, DOS	Free, open source, GPLv2
BRexx370	Mainframes	Free, open source, GPLv2
Rexx/imc	BSD, Unix, Linux	Free, open source, no warranty
IBM REXX	Mainframes	Bundled, Proprietary license
IBM REXX Compiler	Mainframes	Proprietary license
Rexxoid (aka "Rexx for Android")	Android	Free, open source
NetRexx	Anywhere with a Java Virtual Machine	Free, ICU License
ARexx	Amiga-derived systems	Free, bundled, license varies
cREXX	z/VM, others soon	Free, MIT license
R4 (out of support)	Windows	Free. Limited warranty
Roo! (out of support)	Windows	Free. Limited warranty
Reginald (out of support)	Windows	Free. No warranty

All these interpreters meet the TRL-2 Rexx language standard. The single exception is NetRexx, which is best termed a "Rexx-like" language. Any standard Rexx you have installed can be used for working with the sample code in the first half of this book. This includes all the previously listed interpreters (except NetRexx), as well as standard Rexx interpreters bundled with mainframe or other operating systems.

To get you up and programming quickly, we defer closer consideration of the unique strengths of the various Rexx interpreters and the differences between them.

(If you need to know more right now, skip ahead to Chapter 19. That chapter discusses the evolution of Rexx and the roles it plays as a scripting language. It describes all the free Rexx interpreters above and presents the strengths of each. Chapters 20 through 30 then show how and where to download and install each Rexx interpreter. They describe the unique features of each and demonstrate many of them in sample scripts.)

If you're new to Rexx, we recommend starting with Regina Rexx. Regina Rexx is a great place to start for several reasons:

- *Popularity* — Regina is the most widely used free Rexx. Its large user community makes it easy to get help on public forums. More interfaces and tools are tested with Regina than any other Rexx implementation.

- *Runs anywhere* — Rexx is a platform-independent language, and Regina proves the point. Regina runs on almost any operating system including those in these families: Windows, Linux, Unix, BSD, macOS, and many lesser-used systems.
- *Meets all standards* — Regina meets all Rexx standards including the TRL-2 and ANSI- 1996 standards.
- *Well Documented* — Regina comes with complete documentation that precisely and fully explains the product.
- *Open source* — Regina is free and open source and distributed under the GNU Library General Public License. A few Rexx interpreters are free but not open source, as shown in the preceding table.

The code examples in this book all conform to standard Rexx and were tested using Regina Rexx under Windows and/or Linux. Run these scripts under any standard Rexx in any environment. A few scripts require a specific operating system. For example, those in Chapter 14 illustrate how to issue operating system commands and therefore are system-specific. Other scripts later in the book use specific open-source interfaces, tools, or interpreters. Where we present examples that run only in certain environments, we'll point it out.

To get you ready for the rest of the book, the remainder of this chapter shows you how to download and install Regina under Windows, Linux, Unix, BSD, and macOS. You need only install Regina if you don't already have access to a Rexx interpreter.

Downloading Regina Rexx

For a free download of Regina Rexx, go to the Regina project homepage at SourceForge at `https://sourceforge.net/projects/regina-rexx/`. Click on the **Files** tab on the horizontal menu to access the downloads page at `https://sourceforge.net/projects/regina-rexx/files/`.

From the **Files** page, you can click on **regina-documentation** and download Regina's two PDF product manuals. You'll probably want to download the files for the latest product release.

Also on the **Files** page, you can also click on **regina-rexx** to download the product itself. Again, you'll encounter a list of releases. Pick the latest one.

Now you'll face a screen with a long list of downloadable files. (There are so many files because Regina runs on so many different platforms.) Scroll down and you'll see the files grouped by operating system. You'll see groupings for **Windows**, **Linux**, **macOS**, **BSD**, and **Others**.

Once you find the group of downloads for your operating system, it's a simple matter of selecting which download matches your specific computer.

Okay, let's walk through the exact install steps you need to follow to install for Windows, Linux, and other systems like macOS, BSD, and Unix.

Installing Regina On Windows

From the list of downloadable files at the SourceForge project webpage, scroll down to the **Windows** group.

Your goal is to select the correct Windows download for your computer. You want each column in the table to match your computer's specifications.

Select the entry in the **Architecture** column that matches your computer: **x86_64** for 64-bit Windows on Intel processors, **x86** for 32-bit Windows on Intel processors, or **arm64** for 64-bit ARM processors. The **Format** column tells what kind of install format you're downloading.

For example, for my computer I selected the file **Regina__w64.exe** (the underscores represent a variable product release number). That's for 64-bit Windows on Intel computers. The **Format** entry describes this as a "**Self-installing executable.**" Perfect! That makes for a very easy Windows installation.

Once you've downloaded the proper file, double-click on it to start the installation. You'll proceed through a standard Windows install. Accept the defaults all the way through.

This installs Regina's Demos, Development Kit, and Documentation, along with the interpreter itself. It requires very little space (less than ten megabytes).

The Windows default install directory will be `C:\ Program Files\rexx.org\Regina` for 64-bit Intel architecture and ARM processors, and `C:\Program Files (x86)\rexx.org\Regina` for 32-bit Intel architecture processors.

The default install will also add the Regina installation directory to your `PATH` variable and include the `REGINA_HOME` environmental variable. These settings enable you to easily run Regina scripts without any additional steps.

Another screen will prompt you for the filename extensions for your Regina scripts. Traditionally, Regina scripts use filename extension of `.rexx`, so select that.

A final screen asks whether you want to install and optionally auto-start the **Regina Stack Service**. The *stack* is a general-purpose data structure that Regina programs can use to pass or temporarily store data. (Chapter 11 describes the stack in detail.) For learning purposes, it's convenient to install this service.

That's it! You're done. Now, you can navigate to your Windows' installed Program List and see the **ReginaRexx** entry. Beneath it will appear its list of components.

Remember those computer benchmarks on page 5? Let's benchmark your computer and see how it compares. This will also verify that Regina is installed and working properly.

Select **ReginaRexx** from the Windows Program list. Under the **ReginaRexx** entry, pick **Regina Rexx Demos**. Then click on the program **Rexxcps**. The program will run in a command window and give you a benchmark you can directly compare to those on page 5.

So, you can run any Rexx program with the file extension of `.rexx` simply by double-clicking its filename.

Or you can run programs from the command line. Assuming you've navigated to the directory containing the program you want to run, (or that your `PATH` environmental variable contains that directory), you can enter its full filename to run it: `rexxcps.rexx`

The filename extension of `.rexx` tells Windows to run the file using Regina Rexx. Or you can run a file by explicitly invoking the Regina interpreter. Since you've already set the filename extension within Windows, you can either specify the full or partial filename. Either will work:

```
regina rexxcps.rexx        or        regina rexxcps
```

What about entering just the filename, but without its extension? Windows can't run this program because it doesn't know it should invoke Regina to run it. So this will **not** work: `rexxcps`

Installing Regina on Linux / Unix / BSD / macOS

There are several ways to install Regina on Linux and similar computers, such as those running macOS, Unix, and BSD. Perhaps the quickest and easiest is to use your distribution's *package manager*, its software interface for installing and removing software products.

Example package managers include Ubuntu's Software Center, the Synaptic Package Manager, APT, DNF, YUM, and ZYPP. The one you have available depends on which Linux distribution you run. All have easy-to-use GUI interfaces.

Most Linux package managers connect to large repositories of software products. Just search for Regina in that list of installable programs, and then select and install Regina using the package manager's graphical interface.

In most cases, the package manager will direct you to download two Regina files. One is for the interpreter itself, and the other is for its library. For example, using Synaptic Package Manager directed me to download the files `regina-rexx`, and its library, `libregina3`. (The exact file names may vary in your case.) The point is that the package manager will often automatically direct you to download more than a single file to install Regina. Just download these files and double-click to install them.

If you're on a Mac and use HomeBrew, you can install simply by: `brew install regina-rexx`

Installing Regina with your operating system's package manager is quick and easy. But it does have one possible downside. Sometimes repositories don't contain the latest release of the software.

Thus, you may want to install Regina directly from its permanent project homepage at SourceForge.net. This is the same SourceForge webpage mentioned in the above discussion on "Installing Regina on Windows." The web address for downloading Regina files is: `https://sourceforge.net/projects/regina-rexx/files/.`

Once there, you can download the product documentation by selecting the directory **regina-documentation**, and the product itself by choosing **regina-rexx**.

We recommend downloading the two Regina manuals available under the **regina-documentation** directory. You may have need of them later.

In the **regina-rexx** directory, you'll want to select the current release. There you'll view a very long product list. Scroll down to the section labelled **Linux**. This reduces the downloads to a more manageable list.

To find the file to download, scroll through the list under the label **Operating System** and locate yours. (If you can't find an exact match, pick the one closest to your system. For example, someone running a Debian-based distribution that is not explicitly listed could use the Debian packages.)

Once you've located your operating system, ensure that the **Architecture** and **32bit** or **64bit** columns match your system. The rightmost column will then tell you which kind of package is involved: Debian, RPM, Alpine, or whatever.

Now you'll need to download and install two packages. One will be Regina's library package, while the second is the interpreter itself.

Here's an example. At the time of writing, I ran Linux Mint 21.2. I found that Linux Mint did not appear in the list of **Operating Systems** in the **Linux** section**.** A quick web search showed that Mint 21.x is based on Ubuntu 22. So I downloaded the files for Ubuntu 22. They worked fine.

Just as when you install via your Linux package manager, you'll likely need to install two files. I installed these two (the underscores contain the variable product release numbers):

```
libregina3_____-amd64-Ubuntu-22.04.deb
regina-rexx_____-amd64-Ubuntu-22.04.deb
```

As always with package manager files, you just double-click to install them. So first I installed the library `.deb`, then next, the **regina-rexx** package. Done!

Afterwards, you'll want to test the install to ensure it worked correctly. Let's run the benchmarking program that produced the table on page 5. You can benchmark your computer against those in the table.

The benchmarking program is called **rexxcps**. With the default Regina extension of `.rexx`, that means the full name of this program is `rexxcps.rexx`. Use your operating system's Search function to locate this program. Then open a terminal window, change to the directory where it resides, and execute it:

```
regina rexxcps     or     regina  rexxcps.rexx
```

If Regina can't find the program to run it, you can run it by specifying that it resides in the current directory:

```
regina  ./rexxcps       or     regina  ./rexxcps.rexx
```

Manual Installation From Source Code

If the simple "package install" we describe above didn't work for you, or if you have an uncommon system not included in Regina's list of package installs, here we'll describe a simple, generic approach that will work for almost any Unix-derived operating system, including the macOS.

Where slight differences exist between systems, the Regina *Install Notes* will tell you what you need to know. These typically reside in **INSTALL*** or **README*** files. They download with the product. Be sure to read those instructions!

To install Regina under any Linux, Unix, BSD, or macOS operating system, use the `root` user ID and download the source file into an empty directory.

The source file will be compressed. It will thus end in one of these file name extensions: `.tar.bz2`, `.tar.gz`, `.tgz`, `.tar`, or `.zip`. To uncompress this file, just double-click on the file name.

If double-clicking produces a second compressed file (such as a `.tar` file), double-click on that file.

You'll know you're done decompressing when you see a long list of files being created. (So, decompressing might be either a one-step or two-step process, depending on what kind of compressed file you initially downloaded.)

Now, open a terminal window and navigate to the home directory into which you extracted Regina.

Look in the directory into which the files were extracted and find and read the Install Notes. They are usually in a file named `INSTALL*` or `README*`. These notes give operating system specific instructions you must follow to successfully install Regina.

The Install Notes will tell you that you need to enter these two commands as the `root` user id to the operating system:

```
./configure
make install
```

These commands configure and install Regina. Since they compile source code, they require a C compiler to run. Almost all Linux, Unix, and BSD machines will have a C compiler present. If your system does not have one installed, download a free compiler from any of several sites including `www.gnu.org`. For macOS, go to the Apple Store and download Xcode.

The Install Notes will tell you if you need to set any environmental variables. Typically you'll need to set your `PATH` to include the location of the Regina binary. And, you'll need to point the `LD_LIBRARY_PATH` environmental variable to where Regina's shared library file resides (usually in `/usr/local/lib`). Again, refer to the install instructions! They'll give you the exact commands to run, which do vary by the operating system.

Finally, test your installation by running one of the Regina-provided demo scripts. Let's benchmark your system by running the benchmark program used in the table on page 5 of this chapter. You can compare your system's performance to the examples listed in that table.

The program to run is called `rexxcps.rexx`. You may have to look around in the Regina install directories to locate it. Change to that directory, then enter this to run the program:

```
regina rexxcps          or          regina ./rexxcps.rexx
```

If you have any problems with installing, check that your `PATH` variable properly includes the path of the Regina executable, and that the `LD_LIBRARY_PATH` or its equivalent properly points to Regina's shared library.

Also, remember that to run a program you may need to set its permission bits as executable:

```
chmod  +x  your_program_name.rexx
```

Testing Rexx Statements: Rexxtry

As well as the `rexxcps` program that benchmarks your computer's performance, another common program distributed with most Rexx interpreters is called `rexxtry` (if you can't find it, it's included with the sample programs for this book that you can download for free from `www.RexxInfo.org`).

Start this program from the command line, and you can enter various Rexx statements to it. It responds by executing the statement and returning its results to you.

Rexxtry can be useful to learn how Rexx works. Here's an example interaction where we test to see if Rexx treats single and double quotation marks as the same by entering a `say` instruction:

```
babs@dell1:~$ regina rexxtry
Try out Rexx statements interactively.

To end, enter EXIT

Rexxtry:
say 'Are single quotes' "the same as double quotes?"
Are single quotes the same as double quotes?
Rexxtry:
exit
```

Summary

This chapter lists the advantages of scripting in Rexx and suggests where Rexx is most useful. Given its power, flexibility, portability, and ease of use, Rexx is suitable for addressing a wide range of programming problems. The only situations where Rexx does not apply are those oriented toward "systems programming" and programs that demand totally optimized machine utilization.

Rexx distinguishes itself among scripting languages by combining ease of use with power. Rexx uses specific interpreter design techniques to achieve this combination. Rexx has simple syntax, minimal "special variables," no "default variables", and a case-insensitive free-format combined with a small, easily learned instruction set. Its many built-in functions, extensibility, and the ability to issue commands to the operating system and other external interfaces give Rexx power while retaining ease of use.

Ease of use is important even to highly experienced computer professionals because it reduces error rates and determines the life span of their code. Experienced developers leverage a quickly coded language like Rexx to achieve outstanding productivity.

The final part of this chapter showed how to download and install Regina Rexx under Windows, Linux, Unix, BSD, and macOS. This popular Rexx interpreter is a free, open-source product you can use to learn Rexx in the tutorial of the following chapters. Any other standard Rexx interpreter could be used as well. The next several chapters get you quickly up and running Rexx scripts through an example-based tutorial.

Test Your Understanding

1. In what way is Rexx a *higher-level* language than compiled languages like C or C++ ? What's a *glue language*? Why is there an industry-wide trend towards scripting languages?
2. Are developers required to code Rexx instructions starting in any particular column? In upper- or lowercase?
3. Even if you're an expert programmer, why is ease of use still important?
4. What are names of the two object-oriented Rexx interpreters? Will standard or *classic* Rexx scripts run under these OO interpreters without alteration?
5. Does Rexx run on Android? How about departmental servers? PCs? Mainframes?
6. What are the two key Rexx standards? Are these two standards almost the same or significantly different?
7. Traditionally there is a trade-off between ease of use and power. What specific techniques does Rexx employ to gain both attributes and circumvent the trade-off?

2

Language Basics

Overview

This chapter describes the basic elements of Rexx. It discusses the simple components that make up the language. These include script structure, elements of the language, operators, variables, and the like. As a starting point, we explore a simple sample script. We'll walk through this script and explain what each statement means. Then we'll describe the language components individually, each in its own section. We'll discuss Rexx variables, character strings, numbers, operators, and comparisons.

By the end of this chapter, you'll know about the basic components of the Rexx language. You'll be fully capable of writing simple scripts and will be ready to learn about the language features explored more fully in subsequent chapters. The chapters that follow present other aspects of the language, based on sample programs that show its additional features. For example, topics covered in subsequent chapters include directing the logical flow of a script, arrays and tables, input and output, string manipulation, subroutines and functions, and the like. But now, let's dive into our first sample script.

A First Program

Had enough of your job? Maybe it's time to join the lucky developers who create computer games for a living! The complete Rexx program that follows is called the Number Game. It generates a random number between 1 and 10 and asks the user to guess it (well, okay, the playability is a *bit* weak. . . .) The program reads the number the user guesses and states whether the guess is correct.

```
/* The NUMBER GAME - User tries to guess a number between 1 and 10  */

/* Generate a random number between 1 and 10                        */

the_number = random(1,10)

say "I'm thinking of number between 1 and 10. What is it?"
```

```
pull the_guess

if the_number = the_guess then
   say 'You guessed it!'
else
   say 'Sorry, my number was: ' the_number

say 'Bye!'
```

Here are two sample runs of the program:

```
C:\Regina\pgms>number_game.rexx
I'm thinking of number between 1 and 10. What is it?
4
Sorry, my number was: 6
Bye!

C:\Regina\pgms>number_game.rexx
I'm thinking of number between 1 and 10. What is it?
8
You guessed it!
Bye!
```

This program illustrates several Rexx features. It shows that you document scripts by writing whatever description you like between the symbols `/*` and `*/`. Rexx ignores whatever appears between these *comment delimiters*. Comments can be isolated on their own lines, as in the sample program, or they can appear as *trailing comments* after the statement on a line:

```
the_number = random(1,10)  /* Generate a random number between 1 and 10 */
```

Comments can even stretch across multiple lines in *box style*, as long as they start with `/*` and end with `*/`:

```
/*********************************************************************
*  The NUMBER GAME - User tries to guess a number between 1 and 10      *
*  Generate a random number between 1 and 10                            *
*********************************************************************/
```

Rexx is *case-insensitive*. Code can be entered in lowercase, uppercase, or mixed case; Rexx doesn't care. The `if` statement could have been written like this if we felt it were clearer:

```
IF the_number = the_guess THEN
   SAY 'You guessed it!'
ELSE
   SAY 'Sorry, my number was: ' the_number
```

The variable named `the_number` could have been coded as `THE_NUMBER` or `The_Number`. Since Rexx ignores case it considers all these as references to the same variable. The one place where case *does* matter is within *literals* or hardcoded character strings:

```
say 'Bye!'      outputs:      Bye!
```

while

```
say 'BYE!'      displays:     BYE!
```

Character strings are any set of characters occurring between a matched set of either single quotation marks (`'`) or double quotation marks (`"`).

What if you want to encode a quote within a literal? In other words, what do you do when you need to encode a single or double quote as part of the character string itself? To put a single quotation mark within the literal, enclose the literal with double quotation marks:

```
say "I'm thinking of number between 1 and 10. What is it?"
```

To encode double quotation marks within the string, enclose the literal with single quotation marks:

```
say 'I am "thinking" of number between 1 and 10. What is it?'
```

Rexx is a *free-format language*. The spacing is up to you. Insert (or delete) blank lines for readability, and leave as much or as little space between instructions and their operands as you like. Rexx leaves the coding style up to you as much as a programming language possibly can.

For example, here's yet another way to encode the `if` statement:

```
IF the_number = the_guess THEN  SAY 'You guessed it!'
                          ELSE  SAY 'Sorry, my number was: ' the_number
```

About the only situation in which spacing is *not* the programmer's option is when encoding a Rexx *function*. A function is a built-in routine Rexx provides as part of the language; you also may write your own functions. This program invokes the built-in function `random` to generate a random number between 1 and 10 (inclusive). The parenthesis containing the function argument(s) must immediately follow the function name without any intervening space. If the function has no arguments, code it like this:

```
the_number = random()
```

Rexx requires that the parentheses occur *immediately after* the function name to recognize the function properly.

The sample script shows that one does not need to *declare* or predefine variables in Rexx. This differs from languages like C++, Java, COBOL, or Pascal. Rexx variables are established at the time of their first use. The variable `the_number` is defined during the assignment statement in the example. Space for the variable `the_guess` is allocated when the program executes the `pull` instruction to read the user's input:

```
pull  the_guess
```

In this example, the `pull` instruction reads the characters that the user types on the keyboard, until he or she presses the <ENTER> key, into one or more variables and automatically translates them to uppercase. Here the item the user enters is assigned to the newly created variable `the_guess`.

All variables in Rexx are variable-length character strings. Rexx automatically handles string length adjustments. It also manages numeric or data type conversions. For example, even though the variables `the_number` and `the_guess` are character strings, if we assume that both contain strings that represent numbers, one could perform arithmetic or other numeric operations on them:

```
their_sum  =  the_number  +  the_guess
```

Rexx automatically handles all the issues surrounding variable declarations, data types, data conversions, and variable length character strings that programmers must manually manage in traditional compiled languages. These features are among those that make it such a productive, high-level language.

Language Elements

Rexx consists of only two dozen *instructions*, augmented by the power of some 70 *built-in functions*. Figure 2-1 below pictorially represents the key components of Rexx. It shows that the instructions and functions together compose the core of the language, which is then surrounded and augmented by other features. A lot of what the first section of this book is about is introducing the various Rexx instructions and functions.

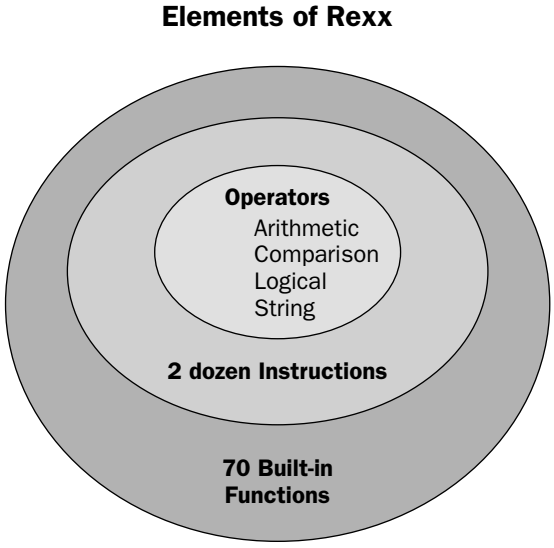

Figure 2-1

Of course, this book also provides a language reference section in the appendices, covering these and other aspects of the language. For example, Appendix B is a reference to all standard Rexx instructions, while Appendix C provides the reference to standard functions.

The first sample program illustrated the use of the instructions `say`, `pull`, and `if`**.** Rexx instructions are typically followed by one or more *operands*, or elements upon which they operate. For example, `say` is followed by one or more elements it writes to the display screen. The `pull` instruction is followed by a list of the data elements it reads.

The sample script illustrated one function, `random`. Functions are always immediately followed by parentheses, usually containing function *arguments*, or inputs to the function. If there are no arguments, the function must be immediately followed by empty parentheses `()` . Rexx functions always return a single result, which is then substituted into the expression directly in place of the function call. For example, the random number returned by the `random` function is actually substituted into the statement that follows, on the right-hand side of the equals sign, then assigned to the variable `the_number`:

```
the_number = random(1,10)
```

Variables are named storage locations that can contain values. They do not need to be declared or defined in advance, but are rather created when they are first referenced. You can declare or define all variables used in a program at the beginning of the script, but Rexx does not require this. Some programmers like to declare all variables at the top of their programs, for clarity, but Rexx leaves the decision whether or not to do this up to you.

All variables in Rexx are internally stored as variable-length strings. The interpreter manages their lengths and data types. Rexx variables are "typeless" in that their contents define their usage. If strings contain digits, you can apply numeric operations to them. If they do not contain strings representing numeric values, numeric operations don't make sense and will fail if attempted. Rexx is simpler than other programming languages in that developers do not have to concern themselves with data types.

Variable names are sometimes referred to as *symbols*. They may be composed of letters, digits, and characters such as `. ! ? _` . A variable name you create must not begin with a digit or period. A *simple variable name* does not include a period. A variable name that includes a period is called a *compound variable* and represents an *array* or *table.* Arrays will be covered in Chapter 4. They consist of groups of similar data elements, typically processed as a group.

If all Rexx variables are *typeless*, how does one create a numeric value? Just place a string representing a valid number into a Rexx variable. Here are *assignment statements* that achieve this:

```
whole_number_example            =  15
decimal_example                 =  14.2
negative_number                 = -21.2
exponential_notation_example    =  14E+12
```

A *number* in Rexx is simply a string of one or more digits with one optional decimal point anywhere in the string. Numbers may optionally be preceded by their sign, indicating a postive or a negative number. Numbers may be represented very flexibly by almost any common notation. Exponential numbers may be represented in either engineering or scientific notation (the default is scientific). The following table shows examples of numbers in Rexx.

Number Type	Also Known As	Examples
Whole	Integer	'3' '+6' '9835297590239032'
Decimal	Fixed point	'0.3' '17.36425'
Exponential	Real --or--	'1.235E+11' (*scientific*, one digit left of decimal point)
	Floating point	'171.123E+11' (*engineering*, 1 to 3 digits left of decimal)

Variables are assigned values through either assignment statements or input instructions. The assignment statement uses the equals sign (=) to assign a value to a variable, as shown earlier. The input instructions are the `pull` or `parse` instructions, which read input values, and the `arg` and `parse arg` instructions, which read command line parameters or input arguments to a script.

If a variable has not yet been assigned a value, it is referred to as *uninitialized*. The value of an uninitialized variable is the name of the variable itself in uppercase letters. This `if` statement uses this fact to determine if the variable `no_value_yet` is uninitialized:

```
if  no_value_yet = 'NO_VALUE_YET'  then
    say 'The variable is not yet initialized.'
```

Character strings or *literals* are any set of characters enclosed in single or double quotation marks (`'` or `"`).

If you need to include either the single or double quote within the literal, simply enclose that literal with the other *string delimiter*. Or you can encode two single or double quotation marks back to back, and Rexx understands that this means that one quote is to be contained within the literal (it knows the doubled quote does not terminate the literal). Here are a few examples:

```
literal= 'Literals contain whatever characters you like: !@#$%^&*()-=+~.<>?/_'
need_a_quote_mark_in_the_string = "Here's my statement."
same_as_the_previous_example    = 'Here''s my statement.'
this_is_the_null_string = ''  /*two quotes back to back are a "null string" */
```

In addition to supporting any typical numeric or string representation, Rexx also supports *hexadecimal* or base 16 numbers. *Hex strings* contain the upper- or lowercase letters `A` through `F` and the digits `0` through `9`, and are followed by an upper- or lowercase `X`:

```
twenty_six_in_hexidecimal = '1a'x  /*  1A is the number 26 in base sixteen      */
hex_string = "3E 11 4A"X           /*  Assigns a hex string value to hex_string */
```

Rexx also supports *binary*, or base two strings. Binary strings consist only of `0`s and `1`s. They are denoted by their following upper- or lowercase `B`:

```
example_binary_string = '10001011'b
another_binary_string = '1011'B
```

Rexx has a full complement of functions to convert between regular character strings and hex and binary strings. Do not be concerned if you are not familiar with the uses of these kinds of strings in programming languages. We mention them only for programmers who require them. Future chapters will explain their use more fully and provide illustrative examples.

Operators

Every programming language has *operators*, symbols that indicate arithmetic operations or dictate that comparisons must be performed. Operators are used in calculations and in assigning values to variables, for example. Rexx supports a full set of operators for the following.

- ❑ Arithmetic
- ❑ Comparison
- ❑ Logical operators
- ❑ Character string concatenation

The arithmetic operators are listed in the following table:

Arithmetic Operator	Use
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Integer division — returns the integer part of the result from division
//	Remainder division — returns the remainder from division
**	Raise to a whole number power
+ (as a prefix)	Indicates a positive number
– (as a prefix)	Indicates a negative number

All arithmetic operators work as one would assume from basic high-school algebra, or from programming in most other common programming languages. Here are a few examples using the less obvious operators:

```
say (5 % 2)   /* Returns the integer part of division result. Displays: 2   */
say (5 // 2)  /* Returns the remainder from division.      Displays: 1      */
say (5 ** 2)  /* Raises the number to the whole power.     Displays: 25     */
```

Remember that because all Rexx variables are strings, arithmetic operators should only be applied to variables that evaluate to valid numbers. Apply them only to strings containing digits, with their optional decimal points and leading signs, or to numbers in exponential forms.

Numeric operations are a major topic in Rexx (as in any programming language). The underlying principle is this — *the Rexx standard ensures that the same calculation will yield the same results even when run under different Rexx implementations or on different computers*. Rexx provides an exceptional level of machine- and implementation-independence compared with many other programming languages.

If you are familiar with other programming languages, you might wonder how Rexx achieves this benefit. Internally, Rexx employs decimal arithmetic. It does not suffer from the approximations caused by languages that rely on floating point calculations or binary arithmetic.

The only arithmetic errors Rexx gives are *overflow* (or *underflow*). These result from insufficient storage to hold exceptionally large results.

To control the number of significant digits in arithmetic results, use the `numeric` instruction. Sometimes the number of significant digits is referred to as the *precision* of the result. Numeric precision defaults to nine digits. This sample statement illustrates the default precision because it displays nine digits to the right of the decimal place in its result:

```
say  2 / 3             /* displays 0.666666667 by default     */
```

This example shows how to change the precision in a calculation. Set the numeric precision to 12 digits by the `numeric` instruction, and you get this result:

```
numeric digits 12      /*  set numeric precision to 12 digits   */
say  2 / 3             /*  displays:  0.666666666667            */
```

Rexx preserves trailing zeroes coming out of arithmetic operations:

```
say  8.80 - 8          /*  displays: 0.80  */
```

If a result is zero, Rexx always displays a single-digit `0`:

```
say  8.80 - 8.80       /*  displays: 0     */
```

Chapter 7 explores computation further. It tells you everything you need to know about how to express numbers in Rexx, conversion between numeric and other formats, and how to obtain and display numeric results. We'll defer further discussion on numbers and calculations to Chapter 7.

Comparison operators provide for numeric and string comparisons. These are the operators you use to determine the equality or inequality of data elements. Use them to determine if one data item is greater than another or if two variables contain equal values.

Since every Rexx variable contains a character string, you might wonder how Rexx decides to perform a character or numeric comparison. The key rule is*: if both terms involved in a comparison are numeric, then the comparison is numeric*. For a numeric comparison, any leading zeroes are ignored and the numeric values are compared. This is just as one would expect.

If either term in a comparison is other than numeric, then a *string comparison* occurs. The rule for string comparison is that leading and trailing blanks are ignored, and if one string is shorter than the other, it is padded with trailing blanks. Then a character-by-character comparison occurs. String comparison is case-sensitive. The character string `ABC` is not equal to the string `Abc`. Again, this is what one would normally assume.

Rexx features a typical set of comparison operators, as shown in the following table:

Comparison Operator	Meaning
=	Equal
\= ¬=	Not equal
>	Greater than
<	Less than
>= \< ¬<	Greater than or equal to, not less than
<= \> ¬>	Less than or equal to, not greater than
>< <>	Greater than or less than (same as not equal)

The "not" symbol for operators is typically written as a backslash, as in "not equal:" `\=` But sometimes you'll see it written as ¬ as in "not equal:" `¬=` Both codings are equivalent in Rexx. The first representation is very common, while the second is almost exclusively associated with mainframe scripting. *Since most keyboards outside of mainframe environments do not include the symbol ¬ we recommend always using the backslash*. This is universal and your code will run on any platform. The backslash is the ANSI-standard Rexx symbol. You can also code "not equal to" as: `<>` or `><` .

In Rexx comparisons, if a comparison evaluates to `TRUE`, it returns 1. A `FALSE` comparison evaluates to 0. Here are some sample numeric and character string comparisons and their results:

```
'37'  = '37'    /*  TRUE  - a numeric comparison */
'0037'= '37'    /*  TRUE  - numeric comparisons disregard leading zeroes */
'37'  = '37   ' /*  TRUE  - blanks disregarded   */
'ABC' = 'Abc'   /*  FALSE - string comparisons are case-sensitive        */
'ABC' = '   ABC  ' /* TRUE- preceding & trailing blanks are irrelevant   */
''    = '     '    /* TRUE- null string is blank-padded for comparison   */
```

Rexx also provides for *strict comparisons* of character strings. *In strict comparisons, two strings must be identical to be considered equal*—leading and trailing blanks count and no padding occurs to the shorter string. Strict comparisons only make sense in string comparisons, not numeric comparisons. Strict comparison operators are easily identified because they contain doubled operators, as shown in the following chart:

Strict Comparison Operator	Meaning
==	Strictly equal
\== ¬==	Strictly not equal
>>	Strictly greater than
<<	Strictly less than
>>= \<< ¬<<	Strictly greater than or equal to, strictly not less than
<<= \>> ¬>>	Strictly less than or equal to, strictly not greater than

Here are sample strict string comparisons:

```
'37'  == '37   '  /*  FALSE - strict comparisons include blanks         */
'ABC' >> 'AB'     /*  TRUE - also TRUE as a nonstrict comparison       */
'ABC' == '   ABC  '   /* FALSE - blanks count in strict comparison     */
''    == '   '       /* FALSE - blanks count in strict comparison     */
```

Logical operators are sometimes called *Boolean operators* because they apply *Boolean logic* to the operands. Rexx's logical operators are the same as the logical operators of many other programming languages. This table lists the logical operators:

Logical Operator	Meaning	Use
`&`	Logical AND	TRUE if both terms are true
`|`	Logical OR	TRUE if either term is true
`&&`	Logical EXCLUSIVE OR	TRUE if either (but not both) terms are true
¬ or `\` (as a prefix)	Logical NOT	Changes TRUE to FALSE and vice versa

Boolean logic is useful in `if` statements with multiple comparisons. These are also referred to as *compound comparisons*. Here are some examples:

```
if  ('A' = var1)  &  ('B' = var2) then
   say 'Displays only if BOTH comparisons are TRUE'

if  ('A' = var1)  |  ('B' = var2) then
   say 'Displays if EITHER comparison is TRUE'

if  ('A' = var1) &&  ('B' = var2) then
   say 'Displays if EXACTLY ONE comparison is TRUE'

if \('A' = var1)  then say 'Displays if A is NOT equal to var1'
```

Concatenation is the process of pasting two or more character strings together. Strings are appended one to the end of the other. *Explicitly* concatenate strings by coding the *concatenation operator* `||` . Rexx also automatically concatenates strings when they appear together in the same statement. Look at these instructions executed in sequence:

```
my_var = 'Yogi Bear'
say 'Hi there,'  ||  ' '  ||  my_var     /*  displays:  'Hi there, Yogi Bear'   */
say 'Hi there,'my_var                    /*  displays:  'Hi there,Yogi Bear'
                                                        no space after the comma */
say 'Hi there,'  my_var                  /*  displays:  'Hi there, Yogi Bear'
                                                       one space after the comma */
```

The second `say` instruction shows *concatenation through abuttal*. A literal string and a variable appear immediately adjacent to one another, so Rexx concatenates them without any intervening blank.

Contrast this to the last `say` instruction, where Rexx concatenates the literal and variable contents, but with one blank between them. If there are one or more spaces between the two elements listed as operands to the `say` instruction, Rexx places exactly one blank between them after concatenation.

Given these three methods of concatenating strings, individual programmers have their own preferences. Using the concatenation operator makes the process more explicit, but it also results in longer statements to build the result string.

Rexx has four kinds of operators: arithmetic, comparison, logical, and concatenation. And there are several operators in each group. If you build a statement with multiple operators, how does Rexx decide which operations to execute first? The order can be important. For example:

4 times 3, then subtract 2 from the result is 10

Perform those same operations with the same numbers in a different order, and you get a different result:

3 subtract 2, then multiple that times 4 yields the result of 4

Both these computations involve the same two operations with the same three numbers but the operations occur in different orders. They yield different results.

Clearly, programmers need to know in what order a series of operations will be executed. This issue is often referred to as the operator *order of precedence*. The order of precedence is a rule that defines which operations are executed in what order.

Some programming languages have intricate or odd orders of precedence. Rexx makes it easy. Its order of precedence is the same as in conventional algebra and the majority of programming languages. (The only minor exception is that the prefix *minus operator* always has higher priority than the exponential operator).

From highest precedence on down, this lists Rexx's order of precedence:

- ❑ Prefix operators `+ - \`
- ❑ Power operator `**`
- ❑ Addition and subtraction `+ -`
- ❑ Concatenation by intervening blanks `||` by abuttal
- ❑ Comparison operators `= == > < >= <=` ...and the others
- ❑ Logical AND `&`
- ❑ Logical OR `|`
- ❑ EXCLUSIVE OR `&&`

If the order of precedence is important to some logic in your program, an easy way to ensure that operations occur in the manner in which you expect is to simply enclose the operations to perform first in parentheses. When Rexx encounters parentheses, it evaluates the entire expression when that term is required. So, you can use parentheses to guarantee any order of evaluation you require. The more deeply nested a set of parentheses is, the higher its order of precedence. The basic rule is this: *when Rexx encounters expressions nested within parentheses, it works from the innermost to the outermost*.

To return to the earlier example, one can easily ensure the proper order of operations by enclosing the highest order operations in parentheses:

```
say  (4 * 3)  - 2        /* displays: 10 */
```

To alter the order in which operations occur, just reposition the parentheses:

```
say  4 * (3 - 2)         /* displays: 4  */
```

Summary

This chapter briefly summarizes the basic elements of Rexx. We've kept the discussion high level and have avoided strict "textbook definitions." We discussed variable names and how to form them, and the difference between simple variable names and the compound variable names that are used to represent tables or arrays. We discussed the difference between strings and numbers and how to assign both to variables.

We also listed and discussed the operators used to represent arithmetic, comparison, logical, and string operations. We gave a few simple examples of how the operators are used; you'll see many more, real-world examples in the sample scripts in the upcoming chapters.

The upcoming chapters round out your knowledge of the language and focus in more detail on its capabilities. They also provide many more programming examples. Their sample scripts use the language elements this chapter introduces in many different contexts, so you'll get a much better feel for how they are used in actual programming.

Test Your Understanding

1. How are comments encoded in Rexx? Can they span more than one line?
2. How does Rexx recognize a function call in your code?
3. Must variables be declared in Rexx as in languages like C++, Pascal, or Java? How are variables established, and how can they be tested to see if they have been defined?
4. What are the two instructions for basic screen input and output?
5. What is the difference between a *comparison* and *strict comparison*? When do you use one versus the other? Does one apply strict comparisons to numeric values?
6. How do you define a numeric variable in Rexx?

3

Control Structures

Overview

Program logic is directed by what are called *control structures* or *constructs* — statements like if- then-else, do-while, and the like. Rexx offers a complete set of control structures in less than a dozen instructions.

Rexx fully supports *structured programming*, a rigorous methodology for program development that simplifies code and reduces programmer error. Invented and defined by such experts as Edsger Dijkstra and Edward Yourdon, structured programming restricts control structures to a handful that permit single points of entry and exit to code blocks. The "golden rule" of structured programming mandates that there should be only a single entry point and a single exit point from any code block, such as a `do` loop. `Goto`'s and unstructured entries and exits are prohibited.

Structured programming encourages *modularity* and reduces complex spaghetti code to short, readable, sections of self-contained code. Small, well-documented routines mean greater clarity and fewer programmer errors. While developer convenience sometimes leads to unstructured code ("Well... it made sense when I wrote it!"), structured, modular code is more readable and maintainable.

We recommend structured programming; nearly all of the examples in this book are structured. But we note that, as a powerful programming language, Rexx includes instructions that permit unstructured coding if desired.

This chapter discusses how to write structured programs with Rexx. We start by listing the Rexx instructions used to implement structured constructs. Then, we describe each in turn, showing how it is used in the language through numerous code snippets. At appropriate intervals, we present complete sample scripts that illustrate the use of the instructions in structured coding.

The latter part of the chapter covers the Rexx instructions for unstructured programming. While we don't recommend their general use, there are special situations in which these instructions are highly convenient. Any full-power scripting language requires a full set of instructions for controlling logical flow, including those that are unstructured.

Structured Programming in Rexx

As we've mentioned, structured programming consists of a set of constructs that enforce coding discipline and organization. These are implemented in Rexx through its basic instructions for the control of program logic. The basic constructs of structured programming and the Rexx instructions used to implement them are listed in this table:

Structured Construct	Rexx Instruction
PROCESS	Any set of instructions, executed one after another. The `exit` or `return` instructions end the code composing a program or routine.
IF-THEN. IF-THEN-ELSE	`if`
DO. DO-WHILE	`do`
CASE	`select`
CALL	`call`

Figure 3-1 illustrates the structured constructs.

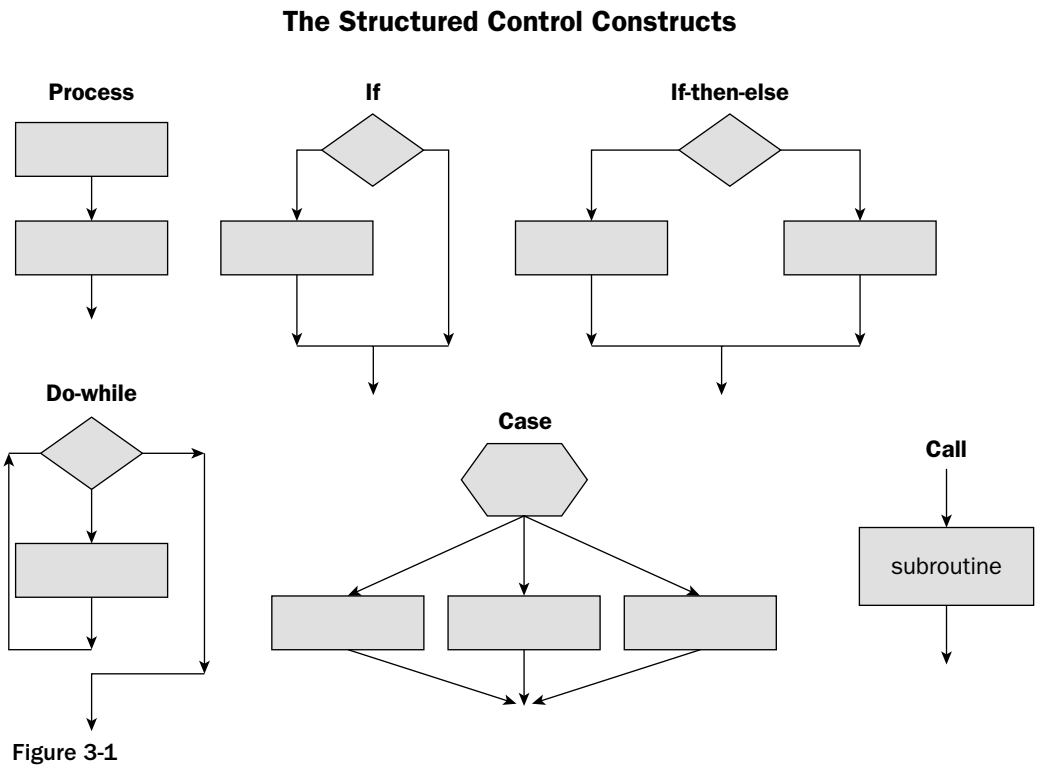

Figure 3-1

IF Statements

`if` statements express conditional logic. Depending on the evaluation of some condition, a different branch of program logic executes. `if` statements are common to nearly all programming languages, and they represent the basic structured instruction for conditional logic. The two basic formats of the Rexx `if` instruction are:

```
IF  expression  THEN  instruction
```

and

```
IF  expression  THEN  instruction  ELSE  instruction
```

Rexx evaluates the *expression* to `1` if it is `TRUE`, and `0` if it is `FALSE`. Here are sample `if` statements:

```
/* A simple IF statement with no ELSE clause */

if  input = 'YES' then
   say 'You are very agreeable'

/* In this example the IF statement tests a two-part or "compound" condition. The
   SAY instruction executes only if BOTH conditions are TRUE, because of the
   AND (&) operator */

if input = 'YES' &  second_input = 'YES' then
   say 'You are doubly agreeable today'

/* This compound IF is true if EITHER of the two expressions are TRUE */

if input = 'YES' |  second_input = 'YES' then
   say 'You are singly agreeable today'

/* Here's a simple IF statement with an ELSE clause.
   The DATATYPE function verifies whether the variable INPUT contains a NUMBER */

if  datatype(input,N) then
   say 'Your input was a number'
else
   say 'Your input was not numeric'

/* This coding is NOT recommended in Rexx, though it is popular in languages
   like C or C++ or many Unix shell languages...
   Variable VAR must be exactly 1 or 0  -- or else a syntax error will occur! */

if (var) then
   say 'VAR evaluated to 1'
else
   say 'VAR evaluated to 0'
```

To execute more than a single instruction after either the `then` or `else` keywords, you *must* insert the multiple instructions between the keywords `do` and `end`. Here is an example:

```
if datatype(input,N) then do
   say 'The input was a number'
   status_record = 'VALID'
   end
else do
   say 'The input was NOT a number'
   status_record = 'INVALID'
end
```

The `do-end` pair groups multiple instructions. This is required when you encode more than one instruction as a logic branch in an `if` instruction. Notice that you must use the `do-end` pair for either branch of the `if` instruction when it executes more than a single statement. In other words, use the `do-end` pair to group more than a single instruction on either the `then` or the `else` branches of the `if` instruction.

You can nest `if` statements, one inside of another. If you nest `if` statements very deeply, it becomes confusing as to which `else` clause matches which `if` instruction. *The important rule to remember is that an else clause is always matched to the nearest unmatched if.* Rexx ignores indentation, so how you indent nested `if` statements has no effect on how Rexx interprets them.

The following code includes comments that show where the programmer sees the end of each `if` instruction. He or she includes these for documentation purposes only, since Rexx ignores comments (regardless of what the comments may say).

```
if age => 70 then
   say 'Person MUST start taking mandatory IRA distributions'
else
   if age >= 65 then
      say 'Person can receive maximum Social Security benefits'
   else
      if  age >= 62 then
         say 'Person may elect reduced Social Security benefits'
      else
         say 'Person is a worker bee, not a beneficiary'
      /* end-if */
   /* end-if */
/* end-if */
```

Here's another style in which to code this example. This series of nested `if` statements is sometimes referred to as an *if-else-if ladder*. The first logic branch that evaluates to `TRUE` executes:

```
if age => 70 then
   say 'Person MUST start taking mandatory IRA distributions'
else if age >= 65 then
   say 'Person can receive maximum Social Security benefits'
else if  age >= 62 then
   say 'Person may elect reduced Social Security benefits'
```

Some languages provide special keywords for this situation, but Rexx does not. (For example, some Unix shell languages provide the `elif` keyword to represent Rexx's `else if` pair). Remember to code a `do – end` pair whenever more than one instruction executes within a branch of the `if` instruction.

The if-else-if ladder embodies another structured construct often referred to the *CASE construct*. In a CASE construct, a set of conditions are tested, then one logic branch is selected from among several.

Rexx provides the `select` instruction to create CASE logic, as will be explained later. In Rexx you can either choose an if-else-if ladder or the `select` instruction to encode CASE logic.

Sometimes, you'll encounter a coding situation where you want to code a logic branch that performs no action. In this case, code the Rexx `nop` instruction. "nop" is a traditional computer science abbreviation or term that means "no operation." The `nop` instruction is a placeholder that results in no action. Here is an example. The `nop` instruction in this code ensures that no action is taken when the `if` statement condition evaluates to `TRUE`:

```
if case_is_undetermined = 'Y' then
   nop      /* No action is taken here. NOP is a placeholder only. */
else do
   say 'Case action completed'
   status_msg = 'Case action completed'
end
```

DO Statements

The `do` instruction groups statements together and optionally executes them repetitively. It comes in several forms, all of which we'll explore in this chapter. `do` instructions permit repetitive execution of one or more statements. They are the basic way you code program "loops" in Rexx.

You are already familiar with the simple `do-end` pair used to group multiple instructions. Here is the generic representation of how this is coded:

```
DO
    instruction_list
END
```

Use the `do-end` group when you must execute multiple instructions in a branch of the `if` instruction, for example. Here's another form of the `do` instruction that repetitively executes a group of instructions while the condition in the `expression` is `TRUE`:

```
DO  WHILE expression
     instruction_list
END
```

This coding example shows how to use this generic format. It employs a `do while` to call a subroutine exactly 10 times:

```
j = 1
do while j <= 10
   call sub_routine
   j = j + 1
end
```

The do instruction is flexible and offers other formats for devising loops. The preceding loop could also be coded with a simpler form of the do instruction:

```
do  10
   call sub_routine
end
```

Or, the example could be coded using a *controlled repetitive loop*:

```
do  j = 1  to  10  by 1
   call sub_routine
end
```

The phrase by 1 is unnecessary because Rexx automatically increases the do loop *control variable* by 1 if this phrase is not coded. But the keyword by could be useful in situations where you want to increase the *loop counter* by some other value:

```
do j = 1 to 20 by 2
   call sub_routine
end
```

In addition to the to and by keywords, for may be used establish another limit on the loop's execution if some other condition does not terminate it first. for is like to, in that Rexx checks it prior to each iteration through the loop. to, by, and for may be coded in any order. In this example, the for keyword limits the do loop to three executions:

```
do j = 1 to 100 by 1 for 3
   say 'Loop executed:'   j  'times.'  /* Ends with: 'Loop executed: 3 times.' */
end
```

You may alter the loop control variable yourself, directly, while inside of the do loop, but this is not a recommended programming practice. It is confusing, and there is always an alternative way to handle such a situation from the logical standpoint. We recommend always using the loop control variable only for controlling an individual loop, and only altering that variable's value through the do instruction condition test.

Rexx also contains unstructured loop control instructions such as leave, iterate, and signal, which we cover later in the section of this chapter on unstructured control constructs. At that time we also cover the do until and do forever forms of do loops, which also fall outside the rules of structured programming.

A Sample Program

This program prompts the user to input a series of words, one at a time. The program identifies words that are four characters long, and concatenates them into a list, which it then displays. The program illustrates a basic do loop, using it to read input from the user. It also shows how to use the if instruction in determining the lengths of the words the user enters.

If you were to enter this sentence to the program (one word at a time):

```
now is the time for all good men to come to the aid of their country
```

the program's output would be:

```
Four letter words:  time good come
```

Here's the sample program:

```
/*  FOUR LETTER WORDS:                                                  */
/*                                                                      */
/*     This program identifies all four letter words in the             */
/*     input and places them into an output list.                       */

four_letter_words  = ''      /* initialize to no 4 letter words found yet */

say "Enter a word: "        /* prompt user to enter 1 word              */
parse pull wordin .       /* the period ensures only 1 word is read in  */
do while wordin \= ''
   if length(wordin) = 4  then
      four_letter_words  = four_letter_words  wordin

   say "Enter a word: "     /* read the next word in                    */
   parse pull wordin .
end

say 'Four letter words:'  four_letter_words       /* display output     */
```

The `do while` loop in this script provides the control structure for the program to prompt the user and read one word after that prompt. The `do while` loop terminates when the user declines to enter a word—after the user just presses the `<ENTER>` key in response to the program's prompt to `Enter a word:` When the user presses the `<ENTER>` key without entering a word, this statement recognizes that fact and terminates the `do while` loop:

```
do while wordin \= ''
```

Recall that the `pull` instruction reads an input and automatically translates it to uppercase. This program uses `parse pull` to read an input *without* the automatic translation to uppercase:

```
parse pull wordin .
```

The period ensures that only the first word is accepted should the user enter more than one. This use of the period is a convention in Rexx, and it's about the only example of *syntax-based coding* in the entire language. You could achieve the same effect by coding:

```
parse pull wordin junk
```

The first word entered by the user is parsed into the variable `wordin`, while any remaining words entered on the input line would be placed into the variable named `junk`.

The program uses the `length` function to determine whether the word the user entered contains four letters. If so, the next statement concatenates the four letter word into a list it builds in the variable named `four_letter_words`.

```
if length(wordin) = 4  then
   four_letter_words  = four_letter_words  wordin
```

The assignment statement relies on the fact that Rexx automatically concatenates variables placed in the same statement, with one space between each. An alternative would have been to use the explicit concatenation operator:

```
four_letter_words  = four_letter_words  ||  wordin
```

But in this case the output would have been:

```
Four letter words:  timegoodcome
```

Explicit concatenation requires explicitly splicing in a blank to achieve properly spaced output:

```
four_letter_words  = four_letter_words  ||  ' '   ||   wordin
```

After the user is done entering words, the program displays the output string through the following statement. Since this is the last statement coded in the program, the script terminates after issuing it:

```
say 'Four letter words:'  four_letter_words       /* display output       */
```

SELECT Statements

The CASE construct tests a series of conditions and executes the set of instructions for the first condition that is `TRUE`. Rexx implements the CASE construct through its `select` instruction. The `select` instruction tests expressions and executes the logic branch of the first one that evaluates to `TRUE`. Here is the generic format of the `select` instruction:

```
        SELECT  when_list   [ OTHERWISE  instruction_list ]  END
```

The `otherwise` branch of the `select` instruction executes if none of the prior `when_list` conditions are found to be `TRUE`. Note that it is possible to code a `select` instruction without an `otherwise` keyword, but if none of the `when_list` conditions execute, an error results. We strongly recommend coding an `otherwise` section on every `select` statement.

The Rexx `select` instruction provides more control than the same CASE construct in some other programming languages because you can encode any expression in the `when` clause. Some languages only permit testing the value of a specified variable.

Here's a simple coding example using `select`:

```
select
   when gender = 'M' then
      say 'Gender is male'
```

```
      when gender = 'F' then do
         say 'Gender is female'
         female_count = female_count + 1
         end
      otherwise
         say 'Error -- Gender is missing or invalid'
         say 'Please check input record'
   end   /* this END pairs with the SELECT instruction itself */
```

If the value in the variable `gender` equals the character `M`, the first logic branch executes. If the value is `F`, the group of instructions associated with the second `when` clause runs. If neither case is true, then the instructions following the `otherwise` keyword execute.

Notice that an `instruction_list` follows the `otherwise` keyword, so if you code more than one statement here you do not need to insert them in a `do-end` pair. Contrast this to the `when` groups, which *do* require a `do-end` pair if they contain more than a single instruction. Don't forget to encode the final `end` keyword to terminate the `select` statement.

CALL Statements

All programming languages provide a mechanism to invoke other scripts or routines. This allows one script, referred to as the *caller*, to run another, the *subroutine*. Rexx's `call` instruction invokes a subroutine, where the subroutine may be one of three kinds:

- *Internal* — Consists of Rexx code residing in the same file as the caller.
- *Built-in* — One of the Rexx built-in functions.
- *External* — Code residing in a different file than the invoking script. An external subroutine may be another Rexx script, or it may be written in any language supporting Rexx's interface.

The subroutine may optionally return one value to the caller through the Rexx *special variable* named `result`. (Rexx has only a handful of special variables and `result` is one of them). Of course, you can have the subroutine send back one or more results by changing the values of variables it has access to. We'll explore all the ways in which caller and subroutines or functions can communicate in detail in Chapter 8, which is on subroutines and modularity. For now, we'll just focus our discussion on the `call` instruction.

Subroutines and *functions* are very similar in Rexx. The one difference is that a function *must* return a value to the caller by its `return` instruction, where a subroutine may elect do so.

The following sample program illustrates the `call` instruction by invoking an internal routine as a subroutine. The subroutine is considered *internal* because its code resides in the same file as that of the program that calls it. The program subroutine squares a number and returns the result.

The main program reads one input number as a *command-line argument* or *input parameter*. To run the program and get the square of four, for example, you enter this line to specify the command-line argument:

```
square.rexx  4
```

Or, you may start the program by entering a line like this:

```
regina  square  4
```

Recall that the first example given earlier *implicitly* invokes the Rexx interpreter, while the second example *explicitly* invokes it. The command-line argument follows the name of the Rexx script you want to run. Here it's a single value, `4`, but other programs might have either many or no command-line arguments.

The program responds to either of the above commands with:

```
You entered: 4   Squared it is: 16
```

Here's the program code:

```
/*  SQUARE:                                                          */
/*                                                                   */
/*     Squares a number by calling an internal subroutine            */

arg number_in .                 /* retrieve the command-line argument    */

call square_the_number number_in
say 'You entered:' number_in '  Squared it is:' result

exit 0


/*  SQUARE_THE_NUMBER:                                               */
/*                                                                   */
/*     Squares the number and RETURNs it into RESULT                 */

square_the_number: procedure

   arg the_number
   return  the_number * the_number
```

The main program or *driver* uses the `arg` instruction to read the command-line argument into variable `number_in`. As with the `pull` and `parse pull` instructions, encode a period (`.`) at the end of this statement to eliminate any extraneous input:

```
arg number_in .                 /* retrieve the command-line argument */
```

The `call` instruction names the internal routine to invoke and passes the variable `number_in` to that routine as its input. The subroutine uses the `arg` instruction to read this parameter (exactly as the main routine did). Here is the encoding of the `call` instruction. The first parameter names the subroutine or function to run, while each subsequent parameter is an input argument sent to the subroutine. In this case, the `call` instruction passes a single argument named `number_in` to the subroutine named `square_the_number`:

```
call square_the_number number_in
```

The first line of the subroutine identifies it as the routine named `square_the_number`. Notice that a colon follows its name on the first line of the subroutine — this identifies a *label* in Rexx. An internal

subroutine starts with the routine's name in the form of a label. The `procedure` instruction on the first line of the subroutine ensures that only the arguments passed to the subroutine will be accessable from within it. No other variables of the calling routine are viewable or changeable by this subroutine. Here is the first executable line of the subroutine:

```
square_the_number: procedure
```

The subroutine reads the number passed into it from its caller by the `arg` instruction. Then, the subroutine returns a single result through its `return` instruction. Here is how this line is encoded. Notice that Rexx evaluates the expression (squaring the number) before executing the `return` instruction:

```
return  the_number * the_number
```

The caller picks up this returned value through the *special variable* named `result`. The main routine displays the squared result to the user through this concatenated display statement:

```
say 'You entered:' number_in '  Squared it is:' result
```

This displays an output similar to this to the user:

```
You entered: 2   Squared it is: 4
```

The driver ends with the instruction `exit  0`. This unconditionally ends the script with a *return code*, or returned value, of `0`. The last statement of the internal subroutine was a `return` instruction. `return` passes control back to the calling routine, in this case passing back the squared number. If the subroutine is a function, a `return` instruction is required to pass back a value.

There is much more to say about subroutines and modular program design. We leave that discussion to Chapter 8. For now, this simple script illustrates the structured CALL construct and how it can be used to invoke a subroutine or function.

Another Sample Program

Here's a program that shows how to build menus and call subroutines based on user input. This program is a fragment of a real production program, slimmed down and simplified for clarity. The script illustrates several instructions, including `do` and `select`. It also provides another example of how to invoke internal subroutines.

The basic idea of the program is that it displays a menu of transaction options to the user. The user picks which transaction to execute. The program then executes that transaction and returns to the user with the menu. Here is how it starts. The program clears the screen and displays a menu of options to the user that looks like this:

```
Select the transaction type by abbreviation:

   Insert = I
   Update = U
   Delete = D
     Exit = X

Your choice => _
```

Based on the user's input, the program then calls the appropriate internal subroutine to perform an Insert, Update, or Delete transaction. (In the example, these routines are "dummied out" and all each really does is display a message that the subroutine was entered). The menu reappears until the user finally exits by entering the menu option 'x'.

Here's the complete program:

```
/*  MENU:                                                          */
/*                                                                 */
/*     This program display a menu and performs updates based      */
/*     on the transaction the user selects.                        */

'cls'                            /* clear the screen (Windows only)  */
tran_type = ''
do while tran_type \= 'X'        /* do until user enters 'X'         */
   say
   say 'Select the transation type by abbreviation:'
   say
   say '   Insert = I '
   say '   Update = U '
   say '   Delete = D '
   say '     Exit = X '
   say
   say 'Your choice => '
   pull tran_type .
   select
      when tran_type = 'I' then
         call insert_routine
      when tran_type = 'U' then
         call update_routine
      when tran_type = 'D' then
         call delete_routine
      when tran_type = 'X' then do
         say
         say 'Bye!'
         end
      otherwise
         say
         say 'You entered invalid transaction type:' tran_type
         say 'Press <ENTER> to reenter the transaction type.'
         pull .
      end
end
exit 0


/* INSERT_ROUTINE goes here                                        */
INSERT_ROUTINE: procedure
   say 'Insert Routine was executed'
   return 0

/* UDPATE_ROUTINE goes here                                        */
UPDATE_ROUTINE: procedure
```

```
   say 'Update Routine was executed'
   return 0

/* DELETE_ROUTINE goes here                                          */
DELETE_ROUTINE: procedure
   say 'Delete Routine was executed'
   return 0
```

The first executable line in the program is this:

```
'cls'                              /* clear the screen (Windows only)  */
```

When the Rexx interpreter does not recognize a statement as part of the Rexx language, it assumes that it is an operating system command and passes it to the operating system for execution. Since there is no such command as `cls` in the Rexx language, the interpreter passes the string `cls` to the operating system for execution as an operating system command.

`cls` is the Windows command to "clear the screen," so what this statement does is send a command to Windows to clear the display screen. Of course, this statement makes this program *operating-system-dependent*. To run this program under Linux or Unix, this statement should contain the equivalent command to clear the screen under these operating systems, which is `clear`:

```
'clear'                            /* clear the screen (Linux/Unix only) */
```

Passing commands to the operating system (or other external environments) is an important Rexx feature. It provides a lot of power and, as you can see, is very easy to code. Chapter 14 covers this topic in detail.

Next in the program, a series of `say` commands paints the menu on the user's screen:

```
say
say 'Select the transation type by abbreviation:'
say
say '   Insert = I '
say '   Update = U '
say '   Delete = D '
say '     Exit = X '
say
say 'Your choice => '
```

A `say` instruction with no operand just displays a blank line and can be used for vertically spacing the output on the user's display screen.

The script displays the menu repeatedly until the user finally enters '`x`' or '`X`'. The `pull` command's automatic translation of user input to uppercase is handy here and eliminates the need for the programmer to worry about the case in which the user enters a letter.

The `select` construct leads to a `call` of the proper internal routine to handle the transaction the user selects:

```
select
      when tran_type = 'I' then
         call insert_routine
      when tran_type = 'U' then
         call update_routine
      when tran_type = 'D' then
         call delete_routine
      when tran_type = 'X' then do
         say
         say 'Bye!'
         end
      otherwise
         say
         say 'You entered invalid transaction type:' tran_type
         say 'Press <ENTER> to reenter the transaction type.'
         pull .
      end
```

The `when` clause where the user enters 'x' or 'X' encloses its multiple instructions within a `do-end` pair. The `otherwise` clause handles the case where the user inputs an invalid character. The final `end` in the code concludes the `select` instruction.

Remember that the logic of the `select` statement is that the first condition that evaluates to `TRUE` is the branch that executes. In the preceding code, this means that the program will call the proper subroutine based on the transaction code the user enters.

Following the `select` instruction, the code for the main routine or driver ends with an `exit 0` statement:

```
exit 0
```

This delimits the code of the main routine from that of the routines that follow it and also sends a return code of `0` to the environment when the script ends. An `exit` instruction is required to separate the code of the main routine from the subroutines or functions that follow it.

The three update routines contain no real code. Each just displays a message that it ran. This allows the user to verify that the script is working. These subroutines cannot access any variables within the main routine, because they have the `procedure` instruction, and no variables are passed into them. Each ends with a `return 0` instruction:

```
return 0
```

While this sample script is simple, it shows how to code a menu for user selection. It also illustrates calling subroutines to perform tasks. This is a nice modular structure that you can expand when coding menus with pick lists. Of course, many programs require graphical user interfaces, or GUIs. There are a variety of free and open-source GUI interfaces available for Rexx scripting. GUI programming is an advanced topic we'll get to in a bit. Chapter 16 shows how to program GUIs with Rexx scripts.

Unstructured Control Instructions

Rexx is a great language for structured programming. It supports all the constructs required and makes structured programming easy. But the language is powerful and flexible, and there are times when unstructured flow of control is necessary (or at least highly convenient). Here are the unstructured instructions that alter program flow in Rexx:

Instruction	Use
`do until`	A form of the `do` instruction that implements a *bottom-drive loop*. Unlike `do-while`, `do-until` will always execute the code in the loop at least one time, because the condition test occurs at the bottom of the loop.
`do forever`	Creates an *endless loop*, a loop that executes forever. This requires an *unstructured exit* to terminate the loop. Code the unstructured exit by either the `leave`, `signal` or `exit` instruction.
`iterate`	Causes control to be passed from the current statement in the `do` loop to the bottom of the loop.
`leave`	Causes an immediate exit from a `do` loop to the statement following the loop.
`signal`	Used to trap *exceptions* (specific program error conditions). Can also be used to unconditionally transfer control to a specified label, similarly to the `GOTO` instruction in other programming languages.

Figure 3-2 below illustrates the unstructured control constructs.

The `do until` and `do forever` are two more forms of the `do` instruction. `do until` implements a bottom-driven loop. Such a loop always executes at least one time. In contrast, the `do while` checks the condition *prior* to entering the loop, so the loop may or may not be executed at least once. `do until` is considered unstructured and the `do while` is preferred. Any logic that can be encoded using `do until` can be coded using `do while` — you just have to think for a moment to see how to change the logic into a `do while`.

Let's look at the difference between `do while` and `do until`. This code will not enter the `do` loop to display the message. The `do while` tests the condition prior to executing the loop, so the loop never executes. The result in this example is that the `say` instruction never executes and does not display the message:

```
ex = 'NO'
do while ex = 'YES'
   say 'Loop 1 was entered'    /* This line is never displayed.     */
   ex = 'YES'
end
```

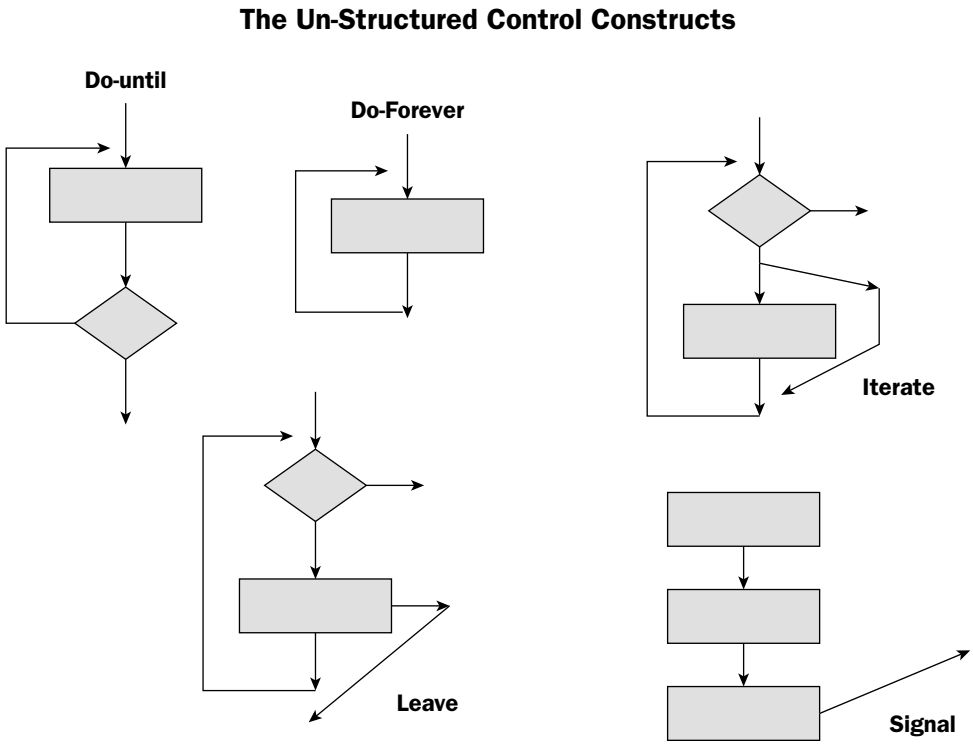

Figure 3-2

If we replace the `do while` loop with a `do until` loop, the code will execute through the loop one time, printing the message once. This is because the condition test is applied only at the bottom of the loop. A `do until` loop will always execute one time, even if the condition test on the `do until` is false, because the test is not evaluated until after the loop executes one time. The result in this example is that the `say` instruction executes once and displays one output line:

```
ex = 'NO'
do until ex = 'YES'
   say 'Loop 2 was entered'    /* This line is displayed one time. */
   ex = 'YES'
end
```

`do forever` creates an *endless loop*. You *must* have some unstructured exit from within the loop or your program will never stop! This example uses the `leave` instruction to exit the endless loop when `j = 4`. The `leave` instruction transfers control to the statement immediately following the `end` that terminates the `do` loop. In this example, it breaks the endless loop and transfers control to the `say` statement immediately following the loop:

```
j = 1
do forever
   /* do some work here */
   j = j + 1
   if j = 4 then leave        /* exits the DO FOREVER loop */
```

```
end
say 'The above LEAVE instruction transfers control to this statement'
```

Another way to terminate the endless loop is to encode the `exit` instruction. `exit` ends a program unconditionally (even if a subroutine is executing or if execution is nested inside of a `do` loop). Control returns to the environment (the operating system) with the optional string encoded on the `exit` statement passed up.

What return code you can pass to the environment or operating system using the `exit` instruction depends on what that operating system accepts. Some systems accept only return codes that are numeric digits between 0 and 127. If your script returns any other string, it is translated into a 0. Other operating systems will accept whatever value you encode on the `exit` instruction.

Here's an example. The following code snippet is the same as the previous one, which illustrates the `leave` instruction, but this time when the condition `j = 4` is attained, the script unconditionally exits and returns 0 to the environment. Since the script ends, the `say` instruction following the `do forever` loop never executes and does not display its output:

```
j = 1
do forever
   /* do some work here */
   j = j + 1
   if j = 4 then
      exit  0   /* unconditionally exits and passes '0' back to the environment */
end
say 'this line will never be displayed'   /* code EXITs, never reaches this line
*/
```

Another instruction for the unstructured transfer of control is `signal`. The `signal` instruction acts much like the `GOTO` statement of other programming languages. It transfers control directly out of any loop, CASE structure, or `if` statement directly to a Rexx label. A *label* is simply a symbol immediately followed by a colon. This sample code snippet is similar to that we've seen earlier, except that this time the `signal` instruction transfers control to a program label. So, once `j = 4` and the `signal` instruction execute, control is transferred to the program label and the `say` instruction displays its output line:

```
j = 1
do forever
   /* do some work here */
   j = j + 1
   if j = 4 then
      signal  my_routine   /* unconditionally go to the label MY_ROUTINE */
end

/*  other code here gets skipped by the SIGNAL instruction */

my_routine:
   say 'SIGNAL instruction was executed, MY_ROUTINE entered...'
```

`signal` differs from the `GOTO` of some other languages in that it terminates all active control structures in which it is encoded. You could not transfer control to another point in a loop using it, for example.

Duplicate labels are allowed within Rexx scripts, but control will always be transferred to the one that occurs first. We recommend that all labels in a program be unique within a program for the sake of readability.

In an entirely different role, the `signal` instruction is also used to capture or "trap" errors and special conditions. Chapter 10 discusses this in detail. This is a special mechanism within the Rexx language designed to capture unusual error conditions or "exceptions."

The last unstructured instruction to discuss is the `iterate` instruction. The `iterate` instruction causes control to be passed from the current statement in the `do` loop to the bottom of the loop. In this example, the `iterate` instruction ensures that the `say` instruction never executes. The `if` instruction's condition evaluates to `TRUE` every time that statement is encountered, so the `iterate` instruction reruns the `do` loop and the `say` instruction never runs:

```
j = 1
do until j = 4
   /* do some work here */
   j = j + 1
   if j > 1 then iterate
      say 'This line is never displayed!'  /* this line will never execute */
end
```

Summary

This chapter summarizes Rexx's structured and unstructured control constructs. These include the `if`, `do`, `select`, `call`, `exit`, and `return` instructions for structured programming, and the unstructured `iterate`, `leave`, and `signal` instructions. The `do until` and `do forever` forms of the `do` instruction are also unstructured.

For more background on why structured programming is beneficial -- especially for larger and more complicated programs -- refer to the works of Edward Yourdon and Edsger Dijkstra.

Use the instructions this chapter covers to direct conditional logic as in most other programming languages. This chapter presented many small code snippets to illustrate how to use the instructions that control program logic. Subsequent chapters will provide many more examples of the use of these instructions. These upcoming examples demonstrate the instructions in the more realistic context of complete programs. They will make the use of the instructions for the control of logical flow much clearer.

Test Your Understanding

1. Why is structured programming recommended? What Rexx instructions implement structured programming? How do subroutines and functions support the benefits of structured programming?
2. How does Rexx determine which `if` instruction each `else` keyword pairs with?
3. Name two ways that a script can test for the end of user input.
4. What are the differences between *built-in*, *internal*, and *external* subroutines? What is the difference between a *function* and a *subroutine*?

5. What are the values of `TRUE` and `FALSE` in Rexx?
6. What is the danger in coding a `do forever` loop? How does one address this danger?
7. What are the two main functions of the `signal` instruction? How does the `signal` instruction differ from the `GOTO` command of many other programming languages?
8. What is the difference between the `do-while` and `do-until` instructions? Why use one versus the other? Are both allowed in structured programming?

4

Arrays

Overview

Every programming language provides for *arrays*. Sometimes they are referred to as *tables*. This basic data structure allows you to build lists of "like elements," which can be stored, searched, sorted, and manipulated by other basic programming operations.

You'll sometimes hear arrays referred to as *compound variables* or *stem variables* in Rexx. We'll explain why in just a moment.

Rexx's implementation of arrays is powerful but easy to use. Arrays can be of any *dimensionality*. They can be a one-dimensional *list*, where all elements in the array are of the same kind. They can be of two dimensions, where there exist pairs of entries. In this case, elements are manipulated by two subscripts (such as I and J). Or, arrays can be of as many dimensions as you like. While Rexx implementations vary, the usual constraint on the size and dimensionality of array is memory. This contrasts with other programming languages that have specific, language-related limitations on array size.

Rexx arrays may be *sparse*. That is, not every array position must have a value or even be initialized. There can be empty array positions, or *slots*, between those that do contain data elements. Or arrays can be *dense*, in which consecutive array slots all contain data elements. Figure 4-1 below pictorially shows the difference between sparse and dense arrays. Dense arrays are also sometimes called *nonsparse* arrays.

Arrays may be initialized by a single assignment statement. But just like other variables, arrays are defined by their first use. You do not have to predefine or preallocate them. Nor must you declare a maximum size for an array. The only limitation on array size in most Rexx implementations is imposed by the amount of machine memory.

Dense versus Sparse Arrays

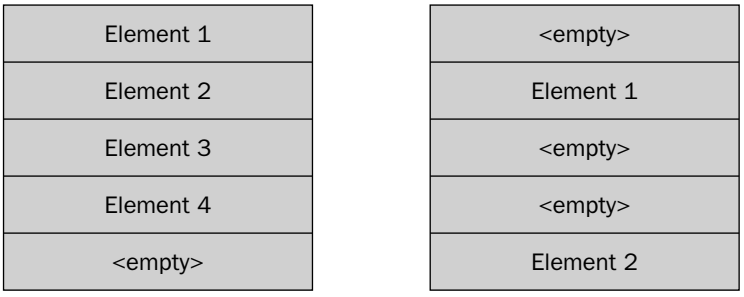

Figure 4-1

You can refer to individual elements within a Rexx arrray by numeric subscripts, as you do in other programming languages. Or, you can refer to array elements by variables that contain character strings. Rexx then uses those character strings as indexes into the array. For this reason, Rexx arrays are sometimes termed *content addressable*. They can be used as a form of *associative memory,* in that they create an association between two values. This permits innovative use of arrays in problem solving. We'll explain what these terms mean and why are they important in more detail later in the chapter. We'll even give several examples of how content addressable memory structures are useful in resolving programming problems. For now, remember that the subscripts you encode to access individual array elements can be either numeric or string variables.

Like many scripting languages, Rexx lacks complex data structures such as lists, trees, records, and the like. These are unnecessary because by understanding content-addressable arrays it is easy to build these structures. Rexx arrays provide the foundation to build any imaginable data structure. We'll show you how later in this chapter. First, let's explore the basics of how to code arrays and process their data elements.

The Basics

To approach the subject of arrays, let's review the way variable names are created. The basis for Rexx arrays are compound variable names or *symbols*. So far we've seen several kinds of symbols within Rexx:

- *Constants* — Literal strings or other values that cannot be changed.
- *Simple symbols* — Variable names that do not begin with a digit and do not contain any embedded period(s).
- *Compound symbols* — The basis for arrays. Like simple symbols, they do not begin with a digit. However, they contain one or more periods.

Simple symbols are synonymous with variable names, as we have known them thus far, while *compound symbols* contain one or more periods. Compound symbols are the basis for arrays.

In compound symbols, the *stem* is the part of the name up to and including the first period. The stem is sometimes called a *stem variable*. The *tail* comprises one or more symbols separated by periods.

Here are a few examples:

- `list.j` — `list` is the name of an array or table.
- `list.j` — `list.` is the stem of the array. Note that the stem name includes the period.
- `books.j.k` — `books.` is the stem, `j.k` is the tail. `j` and `k` are two subscripts.

In these examples, Rexx substitutes in the value of the variables `j` and `k` before referencing into the arrays. These values can be numbers, but they do not have to be. Rexx allows indexing into an array based on any variable value you encode, whether it is numeric or a string value.

Here is a sample series of statements that refer to an array element based on a string value in a variable. The first line below initializes all elements in an array to the null string (represented by two back-to-back quotation marks). The second line assigns a value to a specific array element. The last two statements show how a character string provides a subscript into the array to retrieve that data element from the array:

```
fruit. = ''                 /* initialize all array elements to the null string */
fruit.cherry = 'Tasty!'     /* set the value of an array element                */
subscript_string = cherry   /* establish an index into the array                */
say fruit.subscript_string  /* displays: Tasty!                                 */
```

It is probably worth noting that Rexx uppercases the string `cherry` into `CHERRY` in the subscript assignment statement above because that character string is not enclosed in quotation marks. Rexx also uppercases variable names such as `fruit.cherry` into `FRUIT.CHERRY` internally. Had we coded `subscript_string = 'cherry'` as the third line in the sample code, it would not work properly. The array tail is uppercased internally by Rexx so the subscript string used for finding the data element must also be uppercase.

What happens if you accidentally reference an array with a subscript that is not yet initialized? Recall that in Rexx an uninitialized variable is always its own name in uppercase. So, if `my_index` has not yet been assigned a value, `my_array.my_index` resolves to `MY_ARRAY.MY_INDEX`. Oops! This is probably not what the programmer intended.

Initialize the array as a whole by referring to its stem. The dimensionality of the array does not matter to this operation. We saw one example of initializing an entire array in one line of the sample code. Here are some more examples:

```
list. = 0    /* initialize all possible entries in the array LIST to 0        */
books. = ''  /* initialize all possible entries in BOOKS array to null string */
```

You *cannot* perform other kinds of operations on entire arrays by single statements — in most Rexx implementations. For example, these statements *are invalid* and result in errors:

```
numbers. = numbers. + 5   /* add 5 to each entry in the NUMBERS array */
lista. = listb.        /* move all contents of array LISTB
                             into the array LISTA */
```

To process all the elements in an array, use a `do` loop. This works as long as the array is indexed or subscripted by numeric values, and each position, or slot, in the array contains a value. To process all the elements in the array, you must keep track of the maximum subscript you use. There is no Rexx function that returns the largest numeric subscript you've used for an array. Here is an example that shows how to process all the elements of an array. In this code, each contiguous array position contains an element, and the array subscript is numeric:

```
array_name. = ''        /* initialize all elements to some nonoccurring value   */

number_of_elements = 5  /* initialize to the number of elements in the array    */

/*  place elements into the array here */

/*  This code processes all elements in the array. */
do  j = 1  to number_of_elements
   say  "Here's an array element:" array_name.j
end
```

Another technique for array processing is to initialize the array to zeroes for numeric values, or to the empty string or *null string* for character string entries (represented by two back-to-back quotation marks `''`). Then process the array starting at the beginning until you encounter an entry set to the initialization value. Here's sample code that processes all elements of an array based on this approach:

```
array_name. = ''   /* initialize all array elements to some nonoccurring value */

/*  place elements into the array here */

/*  This code processes all elements in the array. */
do  j = 1  while array_name.j <> ''
   say  "Here's an array element:" array_name.j
end
```

If you take this approach, be sure that the value used for initialization never occurs in the data you place into the array!

This approach also assumes a *nonsparse*, or *dense*, *array* — one in which the positions in the array have been filled consecutively without skipping array slots or positions. For a sparse array, we recommend storing the largest numeric subscript you use in a variable for future reference. Obviously, you cannot simply process a sparse array until you encounter the array initialization value because some positions within the array may not contain data items. In processing a sparse array, your code will have to be able

to distinguish between array positions that contain valid values and those that do not. For this reason, it is useful to initialize all sparse array elements to some unused default value (such as the null string or zeroes) prior to using the array.

In many programming languages, you must be concerned with what the subscript of the first entry in a table is. Is the first numeric subscript 0 or 1? In Rexx, the first subscript is whatever you use! So, input the first array element into position 0 or 1 as you prefer:

```
array_name.0  = 'first element'
```

or

```
array_name.1 = 'first element'
```

Just be sure that whatever choice you make you remember and that you remain consistent in your approach. This flexibility is a handy feature of content-addressable arrays.

As an informal convention, many Rexx programmers store the number of array elements in position 0, then start storing data elements in position 1:

```
array_name.0  = 3       /* store number of elements in the array here */
array_name.1 = 'first element'
array_name.2 = 'second element'
array_name.3 = 'last element'
```

Assuming that the array is not sparse and the index is numeric, process the entire array with code like this:

```
do  j = 1  to array_name.0
   say  "Here's an array element:" array_name.j
end
```

Placing the number of array elements into position 0 in the array is not required and is strictly an informal convention to which many Rexx programmers adhere. But it's quite a useful one, and we recommend it.

A Sample Program

This sample program illustrates basic array manipulation. The program defines two arrays. One holds book titles along with three descriptors that describe each book. The other array contains keywords that will be matched against the three descriptors for each book.

The user starts the program and inputs a "weight" as a command line parameter. Then the program lists all books that have a count of descriptors that match a number of keywords at least equal to the weight. This algorithm is called *weighted retrieval*, and it's often used in library searches and by online bibliographic search services.

Here's the entire program. The main concepts to understand in reviewing it are how the two arrays are set up and initialized at the top of the program, and how they are processed in the body. The `do` loops that process array elements are similar to the ones seen previously.

```
/*  FIND BOOKS:                                                    */
/*                                                                 */
/*     This program illustrates basic arrays by retrieving book    */
/*     titles based on keyword weightings.                         */

keyword. = ''         /* initialize both arrays to all null strings  */
title. = ''

/* the array of keywords to search for among the book descriptors    */

keyword.1 = 'earth'   ;   keyword.2 = 'computers'
keyword.3 = 'life'    ;   keyword.4 = 'environment'

/* the array of book titles, each having 3 descriptors               */

title.1 = 'Saving Planet Earth'
  title.1.1 = 'earth'
  title.1.2 = 'environment'
  title.1.3 = 'life'
title.2 = 'Computer Lifeforms'
  title.2.1 = 'life'
  title.2.2 = 'computers'
  title.2.3 = 'intelligence'
title.3 = 'Algorithmic Insanity'
  title.3.1 = 'computers'
  title.3.2 = 'algorithms'
  title.3.3 = 'programming'


arg weight . /* get number keyword matches required for retrieval    */

say 'For weight of' weight 'retrieved titles are:'  /* output header */

do j=1 while title.j <> ''                   /* look at each book    */
   count = 0
   do k=1 while keyword.k <> ''              /* inspect its keywords */
      do l=1 while title.j.l <> ''
         if  keyword.k = title.j.l  then count = count + 1
      end
   end

   if count >= weight then   /* display titles matching the criteria */
      say title.j
end
```

The program shows that you can place more than one Rexx statement on a line by separating the statements with a semicolon. We use this fact to initialize the searchable keywords. Here's an example with two statements on one line:

```
keyword.1 = 'earth'   ;   keyword.2 = 'computers'
```

To implement the weighted-retrieval algorithm, the outermost `do` loop in the script processes each book, one at a time. This loop uses the variable `j` as its subscript:

```
do j=1 while title.j <> ''                    /* look at each book    */
```

The `do` loop could have included the phrase `by 1`, but this is not necessary. Rexx automatically defaults to incrementing the loop counter by 1 for each iteration. If we were to encode this same line and explicitly specify the increment, it would appear like this. Either approach works just fine:

```
do j=1 by 1 while title.j <> ''               /* look at each book    */
```

The loop that processes each book, one at a time, is the outermost loop in the code. The next, inner loop uses `k` as its control variable and processes all the keywords for one book:

```
   do k=1 while keyword.k <> ''               /* inspect its keywords */
```

The innermost loop uses `l` for loop control and inspects the three descriptors for each book title. This code totals how many of each book's descriptors match keywords:

```
      do l=1 while title.j.l <> ''
         if  keyword.k = title.j.l  then count = count + 1
      end
```

If the count or *weight* this loop totals is at least equal to that input as the command line argument, the book matches the retrieval criteria and its title is displayed on the user's screen.

This script is written such that the programmer does not need to keep track of how many variables any of the arrays contain. The `while` keyword processes items in each `do` loop until a null entry (the null string) is encountered. This technique works fine as long as these two conditions pertain:

- The script initializes each array to the null string.
- Each position or slot in the arrays is filled consecutively.

Its approach to array processing makes the program code independent of the number of books and keywords it must process. This flexibility would allow the same algorithm to process input from files, for example. So, it would be easy to eliminate the static assignment statements in this program and replace them with variable input read in from one or more input files. You can see that the approach this script takes to array processing provides great flexibility.