REBOL/Core Users Guide

For REBOL Version 2.3

Copyright 2005 REBOL Technologies

1	Introduction	An introduction to REBOL/Core, information about the manual, technical support information, and where to send comments.
2	Operation	Installing, starting, and quitting REBOL sessions. Using the REBOL Console. Getting help from REBOL. Error messages. How to upgrade REBOL.
3	Quick Tour	A quick overview that describes values, words, blocks, variables, evaluation, functions, paths, objects, scripts, files, and networking.
4	Expressions	How blocks, values, and words are evaluated. Conditional, repeating, and selective expressions. Stopping evaluation. Error recovery.
5	Scripts	Script headers. Command line arguments to scripts. Loading, saving, and commenting scripts. Style guide on how to write good scripts.
6	Series	Series are the foundation of REBOL. Describes series functions and datatypes. Making and copying series. Series iteration. Searching and sorting series. Series as data sets.
7	Block Series	Specifics on block series. Blocks of blocks. Paths for nested blocks. Arrays. Composing blocks.
8	String Series	Special string functions and converting values to strings.
9	Functions	Evaluating functions and arguments. Defining functions. Nested, unnamed, and conditional functions. Function attributes. Scope of variables. Reflective properties. Online help for functions. Viewing function source code.
10	Objects	Making and cloning objects. Accessing objects. Referring to self. Encapsulation. Reflective properties
11	Math	Scalar datatypes. Evaluation order. Standard functions and operators. Type conversion. Comparison. Log, trig, and logic functions.
12	Files	File names and paths. Reading and writing files. Line conversion and blocks of lines. Directory access and directory file functions.
13	Network Protocols	About REBOL networking. Initial setup. DNS, Whois, Finger Daytime, HTTP, SMTP, POP, FTP, NNTP, CGI, TCP, and UDP.
14	Ports	About I/O ports. Opening, reading, writing, closing ports. Updating and waiting on ports. Other port modes. File permissions. Directory ports.
15	Parsing	Spitting strings. Grammar rules. Skipping input. Match types. Recursion and evaluation.
A1	Values	Summary of REBOL datatype values.
A2	Errors	REBOL error messages. Error categories. Catching errors. The error object. User generated errors.
A3	Console	The command prompt. History recall. Busy indicator. Advanced console operations.

1 - Introduction

1. About REBOL
2. About this Guide
    2.1 Suggestion for New Programmers
    2.2 Suggestion for Experienced Programmers
3. Document Conventions
4. Technical Support
    4.1 Developer News and Information
    4.2 Discussions and Forums
    4.3 Bug Reports and Enhancement Requests
    4.4 REBOL.org Script Library
    4.5 New Alpha and Beta Releases
5. Comments are Welcome

1. About REBOL

Here are a few quick facts about REBOL:

- **REBOL stands for Relative Expression-Based Object Language.**
- **REBOL is pronounced "reb-ol"** as in "rebel with a cause".
- **REBOL is a messaging language.** Its main purpose is to provide a better approach to distributed computing and communication.
- **REBOL was designed by Carl Sassenrath**, the operating system architect responsible for the Amiga OS, the world's first multitasking operating system for personal computers.
- **REBOL is more than just a programming language.** It is also a language for representing data and metadata. It provides a single method for the computation, storage, and exchange of information.
- **REBOL code and data span more than 40 system platforms.** A script written on Windows runs equally well on Linux, UNIX, and many other platforms... with no changes necessary.
- **REBOL introduces the concept of dialecting** - small, efficient, domain-specific sublanguages for code, data, and metadata.
- **REBOL implementations are intentionally kept very small** even though they include hundreds of functions, dozens of datatypes, built-in help, multiple Internet protocols, compression, error handling, a debugging console, encryption, and much more.
- **REBOL programs are easy to write**. All you need is a text editor. A program can be a single line or an entire application consisting of dozens of files.
- **REBOL/Core serves as the foundation for all REBOL technology.** While designed to be simple and productive for novices, the language extends new dimensions of power to professionals.

The graphical version of REBOL, called REBOL/View, builds on top of REBOL/Core. It can be found on the REBOL website.

2. About this Guide

This guide provides the basic information necessary for using REBOL/Core. It assumes that the reader is already familiar with general programming and operating system terminology and concepts.

2.1 Suggestion for New Programmers

If you are new to programming, REBOL provides an excellent way to get started.

There are a few general concepts that REBOL uses everywhere. For example, REBOL's concept of a *series* is used in everything from data structures to code blocks. Once you learn the concepts and methods of series, they can be applied throughout your programs. You should learn these concepts well. It will pay off later. The chapters in this users guide are ordered to best help you get started.

If you experience trouble using REBOL, do not get frustrated. There are many people who can help you. The REBOL Mail List (see support section below) is monitored by hundreds of people who enjoy helping beginners get started. Feel free to visit that forum for any reason.

2.2 Suggestion for Experienced Programmers

If you are already familiar with other programming languages like C, C++, Java, Pascal, Python, PERL, BASIC, etc. we should warn you: **REBOL is quite different**.

You should know that REBOL was not designed just to be different, but rather to give programmers more expressive power. Programmers who have mastered REBOL suggest that the best approach is to forget what you already know about other languages. That's because you don't create REBOL programs in the same way. Sure, you can create REBOL programs that look similar to C, but if you do that, you will lose a lot of the advantages that REBOL offers.

In technical terms, REBOL is a highly reflective, functional, symbolic language with definitional scoping rules. If you don't know what this means, don't worry. REBOL takes advantage of advances in computer science, but you don't need to be a computer scientist to use it.

As an experienced programmer, you may be tempted to skip most of the chapters of this guide. For the most part, that is fine. However, concepts like *series* are critical to understanding REBOL. If you don't take the time to master such concepts, you will find it difficult to become truly fluent in the REBOL language.

3. Document Conventions

The following table describes the typographical conventions used in this guide.

Item	Convention	Example
Words that are part of the REBOL language (as function names, special variables, system objects).	Bold, green, monospace	**`Append at change`**
Non-REBOL words and values such as file names, directory names, program names, and variable names.	Green, monospace	`myfile window-color`
Code examples.	Boxed bold monospace	**`do %feedback.r`**
Results printed at the REBOL console.	Boxed blue monospace	`true`

4. Technical Support

For general questions or feedback regarding REBOL products or our website please visit our feedback page. We usually answer messages within 24 to 48 hours. Be sure to include your email address if you want a reply.

4.1 Developer News and Information

The **REBOL Developer Network (www.REBOL.net)** provides the latest technical news, information, documentation, discussions, releases, and more.

This site is also home to Carl's REBOL Blog, an informative archive of insights, ideas, and queries by REBOL's inventor and founder, Carl Sassenrath.

4.2 Discussions and Forums

- **REBOL Mailing List**
  The REBOL email discussion list is a forum for questions and answers about all topics related to REBOL. You can view prior messages at the mail list archive on REBOL.org.

- **REBOL Talk Forum**
  This is an independent web-based forum for open discussions about REBOL.

- **REBOL Google Group**
  This new web/email discussion group has recently been started on Google Groups. This is still in the experimental stages of use.

- **Other Discussions**
  REBOL Technologies also hosts a variety of private discussion groups using our IOS technology and the AltME IM system from Safeworlds Inc. Look for opening announcements and member requests on www.REBOL.net.

4.3 Bug Reports and Enhancement Requests

REBOL customers, developers, and user can now directly search for information related to known problems, report new problems, or request enhancements using our **RAMBO** database.

4.4 REBOL.org Script Library

The site **www.REBOL.org** is a community-operated website that provides an extensive library of REBOL examples. It also includes a variety of tutorials as well as an archive of REBOL Mailing List messages.

4.5 New Alpha and Beta Releases

We publish interim builds for our products on a regular basis. This service is intended for customers and experienced developers only. These pages hold raw builds not distribution packages.

5. Comments are Welcome

To help us with future releases of this documentation we would like to know what corrections or clarifications you would find useful. Send them to the Feedback page on our website. Please include the title, version, and chapter of the guide.

2 - Operation

1. Installing REBOL
    1.1 Distribution Files
    1.2 Home Environment Variable
    1.3 Network Setup
    1.4 Proxy and Firewall Settings
    1.5 License Agreement
2. Starting REBOL
    2.1 From an Icon
    2.2 From a Shell
    2.3 From Another Application
    2.4 Security Issues
    2.5 Program Arguments
    2.6 Script File
    2.7 Specifying Options
    2.8 File Redirection
    2.9 Script Arguments
    2.10 Startup Files
3. Quitting REBOL
4. Using the Console
    4.1 Mulitple Line Input
    4.2 Interrupting a Script
    4.3 History Recall
    4.4 Word Completion
    4.5 Busy Indicator
    4.6 Network Connections
    4.7 Virtual Terminal
5. Getting Help
    5.1 Online Help
    5.2 Viewing Source Code
    5.3 Download Documents
    5.4 Script Library
    5.5 User Mailing List
    5.6 Contacting Us
6. Errors
    6.1 Error Messages
    6.2 Redirecting Errors
7. Upgrading

1. Installing REBOL

REBOL installation takes only a few seconds and is very easy, non-intrusive, and non-disruptive.

For REBOL/Core the only installation procedure is to uncompress the distribution files and store them in any directory on your system.

For other REBOL products, installation may require you to provide additional information, such as where to store related files. Refer to the release notes that are included with the distribution files.

1.1 Distribution Files

REBOL/Core includes these primary distribution files:

rebol (.exe)	An executable program that starts the REBOL console.
rebol.r	A system bootstrap file. (Not required to run REBOL.)

setup.html	Information about set-up and installation.
changes.html	Changes found in recent releases.
license.txt	REBOL license agreement.

Other files may be provided, depending on the REBOL product and version.

1.2 Home Environment Variable

Although it is not required, if your operating system sets the HOME environment variable, REBOL will use it to locate its startup files. On many operating systems such as UNIX or Linux, the HOME variable may be set by default already. (So you do not need to set it yourself).

1.3 Network Setup

The first time you start REBOL, it prompts you for network information. This information is optional. Some protocols, such as email or FTP, require an email address or an email server name. In addition, if you are behind a firewall or use a proxy server, you need to provide specific information to access the Internet.

To set up your network:

1. Type your email address. For example, name@example.com.
2. Type the name of your email server. For example: mail.example.com. Use the name of the email server you normally use. If you are not sure of the name of the server, contact your network administrator or Internet service provider for the name of your SMTP (email) server.
3. Specify whether you use a proxy server. If you are directly connected to the Internet with a modem or ethernet, type N (no). If you use a proxy or firewall, provide the required information as described in Proxy and Firewall Settings below.

Once the startup questions are answered, REBOL creates a `user.r` file and places your network settings in it. You can change these settings at any time by editing the `user.r` file.

1.4 Proxy and Firewall Settings

Frequently, organizations use a firewall or proxy server to protect access to and from the Internet. Before REBOL can access the Internet through these systems, you need to provide some additional information.

To provide proxy server information:

1. When REBOL asks if you use a proxy server, answer by typing Y (yes).
2. Type the name of your proxy host. This is the computer or firewall on your network serving as a proxy.
3. Type the port number used by the proxy host for proxy requests. Typically, this is port 1080, but this can vary. If you don't know the port number, check your Web browser settings or ask your network administrator.

REBOL defaults to using a SOCKS proxy protocol. You can specify some other type of proxy by editing the `user.r` file and supplying the **`set-net`** function with the appropriate identification for the type of proxy being used. The following settings are supported:

```
socks   - use the latest SOCKS version (5)
socks5  - use socks5 proxy
socks4  - use socks4 proxy
generic - use generic CERN proxy
none    - use no proxy
```

These settings are provided as the sixth argument to the **`set-net`** function called in the `user.r` file. For more information about modifying the proxy settings in the `user`.r file, refer to the Network Protocols Chapter.

1.5 License Agreement

The REBOL end-user license agreement that you agreed to when you downloaded or installed REBOL can be viewed at any time from the REBOL console by typing `license` at the REBOL prompt.

2. Starting REBOL

REBOL runs on a large variety of systems. You start REBOL the same way you start other applications on your system. Depending on the specific operating system, REBOL can be started from one or more of the following: an icon, the command shell, or other applications.

2.1 From an Icon

REBOL can be started by double-clicking the REBOL program icon, an associated .r file, or a REBOL shortcut icon.

If you double-click on the program icon, REBOL boots, displays the console, and provides you with a prompt.

If you want to launch REBOL with a script, you can do so in the following ways:

- Drag the script to the program icon
- Associate the file with the REBOL program
- Create a shortcut or alias icon with program and script information.

See your operating system manual for more information.

2.2 From a Shell

From a shell command line, go to the directory that contains the `rebol.exe` file (or `rebol` on non-Windows systems), and type rebol or ./rebol.

On some operating systems, such as UNIX, you can create alias shell commands that are able to run REBOL with a set of arguments and files. In addition, UNIX enables you to create shell scripts that include a path, such as:

```
!#/path/to/rebol
```

in the top line of the script file. When you type the name of the script file at the command prompt, UNIX will launch REBOL to execute the script.

2.3 From Another Application

For writing and debugging REBOL scripts, it is handy to set up your favorite text editor to run REBOL and pass it the script file you are editing. Each text editor does this differently.

For instance, in the Premia Codewright editor you can use the language compiler options to set up REBOL. Specify the REBOL program rather than a compiler. Then you can press a single key that saves the script and evaluates it.

2.4 Security Issues

By default, security is set to prevent scripts from modifying any of your files or directories.

2.4.1 Port Security

The **secure** function provides flexibility in setting and controlling the security features of REBOL. The current security settings are returned as a result of calling the **secure** function.

Security settings use a REBOL dialect, that is, a language within a language. The normal dialect consists of a block of paired values. The `first` value in the pair specifies what is being secured:

file specifies file security.

net	specifies network security.

A file name or directory path allows you to specify security levels for a specific file or directory.

The `second` value in the pair specifies the level of security. This can be either a security level word or a block of words. The security level words are:

allow	allow access with no restrictions.
ask	ask permission if any restricted access occurs.
throw	throw an error if any restricted access occurs.
quit	quit this REBOL session if any restricted access occurs.

For example, to allow all network access, but to quit on any file access:

```
secure [
    net allow ;allows any net access
    file quit ;any file access will cause the program to quit
]
```

If a block is used instead of a security level word, it can contain pairs of security levels and access types. This lets you specify a greater level of detail about the security you require. The access types allowed are:

read	controls read access.
write	controls write, delete and rename access.
all	controls all access.

The pairs are processed in the order they appear, with later pairs modifying the effect of earlier pairs. This permits setting one type of access without explicitly setting all others. For example:

```
secure [
    net allow
    file [
        ask all
        allow read
    ]
]
```

The above sets the security level to **ask** for all operations except for reading which is to be allowed. This technique can also be used for individual files and directories. For example:

```
secure [
    net allow
    file quit
    %source/ [ask read]
]
```

asks if an attempt is made to read the `%source` directory. Otherwise, it uses the default (**quit**).

There is a special case in which the **secure** function takes a single word argument that must be one of the security access levels. In that case, the security level for all network and file access is set to that level.

```
secure quit
```

The **secure** function also accepts **none**, allowing access with no restrictions (same as **allow**).

```
secure none
```

The default security level is now:

```
secure [
    net allow
    file [
        ask all
        allow read
    ]
]
```

If no security access level is specified for either network or file access, it defaults to **ask**. The current settings will not be modified if an error occurs parsing the security block argument.

2.4.2 Prior Security Settings

The **secure** function now returns the prior security settings before the new settings were made. This is a block with the global network and file settings followed by file or directory settings. The **query** word can be used to obtain the settings without modifying them.

```
current-security: secure query
```

You can modify the current security level by querying the current settings, modifying them, then using the **secure** function to set the new values.

Lowering the security level produces a change security settings request. The exception is when the REBOL session is running in quiet mode which will, instead, terminate the REBOL session. No query is generated when security levels are raised. Note that the security request now includes an option to allow all access for the remainder of the scripts processing.

When running REBOL from the shell, the **-s** argument is equivalent to:

```
secure allow
```

and the **+s** arguments is equivalent to:

```
secure quit
```

You can now follow the **--secure** argument with one of the security access levels for both network and file access:

```
rebol --secure throw
```

2.5 Program Arguments

There are a number of arguments that can be specified in a shell command line, in a batch script, or in the properties of an icon. To view the arguments and options available for any version of the REBOL language, type `usage` at the console prompt.

```
The command line usage is:

    REBOL <options> <script> <arguments>

All fields are optional. Supported options are:

    --cgi (-c)        Check for CGI input
    --do expr         Evaluate expression
    --help (-?)       Display this usage information
    --nowindow (-w)   Do not open a window
    --noinstall (-i)  Do not install (View)
    --quiet (-q)      Don't print banners
    --reinstall (+i)  Force an install (View)
    --script file     Explicitly specify script
    --secure level    Set security level:
                      (allow ask throw quit)
    --trace (-t)      Enable trace mode
    --uninstall (-u)  Uninstall REBOL (View)

Other command line options:

    +q                Force not quiet (View)
    -s                No security
    +s                Full security
    -- args           Provide args without script

Examples:

    REBOL script.r
    REBOL script.r 10:30 test@domain.dom
    REBOL script.r --do "verbose: true"
    REBOL --cgi -s
    REBOL --cgi --secure throw --script cgi.r "debug: true"
    REBOL --secure none
```

Again, the format of the command line is:

```
REBOL options script arguments
```

Where:

options	one or more of the program options. See Specifying Options below for more details.
script	the file name of the script you want to run. If the file name contains spaces, it should be typed in quotes.
arguments	the arguments passed to the script as a string. These arguments can be accessed from within the script.

All of the above arguments are optional, and any combination is permitted.

Shortcut Icons

In some operating systems, like Windows or AmigaOS, you can create icons that supply any of the above options as part of the icon. Using this technique, you can create icons that directly execute REBOL scripts with the correct options.

2.6 Script File

Typically, you run REBOL with the file name of the script that you want it to evaluate. Only one script file is allowed. For example:

```
REBOL script.r
```

If the file name contains spaces, it must be provided in double quotes.

2.7 Specifying Options

Program options are identifed with a plus (+) or minus (-) before a character or by a double dash (--) before a word. This is a standard practice for specifying program options on most operating systems.

Here are several examples of how options are used.

To run a script with an option, such as the `-s` option, which evaluates the script with security turned off, type:

```
REBOL -s script.r
```

To obtain usage information about REBOL, type:

```
REBOL -?
REBOL --help
```

To run REBOL without opening a new window (this is done when you need to redirect output to a file or server), type:

```
REBOL -w
REBOL --nowindow
```

To prevent the printout of startup information which is useful if you are redirecting the output to a file or server, type:

```
REBOL -q
REBOL --quiet
```

To evaluate a REBOL expression from the command line, type:

```
REBOL --do "print 1 + 2"
REBOL --do "verbose: true" script.r
```

This lets you evaluate a remote script with a line such as:

```
REBOL --do "do http://www.rebol.com/speed.r"
```

To change the security level of REBOL, type:

```
REBOL -s script.r
REBOL --secure none script.r
```

To use REBOL scripts for CGI (see the CGI - Common Gateway Interface Section of the Network Protocols Chapter for more information), type:

```
REBOL -c cgi-script.r
REBOL --cgi
```

Multiple options are also allowed. Multiple single character options can be included together. Multiple full word options must be separated with spaces.

```
REBOL -cs cgi-script.r
REBOL --cgi --secure none cgi-script.r
```

The above example runs in CGI mode, with security turned off. The shorthand method is required for various web servers that restrict the number of arguments allowed on the command line (such as the Apache server on Linux).

2.8 File Redirection

On most systems, it is possible to redirect standard input and output from and to files. The example:

```
rebol -w script.r > output-file
```

redirects output to a file. Similarly,

```
rebol -w script.r < input-file
```

redirects input from a file.

When Redirecting File IO...

Use the `-w` option to prevent the REBOL console window from opening, as it interferes with standard input and output redirection.

2.9 Script Arguments

Everything on the command line that follows the script file name is passed to the script as its argument. This allows you to write scripts that accept arguments directly from the command line. For example, if you start REBOL with the line:

```
REBOL script.r 10:30 test@domain.dom
```

There are two ways to obtain the command line arguments. The first method returns the arguments as a block of REBOL values:

```
probe system/options/args
["10:30" "test@domain.dom"]
```

The second method returns the arguments as a string:

```
probe system/script/args
"10:30 test@domain.dom"
```

Version dependent note:

Earlier versions of REBOL returned the block of values for the script/args field (similar to options/args). It is advised that you verify that your script receives the correct args datatype as shown above.

2.10 Startup Files

When REBOL starts, it attempts to load the `rebol.r` and `user.r` boot files. These files are optional, but when found, they can be used to set up networking, define common functions, and initialize data used by scripts.

The `rebol.r` script file holds special functions or extensions to REBOL that are provided as part of the standard distribution. It is suggested that you do not edit this file as it is overwritten with each new release of REBOL.

The `user.r` script file holds user preferences. You can edit this file and add whatever definitions or data you require.

On multi-user systems, there can be a different `user.r` for every user. While the `user.r` file is not part of the distribution, it is automatically generated if it does not exist.

When REBOL starts, it looks for the `rebol.r` and `user.r` files first in the home directory and, if not found there, then in the current directory.

To set a HOME directory, you can set an environment variable in the appropriate login or startup script for your system. Note that some systems, such as UNIX or Linux may already do this, so you do not need to.

For example, on Windows NT to set HOME you can add:

```
set HOME=C:\REBOL
```

to your environment by following these steps:

1. Choose Settings Control Panel in the Windows Start Menu,
2. Double-click the System icon, and select the Environment tab.
3. Type HOME in the variable field and C:\REBOL (or the path to where you put the REBOL program) in the value field.

On Unix systems, you can set the path to REBOL by adding a line like the following in your login shell script

or profile:

```
set HOME=/usr/bin/rebol
```

For some versions of REBOL, the path is stored in a `.rebol` file that is located in your home directory.

3. Quitting REBOL

To exit REBOL at any time, select **Quit** from the **Console File** menu or by typing **quit** or **q** at the prompt.

You can also quit the program from within a script:

```
if now/time > 12:00 [quit]
```

The REBOL console may also quit if an error occurs during startup.

> **Exit Does Not Quit**
>
> Do not use the word **exit** to quit REBOL. This word is used for exiting functions and it will return an error if typed at the console.

4. Using the Console

Whenever you run REBOL/Core, it opens a console to display output and accept input. If you provide a script argument to the program, the script is run, and you see the output from that script. If you do not provide a script file, the console prompts you for input. The input prompt looks like this:

```
>>
```

If you type an expression at the input prompt, it is evaluated and any returned values are displayed following the result indicator:

```
==
```

For example:

```
>> 100 + 20
== 120
>> now - 7-Dec-1944
== 20341
```

> **Changing the Prompt**
>
> The prompt characters can be changed. See the Console Appendix for more information.

The console also becomes active if a script encounters an error or if the script calls the **halt** function.

4.1 Mulitple Line Input

If you begin a block on the command line and don't end it, the block is extended to the next line. This is indicated by a prompt that begins with a bracket and is followed by indentation. The line will be indented four spaces for each open block. For example:

```
loop 10 [
[    print "example"
[    if odd? random 10 [
[        print "here"
[        ]
[    ]
```

This is also true for multilined strings enclosed in braces.

```
Print {This is a long
{    string that has more
{    than one line.}
```

Brackets and braces that appear within quoted strings are ignored. You can escape from input at any time by pressing the ESCAPE key.

4.2 Interrupting a Script

A script can be interrupted by pressing the ESCAPE key, which returns immediately to the command prompt.

During some types of operating system or network activity there may be a delay in responding to the ESCAPE interrupt.

4.3 History Recall

Each line that is typed into REBOL is stored for later recall. The up and down arrow keys are used to scroll through the list of previous lines. For instance, pressing the up arrow once recalls the prior input line.

History lines can be written to a file by saving the history block. See the Console Appendix for more information.

4.4 Word Completion

To help speed typing of long words and file names, the REBOL console has word and file name completion. After typing a few letters of a word, press the tab key. If the letters uniquely identify the word, the rest of the word is displayed. For example, typing:

```
>> sq
```

then pressing tab results in:

```
>> square-root
```

If the letters do not uniquely identify the word, you can press tab again to get a list of choices. For example, typing:

```
>> so
```

then pressing tab twice results in:

```
>> sort source
```

so

and you can type the rest of the word or enough of it to be unique.

Completion works for all words, including user-defined words. It also works for files when they are preceded by a percent sign.

```
>> print read %r
```

Pressing tab might produce:

```
>> print read %rebol.r
```

depending on your current directory.

4.5 Busy Indicator

When REBOL waits for a network operation to complete, a busy indicator appears to indicate that something is happening. You can change the indicator to your own character pattern. See the Console Appendix for more information.

4.6 Network Connections

As network connections are initiated, a message appears on the console. For instance, typing:

```
>> read http://www.rebol.com
connecting to: www.rebol.com
```

If necessary, you can disable this output by setting the quiet flag. See the Console Appendix for more information.

4.7 Virtual Terminal

The console provides `virtual` terminal capability that allows you to perform operations such as cursor movement, cursor addressing, line editing, screen clearing, control key input, and cursor position querying.

The virtual terminal uses the standard ANSI character sequences. This allows you to write platform-independent terminal programs such as text editors, email clients, or telnet emulators.

More information can be found in the Console Appendix.

5. Getting Help

Several sources of information exist: online help built into REBOL, the **`source`** function, documents on the REBOL website, the REBOL script library, the REBOL mailing list, and sending feedback to REBOL.

5.1 Online Help

The online **help** function provides a quick way to obtain summary information about REBOL words. There are several ways to use help.

Type **help** or **?** at the console prompt to view a summary of help:

```
The help function provides a simple way to get
information about words and values.  To use it
supply a word or value as its argument:

    help insert
    help find

To view all words that match a pattern:

    help "path"
    help to-

To view all words of a specified datatype:

    help native!
    help datatype!

There is also word completion from the command
line.  Type a few chars and press TAB to complete
the word.  If nothing happens, there is more than
one word that matches.  Enough chars are needed
to uniquely identify the word.

Other useful functions:

    about - for general info
    usage - for the command line arguments
    license - for the terms of user license
    source func - print source for given function
    upgrade - updates your copy of REBOL

For more information, see the user guides.
```

If you provide a function word as an argument, **help** prints all of the information that was provided about the function. For instance, if you type:

```
help insert
```

you will see:

```
USAGE:
    INSERT series value /part range /only /dup count

DESCRIPTION:
     Inserts a value into a series and returns the series after the insert.
     INSERT is an action value.

ARGUMENTS:
     series -- Series at point to insert (Type: series port bitset)
     value -- The value to insert (Type: any-type)

REFINEMENTS:
     /part -- Limits to a given length or position.
         range -- (Type: number series port)
```

```
     /only -- Inserts a series as a series.
     /dup -- Duplicates the insert a specified number of times.
         count -- (Type: number)
```

The **help** function also finds words that contain a specified string. For instance, to find all of the words that include the string path, type:

```
? "path"
```

and the result will be:

```
Found these words:
     clean-path      (function)
     lit-path!       (datatype)
     lit-path?       (action)
     path            (file)
     path!           (datatype)
     path-thru       (function)
     path?           (action)
     set-path!       (datatype)
     set-path?       (action)
     split-path      (function)
     to-lit-path     (function)
     to-path         (function)
     to-set-path     (function)
```

You can also search for all globally defined words that are of a given data type. For example, to list all words that are function! data types, type:

```
? function!
```

and the result would be:

```
Found these words:
     ?               (function)
     ??              (function)
     about           (function)
     alert           (function)
     alter           (function)
     append          (function)
     array           (function)
     ask             (function)
...
```

To obtain a list of all REBOL datatypes, type:

```
? datatype!

Found these words:
     action!         (datatype)
     any-block!      (datatype)
     any-function!   (datatype)
     any-string!     (datatype)
     any-type!       (datatype)
     any-word!       (datatype)
```

```
     binary!         (datatype)
     bitset!         (datatype)
     block!          (datatype)
     char!           (datatype)
     datatype!       (datatype)
     date!           (datatype)
     decimal!        (datatype)
     email!          (datatype)
...
```

No Help for Objects

The **help** function does not provide useful information about the objects of the system, for example:

```
help system/options/home
system/options/home is a path.
```

Do not attempt to do:

```
help system
```

as it can take several minutes and produce over a megabyte of text output.

5.2 Viewing Source Code

Advanced users can learn more about specific REBOL functions by examining their source code. The **source** function displays the code for any REBOL defined function. If you type:

```
source join
```

The source to the **join** function will be returned:

```
join: func [
    "Concatenates values."
    value "Base value"
    rest "Value or block of values"
][
    value: either series? value [copy value] [form value]
    repend value rest
]
```

REBOL defined functions include the ***mezzanine functions*** (built-in functions implemented in REBOL) and user defined functions. ***Native functions*** are built-in functions implemented in machine code, and their source cannot be displayed.

5.3 Download Documents

Check the REBOL Web site http://www.rebol.com/ for a list of the current documentation.

In addition to this manual, there is a *REBOL Dictionary* that covers all the predefined words available in REBOL. If the console help or this guide does not contain sufficient information about a REBOL word, look in the dictionary for a detailed description.

The dictionary is updated with each release of REBOL and is available at http://www.REBOL.com/docs/dictionary.html.

5.4 Script Library

The REBOL Web site contains a library with numerous useful debugged scripts that cover a variety of topics. The library is divided into categories to make it easy to find a script specific to a given function. You can also search the library for scripts that contain a specific word.

The script library can be found at http://www.REBOL.com/library/library.html.

5.5 User Mailing List

You can also obtain help from a community of REBOL users by joining the REBOL email discussion list. To sign up, send an email to rebol-request@rebol.com with the subject line containing the word "subscribe". For example:

```
send rebol-request@rebol.com "subscribe"
```

Be sure that your correct email address has been set up in advance with **set-net**.

5.6 Contacting Us

We want to know what you think; please contact us to:

- Report crashes or problems
- Tell us how you are using REBOL
- Make suggestions
- Request more information about our products.

You can reach us through the Feedback page on our website.

6. Errors

6.1 Error Messages

There are several types of errors within REBOL. When an error occurs a message is displayed that tells you what the error was and approximately where it occurred. For instance if you type:

```
abc
** Script Error: abc has no value.
** Where: abc
```

The type of error is indicated by the first few words of the message. In the above example, the error is a `Script` Error. Script errors are the most common and occur when you use a function of the language in the wrong way or with improper arguments. Other types of errors are described in Error Types.

Error Type	Description
`Syntax errors`	Occur when the script contains an invalid value or a missing header, quote, bracket, or parenthesis.
`Math errors`	Occur when dividing a number by zero or there was a math overflow or underflow.
`Access errors`	Occur when a file, directory, or network operation cannot be accessed or access permissions are restricted.
`Throw errors`	Occur when a break, exit, or throw is used in an improper manner.
`User errors`	Defined by the user's script.

`Internal errors`	Returned when a problem occurs within the REBOL system. If you encounter one of these types of errors, please report it to Feedback.

Most types of errors can be trapped and processed by your script. See Trying Blocks for a description of the try function.

The Errors Appendix also includes useful information about errors.

6.2 Redirecting Errors

When errors are encountered in non-interactive sessions, such as when running in CGI mode (-c or --cgi) or in no window mode (-w or --nowindow), the session is automatically terminated.

If a script terminates while running in non-interactive mode, you can use shell redirection to output the error to a file:

```
REBOL -cs my_script.r >> my_script.log
```

This appends the output to the file in most operating systems.

7. Upgrading

On initialization, a banner is displayed that identifies the program version. Version numbers have the format:

```
version.revision.update.platform.variation
```

For example, the version number:

```
2.3.0.3.1
```

indicates that you are running version 2, revision 3, update 0, for Windows 95/98/NT (REBOL platform number 3.1). A complete list of all platform numbers is available from http://www.rebol.com. This HTML file also contains a hidden REBOL database that can be used for determining the platform.

You can obtain the version number from the REBOL prompt with:

```
print system/version
```

Only the latest release of REBOL is supported by REBOL Technologies. You can verify that you have the latest version and automatically update it if out of date. To do so, be sure that you are connected to the Internet, then from within REBOL type:

```
upgrade
```

REBOL returns one of the following messages about your version:

```
This copy of Windows 95/98/NT iX86 REBOL/core 2.3.0.3.1
is currently up to date.
```

or:

```
This copy of Windows 95/98/NT iX86 REBOL/core 2.1.2.3.1
is not up to date. Current version is: 2.3.0.3.1.
Download current version?
```

To upgrade to the latest version, type `Y` (yes). Otherwise, type `N` (no).

3 - Quick Tour

1. Overview
2. Values
    2.1 Numbers
    2.2 Times
    2.3 Dates
    2.4 Money
    2.5 Tuples
    2.6 Strings
    2.7 Tags
    2.8 Email Addresses
    2.9 URLs
    2.10 Filenames
    2.11 Pairs
    2.12 Issues
    2.13 Binary
3. Words
4. Blocks
5. Variables
6. Evaluation
7. Functions
8. Paths
9. Objects
10. Scripts
11. Files
12. Networking
    12.1 HTTP
    12.2 FTP
    12.3 SMTP
    12.4 POP
    12.5 NNTP
    12.6 Daytime
    12.7 Whois
    12.8 Finger
    12.9 DNS
    12.10 TCP

1. Overview

This chapter provides a quick way to familiarize yourself with the REBOL language. Using examples, this chapter presents the basic concepts and structure of the language, illustrating everything from data values to performing network operations.

2. Values

A script is written with a sequence of ***values***. A wide variety of values exist and you are familiar with many of them from daily experience.

When possible, REBOL also allows the use of international formats for values such as decimal numbers, money, time, and date.

2.1 Numbers

Numbers are written as integers, decimals, or scientific notation. For example:

```
1234 -432 3.1415 1.23E12
```

And can also be written in European format:

```
123,4 0,01 1,2E12
```

2.2 Times

Time is written in hours and minutes with optional seconds, each separated by colons. For example:

```
12:34 20:05:32 0:25.345 0:25,345
```

Seconds can include a decimal sub-second. Times can also include AM and PM appended without intervening spaces:

```
12:35PM 9:15AM
```

2.3 Dates

Dates are written in either international format: day-month-year or year-month-day. A date can also include a time and a time zone. The name or abbreviation of a month can be used to make its format more identifiable in the United States. For example:

```
20-Apr-1998 20/Apr/1998 (USA friendly)

20-4-1998 1998-4-20     (international)

1980-4-20/12:32         (date with time)

1998-3-20/8:32-8:00     (with time zone)
```

2.4 Money

Money is written as an optional international three-letter currency symbol followed by a numeric amount. For example:

```
$12.34  USD$12.34  CAD$123.45  DEM$1234,56
```

2.5 Tuples

Tuples are used for version numbers, RGB color values, and network addresses. They are written as integers that range from 0 to 255 and are separated by dots. For example:

```
2.3.0.3.1  255.255.0  199.4.80.7
```

At least two dots are required (otherwise the number will be interpreted as a decimal value, not a tuple). Example:

```
2.3.0  ; tuple
2.3.   ; tuple
2.3    ; decimal
```

2.6 Strings

Strings are written in a single-line format or a multiline format. Single-line format strings are enclosed in quotes. Multiline format strings are enclosed in braces. Strings that include quotes, tabs, or line breaks must be enclosed in braces using the multiline format. For example:

```
"Here is a single-line string"

{Here is a multiline string that
contains a "quoted" string.}
```

Special characters (escapes) within strings are indicated with the caret character (^). See the string section in the Values Chapter for the table of escape sequences.

2.7 Tags

Tags are useful for markup languages such as XML and HTML. Tags are enclosed in angle brackets For example:

```
<title> </body>

<font size="2" color="blue">
```

2.8 Email Addresses

Email addresses are written directly in REBOL. They must include an at sign (@). For example:

```
info@rebol.com

pres-bill@oval.whitehouse.gov
```

2.9 URLs

Most types of Internet URLs are accepted directly by REBOL. They begin with a scheme name (HTTP for example) followed by a path. For example:

```
http://www.rebol.com

ftp://ftp.rebol.com/sendmail.r
```

```
ftp://freda:grid@da.site.dom/dir/files/

mailto:info@rebol.com
```

2.10 Filenames

Filenames are preceded by a percent sign to distinguish them from other words. For example:

```
%data.txt

%images/photo.jpg

%../scripts/*.r
```

2.11 Pairs

Pairs are used to indicate spatial coordinates, such as positions on a display. They are used to indicate both positions and sizes. Coordinates are separated by an `x`. For example:

```
100x50

1024x800

-50x200
```

2.12 Issues

Issues are identification numbers, such as telephone numbers, model numbers, credit card numbers. For example:

```
#707-467-8000

#0000-1234-5678-9999

#MFG-932-741-A
```

2.13 Binary

Binary values are byte strings of any length. They can be encoded directly as hexadecimal or base-64. For example:

```
#{42652061205245424F4C}

64#{UkVCT0wgUm9ja3Mh}
```

3. Words

Words are the symbols used by REBOL. A word may or may not be a variable, depending on how it is used. Words are also used directly as symbols.

```
show next image

Install all files here

Country State City Street Zipcode

on off true false one none
```

REBOL has no keywords; there are no restrictions on what words are used or how they are used. For instance, you can define your own function called `print` and use it instead of the predefined function for printing values.

Words are not case sensitive and can include hyphens and a few other special characters such as:

```
+ - `  * ! ~ & ? |
```

The following examples illustrate valid words:

```
number?  time?  date!
image-files  l'image
++ -- == +-
***** *new-line*
left&right left|right
```

The end of a word is indicated by a space, a line break, or one of the following characters:

```
[ ] ( ) { } " : ; /
```

The following characters are not allowed in words:

```
@ # $ % ^ ,
```

4. Blocks

REBOL is composed by grouping values and words into ***blocks***. Blocks are used for code, lists, arrays, tables, directories, associations, and other sequences.

A block is a type of *series* which is a *collection of values* organized in a *specific order*.

A block is enclosed in square brackets [ ]. Within a block, values and words can be organized in any order and can span any number of lines. The following examples illustrate the valid forms of blocks:

```
[white red green blue yellow orange black]

["Spielberg" "Back to the Future" 1:56:20 MCA]

[
```

```
    Ted     ted@gw2.dom   #213-555-1010
    Bill    billg@ms.dom  #315-555-1234
    Steve   jobs@apl.dom  #408-555-4321
]

[
    "Elton John"  6894  0:55:68
    "Celine Dion" 68861 0:61:35
    "Pink Floyd"  46001 0:50:12
]
```

Blocks are used for code as well as for data, as shown in the following examples:

```
loop 10 [print "hello"]

if time > 10:30 [send jim news]

sites: [
    http://www.rebol.com [save %reb.html data]
    http://www.cnn.com   [print data]
    ftp://www.amiga.com  [send cs@org.foo data]
]

foreach [site action] sites [
    data: read site
    do action
]
```

A script file itself also is a block. Although it does not include the brackets, the block is implied. For example, if the lines below were put in a script file:

```
red
green
blue
yellow
```

When the file is loaded, it will be a block that contains the words red, green, blue, and yellow. It is equivalent to writing:

```
[red green blue yellow]
```

5. Variables

Words can be used as variables that refer to values. To define a word as a variable, follow the word with a colon (:), then the value to which the variable refers as shown in the following examples:.

```
age: 22
snack-time: 12:32
birthday: 20-Mar-1997
friends: ["John" "Paula" "Georgia"]
```

A variable can refer to any type of value, including functions (see Functions) and objects (see Objects).

A variable refers to a specific value only within a defined context, such as a block, a function, or an entire program. Outside that context the variable can refer to some other value or to no value at all. The context of a variable can span an entire program or it can be restricted to a particular block, function, or object. In other languages, the context of a variable is often referred to as the scope of a variable.

6. Evaluation

Blocks are evaluated to compute their results. When a block is evaluated the values of its variables are obtained. The following examples evaluate the variables `age`, `snack-time`, `birthday`, and `friends` that were defined in the previous section:

```
print age
22
if current-time > snack-time [print snack-time]
12:32
print third friends
Georgia
```

A block can be evaluated multiple times by using a loop, as shown in the following examples:

```
loop 10 [prin "*"]  ;("prin" is not a typo, see manual)
**********
loop 20 [
    wait 8:00
    send friend@rebol.com read http://www.cnn.com
]

repeat count 3 [print ["count:" count]]
count: 1
count: 2
count: 3
```

The evaluation of a block returns a result. In the following examples, `5` and `PM` are the results of evaluating each block:

```
print do [2 + 3]
5
print either now/time < 12:00 ["AM"]["PM"]
PM
```

In REBOL there are no special operator precedence rules for evaluating blocks. The values and words of a block are always evaluated from first to last, as shown in the following example:

```
print 2 + 3 * 10
50
```

Parentheses can be used to control the order of evaluation, as shown in the following examples:

```
2 + (3 * 10)
32
(length? "boat") + 2
6
```

You can also evaluate a block and return each result that was computed within it. This is the purpose of the **reduce** function:

```
reduce [1 + 2  3 + 4  5 + 6]
3 7 11
```

7. Functions

A function is a block with variables that are given new values each time the block is evaluated. These variables are called the arguments of the function.

In the following example, the word `sum` is set to refer to a function that accepts two arguments, `a` and `b` :

```
sum: func [a b] [a + b]
```

In the above example, **func** is used to define a new function. The first block in the function describes the arguments of the function. The second block is the block of code that gets evaluated when the function is used. In this example, the second block adds two values and returns the result.

The next example illustrates one use of the function `sum` that was defined in the previous example:

```
print sum 2 3
5
```

Some functions need local variables as well as arguments. To define this type of function, use **function**, instead of **func**, as shown in the following example:

```
average: function [series] [total] [
    total: 0
    foreach value series [total: total + value]
    total / (length? series)
]

print average [37 1 42 108]
47
```

In the above example, the word `series` is an argument and the word `total` is a local variable used by the function for calculation purposes.

The function argument block can contain strings to describe the purpose of a function and its argument, as shown in the following example:

```
average: function [
    "Return the numerical average of numbers"
    series "Numbers to average"
] [total] [
    total: 0
    foreach value series [total: total + value]
    total / (length? series)
]
```

These descriptive strings are kept with the function and can be viewed by asking for help about the function, as shown below:

```
help average
USAGE:
    AVERAGE series
DESCRIPTION:
    Return the numerical average of numbers
    AVERAGE is a function value.
ARGUMENTS:
    series -- Numbers to average (Type: any)
```

8. Paths

If you are using files and URLs, then you are already familiar with the concept of paths. A path provides a set of values that are used to navigate from one point to another. In the case of a file, a path specifies the route through a set of directories to the location of the file. In REBOL, the values in a path are called ***refinements***.

A slash (/) is used to separate words and values in a path, as shown in the following examples of a file path and a URL path:

```
%source/images/globe.jpg

http://www.rebol.com/examples/simple.r
```

Paths can also be used to select values from blocks, pick characters from strings, access variables in objects, and refine the operation of a function, as shown in the following examples:

```
USA/CA/Ukiah/size  (block selection)
names/12           (string position)
account/balance    (object function)
match/any          (function option)
```

The `print` function in next example shows the simplicity of using a path to access a mini-database created from a few blocks:

```
towns: [
    Hopland [
        phone #555-1234
        web   http://www.hopland.ca.gov
    ]
```

```
        Ukiah [
        phone #555-4321
        web   http://www.ukiah.com
        email info@ukiah.com
    ]
]

print towns/ukiah/web
http://www.ukiah.com
```

9. Objects

An object is a set of variables that have specific values in a context. Objects are used for managing data structures and more complex behavior. The following example shows how a bank account is set up as an object to specify its attributes and functions:

```
account: make object! [
  name: "James"
  balance: $100
  ss-number: #1234-XX-4321
  deposit:  func [amount] [balance: balance + amount]
  withdraw: func [amount] [balance: balance - amount]
]
```

In the above example, the words `name`, `balance`, `ss-number`, `deposit`, and `withdraw` are local variables of the `account` object. The `deposit` and `withdraw` variables are functions that are defined within the object. The variables of the account can be accessed with a path, as shown in the next example:

```
print account/balance
$100.00
account/deposit $300

print ["Balance for" account/name "is" account/balance]
Balance for James is $400.00
```

The next example shows how to make another account with a new balance but with all other values remaining the same:

```
checking-account: make account [
    balance: $2000
]
```

Notice that the new object is called checking-account, and it shares the values of the original account, but provides a new balance amount.

Just as easily you can create a new account object that extends the old account object, adding new variables for the bank name and last activity date:

```
account: make account [
    bank: "Savings Bank"
    last-active: 20-Jun-2000
]
```

```
print account/balance
$400.00
print account/bank
Savings Bank
print account/last-active
20-Jun-2000
```

10. Scripts

A script is a file that holds a block that can be loaded and evaluated. The block can contain code or data, and typically contains a number of sub-blocks.

Scripts require a header to identify the presence of code. The header can include the script title, date, and other information. In the following example of a script, the first block contains the header information:

```
REBOL [
    Title: "Web Page Change Detector"
    File:  %webcheck.r
    Author: "Reburu"
    Date:  20-May-1999
    Purpose: {
        Determine if a web page has changed since it was
        last checked, and if it has, send the new page
        via email.
    }
    Category: [web email file net 2]
]

page: read http://www.rebol.com

page-sum: checksum page

if any [
    not exists? %page-sum.r
    page-sum <> (load %page-sum.r)
][
    print ["Page Changed" now]
    save %page-sum.r page-sum
    send luke@rebol.com page
]
```

11. Files

In REBOL, files are easily accessed. The following table describes some of the ways to access files.

You can read a text file with:

```
data: read %plan.txt
```

You can display a text file with:

```
print read %plan.txt
```

To write a text file:

```
write %plan.txt data
```

For instance, you could write out the current time with:

```
write %plan.txt now
```

You can also easily append to the end of a file:

```
write/append %plan data
```

Binary files can be read and written with:

```
data: read/binary %image.jpg

write/binary %new.jpg data
```

To load a file as a REBOL block or value:

```
data: load %data.r
```

Saving a block or a value to a file is just as easy:

```
save %data.r data
```

To evaluate a file as a script (it needs a header to do this.):

```
do %script.r
```

You can read a file directory with:

```
dir: read %images/
```

and, you can then display the file names with:

```
foreach file dir [print file]
```

To make a new directory:

```
make-dir %newdir/
```

To find out the current directory path:

```
print what-dir
```

If you need to delete a file:

```
delete %oldfile.txt
```

You can also rename a file with:

```
rename %old.txt %new.txt
```

To get information about a file:

```
print size? %file.txt

print modified? %file.txt

print dir? %image
```

12. Networking

There are a number of Internet protocols built into REBOL. These protocols are easy to use and require very little knowledge of networking.

12.1 HTTP

The following example shows how to use the HTTP protocol to read a web page:

```
page: read http://www.rebol.com
```

The next example fetches an image from a web page and writes it to a local file:

```
image: read/binary http://www.page.dom/image.jpg

write/binary %image.jpg image
```

12.2 FTP

The following reads and writes files to a server using the file transfer protocol (FTP):

```
file: read ftp://ftp.rebol.com/test.txt

write ftp://user:pass@site.dom/test.txt file
```

The next example gets a directory listing from FTP:

```
print read ftp://ftp.rebol.com/pub
```

12.3 SMTP

The following example sends email with the simple mail transfer protocol (SMTP):

```
send luke@rebol.com "Use the force."
```

The next example sends the text of a file as an email:

```
send luke@rebol.com read %plan.txt
```

12.4 POP

The following example fetches email with the post office protocol (POP) and prints all of the current messages but leaves them on the server:

```
foreach message read pop://user:pass@mail.dom [
    print message
]
```

12.5 NNTP

The following example fetches news with the network news transfer protocol (NNTP), reading all of the news in a particular news group:

```
messages: read nntp://news.server.dom/comp.lang.rebol
```

The next example reads a list of all news group and prints them:

```
news-groups: read nntp://news.some-isp.net

foreach group news-groups [print group]
```

12.6 Daytime

The following example gets the current time from a server:

```
print read daytime://everest.cclabs.missouri.edu
```

12.7 Whois

The following example finds out who is in charge of a domain using the whois protocol:

```
print read whois://rebol@rs.internic.net
```

12.8 Finger

The following example gets user information with the finger protocol:

```
print read finger://username@host.dom
```

12.9 DNS

The following example determines an Internet address from a domain name and a domain name from an address:

```
print read dns://www.rebol.com

print read dns://207.69.132.8
```

12.10 TCP

Direct connections with TCP/IP are also possible in REBOL. The following example is a simple, but useful, server that waits for connections on a port, then executes whatever has been sent:

```
server-port: open/lines tcp://:9999

forever [
    connection-port: first server-port
    until [
        wait connection-port
        error? try [do first connection-port]
    ]
    close connection-port
]
```

4 - Expressions

1. Overview
2. Blocks
3. Values
    3.1 Direct and Indirect Values
    3.2 Datatypes of Values
4. Evaluating Expressions
    4.1 Evaluating Console Input
    4.2 Evaluating Simple Values
    4.3 Evaluating Blocks
    4.4 Reducing Blocks
    4.5 Evaluating Scripts
    4.6 Evaluating Strings
    4.7 Evaluation Errors
5. Words
    5.1 Word Names
    5.2 Word Usage
    5.3 Setting Words
    5.4 Getting Words
    5.5 Literal Words
    5.6 Unset Words
    5.7 Protecting Words
6. Conditional Evaluation
    6.1 Conditional Blocks
    6.2 Any and All
    6.3 Conditional Loops
    6.4 Common Mistakes
7. Repeated Evaluation
    7.1 Loop
    7.2 Repeat
    7.3 For
    7.4 Foreach
    7.5 Forall and Forskip
    7.6 Forever
    7.7 Break
8. Selective Evaluation
    8.1 Select
    8.2 Switch
9. Stopping Evaluation
10. Trying Blocks

1. Overview

The foremost goal of REBOL is to establish a standard method of communication that spans all computer systems. REBOL provides a simple, direct means of expressing any kind of information with optimal flexibility and minimal syntax. For example, examine the following line:

```
Sell 100 shares of "Acme" at $47.97 per share
```

The line looks a lot like English making it easy to compose if you are sending it and easy to understand if you are receiving it. However, this line is actually a valid expression in REBOL, so

your computer could also understand and act on it. (Note that the line is a "dialect" of REBOL. It can be in-directly evaluated. More on this concept below.)

REBOL provides a common language between you and your computer. In addition, if your computer sends this expression to your stock broker's computer, which is also running REBOL, your stock broker's computer can understand the expression and act on it. So, REBOL provides a common language between computers. And, the line could be sent to millions of other computer systems that could also act on it.

The following line is another example of a REBOL expression:

```
Reschedule exam for 2-January-1999 at 10:30
```

The expression shown in the above example (written in another dialect) may have come from your doctor typing it, or perhaps it originated from an application that was run by your doctor. It does not matter. What is important is that the expression can be acted upon regardless of the type of computer, hand-held device, kiosk, or television console you are using.

The data values (numbers, strings, prices, dates, and times) in all of the expressions shown in the previous examples are standardized valid REBOL formats. The words, however, depend on a specific context of interpretation to convey their meaning. Words such as `sell`, `at`, and `read` have different meanings in different contexts. The words are relative expressions -- their meaning is context dependent.

Expressions can be processed in one of two ways: ***directly*** by the REBOL interpreter, or ***indirectly*** by a REBOL script. An expression processed indirectly is called a ***dialect***. The previous examples are dialects, so they are processed by a script. The following example is not a dialect and is processed directly by the REBOL interpreter:

```
send master@rebol.com read http://www.rebol.com
```

In this example the words **`send`** and **`read`** are functions that are processed by the REBOL interpreter.

The distinction REBOL makes is that **information is either directly or indirectly interpreted**. The distinction is not whether information is code or data, but how it is processed. In REBOL code is often handled as data and data is frequently processed as code, so the traditional division between code and data blurs. How information is processed determines whether it is code or data.

2. Blocks

REBOL Expressions are based on this concept: you combine ***values*** and ***words*** into ***blocks***.

In scripts, a block is normally enclosed with square brackets [ ]. Everything within the square brackets is part of the block. The block contents can span any number of lines, and its format is completely freeform. The following examples show various ways of formatting block content:

```
[white red green blue yellow orange black]

["Spielberg" "Back to the Future" 1:56:20 MCA]

[
    "Bill"  billg@ms.dom  #315-555-1234
    "Steve" jobs@apl.dom  #408-555-4321
    "Ted"   ted@gw2.dom   #213-555-1010
```

```
]

sites: [
    http://www.rebol.com [save %reb.html data]
    http://www.cnn.com   [print data]
    ftp://www.amiga.com  [send cs@org.foo data]
]
```

Some blocks do not require square brackets, because they are implied. For example, in a REBOL script, there are no brackets around the entire script, however, the script content is a block. The square brackets of an ***outer-block*** of the script are implied. The same is true for expressions typed at the command prompt or for REBOL messages sent between computers--each is an implied block.

Another important aspect of blocks is that they imply additional information. Blocks ***group*** a set of values in a particular ***order***. That is, a block can be used as a data set as well as a sequence. This will be described in more detail in the Series Chapter.

3. Values

REBOL provides a built-in set of ***values*** that can be expressed and exchanged between all systems. Values are the primary elements for composing all REBOL expressions.

3.1 Direct and Indirect Values

Values can be ***directly*** or ***indirectly*** expressed.

A directly expressed value is ***known*** as it is lexically, or literally, written. For instance, the number 10 or the time 10:30 are directly expressed values.

An indirectly expressed value is ***unknown*** until it is evaluated. The values `none`, `true`, and `false` all require words to represent them. These values are indirectly expressed because they must be evaluated for their values to be known. This is also true of other values, such as lists, hashes, functions, objects.

3.2 Datatypes of Values

Every REBOL value is of a particular ***datatype***. The datatype of a value defines:

1. The range of possible values for the datatype. For example, the logic datatype can only be true or false.
2. The operations that can be performed. For example, you can add two integers, but you cannot add two logical values.
3. The way in which the values are stored in memory. Some datatypes can be stored directly (such as numbers), while others are stored indirectly (such as strings).

By convention, REBOL datatype words are followed by an exclamation point (!) to help make them stand out. For example:

```
integer!
char!
word!
string!
```

Datatype Words are Just Words

The words used for datatypes are just like any other words in REBOL. There is nothing magic about the ! used to represent them.

See the Values Appendix for a description of all the REBOL datatypes.

4. Evaluating Expressions

To ***evaluate*** an expression is to compute its value. REBOL operates by evaluating the series of expressions constituting a script and then returning the result. Evaluating is also called running, processing, or executing a script.

Evaluation is performed on blocks. Blocks can be typed at the console or loaded from a script file. Either way, the process of evaluation is the same.

4.1 Evaluating Console Input

Any expression that can be evaluated in a script, can also be evaluated from the REBOL prompt, providing a simple means of testing individual expressions in a script.

For example, if you type the following expression at the console prompt:

```
>> 1 + 2
```

the expression is evaluated and the following result is returned:

```
== 3
```

About The Code Examples...

In the example above, the console prompt (>>) and result indicator (==) are shown to give you an idea of how they appear in the console. For the examples that follow, the prompt and result strings are not shown. However, you can assume that these examples can be typed into the console to verify their results.

4.2 Evaluating Simple Values

Since the value of directly expressed values is known, they simply return their values. For example, if you type the following line:

```
10:30
```

the value 10:30 is returned. This is the behavior of all directly expressed values. It includes:

```
integer    1234
decimal    12.34
string     "REBOL world!"
time       13:47:02
date       30-June-1957
```

```
tuple       199.4.80.1
money       $12.49
pair        100x200
char        #"A"
binary      #{ab82408b}
email       info@rebol.com
issue       #707-467-8000
tag         <IMG SRC="xray.jpg">
file        %xray.jpg
url         http://www.rebol.com/
block       [milk bread butter]
```

4.3 Evaluating Blocks

Normally, blocks are ***not*** evaluated. For example, typing the following block:

```
[1 + 2]
```

returns the same block:

```
[1 + 2]
```

The block is not evaluated; it is simply treated as data.

To evaluate a block, use the **do** function, as shown in the following example:

```
do [1 + 2]
3
```

The **do** function returns the result of the evaluation. In the previous example, the number 3 is returned.

If a block contains multiple expressions, only the result of the last expression is returned:

```
do [
    1 + 2
    3 + 4
]
7
```

In this example, both expressions are evaluated, but only the result of the 3 + 4 expression is returned.

There are a number of functions such as **if**, **loop**, **while**, and **foreach** that evaluate a block as part of their function. These functions are discussed in detail later in this chapter, but here are a few examples:

```
if time > 12:30 [print "past noon"]
past noon
loop 4 [print "looping"]
```

```
looping
looping
looping
looping
```

This is important to remember: blocks are treated as data until they are explicitly evaluated by a function. Only a function can cause them to be evaluated.

4.4 Reducing Blocks

When you evaluate a block with **do**, only the value of its last expression is returned as a result. However, there are times when you want the values of all the expressions in a block to be returned. To return the results of all of the expressions in a block, use the **reduce** function. In the following example, **reduce** is used to return the results of both expressions in the block:

```
reduce [
    1 + 2
    3 + 4
]
[3 7]
```

In the above example, the block was ***reduced*** to its evaluation results. The **reduce** function returns results in a block.

The **reduce** function is important because it enables you to create blocks of expressions that are evaluated and passed to other functions. **Reduce** evaluates each expression in a block and puts the result of that expression into a new block. That new block is returned as the result of **reduce**.

Some functions, like **print**, use **reduce** as part of their operation, as shown in the following example:

```
print [1 + 2  3 + 4]
3 7
```

The **rejoin**, **reform**, and **remold** functions also use **reduce** as part of their operation, as shown in the following examples:

```
print rejoin [1 + 2  3 + 4]
37
print reform [1 + 2  3 + 4]
3 7
print remold [1 + 2  3 + 4]
[3 7]
```

The **rejoin**, **reform**, and **remold** functions are based on the **join**, **form**, and **mold** functions, but reduce their blocks first.

4.5 Evaluating Scripts

The **do** function can be used to evaluate entire scripts. Normally, **do** evaluates a block, as shown in the following example:

```
do [print "Hello!"]
Hello!
```

But, when **do** evaluates a file name instead of a block, the file will be loaded into the interpreter as a block, then evaluated as shown in the following example:

```
do %script.r
```

A REBOL Header is Required

For a script file to be evaluated, it must include a valid REBOL header, which is described in the Scripts Chapter. The header identifies that the file contains a script and not just random text.

4.6 Evaluating Strings

The **do** function can be used to evaluate expressions that are found within text strings. For example, the following expression:

```
do "1 + 2"
3
```

returns the result 3. First the string is converted to a block, then the block is evaluated.

Evaluating strings can be handy at times, but it should be done only when necessary. For example, to create a REBOL console line processor, type the following expression:

```
forever [probe do ask "=> "]
```

The above expression would prompt you with => and wait for you to type a line of text. The text would then be evaluated, and its result would be printed. (Of course, it's not really quite this simple, because the script could have produced an error.)

Unless it is necessary, evaluating strings is not generally a good practice. Evaluating strings is less efficient than evaluating blocks, and the context of words in a string is not known. For example, the following expression:

```
do form ["1" "+" "2"]
```

is much less efficient than typing:

```
do [1 + 2]
```

REBOL blocks can be constructed just as easily as strings, and blocks are better for expressions that need to be evaluated.

4.7 Evaluation Errors

Errors may occur for many different reasons during evaluation. For example, if you divide a number by zero, evaluation is stopped and an error is displayed

```
100 / 0
** Math Error: Attempt to divide by zero.
** Where: 100 / 0
```

A common error is using a word before it has been defined:

```
size + 10
** Script Error: size has no value.
** Where: size + 10
```

Another common error is not providing the proper values to a function in an expression:

```
10 + [size]
** Script Error: Cannot use add on block! value.
** Where: 10 + [size]
```

Sometimes errors are not so obvious, and you will need to experiment to determine what is causing the error.

5. Words

Expressions are built from values and words. Words are used to represent meaning. A word can represent an idea or it can represent a specific value.

In the previous examples in this chapter, a number of words were used within expressions without explanation. For instance, the **do**, **reduce**, and **try** words are used, but not explained.

Words are evaluated somewhat differently than directly expressed values. When a word is evaluated, its value is looked up, evaluated, and returned as a result. For example, if you type the following word:

```
zero
0
```

the value 0 is returned. The word **zero** is predefined to be the number zero. When the word is looked up, a zero is found and is returned.

When words like **do** and **print** are looked up, their values are found to be functions, rather than simple values. In such cases, the function is evaluated, and the result of the function is returned.

5.1 Word Names

Words are composed of alphabetic characters, numbers, and any of the following characters:

```
? ! . ' + - * & | = _ ~
```

A word cannot begin with a number, and there are also some restrictions on words that could be interpreted as numbers. For example, -1 and +1 are numbers, not words.

The end of a word is marked by a space, a new line, or one of the following characters:

```
[ ] ( ) { } " : ; /
```

Thus, the brackets of a block are not part of a word. For example, the following block contains the word `test` :

```
[test]
```

The following characters are not allowed in words as they cause words to be misinterpreted or to generate an error:

```
@ # $ % ^ ,
```

Words can be of any length, but words cannot extend past the end of a line:

```
this-is-a-very-long-word-used-as-an-example
```

The following lines provide examples of valid words:

```
Copy print test
number?  time?  date!
image-files  l'image
++ -- == +-
***** *new-line*
left&right left|right
```

REBOL is not case sensitive. The following words all refer to the same word:

```
blue
Blue
BLUE
```

The case of a word is preserved when it is printed.

Words can be reused. The meaning of a word is dependent on its context, so words can be reused in different contexts. There are no keywords in REBOL. You can reuse any word, even those that are predefined in REBOL. For instance, you can use the word **`if`** in your code differently than the REBOL interpreter uses this word.

Pick Good Words

Pick the words you use carefully. Words are used to associate meaning. If you pick your words well, it will be easier for you and others to understand your scripts.

5.2 Word Usage

Words are used in two ways: as ***symbols*** or as ***variables***. In the following block, words are used as symbols for colors.

```
[red green blue]
```

In the following line:

```
print second [red green blue]
green
```

the words have no meaning other than their use as names for colors. All words used within blocks serve as symbols until they are evaluated.

When a word is evaluated, it is used as a variable. In the previous example, the words **print** and **second** are variables that hold native functions which perform the required processing.

A word can be written in four ways to indicate how it is to be treated, as shown in Word Formats.

Format	What It Does
`word`	Evaluates the word. This is the most natural and common way to write words. If the word holds a function, it will be evaluated. Otherwise, the value of the word will be returned.
`word:`	Defines or sets the value of a word. It is given a new value. The value can be anything, including a function. See Setting Words below.
`:word`	Gets the word's value, but doesn't evaluate it. This is useful for referring to functions and other data without evaluating them. See Getting Words below.
`'word`	Treats the word as a symbol, but does not evaluate it. The word itself is the value.

5.3 Setting Words

A word followed by a colon (:) is used to define or set its value:

```
age: 42
lunch-time: 12:32
birthday: 20-March-1990
town: "Dodge City"
test: %stuff.r
```

You can set a word to be any type of value. In the previous examples, words are defined to be integer, time, date, string, and file values. You can also set words to be more complex types of values. For example, the following words are set to block and function values:

```
towns: ["Ukiah" "Willits" "Mendocino"]
code: [if age > 32 [print town]]
say: func [item] [print item]
```

Why Words Are Set This Way

In many langages words are set with an equal sign, such as:

```
age = 42
```

In REBOL words are set with a colon. The reason for this is important. It makes the set operation on words into a single lexical value. The representation for the set operation is ***atomic***.

The difference between the two approaches can be seen in this example.

```
print length? [age: 42]
2
print length? [age = 42]
3
```

REBOL is a ***reflective*** language, it is able to manipulate its own code. This method of setting values allows you to write code that easily manipulates **`set-word`** operations as a single unit.

Of couse, the other reason is that the equal sign (=) is used as a comparision operator.

Multiple words can be set at one time by cascading the word definitions. For example, each of the following words are set to `42:`

```
age: number: size: 42
```

Words can also be set with the **`set`** function:

```
set 'time 10:30
```

In this example, the line sets the word `time` to 10:30. The word `time` is written as a literal (using a single quote) so that it will not be evaluated.

The **`set`** function can also set multiple words:

```
set [number num ten] 10

print [number num ten]
10 10 10
```

In the above example, notice that the words do not need to be quoted because they are within a block, which is not evaluated. The **print** function shows that each word is set to the integer 10.

If **set** is provided a block of values, each of the individual values are set to the words. In this example, one, two, and three are set to 1, 2, and 3:

```
set [one two three] [1 2 3]

print three
3
print [one two three]
1 2 3
```

See the Words Section in the Values Appendix for more about setting words.

5.4 Getting Words

To get the value of a word that was previously defined, place a colon (:) at the front of the word. A word prefixed with a colon obtains the value of the word, but does not evaluate it further if it is a function. For example, the following line:

```
drucken: :print
```

defines a new word, drucken (which is German for print), to refer to the same function **print** does. This is possible because **:print** returns the function for **print**, but does not evaluate it.

Now, drucken performs the same function as **print** :

```
drucken "test"
test
```

Both **print** and drucken are set to the same value, which is the function that does printing.

This can also be accomplished with the **get** function. When given a literal word, **get** returns its value, but does not evaluate it:

```
stampa: get 'print

stampa "test"
test
```

The ability to get the value of a word is also important if you want to determine what the value is without evaluating it. For example, you can determine if a word is a native function using the following line:

```
print native? :if
true
```

Here the get returns the function for **if**. The **if** function is not evaluated, but rather it is passed to the **native?** function which checks if it is a native datatype. Without the colon, the **if**

function would be evaluated, and, because it has no arguments, an error would occur.

5.5 Literal Words

The ability to deal with a word as a literal is useful. Both **set** and **get**, as well as other functions like **value?**, **unset**, **protect**, and **unprotect**, expect a literal value.

Literal words can be written in one of two ways: by prefixing the word with a single quotation mark, also known as a tick, (`) or by placing the word in a block.

You can use a tick in front of a word that is evaluated:

```
word: 'this
```

In the above example, the `word` variable is set to the literal word `this`, not to the value of `this`. The `word` variable just uses the name symbolically. The example below shows that if you print the value of the word, you will see the this word:

```
print word
this
```

You can also obtain literal words from an unevaluated block. In the following example, the **first** function fetches the first word from the block. This word is then set to the word variable.

```
word: first [this and that]
```

Any word can be used as a literal. It may or may not refer to a value. For example, in the example below the word `here` has no value. The word **print** does have a value, but it can still be used as a literal because literal words are not evaluated.

```
word: 'here
print word
here
word: 'print
print word
print
```

The next example illustrates the importance of literal values:

```
video: [
    title "Independence Day"
    length 2:25:24
    date   4/july/1996
]
print select video 'title
Independence Day
```

In this example, the word `title` is searched for in a block. If the tick was missing from `title`, then its natural value would be used. If `title` has no natural value, an error is displayed.

See the Words Section in the Values Appendix for more information about word literals.

5.6 Unset Words

A word that has no value is `unset`. If an unset word is evaluated, an error will occur:

```
>> outlook
** Script Error: outlook has no value.
** Where: outlook
```

The error message in the previous example indicates that the word has not been set to a value. The word is unset. Do not confuse this with a word that has been set to `none`, which is a valid value.

A previously defined word can be unset at any time using **`unset`** :

```
unset 'word
```

When a word is unset, its value is lost.

To determine if a word has been set, use the **`value?`** function, which takes a literal word as its argument:

```
if not value? 'word [print "word is not set"]
word is not set
```

Determining whether a word is set can be useful in scripts that call other scripts. For instance, a script may set a default parameter that was not previously set:

```
if not value? 'test-mode [test-mode: on]
```

5.7 Protecting Words

You can prevent a word from being set with the **`protect`** function:

```
protect 'word
```

An attempt to redefine a protected word causes an error:

```
word: "here"
** Script Error: Word word is protected, cannot modify.
** Where: word: "here"
```

A word can be unprotected as well using **`unprotect`** :

```
unprotect 'word
```

```
word: "here"
```

The **protect** and **unprotect** functions also accept a block of words:

```
protect [this that other]
```

Important function and system words can be protected using the **protect-system** function. Protecting function and system words is especially useful for beginners who might accidentally set important words. If **protect-system** is placed in your user.r file, then all predefined words are protected.

6. Conditional Evaluation

As previously mentioned, blocks are not normally evaluated. A **do** function is required to force a block to be evaluated. There are times when you may need to conditionally evaluate a block. The following section describes several ways to do this.

6.1 Conditional Blocks

The **if** function takes two arguments. The first argument is a condition and the second argument is a block. If the condition is true , the block is evaluated, otherwise it is not evaluated.

```
if now/time > 12:00 [print "past noon"]
past noon
```

The condition is normally an expression that evaluates to true or false ; however, other values can also be supplied. Only a false or a none value prevents the block from being evaluated. All other values (including zero) are treated as true, and cause the block to be evaluated. This can be useful for checking the results of **find**, **select**, **next**, and other functions that return none :

```
string: "let's talk about REBOL"
if find string "talk" [print "found"]
found
```

The **either** function extends **if** to include a third argument, which is the block to evaluate if the condition is false:

```
either now/time > 12:00 [
    print "after lunch"
][
    print "before lunch"
]
after lunch
```

The **either** function also interprets a none value as false.

Both the **if** and **either** functions return the result of evaluating their blocks. In the case of an **if**, the block value is only returned if the block is evaluated; otherwise, a none is returned. The **if** function is useful for conditional initialization of variables:

```
flag: if time > 13:00 ["lunch eaten"]

print flag
lunch eaten
```

Making use of the result of the **either** function, the previous example could be rewritten as follows:

```
print either now/time > 12:00 [
    "after lunch"
][
    "before lunch"
]
after lunch
```

Since both **if** and **either** are functions, their block arguments can be any expression that results in a block when evaluated. In the following examples, words are used to represent the block argument for **if** and **either**.

```
notice: [print "Wake up!"]
if now/time > 7:00 notice
Wake up!
notices: [
    [print "It's past sunrise!"]
    [print "It's past noon!"]
    [print "It's past sunset!"]
]
if now/time > 12:00 second notices
It's past noon!
sleep: [print "Keep sleeping"]
either now/time > 7:00 notice sleep
Wake up!
```

The conditional expressions used for the first argument of both **if** and **either** can be composed from a wide variety of comparison and logic functions. Refer to the Math Chapter for more information.

Avoid This Common Mistake

The most commonly made mistake in REBOL is to forget the second block on **either** or add a second block to **if**. These examples both creator hard-to-find errors:

```
either age > 10 [print "Older"]

if age > 10 [print "Older"] [print "Younger"]
```

These types of errors may be difficult to detect, so keep this in mind if these functions do not seem to be doing what you expect.

6.2 Any and All

The **any** and **all** functions offer a shortcut to evaluating some types of conditional expressions. These functions can be used in a number of ways:either in conjunction with **if**, **either**, and other conditional functions, or separately.

Both **any** and **all** accept a block of expressions, which is evaluated one expression at a time. The **any** function returns on the first true expression, and the **all** function returns on the first false expression. Keep in mind that a false expression can also be none, and that a true expression is any value other than false **or** none.

The **any** function returns the first value that is not false, otherwise it returns none. The **all** function returns the last value if all the expressions are not false, otherwise it returns none.

Both the **any** and **all** functions only evaluate as much as they need. For example, once **any** has found a true expression, none of the remaining expressions are evaluated. Here is an example of using **any** :

```
size: 50
if any [size < 10 size > 90] [
    print "Size is out of range."
]
```

The behavior of **any** is also useful for setting default values. For example, the following lines set a number to 100, but only when its value is none :

```
number: none
print number: any [number 100]
100
```

Similarly, if you have various potential values, you can use the first one that actually has a value (is not none):

```
num1: num2: none
num3: 80
print number: any [num1 num2 num3]
80
```

You can use **any** with functions like **find** to always return a valid result:

```
data: [123 456 789]
print any [find data 432 999]
999
```

Similarly, **all** can be used for conditions that require all expressions to be true :

```
if all [size > 10 size < 90] [print "Size is in range"]
Size is in range
```

You can verify that values have been set up before evaluating a function:

```
a: "REBOL/"
b: none
probe all [string? a string? b append a b]
none
b: "Core"
probe all [string? a string? b append a b]
REBOL/Core
```

6.3 Conditional Loops

The **until** and **while** functions repeat the evaluation of a block until a condition is met.

The **until** function repeats a block until the evaluation of the block returns `true` (that is, not `false` or `none`). The evaluation block is always evaluated at least once. The **until** function returns the value of its block.

The example below will print each word in the color block. The block begins by printing the first word of the block. It then moves to the next color for each color in the block. The **tail?** function checks for the end of the block, and will return `true`, which will cause the **until** function to exit.

```
color: [red green blue]
until [
    print first color
    tail? color: next color
]
red
green
blue
```

The **break** function can be used to escape from the **until** loop at any time.

The **while** function repeats the evaluation of its two block arguments while the first block returns `true`. The first block is the condition block, the second block is the evaluation block. When the condition block returns `false` or `none`, the expression block will no longer be evaluated and the loop terminates.

Here is a similar example to that show above. The **while** loop will continue to print a color while there are still colors to print.

```
color: [red green blue]
while [not tail? color] [
    print first color
    color: next color
]
red
green
blue
```

The condition block can contain any number of expressions, so long as the last expression returns the condition. To illustrate this, the next example adds a print to the condition block. This will print the index value of the color. It will then check for the tail of the color block, which is the condition used for the loop.

```
color: [red green blue]
```

```
while [
    print index? color
    not tail? color
][
    print first color
    color: next color
]
1
red
2
green
3
blue
4
```

The last value of the block is returned from the **while** function.

A **break** can be used to escape from the loop at any time.

6.4 Common Mistakes

Conditional expressions are only false when they return `false` or `none`, and they are `true` when they return any other value. All of the conditional expressions in the following examples return `true`, even the zero and empty block values:

```
if true [print "yep"]
yep
if 1 [print "yep"]
yep
if 0 [print "yep"]
yep
if [] [print "yep"]
yep
```

The following conditional expressions return `false` :

```
if false [print "yep"]

if none [print "yep"]
```

Do not enclose conditional expressions in a block. Conditional expressions enclosed in blocks, always return a `true` result:

```
if [false] [print "yep"]
yep
```

Do not confuse **either** with **if**. For example, if you intend to write:

```
either some-condition [a: 1] [b: 2]
```

but write this instead:

```
if some-condition [a: 1] [b: 2]
```

the **if** function would ignore the second block. This would not cause an error, but the second block would never get evaluated.

The opposite is also true. If you write the following line, omitting a second block:

```
either some-condition [a: 1]
```

the **either** function will not evaluate the correct code and may produce an erroneous result.

7. Repeated Evaluation

The **while** and **until** functions above where used to loop until a condition was met. There are also several functions that let you loop for a specified a number of times.

7.1 Loop

The **loop** function evaluates a block a specified number of times. The following example prints a line of 40 dashes:

```
loop 40 [prin "-"]
----------------------------------------
```

Note that the **prin** function is similar to the **print** function, but prints its argument without a line termination.

The **loop** function returns the value of the final evaluation of the block:

```
i: 0
print loop 40 [i: i + 10]
400
```

7.2 Repeat

The **repeat** function extends **loop** by allowing you to monitor the loop counter. The **repeat** function's first argument is a word that will be used to hold the count value:

```
repeat count 3 [print ["count:" count]]
count: 1
count: 2
count: 3
```

The final block value is also returned:

```
i: 0
print repeat count 10 [i: i + count]
```

```
55
```

In the previous examples, the `count` word only has its value within the **`repeat`** block. In other words, the value of `count` is local to the block. After **`repeat`** finishes, `count` returns to any previous set value.

7.3 For

The **`for`** function extends **`repeat`** by allowing the starting value, the ending value, and the increment to the value to be specified. Any of the values can be positive or negative.

The example below begins at zero and counts to 50 by incrementing 10 each time through the loop.

```
for count 0 50 10 [print count]
0
10
20
30
40
50
```

The **`for`** function cycles through the loop up to and including the ending value. However, if the count exceeds the ending value, the loop is still terminated. The example below specifies an ending value of 55. That value will never be hit because the loop increments by 10 each time. The loop stops at 50.

```
for count 0 55 10 [prin [count " "]]
0 10 20 30 40 50
```

The next example shows how to count down. It begins at four and counts down to zero one at a time.

```
for count 4 0 -1 [print count]
4
3
2
1
0
```

The **`for`** function also works for decimal numbers, money, times, dates, series, and characters. Be sure that both the starting and ending values are of the same datatype. Here are several examples of using the **`for`** loop with other datatypes.

```
for count 10.5 0.0 -1 [prin [count " "]]
10.5 9.5 8.5 7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5
for money $0.00 $1.00 $0.25 [prin [money " "]]
$0.00 $0.25 $0.50 $0.75 $1.00
for time 10:00 12:00 0:20 [prin [time " "]]
10:00 10:20 10:40 11:00 11:20 11:40 12:00
for date 1-jan-2000 4-jan-2000 1 [prin [date " "]]
1-Jan-2000 2-Jan-2000 3-Jan-2000 4-Jan-2000
for char #"a" #"z" 1 [prin char]
```

```
abcdefghijklmnopqrstuvwxyz
```

The **for** function also works on series. The following example uses **for** on a string value. The word end is defined as the string with its current index at the d character. The **for** function moves through the string series one character at a time and stops when it reaches the character position defined to end:

```
str: "abcdef"
end: find str "d"
for s str end 1 [print s]
abcdef
bcdef
cdef
def
```

7.4 Foreach

The **foreach** function provides a convenient way to repeat the evaluation of a block for each element of a series. It works for all types of block and string series.

In the example below, each word in the block will be printed:

```
colors: [red green blue]
foreach color colors [print color]
red
green
blue
```

In the next example, each character in a string will be printed:

```
string: "REBOL"
foreach char string [print char]
R
E
B
O
L
```

In the example below, each filename in a directory block will be printed:

```
files: read %.
foreach file files [
    if find file ".t" [print file]
]
file.txt
file2.txt
newfile.txt
output.txt
```

When a block contains groups of values that are related, the **foreach** function can fetch all the values of the group at the same time. For example, here is a block that contains a time, string, and price. By providing the **foreach** function with a block of words for the group, each of their

values can be fetched and printed.

```
movies: [
     8:30 "Contact"      $4.95
    10:15 "Ghostbusters" $3.25
    12:45 "Matrix"       $4.25
]

foreach [time title price] movies [
    print ["watch" title "at" time "for" price]
]
watch Contact at 8:30 for $4.95
watch Ghostbusters at 10:15 for $3.25
watch Matrix at 12:45 for $4.25
```

In the above example, the **foreach** value block, [time title price], specifies that three values are to be fetched from movies for each evaluation of the block.

The variables used to hold the **foreach** values are local to the block. Their value are only set within the block that is being repeated. Once the loop has exited, the variables return to their previously set values.

7.5 Forall and Forskip

Similar to **foreach**, the **forall** function evaluates a block for every value in a series. However, there are some important differences. The **forall** function is handed the series that is set to the beginning of the loop. As it proceeds through the loop, **forall** modifies the position within the series.

```
colors: [red green blue]
forall colors [print first colors]
red
green
blue
```

In the above example, after each evaluation of the block, the series is advanced to its next position. When **forall** returns, the color index is at the tail of the series.

To continue to use the series you will need to return it to its head position with the following line:

```
colors: head colors
```

The **forskip** function evaluates a block for groups of values in a series. The second argument to **forskip** is the count of how many elements to move forward after each cycle of the loop.

Like **forall**, **forskip** is handed the series with the series index set to where it is to begin. Then, **forskip** modifies the index position as it continues the loop. After each evaluation of the body block, the series index is advanced by the skip amount to its next index position. The following example demonstrates **forskip** :

```
movies: [
     8:30 "Contact"      $4.95
    10:15 "Ghostbusters" $3.25
    12:45 "Matrix"       $4.25
```

```
]

forskip movies 3 [print second movies]
Contact
Ghostbusters
Matrix
```

In the above example, **forskip** returns with the movies series at its tail position. You will need to use the **head** function to return the series back to its head position.

7.6 Forever

The **forever** function evaluates a block endlessly or until a it encounters the **break** function.

The following example uses **forever** to check for the existence of a file every ten minutes:

```
forever [
    if exists? %datafile [break]
    wait 0:10
]
```

7.7 Break

You can stop the repeated evaluation of a block with the **break** function. The **break** function is useful when a special condition is encountered and the loop must be stopped. The **break** function works with all types of loops.

In the following example, the loop will **break** if a number is greater than 5.

```
repeat count 10 [
    if (random count) > 5 [break]
    print "testing"
]
testing
testing
testing
```

The **break** function does not return a value from the loop unless a **/return** refinement is used:

```
print repeat count 10 [
    if (random count) > 5 [break/return "stop here"]
    print "testing"
    "normal exit"
]
testing
testing
testing
stop here
```

In the above example, if the repeat terminates without the condition occurring, the block returns the string normal exit. Otherwise, **break/return** will return the string stop here.

8. Selective Evaluation

There are several methods to selectively evaluate expressions in REBOL. These methods provide a way for evaluation to branch many different ways, based on a key value.

8.1 Select

The **select** function is often used to obtain a particular value or block, given a target value. If you define a block of values and actions, you can use **select** to search for the action that corresponds to a value.

```
cases: [
    center [print "center"]
    right  [print "right"]
    left   [print "left"]
]
action: select cases 'right
if action [do action]
right
```

In the previous example, the `select` function finds the word `right` and returns the block that follows it. (If for some reason the block was not found, then `none` would have been returned.) The block is then evaluated. The values used in the example are words, but they can be any kind of value:

```
cases: [
     5:00 [print "everywhere"]
    10:30 [print "here"]
    18:45 [print "there"]
]
action: select cases 10:30
if action [do action]
here
```

8.2 Switch

The **select** function is used so often that there is a special version of it called **switch**, which includes the evaluation of the resulting block. The **switch** function makes it easier to perform inline selective evaluation. For instance, to switch on a simple numeric case:

```
switch 22 [
    11 [print "here"]
    22 [print "there"]
]
there
```

The **switch** function also returns the value of the block it evaluates, so the previous example can also be written as:

```
str: copy "right "

print switch 22 [
    11 [join str "here"]
    22 [join str "there"]
]
```

```
right there
```

and:

```
car: pick [Ford Chevy Dodge] random 3
print switch car [
    Ford  [351 * 1.4]
    Chevy [454 * 5.3]
    Dodge [154 * 3.5]
]
2406.2
```

The cases can be any valid datatype, including numbers, strings, words, dates, times, urls, and files. Here are some examples:

Strings:

```
person: "kid"
switch person [
    "dad" [print "here"]
    "mom" [print "there"]
    "kid" [print "everywhere"]
]
everywhere
```

Words:

```
person: 'kid
switch person [
    dad [print "here"]
    mom [print "there"]
    kid [print "everywhere"]
]
everywhere
```

Datatypes:

```
person: 123
switch type?/word [
    string! [print "a string"]
    binary! [print "a binary"]
    integer! [print "an integer number"]
    decimal! [print "a decimal number"]
]
an integer number
```

Files:

```
file: %rebol.r
switch file [
    %user.r [print "here"]
```

```
    %rebol.r [print "everywhere"]
    %file.r [print "there"]
]
everywhere
```

URLs:

```
url: ftp://ftp.rebol.org
switch url [
    http://www.rebol.com [print "here"]
    http://www.cnet.com [print "there"]
    ftp://ftp.rebol.org [print "everywhere"]
]
everywhere
```

Tags:

```
tag: <LI>
print switch tag [
    <PRE>   ["Preformatted text"]
    <TITLE> ["Page title"]
    <LI>    ["Bulleted list item"]
]
Bulleted list item
```

Times:

```
time: 12:30
switch time [
     8:00 [send wendy@domain.dom "Hey, get up"]
    12:30 [send cindy@rebol.dom "Join me for lunch."]
    16:00 [send group@every.dom "Dinner anyone?"]
]
```

8.2.1 Default Case

A default case can be specified when none of the other cases match. Use the **default** refinement to specify a default:.

```
time: 7:00
switch/default time [
     5:00 [print "everywhere"]
    10:30 [print "here"]
    18:45 [print "there"]
] [print "nowhere"]
nowhere
```

8.2.2 Common Cases

If you have common cases, where the result would be the same for several values, you can define a word to hold a common block of code:

```
case1: [print length? url]    ; the common block

url: http://www.rebol.com
switch url [
    http://www.rebol.com case1
    http://www.cnet.com [print "there"]
    ftp://ftp.rebol.org case1
]
20
```

8.2.3 Other Cases

More than just blocks can be evaluated for cases. This example evaluates the file that corresponds to a day of the week:

```
switch now/weekday [
    1 %monday.r
    5 %friday.r
    6 %saturday.r
]
```

So, if it's Friday, the `friday`.r file is evaluated and its result is returned from the switch. This type of evaluation also works for URLs:

```
switch time [
     8:30 ftp://ftp.rebol.org/wakeup.r
    10:30 http://www.rebol.com/break.r
    18:45 ftp://ftp.rebol.org/sleep.r
]
```

The cases for **`switch`** are enclosed in a block, and therefore can be defined apart from the switch statement:

```
schedule: [
     8:00 [send wendy@domain.dom "Hey, get up"]
    12:30 [send cindy@dom.dom "Join me for lunch."]
    16:00 [send group@every.dom "Dinner anyone?"]
]

switch 8:00 schedule
```

9. Stopping Evaluation

Evaluation of a script can be stopped at any time by pressing the escape key (ESC) on the keyboard or by using the **`halt`** and **`quit`** functions.

The **`halt`** function stops evaluation and returns you to the REBOL console prompt:

```
if time > 12:00 [halt]
```

The **quit** function stops evaluation and exits the REBOL interpreter:

```
if error? try [print test] [quit]
```

10. Trying Blocks

There are times when you want to evaluate a block, but should an error occur, you do not want to stop the evaluation of the rest of your script.

For example, you might be performing a number division, but do not want your script to stop if a divide-by-zero occurs.

The **try** function allows you to catch errors during the evaluation of a block. It is almost identical to **do**. The **try** function will normally return the result of the block; however, if an error occurs, it will return an error value instead.

In the following example, when the divide by zero occurs, the script will pass an error back to the try function, and evaluation will continue from that point.

```
for num 5 0 -1 [
    if error? try [print 10 / num] [print "error"]
]
2
2.5
3.33333333333333
5
10
error
```

More about error handling can be found in the Errors Appendix.

5 - Scripts

1. Overview
    1.1 File Suffix
    1.2 Structure
2. Headers
3. Script Arguments
    3.1 Program Options
4. Running Scripts
    4.1 Loading Scripts
    4.2 Saving Scripts
    4.3 Commenting Scripts
5. Style Guide
    5.1 Formatting
    5.2 Word Names
    5.3 Script Headers
    5.4 Function Headers
    5.5 Script File Names
    5.6 Embedded Examples
    5.7 Embedded Debugging
    5.8 Minimize Globals
6. Script Cleanup

1. Overview

The term ***script*** refers not only to single files that are evaluated but also to ***source*** text embedded within other types of files (such as, web pages), or ***fragments*** of source text that are saved as data files or passed as messages.

1.1 File Suffix

REBOL scripts typically append a .r suffix to file names; however, this convention is not required. The interpreter reads files with any suffix and scans the contents for a valid REBOL script header.

1.2 Structure

The structure of a script is free-form. Indentation and spacing can be used to clarify the structure and content of the script. In addition, you are encouraged to use the standard scripting style to make scripts more universally readable. See the Style Guide for more information.

2. Headers

Directly preceding the script body, every script must have a header that identifies its purpose and other script attributes. A header can contain the script name, author, date, version, file name, and additional information. REBOL data files that are not intended for direct evaluation do not require a header.

Headers are useful for several reasons.

- They identify a script as being valid source text for the REBOL interpreter.
- The interpreter uses the header to print out the script's title and determine what resources and versions it needs before evaluating the script.

- Headers provide a standard way to communicate the title, purpose, author, and other details of scripts. You can often determine from a script's header if a script interests you.
- Script archives and web sites use headers for generating script directories, categories, and cross references.
- Some text editors access and update a script's header to keep track of information such as the author, date, version, and history.

The general form of a script header is:

```
REBOL [block]
```

For the interpreter to recognize the header, the block must immediately follow the word **REBOL**. Only white space (spaces, tabs, and lines) is permitted between the word **REBOL** and the block.

The block that follows the **REBOL** word is an object definition that describes the script. The preferred minimal header is:

```
REBOL [
    Title:  "Scan Web Sites"
    Date:   2-Feb-2000
    File:   %webscan.r
    Author: "Jane Doer"
    Version: 1.2.3
]
```

When a script is loaded, the header block is evaluated and its words are set to their defined values. These values are used by the interpreter and can also be used by the script itself.

Note that words defined as a single value can also be defined as multiple values by providing them in a block:

```
REBOL [
    Title: "Scan Web Sites"
    Date:   12-Nov-1997
    Author: ["Ema User" "Wasa Writer"]
]
```

Headers can be more complex, providing information about the author, copyright, formatting, version requirements, revision history, and more. Because the block is used to construct the header object, it can also be extended with new information. This means that a script can extend the header as needed, but it should be done with care to avoid ambiguous or redundant information.

A full header might look something like this:

```
REBOL [
    Title:   "Full REBOL Header Example"
    Date:    8-Sep-1999
    Name:    'Full-Header  ; For window title bar

    Version: 1.1.1
    File:    %headfull.r
    Home:    http://www.rebol.com/rebex/

    Author:  "Carl Sassenrath"
```

```
    Owner:   "REBOL Headquarters"
    Rights:  "Copyright (C) Carl Sassenrath 1999"

    Needs:   [2.0 ODBC]
    Tabs:    4

    Purpose: {
        The purpose or general reason for the program
        should go here.
    }

    Note: {
        An important comment or notes about the program
        can go here.
    }

    History: [
        0.1.0 [5-Sep-1999 "Created this example" "Carl"]
        0.1.1 [8-Sep-1999 {Moved the header up, changed
            comment on extending the header, added
            advanced user comment.} "Carl"]
    ]

    Language: 'English
]
```

2 Prefaced Scripts

Script text does not need to begin with a header. Scripts can begin with any text, allowing them to be inserted into email messages, web pages, and other files.

The header marks the beginning of the script, and the text that follows is the body of the script. Text that appears before the header is called the ***preface*** and is ignored during evaluation.

```
The text that appears before the header is ignored
by REBOL and can be used for comments, email headers,
HTML tags, etc.

REBOL [
  Title:   "Preface Example"
  Date:    8-Jul-1999
]

print "This file has a preface before the header"
```

2 Embedded Scripts

If a script is to be followed by other text unrelated to the script itself, the script must be enclosed with square brackets [ ]:

```
Here is some text before the script.
[
    REBOL [
        Title:   "Embedded Example"
        Date:    8-Nov-1997
    ]
    print "done"
]
```

```
Here is some text after the script.
```

Only white space is permitted between the initial bracket and the word REBOL.

3. Script Arguments

When a script is evaluated, it has access to information about itself. This is found in the `system/script` object. The object contains the fields listed in Object Fields for system/script.

Header	The header object of the script. This can be used to access the script's title, author, version, date, and other fields.
Parent	If the script was evaluated from another script, this is the system/script object for the parent script.
Path	The file directory path or URL to the script being evaluated.
Args	The arguments to the script. These are passed from the operating system command line or from the do function that was used to evaluate the script.

Examples of using the script object are:

```
print system/script/title

print system/script/header/date

do system/script/args

do system/script/path/script.r
```

The last example evaluates a script called `script`.r in the same directory as the script that is currently running.

3.1 Program Options

Scripts also have access to the options provided to the REBOL interpreter when it was started. These are found in the system/options object. The object contains the fields listed in Object Fields for system/options.

Home	The file path as determined by your operating system's environment. This is the path set in the HOME environment variable or system registry for systems that support it. This is the path used to find the rebol.r and user.r files.
Script	The file name of the initial script provided when the interpreter was launched.
Path	The path to the current directory.
Args	The initial arguments provided to the interpreter on the command line.
Do-arg	The string provided as an argument to the --do option on the command line.

The system/options object also contains additional options that were provided on the command line. Type

```
probe system/options
```

to examine the contents of the options object.

Examples:

```
print system/options/script

probe system/options/args

print read system/options/home/user.r
```

4. Running Scripts

There are two ways to run a script: as the initial script when the REBOL interpreter is started, or from the **do** function.

To run a script when starting the interpreter, provide the script name on the command line following the REBOL program name:

```
rebol script.r
```

As soon as the interpreter initializes, the script is evaluated.

From the **do** function, provide the script file name or URL as an argument. The file is loaded into the interpreter and evaluated:

```
do %script.r

do http://www.rebol.com/script.r
```

The **do** function returns the result of the script when it finishes evaluation.

Note that the script file must include a valid REBOL header.

4.1 Loading Scripts

Script files can be loaded as data with the **load** function. This function reads the script and translates the script into values, words, and blocks, but does not evaluate the script. The result of the **load** function is a block, unless only a single value was loaded, then that value is returned.

The script argument to the **load** function is a file name, URL, or a string.

```
load %script.r
load %datafile.txt
load http://www.rebol.org/script.r
load "print now"
```

The **load** function performs the following steps:

- Reads the text from the file, URL, or string.
- Searches for a script header, if present.
- Translates data beginning after the header, if found.
- Returns a block containing the translated values.

For example, if a script file buy.r contained the text:

```
Buy 100 shares at $20.00 per share
```

it could be loaded with the line:

```
data: load %buy.r
```

which would result in a block:

```
probe data
[Buy 100 shares at $20.00 per share]
```

> It should be noted that the "Buy" example above is a dialect of REBOL, not directly executable code. See Chapter 4 on Expressions or Chapter 15 on Parsing for more information.

Note that a file does not require a header to be loaded. The header is necessary only if the file is to be run as a script.

The **load** function supports a few refinements. The load Function Refinements lists the refinements and a description of their functionality:

/header	Includes the header if present.
/next	Loads only the next value, one value at a time. This is useful for parsing REBOL scripts.
/markup	Treats the file as an HTML or XML file and returns a block that holds its tags and text.

Normally, **load** does not return the header from the script. But, if the **/header** refinement is used the returned block contains the header object as its first argument.

The **/next** refinement loads the next value and returns a block containing two values. The first returned value is the next value from the series. The second returned value is the string position immediately following the last item loaded.

The **/markup** refinement loads HTML and XML data as a block of tags and strings. All tags are tag data types. All other data are treated as strings.

If the following file contents where loaded with **load/markup:**

```
<title>This is an example</title>
```

a block would be produced:

```
probe data
[<title> "This is an example" </title>]
```

4.2 Saving Scripts

Data can be saved to a script file in a format that can be loaded into REBOL with the **load** function. This is a useful way to save data values and blocks of data. In this fashion, it is possible to create entire mini-databases.

The **save** function expects two arguments: a file name and either a block or a value to be saved:

```
data: [Buy 100 shares at $20.00 per share]

save %data.r data
```

The data is written out in REBOL source text format, which can be loaded later with:

```
data: load %data.r
```

Simple values can also be saved and loaded. For instance, a date stamp can be saved with:

```
save %date.r now
```

and later reloaded with:

```
stamp: load %date.r
```

In the previous example, because stamp is a single value, it is not enclosed in a block when loaded.

To save a script file with a header, the header can be provided in a refinement as either an object or a block:

```
header: [Title: "This is an example"]

save/header %data.r data header
```

4.3 Commenting Scripts

Commenting is useful for clarifying the purpose of sections of a script. Script headers provide a high level description of the script and comments provide short descriptions of functions. It is also a good idea to provide comments for other parts of your code as well.

A single-line comment is made with a semicolon. Everything following the semicolon to the end of the line is part of the comment:

```
zertplex: 10   ; set to the highest quality
```

You can also use strings for comments. For instance, you can create multi-line comments with a string enclosed in braces:

```
{
    This is a long multilined comment.
}
```

This technique of commenting works only when the string is not interpreted as an argument to a function. If you want to make sure that a multi-line comment is recognized as a comment and is not interpreted as code, precede the string with the word **comment** :

```
comment {
    This is a long multilined comment.
}
```

The **comment** function tells REBOL to ignore the following block or string. *Note that string and block comments are actually part of the script block. Care should be taken to avoid placing them in data blocks, because they would appear as part of the data.*

5. Style Guide

REBOL scripts are free-form. You can write a script using the indenting, spacing, line length, and line terminators you prefer. You can put each word on a separate line or join them together on one long line.

While the formatting of your script does not affect the interpreter, it does affect its human readability. Because of this, REBOL Technologies encourages you to follow the standard scripting style described in this section.

Of course, you don't have to follow any of these suggestions. However, scripting style is more important than it first seems. It can make a big difference in the readability and reuse of scripts. Users may judge the quality of your scripts by the clarity of their style. Sloppy scripts often mean sloppy code. Experienced script writers usually find that a clean, consistent style makes their code easier to produce, maintain, and revise.

5.1 Formatting

Use the following guidelines for formatting REBOL scripts for clarity.

5.1.1 Indent Content for Clarity

The contents of a block are indented, but the block's enclosing square brackets [ ] are not. That's because the square brackets belong to the prior level of syntax, as they define the block but are not contents of the block. Also, it's easier to spot breaks between adjacent blocks when the brackets stand out.

Where possible, an opening square bracket remains on the line with its associated expression. The closing bracket can be followed by more expressions of that same level. These same rules apply equally to parenthesis ( ) and braces { }.

```
if check [do this and that]

if check [
    do this and do that
    do another thing
    do a few more things
]

either check [do something short][
```

```
    do something else]

either check [
    when an expression extends
    past the end of a block...
][
    this helps keep things
    straight
]

while [
    do a longer expression
    to see if it's true
][
    the end of the last block
    and start of the new one
    are at the WHILE level
]

adder: func [
    "This is an example function"
    arg1 "this is the first arg"
    arg2 "this is the second arg"
][
    arg1 + arg2
]
```

An exception is made for expressions that normally belong on a single line, but extend to multiple lines:

```
if (this is a long conditional expression that
    breaks over a line and is indented
)[
    so this looks a bit odd
]
```

This also applies to grouped values that belong together, but must be wrapped to fit on the line:

```
[
    "Hitachi Precision Focus" $1000 10-Jul-1999
        "Computers Are Us"

    "Nuform Natural Keyboard" $70 20-Jul-1999
        "The Keyboard Store"
]
```

5.1.2 Standard Tab Size

REBOL standard tab size is four spaces. Because people use different editors and readers for scripts, you can elect to use spaces rather than tabs.

5.1.3 Detab Before Posting

The tab character (ASCII 9) does not indent four spaces in many viewers, browsers, or shells, so use an editor or REBOL to detab a script before publishing it to the net. The following function detabs a file with standard four-space tabs:

```
detab-file: func [file-name [file!]] [
    write file-name detab read file-name
]
detab-file %script.r
```

The following function converts an eight-space tabs to four-space tabs:

```
detab-file: func [file-name [file!]] [
    write file-name detab entab/size read file-name 8
]
```

5.1.4 Limit Line Lengths to 80 Characters

For ease of reading and portability among editors and email readers, limit lines to 80 characters. Long lines that get wrapped in the wrong places by email clients are difficult to read and have problems loading.

5.2 Word Names

Words are a user's first exposure to your code, so it is critical to choose words carefully. A script should be clear and concise. When possible, the words should relate to their English or other human language equivalent, in a simple, direct way.

The following are standard naming conventions for REBOL.

5.2.1 Use the Shortest Word that Communicates the Meaning

Short, crisp words work best where possible:

```
size  time  send  wait  make  quit
```

Local words can often be shortened to a single word. Longer, more descriptive words are better for global words.

5.2.2 Use Whole Words Where Possible

What you save when abbreviating a word is rarely worth it. Type date not dt, or image-file not imgfl.

5.2.3 Hyphenate Multiple Word Names

The standard style is to use hyphens, not character case, to distinguish words.

```
group-name image-file  clear-screen  bake-cake
```

5.2.4 Begin Function Names with a Verb

Function names begin with a verb and are followed by a noun, adverb, or adjective. Some nouns can also be used as verbs.

```
make  print  scan  find  show  hide  take
```

```
rake-coals   find-age   clear-screen
```

Avoid unnecessary words. For instance, quit is just as clear as quit-system.

When using a noun as a verb, use special characters such as ? where applicable. For instance, the function for getting the length of a series is `length?`. Other REBOL functions using this naming convention are:

```
size?   dir?   time?   modified?
```

5.2.5 Begin Data Words with Nouns

Words for objects or variables that hold data should begin with a noun. They can include modifiers (adjectives) as needed:

```
image   sound   big-file   image-files   start-time
```

5.2.6 Use Standard Names

There are standard names in REBOL that should be used for similar types of operations. For instance:

```
make-blub           ;creating something new
free-blub           ;releasing resources of something
copy-blub           ;copying the contents of something
to-blub             ;converting to it
insert-blub         ;inserting something
remove-blub         ;removing something
clear-blub          ;clearing something
```

5.3 Script Headers

The advantage of using headers is clear. Headers give users a summary of a script and allow other scripts to process the information (like a cataloging script). A minimum header provides a title, date, file name and purpose. Other fields can also be provided such as author, notes, usage, and needs.

```
REBOL [
    Title: "Local Area Defringer"
    Date:  1-Jun-1957
    File:  %defringe.r
    Purpose: {
        Stabilize the wide area ignition transcriber
        using a double ganged defringing algorithm.
    }
]
```

5.4 Function Headers

It is useful to provide a description in function specification blocks. Limit such text to one line of 70 characters or less. Within the description, mention what type of value the function normally

returns.

```
defringe: func [
    "Return the defringed localization radius."
    area "Topo area to defringe"
    time "Time allotted for operation"
    /cost num "Maximum cost permitted"
    /compound "Compound the calculation"
][
    ...code...
]
```

5.5 Script File Names

The best way to name a file is to think about how you can best find that file in a few months. Short and clear names are often enough. Plurals should be avoided, unless meaningful.

In addition, when naming a script, consider how the name will sort in a directory. For instance, keep related files together by starting them with a common word.

```
%net-start.r
%net-stop.r
%net-run.r
```

5.6 Embedded Examples

Where appropriate, provide examples within a script to show how the script operates and to give users a quick way of verifying that the script works correctly on their system.

5.7 Embedded Debugging

It is often useful to build in debugging functions as part of the script. This is especially true of networking and file handling scripts where it is not desirable to send and write files while running in test mode. Such tests can be enabled with a control variable at the head of the script.

```
verbose: on
check-data: off
```

5.8 Minimize Globals

In large scripts and where possible, avoid using global variables that carry their internal state from one module or function to another. For short scripts, this isn't always practical. But recognize that short scripts may become longer scripts over time.

If you have a collection of global variables that are closely related, consider using an object to keep track of them:

```
user: make object! [
    name:  "Fred Dref"
    age:   94
    phone: 707-555-1234
    email: dref@fred.dom
```

```
]
```

6. Script Cleanup

Here is a short script that can be used to clean up the indentation of a script. It works by parsing the REBOL syntax and reconstructing each line of the script. This example can be found in the REBOL Script Library at www.REBOL.com.

```
out: none ; output text
spaced: off ; add extra bracket spacing
indent: "" ; holds indentation tabs

emit-line: func [] [append out newline]

emit-space: func [pos] [
    append out either newline = last out [indent] [
        pick [#" " ""] found? any [
            spaced
            not any [find "[(" last out
                     find ")]" first pos]
        ]
    ]
]

emit: func [from to] [
    emit-space from append out copy/part from to
]

clean-script: func [
    "Returns new script text with standard spacing."
    script "Original Script text"
    /spacey "Optional spaces near brackets/parens"
    /local str new
] [
    spaced: found? spacey
    out: append clear copy script newline
    parse script blk-rule: [
        some [
            str:
            newline (emit-line) |
            #";" [thru newline | to end] new:
                (emit str new) |
            [#"[" | #"("]
                (emit str 1 append indent tab)
                blk-rule |
            [#"]" | #")"]
                (remove indent emit str 1) |
            skip (set [value new]
                load/next str emit str new) :new
        ]
    ]
    remove out ; remove first char
]

script: clean-script read %script.r

write %new-script.r script
```

6 - Series

1. Basic Concepts
    1.1 Traversing a Series
    1.2 Skipping Around
    1.3 Extracting Values
    1.4 Extracting a Sub-series
    1.5 Inserting and Appending
    1.6 Removing Values
    1.7 Changing Values
2. Series Functions
    2.1 Creation Functions
    2.2 Navigation Functions
    2.3 Information Functions
    2.4 Extraction Functions
    2.5 Modification Functions
    2.6 Search Functions
    2.7 Ordering Functions
    2.8 Data Set (Group) Functions
3. Series Datatypes
    3.1 Block Types
    3.2 String Types
    3.3 Pseudo-types
    3.4 Type Test Functions
4. Series Information
    4.1 Length?
    4.2 Head?
    4.3 Tail?
    4.4 Index?
    4.5 Offset?
5. Making and Copying Series
    5.1 Partial Copies
    5.2 Deep Copies
    5.3 Initial Copies
6. Series Iteration
    6.1 Foreach Loop
    6.2 While Loop
    6.3 Forall Loop
    6.4 Forskip Loop
    6.5 The Break Function
7. Searching Series
    7.1 Simple Find
    7.2 Refinement Summary
    7.3 Partial Searches
    7.4 Tail Positions
    7.5 Backward Searches
    7.6 Repeated Searches
    7.7 Matching
    7.8 Wildcard Searches
    7.9 Select
    7.10 Search and Replace
8. Sorting Series
    8.1 Simple Sorting
    8.2 Group Sorting
    8.3 Comparison Functions
9. Series as Data Sets

9.1 Unique
9.2 Intersect
9.3 Union
9.4 Exclude
9.5 Difference
9.6 Exclude
10. Multiple Series Variables
11. Modification Refinements
11.1 Part
11.2 Only
11.3 Dup

1. Basic Concepts

The concept of a series is simple, and it is the fundamental concept that is used everywhere and for nearly everything in REBOL. In order to understand REBOL, you must understand how to create and manipulate series.

A **series** is a **set of values** arranged in a **specific order**.

It's that simple. For example, these are all series:

```
1 2 3 4

A B C D

"ABCD"

10:30 4:20 7:11
```

There are many types of series in REBOL. A block, a string, a list, a URL, a path, an email, a file, a tag, a binary, a bitset, a port, a hash, an issue, and an image are all series and can be accessed and processed in the same way with the same small set of series functions.

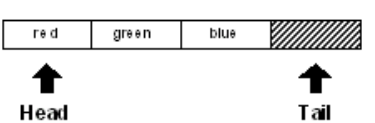
1.1 Traversing a Series

Since a series is an ordered set of values, you can traverse it from one position to another. As an example, take a series of three colors defined by the following block:

```
colors: [red green blue]
```

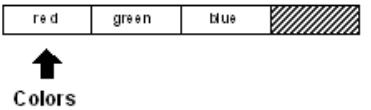
There is nothing special about this block. It is a series containing three words. It has a set of values: red, green, and blue. The values are organized into a specific order: red is first, green is second, and blue is third.

The first position of the block is called its `head`. This is the position occupied by the word `red`. The last position of the block is called its tail. This is the position immediately after the last word in the block. If you were to draw a diagram of the block, it would look like this:



Notice that the tail is just past the end of the block. The importance of this will become more clear shortly.

The variable `colors` is used to refer to the block. It is currently set to the head of the block:




```
print head? colors
true
```

The `colors` variable is at the first index position of the block.

```
print index? colors
1
```

The block has a length of three:

```
print length? colors
3
```

The first item in the block is:

```
print first colors
red
```

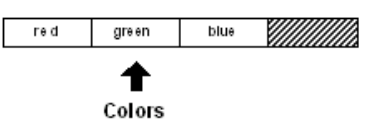
The second item in the block is:

```
print second colors
green
```

You can reposition the `colors` variable in the block using various functions. To move the `colors` variable to the next position in the colors block, use the **`next`** function:

```
colors: next colors
```

The **`next`** function moves forward one value in the block and returns that position as a result. The `colors` variable is now set to that new position:



The position of the `colors` variable has changed. Now the variable is no longer at the head of the block:

```
print head? colors
false
```

It is at the second position in the block:

```
print index? colors
2
```

However, if you obtain the first item of colors, you get:

```
print first colors
green
```

The position of the value that is returned by the **first** function is relative to the position that colors has in the block. The returned value is not the first color in the block, but the first color immediately following the current position of the block.

Similarly, if you ask for the length or the second color, you find that these are relative as well:

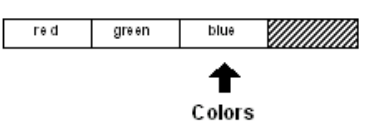
```
print length? Colors
2
print second colors
blue
```

You could move to the next position, and get a similar set of results:

```
colors: next colors

print index? colors
3
print first colors
blue
print length? colors
1
```

The block diagram now looks like this:



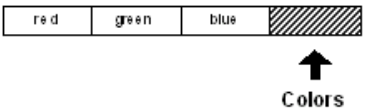
The colors variable is now at the last color in the block, but it is not yet to the tail position.

```
print tail? colors
false
```

To reach the tail, it has to be moved to the next position:

```
colors: next colors
```

Now the `colors` variable is resting at the tail of the block. It is no longer positioned at a valid color. It is past the end of the block.



If you try your code, you will get:

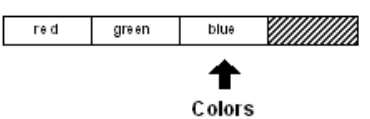
```
print tail? colors
true
print index? colors
4
print length? Colors
0
print first colors
** Script Error: Out of range or past end.
** Where: print first colors
```

You receive an error in this last case because there is no valid first item when you are past the end of the block.

It is also possible to move backwards in the block. If you write:

```
colors: back colors
```

you will move the `colors` variable back one position in the series:

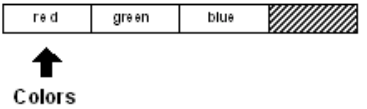


All of the same code will work as before:

```
print index? colors
3
print first colors
blue
```

1.2 Skipping Around

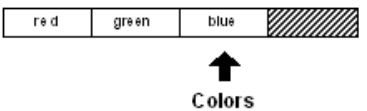
The previous examples move through the series one item at a time. However, there are times when you want to skip past multiple items using the **`skip`** function. Assume that the `colors` variable is positioned at the head of a series:



You can skip forward two items using:

```
colors: skip colors 2
```

The **skip** function is similar to **next** in that **skip** returns the series at the new position.



The following code confirms the new position:

```
print index? colors
3
print first colors
blue
```

To move backward, use **skip** with negative values:

```
colors: skip colors -1
```

This is similar to **back**. In the above example, a skip of -1 moves back one item.

```
print first colors
green
```

Note that you cannot skip past the tail or the head of a series. If you attempt to do so, **skip** only goes as far as it can. It will not generate an error.

If you skip too far forward, **skip** returns the tail of the series:

```
colors: skip colors 20

print tail? colors
true
```

If you skip too far back, **skip** returns the head of the series:

```
colors: skip colors -100

print head? colors
true
```

To skip directly to the head of the series, use the **head** function:

```
colors: head colors

print head? colors
true
print first colors
red
```

You can return to the tail with the **tail** function:

```
colors: tail colors

print tail? colors
true
```

1.3 Extracting Values

Some of the previous examples made use of the **first** and **second** ordinal functions to extract specific values from a series. The full set of ordinal functions is:

```
first
second
third
fourth
fifth
last
```

Ordinal functions are provided as a convenience, and are used for picking values from the most common position in a series. Here are some examples:

```
colors: [red green blue gold indigo teal]

print first colors
red
print third colors
blue
print fifth colors
indigo
print last colors
teal
```

To extract from a numeric position, use the pick function:

```
print pick colors 3
```

```
blue
print pick colors 5
indigo
```

A shorthand notation for pick is to use a path:

```
print colors/3
blue
print colors/5
indigo
```

Remember, as shown earlier, extraction is performed `relative` to the series variable that you provide. If the `colors` variable were at another position in the series, the results would be different.

Extracting a value past the end of its series generates an error in the case of the ordinal functions and returns **`none`** in the case of the **`pick`** function or a pick path:

```
print pick colors 10
none
print colors/10
none
```

1.4 Extracting a Sub-series

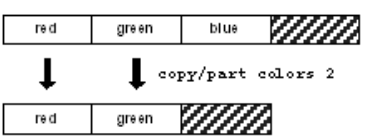
You can extract multiple values from a series with the **`copy`** function. To do so, use **`copy`** with the **`/part`** refinement, which specifies the number of values that you want to extract:

```
colors: [red green blue]

sub-colors: copy/part colors 2

probe sub-colors
[red green]
```

Graphically, this would look like:

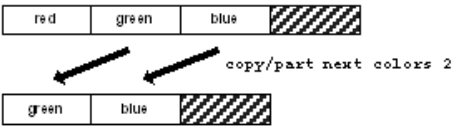


To copy a sub-series from any position within the series, first traverse to the starting position. The following example moves forward to the second position in the series using **`next`** before performing the copy:

```
sub-colors: copy/part next colors 2

probe sub-colors
[green blue]
```

This would be diagrammed as:



The length of the series to copy can be specified as an ending position, as well as a copy count. Note that the position indicates `where` the copy should stop, not the ending position.

```
probe copy/part colors next colors
[red]
probe copy/part colors back tail colors
[red green]
probe copy/part next colors back tail colors
[green]
```

This can be useful when the ending position is found as the result of the **`find`** function:

```
file: %image.jpg

print copy/part file find file "."
image
```

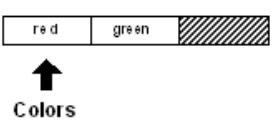
1.5 Inserting and Appending

You can insert one or more new values into any part of a series using the **`insert`** function. When you insert a value at a position in a series, space is made by shifting its prior values toward the tail of the series.

For instance, the block:

```
colors: [red green]
```

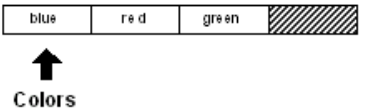
would be shown as:



To insert a new value at the head of the block where the `colors` variable is now positioned:

```
insert colors 'blue
```

The `red` and `green` words are shifted over and the `blue` word (which is prefixed with a tick because it is a word and should not be evaluated) is inserted at the head of the list.



Note that the `colors` variable remains positioned at the head of the list.

```
probe colors
[blue red green]
```

Also note that the return from the **`insert`** function was not used because it was not set to a variable or passed along to another function. If the return had been used to set the value of the `colors` variable with the line:

```
colors: insert colors 'blue
```

the effect on the block would have been the same, but the position of the `colors` variable would have changed as a result of setting the return value. The position returned from **`insert`** is immediately following the insertion point.

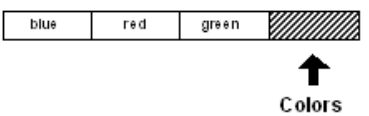
An insertion can be made anywhere in the series. The position of the insert can be specified, and it can include the tail. Inserting at the tail has the effect of appending to the series.

```
colors: tail colors

insert colors 'gold

probe colors
[blue red green gold]
```

Before the insertion:



After the insertion:

The word `gold` has been inserted at the tail of the series.

Another way to insert at the tail of a series is with the **`append`** function. The **`append`** function

works in the same way as **insert**, but always inserts at the tail. The previous example would become:

```
append colors 'gold
```

The result is the same as the previous example.

The **insert** and **append** function also accept a block of arguments to insert. As an example:

```
colors: [red green]

insert colors [blue yellow orange]

probe colors
[blue yellow orange red green]
```

If you want to insert the new values between the red and green words:

```
colors: [red green]

insert next colors [blue yellow orange]

probe colors
[red blue yellow orange green]
```

The **insert** and **append** functions have other capabilities that are covered in more detail in a later section.

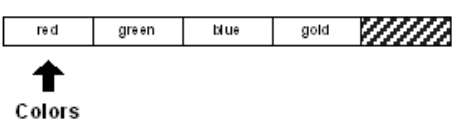
1.6 Removing Values

You can remove one or more values from any part of a series by using the **remove** function.

For instance, starting with the block:

```
colors: [red green blue gold]
```

As shown here:



You can remove the first value from the block with the line:

```
remove colors
```

The block becomes:

It can be printed with:

```
probe colors
[green blue gold]
```

The **remove** function removes values relative to the position of the `colors` variable. You can remove values from anywhere in the series by setting the position.

```
remove next colors
```

The block is now:

Multiple values can be removed by supplying the **/part** refinement.

```
remove/part colors 2
```

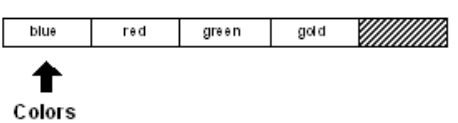
This removes the remaining values, leaving an empty block:

Similar to **insert/part**, the argument to **remove/part** can also be a position within the block.

Removing all of the remaining values is a common operation. The **clear** function is provided to make this more direct. **Clear** removes all values from the current position to the tail. For example:

```
Colors: [blue red green gold]
```

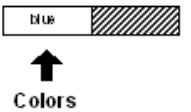
As shown here:



Everything after blue can be removed with:

```
clear next colors
```

The block becomes:



You can easily clear the entire block with:

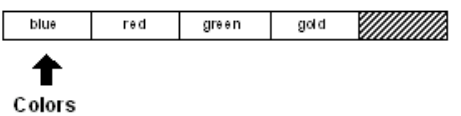
```
clear colors
```

1.7 Changing Values

One additional set of functions is provided for changing values in a series. The **change** function replaces one or more values with new values. Although this can be accomplished by removing and inserting values, it is more efficient to use **change**.

Defining the block:

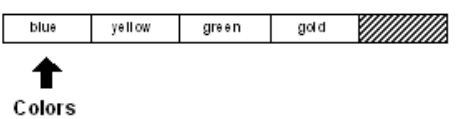
```
colors: [blue red green gold]
```



Its second value could be changed with the line:

```
change next colors 'yellow
```

And it would become:



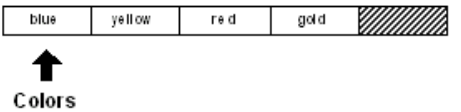
The block would now become:

```
probe colors
[blue yellow green gold]
```

The **poke** function allows you to specify that the change occur at a particular position relative to the `colors` variable. The **poke** function is similar to the **pick** function described earlier.

```
poke colors 3 'red
```

The block is now:



As proven by:

```
probe colors
[blue yellow red gold]
```

The **change** function has additional refinements that are described later in this chapter.

2. Series Functions

Here is a summary of the functions that operate on series. Most of these were described in detail in the previous section. Others will be covered in more detail in this section.

2.1 Creation Functions

Function	Description
make	Makes a new series of the given type.
copy	Copies a series.

2.2 Navigation Functions

Function	Description
next	Returns the next position in a series.
back	Returns the previous position in a series.
head	Returns the head position of a series.
tail	Returns the tail position of a series.
skip	Returns the position plus or minus an integer.
at	Returns the position plus or minus an integer, but uses the same indexing as **pick**.

2.3 Information Functions

Function	Description
head?	Returns `true` if at the head of the series.
tail?	Returns `true` if at the tail of the series.
index?	Returns the offset from the head of the series.
length?	Returns the length of a series from the current position.
offset?	Returns the distance between two series positions.

empty?	Returns `true` if the series is empty from this position.

2.4 Extraction Functions

Function	Description
pick	Extracts a single value from a position in a series.
copy/part	Extracts a sub-series from a series.
first	Extracts the first value from a series.
second	Extracts the second value from a series.
third	Extracts the third value from a series.
fourth	Extracts the fourth value from a series.
fifth	Extracts the fifth value from a series.
last	Extracts the last value from a series.

2.5 Modification Functions

Function	Description
insert	Inserts values into a series.
append	Appends values to the tail of a series.
remove	Removes values from a series.
clear	Clears values to the tail of a series.
change	Changes values in a series.
poke	Changes values at a position in a series.

2.6 Search Functions

Function	Description
find	Finds a value in a series.
select	Finds an associated value in a series.
replace	Searches and replaces values in a series.
parse	Parses values in a series.

2.7 Ordering Functions

Function	Description
sort	Sorts the values in a series into an order.
reverse	Reverse the order of values in a series

2.8 Data Set (Group) Functions

Function	Description
unique	Returns a unique set of values, removing duplicates.
intersect	Returns only the values found in both series.
union	Returns the combined values from two series.

exclude	Returns one series less another.
difference	Returns the values not found in either series.

3. Series Datatypes

All series datatypes can be divided into two broad classes. Each includes a datatype value and a type test function.

3.1 Block Types

Block Type	Description
Block!	Blocks of values
Paren!	Blocks of values enclosed in parentheses
Path!	Paths of values
List!	Linked lists
Hash!	Associative arrays

3.2 String Types

String Type	Description
String!	Character strings
Binary!	Byte strings
Tag!	HTML and XML tags
File!	File names
URL!	Internet uniform resource locators
Email!	Email names
Image!	Image data
Issue!	Sequence codes

3.3 Pseudo-types

Series datatypes are grouped into a few pseudo-types that make function argument and type testing easier:

Pseudo-type	Description
Series!	A series datatype
Any-block!	Any of the block datatypes
Any-string!	Any of the string datatypes

3.4 Type Test Functions

Block type tests:

```
Block? Paren? Path? List? Hash?
```

String type tests:

```
String? Binary? Tag? File? URL?

Email? Image? Issue?
```

Other series type tests:

```
Series? Any-block? Any-string?
```

4. Series Information

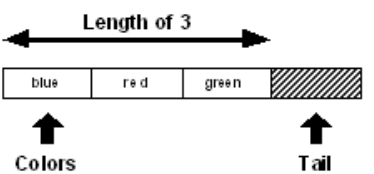
4.1 Length?

The length of a series is the number of items (values for a block or characters for a string) from the `current` position to the **`tail`**. If the current position is the head of the series, then the length is the number of items in the entire series.

The **`length?`** function returns the number of items to the tail.

```
colors: [blue red green]
print length? colors
3
```

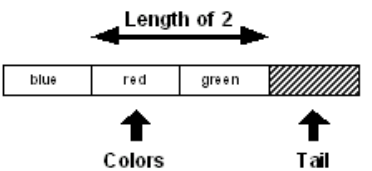
All three values are part of the length:



If the position of the `color` variable is advanced to the next value:

```
color: next color
print length? color
2
```

the length becomes two:



Other examples of **`length?:`**

```
print length? "Ukiah"
5
print length? []
0
print length? ""
0
data: [1 2 3 4 5 6 7 8]
print length? data
8
data: next data
print length? data
7
data: skip data 5
print length? data
2
```

4.2 Head?

The head of a series is the position of its first value. If a series is at its head, the **head?** function returns true:

```
data: [1 2 3 4 5]
print head? data
true
data: next data
print head? data
false
```

4.3 Tail?

The tail of a series is the position `immediately` following the last value. If a series variable is at the tail, the **tail?** function returns true:

```
data: [1 2 3 4 5]
print tail? data
false
data: tail data
print tail? data
true
```

The **empty?** function is equivalent to the **tail?** function.

```
print empty? data
true
```

If **empty?** returns `true`, it means there are no values between the current position and the **tail;** however, there still may be values in the series. Values can still be present before the current position. If you need to determine if the series is empty from head to tail, use:

```
print empty? head data
false
```

4.4 Index?

The `index` is the position in a series relative to the head of the series. To determine the index position for a series variable, use the **`index?`** function:

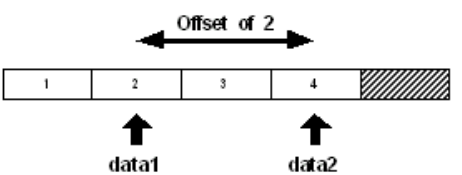
```
data: [1 2 3 4 5]
print index? data
1
data: next data
print index? data
2
data: tail data
print index? data
6
```

4.5 Offset?

The distance between two positions in a series can be determined with the **`offset?`** function.

```
data: [1 2 3 4]
data1: next data
data2: back tail data
print offset? data1 data2
4
```

In this example, the offset is the difference between position 2 and position 4:



5. Making and Copying Series

New series are created with the **`make`** and **`copy`** functions.

Use the **`make`** function to create a new series from a series datatype and an initial size. The size is an estimate of the size needed for the series. If the initial size is too small, the series will automatically expand, but at a slight performance cost.

```
block: make block! 50

string: make string! 10000

list: make list! 128

file: make file! 64
```

The **`copy`** function creates a new series by copying an existing series:

```
string: copy "Message in a bottle"

new-string: copy string

block: copy [1 2 3 4 5]

new-block: copy block
```

Copying is also important for use with functions that modify the contents of a series. For instance, if you want to change the case of a string without modifying the original, use the **copy:**

```
string: uppercase copy "Message in a bottle"
```

5.1 Partial Copies

The **copy** function **/part** refinement takes a single argument, which is either an integer specifying the number of items to copy or a position within the series indicating the last position to copy.

```
str: "Message in a bottle"
print str
Message in a bottle
print copy/part str find str " "
Message
new-str: copy/part (find str "in") (find str "bottle")
print new-str
in a
blk: [ages [10 12 32] sizes [100 20 30]]
new-blk: copy/part blk 2
probe new-blk
[ages [10 12 32]]
```

5.2 Deep Copies

Many blocks contain other blocks and strings. When such a block is copied, its sub-series are not copied. The sub-series are referred to directly and are the same series data as the original block. If you modify any of these sub-series, you modify them in the original block as well.

The **copy/deep** refinement forces a copy of all series values within a block:

```
blk-one: ["abc" [1 2 3]]
probe blk-one
["abc" [1 2 3]]
```

The next example assigns a normal copy of `blk-one` to `blk-two:`

```
blk-two: copy blk-one
probe blk-one
["abc" [1 2 3]]
probe blk-two
["abc" [1 2 3]]
```

If either the string or block contained in `blk-two` is modified, the series values in `blk-one` are also modified.

```
append blk-two/1 "DEF"
append blk-two/2 [4 5 6]
probe blk-one
["abcDEF" [1 2 3 4 5 6]]
probe blk-two
["abcDEF" [1 2 3 4 5 6]]
```

Using **`copy/deep`** makes a copy of all series values found in the block:

```
blk-two: copy/deep blk-one
append blk-two/1 "ghi"
append blk-two/2 [7 8 9]
probe blk-one
["abcDEF" [1 2 3 4 5 6]]
probe blk-two
["abcDEFghi" [1 2 3 4 5 6 7 8 9]]
```

5.3 Initial Copies

When initializing a string or block series, use **`copy`** on the value to make is a unique series:

```
str: copy ""
blk: copy []
```

Using **`copy`** assures that a new series is created for the word every time the word is initialized. Here is an example of why this is important.

```
print-it: func [/local str] [
    str: ""
    insert str "ha"
    print str
]

print-it
ha
print-it
haha
print-it
hahaha
```

In this example, because **`copy`** wasn't used, the empty string series is modified with every call of `print-it`. The string series `ha` is inserted into `str` each time `print-it` is called.

Examining the source of the function as it now exists exposes the root of the problem:

```
source print-it
print-it: func [/local str] [
    str: "hahaha"
    insert str "ha"
```

```
    print str
]
```

Although `str` is a local variable, its string value is global. To avoid this problem, the function should **`copy`** the empty string or use **`make`** on the string.

```
print-it: func [/local str] [
    str: copy ""
    insert str "ha"
    print str
]

print-it
ha
print-it
ha
print-it
ha
```

6. Series Iteration

You can use a loop to traverse a series. There are a few loop functions that can help automate the iteration process.

6.1 Foreach Loop

The **`foreach`** loop moves through a series setting a word or multiple words in to the values in the series.

The **`foreach`** loop takes three arguments: a word or a block of words that holds the values for each iteration, a series, and a block to evaluate for each iteration.

```
colors: [red green blue yellow orange gold]
foreach color colors [print color]
red
green
blue
yellow
orange
gold
foreach [c1 c2] colors [print [c1 c2]]
red green
blue yellow
orange gold
foreach [c1 c2 c3] colors [print [c1 c2 c3]]
red green blue
yellow orange gold
```

This is very useful with blocks that contain related values:

```
people: [
    "Bob" bob@example.com 12
    "Tom" tom@example.net 40
    "Sam" sam@example.org 22
]
```

```
foreach [name email age] people [
    print [name email age]
]
Bob bob@example.com 12
Tom tom@example.net 40
Sam sam@example.org 22
```

Note that the **foreach** loop does not advance the current index through the series, so there is no need to reset its series variable.

6.2 While Loop

The most flexible approach is to use a **while** loop, which allows you to do just about anything to the series without problems.

```
colors: [red green blue yellow orange]

while [not tail? colors] [
    print first colors
    colors: next colors
]
red
green
blue
yellow
orange
```

The method shown below allows you to **insert** values without hitting a value twice:

```
colors: head colors

while [not tail? colors] [
    if colors/1 = 'yellow [
        colors: insert colors 'blue
    ]
    colors: next colors
]
```

This example illustrates that the **insert** returns the position immediately following the insertion.

To **remove** a value without accidentally skipping a value, use the following code:

```
colors: head colors

while [not tail? colors] [
    either colors/1 = 'blue [
        remove colors
    ][
        colors: next colors
    ]
]
```

Notice that if a removal is done, the **next** function is not performed.

6.3 Forall Loop

The **forall** loop is similar to the **while** loop, but eliminates some of the effort required. The **forall** loop starts from the current index and advances through a series to its tail evaluating a block for every value.

The **forall** loop takes two arguments: a series variable and a block to evaluate for each iteration.

```
colors: [red green blue yellow orange]

forall colors [print first colors]
red
green
blue
yellow
orange
```

The **forall** advances the variable position through the series, so when it returns the variable is left at its tail:

```
print tail? colors
true
```

Therefore, the variable must be reset before it is used again:

```
colors: head colors
```

Also, if the block modifies the series, be careful to avoid missing or repeating a value. The **forall** loop works in some cases; however, if you are uncertain, use the **while** loop instead.

```
forall colors [
    if colors/1 = 'blue [remove colors]
    print first colors
]
red
green
yellow
orange
```

6.4 Forskip Loop

Similar to **forall**, the **forskip** loop advances through a series starting at the current position, but skips the specified number of values each time.

The **forskip** loop takes three arguments: a series variable, the skip between each iteration, and a block to evaluate for each iteration.

```
colors: [red green blue yellow orange]

forskip colors 2 [print first colors]
```

```
red
blue
orange
```

The forskip loop leaves the series at its tail, requiring you to reset it.

```
print tail? colors
true
colors: head colors
```

6.5 The Break Function

Any of the loops can be stopped at any time by evaluating the **break** function from within the evaluation block. See the Expressions Chapter for more information about the **break** function.

7. Searching Series

The **find** function searches through block or string series for a value or pattern. This function has many refinements that permit a wide range of variations in search parameters.

7.1 Simple Find

The simplest and most common use of **find** is to search a block or string for a value. In this case, **find** requires only two arguments: the series to search and the value to find.

An example of using **find** on a block is:

```
colors: [red green blue yellow orange]
where: find colors 'blue
probe where
[blue yellow orange]
print first where
blue
```

The **find** function can also search for values by datatype. This can be quite useful.

```
items: [10:30 20-Feb-2000 Cindy "United"]
where: find items date!
print first where
20-Feb-2000
where: find items string!
print first where
United
```

An example of using **find** on a string is:

```
colors: "red green blue yellow orange"
where: find colors "blue"
print where
blue yellow orange
```

When a search fails, **none** is returned.

```
colors: [red green blue yellow orange]
probe find colors 'indigo
none
```

7.2 Refinement Summary

Find has many refinements that support a wide variety of search parameters:

Refinement	Description
/part	Limits a search on a series to a given length or ending position.
/only	Treats a series value as a single value.
/case	Uses case-sensitive string comparison.
/any	Allows the use of pattern wildcards that allow matches to be made with any character. An asterisk (*) in the pattern matches any string, and a question mark (?) in the pattern matches any character.
/with	Allows pattern wildcards with different characters other than asterisk (*) and (?). This allows a pattern to contain asterisks and question marks.
/match	Matches a pattern beginning at the current series position, rather than finding the first occurrence of a value or string. Returns the tail position if the match is found.
/tail	Return the tail position of a match on a successful search, rather than returning the point at which the match was found.
/last	Searches backwards for the match, starting at the tail of the series.
/reverse	Searches backwards for the match, starting at the current position.

7.3 Partial Searches

The **/part** refinement allows a search to be confined to a specific portion of a series. For instance, you may want to restrict a search to a given line or section of text.

Similar to **insert/part** and **remove/part**, **find/part** takes either a count or an ending position. The following example uses a count and restricts the search to the first three items:

```
colors: [red green blue yellow blue orange gold]
probe find/part colors 'blue
[blue yellow blue orange gold]
```

The next search is restricted to the first 15 characters:

```
text: "Keep things as simple as you can."
print find/part text "as" 15
as simple as you can.
```

The next example uses an ending position. The search is restricted to a single line of text:

```
text: {
    This is line one.
```

```
    This is line two.
}

start: find text "this"
end: find start newline
item: find/part start "line" end
print item
line one.
```

7.4 Tail Positions

The **find** function returns the position in the series where an item was found. The **/tail** refinement returns the position immediately following the item that was found. Here's an example:

```
filename: %script.txt

print find filename "."
.txt
print find/tail filename "."
txt
clear change find/tail filename "." "r"
print filename
script.r
```

In this example, **clear** is necessary to remove `xt`, which follows `t`.

7.5 Backward Searches

The last example in the previous section would fail if the filename had more than one period. For instance:

```
filename: %new.script.txt
print find filename "."
.script.txt
```

In this example we want the last occurrence of the period in the string, which can be found using the **/last** refinement. The **/last** refinement searches backward through a series.

```
print find/last filename "."
.txt
```

The **/last** refinement can be combined with**/tail** to produce:

```
print find/last filename "."
txt
```

If you want to continue to search backward through the string, you need the **/reverse** refinement. This refinement performs a search from the current position backward toward the head, rather than forward toward the tail.

```
where: find/last filename "."
print where
.txt
print find/reverse where "."
.script.txt
```

Notice that **/reverse** continues the search just before the position of the last match. This prevents it from finding the same period again.

7.6 Repeated Searches

You can easily repeat the **find** function to search for multiple occurrences of a value or string. Here is an example that would print all the strings found in a block:

```
blk: load %script.r
while [blk: find blk string!] [
    print first blk
    blk: next blk
]
```

The next example counts the number of new lines in a script. It uses the **/tail** refinement to prevent an infinite loop and returns the position immediately following the match.

```
text: read %script.r
count: 0
while [text: find/tail text newline] [count: count + 1]
```

To perform a repeated search in reverse, use the **/reverse** refinement. The following example prints all of the index positions in reverse order for the text of a script.

```
while [text: find/reverse tail text newline] [
    print index? text
]
```

7.7 Matching

The **/match** refinement modifies the behavior of **find** to perform pattern matching on the current position of a series. This refinement allows parsing operations to be performed by matching the next part of a series with expected patterns. See the chapter on Parsing for another way to match series.

A simple example of matching is as follows:

```
blk: [1342 "Franklin Pike Circle"]
probe find/match blk integer!
["Franklin Pike Circle"]
probe find/match blk 1432
["Franklin Pike Circle"]
probe find/match blk "test"
none
str: "Keep things simple."
probe find/match str "keep"
```

```
" things simple."
print find/match str "things"
none
```

Notice in the example that a search is not performed. The beginning of the series either matches or it does not. If it does match, the series is advanced the position immediately following the match point, allowing you to match the next sequence.

Here is a simple parser written with **find/match:**

```
grammar: [
    ["keep" "make" "trust"]
    ["things" "life" "ideas"]
    ["simple" "smart" "happy"]
]

parse-it: func [str /local new] [
    foreach words grammar [
        foreach word words [
            if new: find/match str word [break]
        ]
       if none? new [return false]
       str: next new  ;skip space
   ]
   true
]

print parse-it "Keep things simple"
true
print parse-it "Make things smart"
true
print parse-it "Trust life well"
false
```

Matching can be made case-sensitive with the **/case** refinement.

The capability of **/match** can be greatly extended with the addition of the **/any** refinement as discussed below.

7.8 Wildcard Searches

The **/any** refinement enables wildcard pattern matching. The question mark (?) and asterisk (*) characters act as wildcards for matching any single character or any number of characters respectively. The **/any** refinement can be used in conjunction with **find** with or without the **/match** refinement.

Examples:

```
str: "abcdefg"
print find/any str "c*f"
cdefg
print find/any str "??d"
bcdefg
email-list: [
    mack@rebol.dom
    judy@somesite.dom
    jack@rebol.dom
    biff@rebol.dom
```

```
    jenn@somesite.dom
]
foreach email email-list [
    if find/any email *@rebol.dom [print email]
]
mack@rebol.dom jack@rebol.dombiff@rebol.dom
```

The next example uses the **/match** refinement to attempt to match the pattern to the next part of the series:

```
file-list: [
    %rebol.exe
    %notes.html
    %setup.html
    %feedback.r
    %nntp.r
    %rebdoc.r
    %rebol.r
    %user.r
]

foreach file file-list [
    if find/match/any file %reb*.r [print file]
]
rebdoc.rrebol.r
```

If either of the wildcard characters are part of what is to be matched, substitute wildcard characters can be provided using the **/with** refinement.

7.9 Select

A useful variation of the **find** function is the **select** function, which returns the value following the one found. The **select** function is often used to lookup a value in tagged blocks of data. The **select** function takes the same arguments as **find:** the series to search and the value find. However, unlike **find**, which returns a series position, the **select** function returns the value that follows the match.

```
colors: [red green blue yellow orange]
print select colors 'green
blue
```

Given a simple database, the **select** function can be used to access its values:

```
email-book: [
    "George" harrison@guru.org
    "Paul" lefty@bass.edu
    "Ringo" richard@starkey.dom
    "Robert" service@yukon.dom
]
```

The following code locates a specific email address:

```
print select email-book "Paul"
```

```
lefty@bass.edu
```

Use the **select** function to find a block of expressions to evaluate. For example, given the following data:

```
cases: [
    10 [print "ten"]
    20 [print "twenty"]
    30 [print "thirty"]
]
```

a block can be evaluated based on a selector:

```
do select cases 10
ten
do select cases 30
thirty
```

7.10 Search and Replace

To replace values throughout a series, you can use the **replace** function. This function searches for a specific value in a series, then replaces it with a new value.

The **replace** function takes three arguments: the series to search, value to replace, and the new value.

```
str: "hello world hello"
probe replace str "hello" "aloha"
"aloha world hello"
data: [1 2 8 4 5]
probe replace data 8 3
[1 2 3 4 5]
probe replace data 4 `four
[1 2 3 four 5]
probe replace data integer! 0
[0 2 3 four 5]
```

Use the **/all** refinement to replace all occurrences of the value from the current position to the tail.

```
probe replace/all data integer! 0
[0 0 0 four 0]
code: [print "hello" print "world"]
replace/all code 'print 'probe
probe code
[probe "hello" probe "world"]
do code
helloworld
str: "hello world hello"
probe replace/all str "hello" "aloha"
"aloha world aloha"
```

8. Sorting Series

The **sort** function offers a simple, quick method of sorting series. It is most useful for blocks of data, but can also be used on strings of characters.

8.1 Simple Sorting

The simplest examples of **sort** are:

```
names: [Eve Luke Zaphod Adam Matt Betty]
probe sort names
[Adam Betty Eve Luke Matt Zaphod]
print sort [321.3 78 321 42 321.8 12 98]
12 42 78 98 321 321.3 321.8
print sort "plosabelm"
abellmops
```

Notice that **sort** is destructive to its argument series. It reorders the original data. To prevent this, use **copy**, as in the following example:

```
probe sort copy names
```

By default, sorting is case insensitive:

```
print sort ["Fred" "fred" "FRED"]
Fred fred FRED
print sort "G4C28f9I15Ed3bA076h"
0123456789AbCdEfGhI
```

Providing the **/case** refinement makes sorting case sensitive:

```
print sort/case "gCcAHfiEGeBIdbFaDh"
ABCDEFGHIabcdefghi
print sort/case ["Fred" "fred" "FRED"]
FRED Fred fred
print sort/case "g4Dc2BI8fCF9i15eAd3bGaE07H6h"
0123456789ABCDEFGHIabcdefghi
```

Many other datatypes can be sorted:

```
print sort [1.3.3.4 1.2.3.5 2.2.3.4 1.2.3.4]
1.2.3.4 1.2.3.5 1.3.3.4 2.2.3.4
print sort [$4.23 $23.45 $62.03 $23.23 $4.22]
$4.22 $4.23 $23.23 $23.45 $62.03
print sort [11:11:43 4:12:53 4:14:53 11:11:42]
4:12:53 4:14:53 11:11:42 11:11:43
print sort [11-11-1999 10-11-9999 11-4-1999 11-11-1998]
11-Nov-1998 11-Apr-1999 11-Nov-1999 10-Nov-9999
print sort [john@doe.dom jane@doe.dom jack@jill.dom]
jack@jill.dom jane@doe.dom john@doe.dom
print sort [%user.r %rebol.r %history.r %notes.html]
```

```
history.r notes.html rebol.r user.r
```

8.2 Group Sorting

Often it is necessary to sort a data set that has more than one value per record. The **/skip** refinement supports this for sorting records that have a fixed length. The refinement takes one additional argument: an integer specifying length of each record.

Here is an example that sorts a block that contains first name, last name, ages, and emails. The block is sorted by its first column, first-name.

```
names: [
    "Evie" "Jordan" 43 eve@jordan.dom
    "Matt" "Harrison" 87 matt@harrison.dom
    "Luke" "Skywader" 32 luke@skywader.dom
    "Beth" "Landwalker" 104 beth@landwalker.dom
    "Adam" "Beachcomber" 29 adam@bc.dom
]
sort/skip names 4
foreach [first-name last-name age email] names [
    print [first-name last-name age email]
]
Adam Beachcomber 29 adam@bc.dom
Beth Landwalker 104 beth@landwalker.dom
Evie Jordan 43 eve@jordan.dom
Luke Skywader 32 luke@skywader.dom
Matt Harrison 87 matt@harrison.dom
```

8.3 Comparison Functions

The **/compare** refinement allows you to perform custom comparisons on the data being sorted. This refinement takes an additional argument, which is the comparison function to use for ordering the data.

A comparison function is written as a regular function that takes two arguments. These arguments are the values to be compared. A comparison function returns `true` if the first value should be placed before the second value and `false` if the first value should be placed after the second value.

A normal comparison places data in ascending order:

```
ascend: func [a b] [a < b]
```

If the first value is less than the second, then `true` is returned from the function and the first value is placed before the second value.

```
data: [100 101 -20 37 42 -4]
probe sort/compare data :ascend
[-20 -4 37 42 100 101]
```

Similarly:

```
descend: func [a b] [a > b]
```

If the first value is greater than the second value, then `true` is returned and the data is sorted with greater values first. The sort will descend from greater values.

```
probe sort/compare data :descend
[101 100 42 37 -4 -20]
```

Notice that in both cases the comparison function is passed by providing its name preceded with a colon. The name preceded with a colon causes the function to be passed to **sort** without first being evaluated. The comparison function could also be provided directly with:

```
probe sort/compare data func [a b] [a > b]
[101 100 42 37 -4 -20]
```

9. Series as Data Sets

There are a few functions that operate on series as data sets. These functions allow you to perform operations such as finding the union or intersection between two series.

9.1 Unique

The **unique** function returns a unique set that contains no duplicate values.

Examples:

```
data: [Bill Betty Bob Benny Bart Bob Bill Bob]
probe unique data
[Bill Betty Bob Benny Bart]
print unique "abracadabra"
abrcd
```

9.2 Intersect

The **intersect** function takes two series and returns a series that contains the values that are present in both series.

Examples:

```
probe intersect [Bill Bob Bart] [Bob Ted Fred]
[Bob]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe intersect lunch dinner
[ham carrot]
print intersect [1 3 2 4] [3 5 4 6]
3 4
string1: "CBAD"     ; A B C D scrambled
string2: "EDCF"     ; C D E F scrambled
print sort intersect string1 string2
```

```
CD
```

The intersection can be found between bitsets:

```
all-chars: "ABCDEFGHI"
charset1: charset "ABCDEF"
charset2: charset "DEFGHI"
charset3: intersect charset1 charset2

print find charset3 "E"
true
print find charset3 "B"
false
```

The **/case** refinement allows case-sensitive intersection:

```
probe intersect/case [Bill bill Bob bob] [Bart bill Bob]
[bill Bob]
```

9.3 Union

The **union** function takes two series and returns a series that contains all the values from both series, but no duplicates.

Examples:

```
probe union [Bill Bob Bart] [Bob Ted Fred]
[Bill Bob Bart Ted Fred]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe union lunch dinner
[ham cheese bread carrot salad rice]
print union [1 3 2 4] [3 5 4 6]
1 3 2 4 5 6
string1: "CBDA"     ; A B C D scrambled
string2: "EDCF"     ; C D E F scrambled
print sort union string1 string2
ABCDEF
```

The **union** function can also be used on bitsets:

```
charset1: charset "ABCDEF"
charset2: charset "DEFGHI"
charset3: union charset1 charset2

print find charset3 "C"
true
print find charset3 "G"
true
```

The **/case** refinement allows case-sensitive unions:

```
probe union/case [Bill bill Bob bob] [bill Bob]
[Bill bill Bob bob]
```

9.4 Exclude

The **exclude** function takes two series and returns a series that contains all the values of the first series, less the values of the second.

```
probe exclude [1 2 3 4] [1 2 3 5]
[4]
probe exclude [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe exclude lunch dinner
[cheese bread]
string1: "CBAD"     ; A B C D scrambled
string2: "EDCF"     ; C D E F scrambled
print sort difference string1 string2
AB
```

The **/case** refinement allows case-sensitive exclusion:

```
probe exclude/case [Bill bill Bob bob] [Bart bart bill Bob]
[Bill bob]
```

9.5 Difference

The **difference** function takes two series and returns a series that contains all of the values not in common with both series.

Examples:

```
probe difference [1 2 3 4] [1 2 3 5]
[4 5]
probe difference [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart Ted Fred]
lunch: [ham cheese bread carrot]
dinner: [ham salad carrot rice]
probe difference lunch dinner
[cheese bread salad rice]
string1: "CBAD"     ; A B C D scrambled
string2: "EDCF"     ; C D E F scrambled
print sort difference string1 string2
ABEF
```

The **/case** refinement allows case-sensitive differences.

```
probe difference/case [Bill bill Bob bob] [Bart bart bill Bob]
[Bill bob Bart bart]
```

9.6 Exclude

A variation of the **difference** function is the **exclude** function. It returns the values that are in the first series but not found in the second series. For example:

```
probe exclude [1 2 3 4] [1 2 3 5]
[4]
```

Notice that the above result does not contain 5, as was the case with **difference** in the prior section.

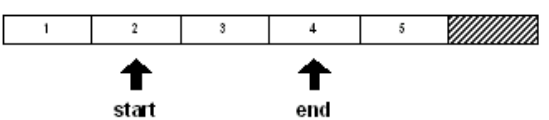
```
probe exclude [Bill Bob Bart] [Bob Ted Fred]
[Bill Bart]
probe exclude "abcde" "ace"
"bd"
```

10. Multiple Series Variables

Multiple variables can refer to the same series. For instance:

```
data: [1 2 3 4 5]
start: find data 3
end: find start 4
print first start
2
print first end
4
```

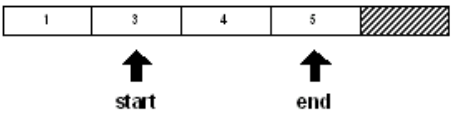
Both the `start` and `end` variables refer to the series. They have different positions, but the series they reference is the same.



If an **insert** or **remove** function is performed on a series, the values in the series will shift and the `start` and `end` variables may no longer refer to the same values. For instance, if a value is removed from the series at the start position:

```
remove start
print first start
3
print first end
5
```

The series has shifted to the left and the variables now refer to different values.



Notice that the index positions of the variables have not changed, but the values in the series have changed. The same situation would occur when using **insert**.

Sometimes this side effect will work to your advantage. Sometimes it will not, and you will need to correct for it in your code.

11. Modification Refinements

The **change**, **insert**, and **remove** functions can take additional refinements to modify their operation.

11.1 Part

The **/part** refinement accepts a count or a position in the series and uses it to limit the effect of the function.

For example, using the following series:

```
str: "abcdef"
blk: [1 2 3 4 5 6]
```

you can change part of `str` and `blk` using **change/part:**

```
change/part str [1 2 3 4] 3
probe str
1234def
change/part blk "abcd" 3
probe blk
["abcd" 4 5 6]
```

You can insert part of a series into the tail of `str` and `blk` using **insert/part**.

```
insert/part tail str "-ghijkl" 4
probe str
1234def-ghi
insert/part tail blk ["--" 7 8 9 10 11 12] 4
probe blk
["abcd" 4 5 6 "--" 7 8 9]
```

To remove part of the `str` and `blk` series, use **remove/part**. Note how **find** is used to obtain the series position:

```
remove/part (find str "d") (find str "-")
probe str
1234-ghi
remove/part (find blk 4) (find blk "--")
probe blk
```

```
["abcd" "--" 7 8 9]
```

11.2 Only

The **/only** refinement changes or inserts a block as a block, rather than its individual values.

Examples:

```
blk: [1 2 3 4 5 6]
```

You can replace the `2` in `blk` with the block `[a b c]` and insert the block `[$1 $2 $3]` at the position of the `5`.

```
change/only (find blk 2) [a b c]
probe blk
[1 [a b c] 3 4 5 6]
insert/only (find blk 5) [$1 $2 $3]
probe blk
[1 [a b c] 3 4 [$1.00 $2.00 $3.00] 5 6]
```

11.3 Dup

The **/dup** refinement changes or inserts a value a specified number of times.

Examples:

```
str: "abcdefghi"
blk: [1 2 3 4 5 6]
```

You can change the first four values in a string or block series to an asterisk `(*)` with:

```
change/dup str "*" 4
probe str
****efghi
change/dup blk "*" 4
probe blk
["*" "*" "*" "*" 5 6]
```

To insert a dash (-) four times before the last value in a string or block:

```
insert/dup (back tail str) #"-" 4
probe str
****efgh----i
insert/dup (back tail blk) #"-" 4
probe blk
["*" "*" "*" "*" 5 #"-" #"-" #"-" #"-" 6]
```

7 - Block Series

1. Blocks of Blocks
2. Paths for Nested Blocks
3. Arrays
    3.1 Creating Arrays
    3.2 Initial Values
4. Composing Blocks

1. Blocks of Blocks

When a block appears as a value within another block, it counts as a single value regardless of how many values it contains. For example:

```
values: [
    "new" [1 2]
    %file1.txt ["one" ["two" %file2.txt]]
]
probe values
["new" [1 2] %file1.txt ["one" ["two" %file2.txt]]]
```

The **length?** of values is four. The second and fourth values are counted as single values:

```
print length? values
4
```

The block values within the values block can be used as a block as well. In the following examples, **second** is used to extract the second value from values.

To print the block, type:

```
probe second values
[1 2]
```

To get the length of the block, type:

```
print length? second values
2
```

To print the data type of the block, type:

```
print type? second values
```

```
block
```

In the same way, series operations can be performed on other types of series values in blocks. In the following examples, **pick** is used to extract `%file1.txt` from `values`.

To look at the value, type:

```
probe pick values 3
%file1.txt
```

To get the length of the value:

```
print length? pick values 3
9
```

to see the data type of the value:

```
print type? pick values 3
file
```

2. Paths for Nested Blocks

The path notation is useful for nested blocks.

The fourth value in `values` is a block containing another block. The following examples use a path to get information about this value.

To look at nested values, type:

```
probe values/4
["one" ["two" %file2.txt]]
probe values/4/2
["two" %file2.txt]
```

To get the lengths of nested values, type:

```
print length? values/4
2
print length? values/4/2
2
```

To see what the data type of a nested value, type:

```
print type? values/4
block
print type? values/4/2
```

```
block
```

The two series values in the fourth value's block can also be accessed.

To look at the values, type:

```
probe values/4/2/1
two
probe values/4/2/2
%file2.txt
```

To get the lengths of the values:

```
print length? values/4/2/1
3
print length? values/4/2/2
9
```

To see what data type the values are:

```
print type? values/4/2/1
string
print type? values/4/2/2
file
```

To modify the values:

```
change (next values/4/2/1) "o"
probe values/4/2/1
too
change/part (next find values/4/2/2 ".") "r" 3
probe values/4/2/2
%file2.r
```

The above examples illustrate REBOL's ability to operate on values nested inside blocks. Note that in the last series of examples, **change** is used to modify a string and file series three layers deep in `values`. Printing out the values block produces:

```
probe values
["new" [1 2] %file1.txt ["one" ["too" %file2.r]]]
```

3. Arrays

Blocks are used for arrays.

An example of a statically defined two dimensional array is:

```
arr: [
```

```
    [1   2   3  ]
    [a   b   c  ]
    [$10 $20 $30]
]
```

You can obtain the values of an array with the series extraction functions:

```
probe first arr
[1 2 3]
probe pick arr 3
[$10.00 $20.00 $30.00]
probe first first arr
1
```

You can also use paths to obtain values from the array:

```
probe arr/1
[1 2 3]
probe arr/3
[$10.00 $20.00 $30.00]
probe arr/3/2
$20.00
```

Paths can also be used to change the values in an array:

```
arr/1/2: 20
```

probe arr/1 == [1 20 3]

```
arr/3/2: arr/3/1 + arr/3/3
```

probe arr/3/2 == $40.00

3.1 Creating Arrays

The **array** function creates arrays dynamically. The function takes an argument that is either an integer or a block of integers and returns a block that is the array. By default, the cells of an array are initialized to none. To initialize array cells to some other value, use the /initial refinement explained in the next section.

When array is supplied with a single integer, a one-dimensional array of that size is returned:

```
arr: array 5
probe arr
[none none none none none]
```

When a block of integers is provided, the array has multiple dimensions. Each integer provides the size of the corresponding dimension.

Here is an example of a two dimensional array that has six cells, two rows of three columns:

```
arr: array [2 3]
probe arr
[[none none none] [none none none]]
```

This can be made into a three dimensional array by adding another integer to the block:

```
arr: array [2 3 2]
foreach lst arr [probe lst]
[[none none] [none none] [none none]]
[[none none] [none none] [none none]]
```

The block of integers that is passed to **array** can be as big as your memory will support.

3.2 Initial Values

To initialize the cells of an array to a value other than none, use the **/initial** refinement. This refinement takes one argument: the initial value. Here are some examples:

```
arr: array/initial 5 0
probe arr
[0 0 0 0 0]
arr: array/initial [2 3] 0
probe arr
[[0 0 0] [0 0 0]]
arr: array/initial 3 "a"
probe arr
["a" "a" "a"]
arr: array/initial [3 2] 'word
probe arr
[[word word] [word word] [word word]]
arr: array/initial [3 2 1] 11:11
probe arr
[[[11:11] [11:11]] [[11:11] [11:11]] [[11:11] [11:11]]]
```

4. Composing Blocks

The **compose** function is handy for creating blocks from dynamic values. It can be used for creating both data and code.

The **compose** function takes a block as an argument and returns a block that has each value in the argument block. Values in parentheses are evaluated before the block is returned. For example:

```
probe compose [1 2 (3 + 4)]
[1 2 7]
probe compose ["The time is" (now/time)]
["The time is" 10:32:45]
```

If the values in parentheses return a block, that block's individual values are used:

```
probe compose [a b ([c d])]
[a b c d]
```

To prevent this, you need to enclose the result in an extra block:

```
probe compose [a b ([[c d]])]
[a b [c d]]
```

An empty block inserts nothing:

```
probe compose [a b ([]) c d]
[a b c d]
```

When compose is given a block that contains sub-blocks, the sub-blocks are not evaluated, even if they contain parentheses:

```
probe compose [a b [c (d e)]]
[a b [c (d e)]]
```

If you would like the sub-blocks to be evaluated, use the **/deep** refinement. The **/deep** refinement causes all parentheses to be evaluated, regardless of where they are:

```
probe compose/deep [a b [c (d e)]]
[a b [c d e]]
```

8 - String Series

1. String Functions
2. Converting Values to Strings
   - 2.1 Join
   - 2.2 Rejoin
   - 2.3 Form
   - 2.4 Reform
   - 2.5 Mold
   - 2.6 Remold
   - 2.7 String Spacing Functions
   - 2.8 Uppercase and Lowercase
   - 2.9 Checksum
   - 2.10 Compression and Decompression
   - 2.11 Number Base Conversion
   - 2.12 Internet Hexadecimal Decoding

1. String Functions

There are a wide variety of functions that operate on or produce strings. Functions are available for modifying strings, searching strings, compressing and decompressing strings, changing the spacing of strings, parsing strings, and converting strings. These functions operate on all string related datatypes, such as **string!**, **binary!**, **tag!**, **file!**, **URL!**, **email!**, and **issue!**.

The string creation, modification and search functions are covered in the Series chapter. They include the items listed in String Functions.

copy	copy all or part of a string
make	allocate storage for a string
insert	insert a character or substring into another string
remove	remove one or more characters from a string
change	change one or more characters in a string
append	insert a character or substring at the tail of a string
find	find or match a character or string in another string
replace	find a string and replace it with another string

In addition, the series traversing functions like **next**, **back**, **head**, and **tail** were covered. They are used to reposition in strings. In addition, the series test functions allow you to determine your position within a string.

This chapter will introduce functions that convert REBOL values into strings. These functions are used often, and they are also used by the **print** and **probe** functions. They include:

form	convert values with spaces and in human readable format
mold	convert values in REBOL readable format
join	convert values with no spaces
reform	reduces values before forming them
remold	reduces values before molding them

rejoin	reduces values before joining them

This chapter will also describes these string functions:

detab	replace tabs with spaces
entab	replace spaces with tabs
trim	remove white space or lines around strings
uppercase	convert string to uppercase
lowercase	convert string to lowercase
checksum	compute a checksum for string
compress	compress string
decompress	decompress string
enbase	convert a string to base value
debase	convert an enbased string to a string
dehex	convert hexadecimal ASCII values to characters

2. Converting Values to Strings

2.1 Join

The **join** function takes two arguments and concatenates them into a single series.

The data type of series returned is based on the value of the first argument. When the first argument is a series value, that series type is returned.

```
str: "abc"
file: %file
url: http://www.rebol.com/

probe join str [1 2 3]
abc123
probe join file ".txt"
%file.txt
probe join url %index.html
http://www.rebol.com/index.html
```

When the first argument is not a series, the **join** converts it to a string first, then performs the append:

```
print join $11 " dollars"
$11.00 dollars
print join 9:11:01 " elapsed"
9:11:01 elapsed
print join now/date " -- today"
30-Jun-2000 -- today
print join 255.255.255.0 " netmask"
255.255.255.0 netmask
print join 412.452 " light-years away"
412.452 light-years away
```

When the second argument to **join** is a block, the values of that block are evaluated and appended to

the series returned.

```
print join "a" ["b" "c" 1 2]
abc12
print join %/ [%dir1/ %sub-dir/ %filename ".txt"]
%/dir1/sub-dir/filename.txt
print join 11:09:11 ["AM" " on " now/date]
11:09:11AM on 30-Jun-2000
print join 312.423 [123 987 234]
312.423123987234
```

2.2 Rejoin

The **rejoin** function is identical to join, except that it takes one argument, a block.

```
print rejoin ["try" 1 2 3]
try123
print rejoin ["h" 'e #"l" (to-char 108) "o"]
hello
```

2.3 Form

The **form** function converts a value to a string:

```
print form $1.50
$1.50
print type? $1.50
money
print type? form $1.50
string
```

The following example uses **form** to find a number by its decimal value:

```
blk: [11.22 44.11 11.33 11.11]
foreach num blk [if find form num ".11" [print num]]
44.11
11.11
```

When **form** is used on a block, all values in the block are converted to string values with spaces between each value:

```
print form [11.22 44.11 11.33]
11.22 44.11 11.33
```

The **form** function does not evaluate the values of a block. This results in words being converted to string values:

```
print form [a block of undefined words]
a block of undefined words
```

```
print form [33.44 num "-- unevaluated string:" str]
33.44 num -- unevaluated string: str
```

2.4 Reform

The **reform** function is like **form**, except that blocks are reduced before being converted.

```
str1: "Today's date is:"
str2: "The time is now:"
print reform [str1 now/date newline str2 now/time]
Today's date is: 30-Jun-2000 The time is now: 14:41:44
```

The **print** function is based on the **reform** function.

2.5 Mold

The **mold** function converts a value to a string that is usable by REBOL. Strings created with **mold** can be converted back to values with the **load** function.

```
blk: [[11 * 4] ($15 - $3.89) "eleven dollars"]
probe blk
[[11 * 4] ($15.00 - $3.89) "eleven dollars"]
molded-blk: mold blk
probe molded-blk
{[[11 * 4] ($15.00 - $3.89) "eleven dollars"]}
print type? blk
block
print type? molded-blk
string
probe first blk
[11 * 4]
<A name=pgfId-539552>probe first molded-blk
#"["
```

The strings returned from **mold** can be loaded by REBOL:

```
new-blk: load molded-blk
probe new-blk
[[11 * 4] ($15.00 - $3.89) "eleven dollars"]
print type? new-blk
block
probe first new-blk
[11 * 4]
```

The **mold** function does not evaluate the values of a block.

```
money: $11.11
sub-blk: [inside another block mold this is unevaluated]
probe mold [$22.22 money "-- unevaluated block:" sub-blk]
{[$22.22 money "-- unevaluated block:" sub-blk]}
probe mold [a block of undefined words]
[a block of undefined words]
```

2.6 Remold

The **remold** function works just like **mold**, except that blocks are reduced before being converted.

```
str1: "Today's date is:"
probe remold [str1 now/date]
{["Today's date is:" 30-Jun-2000]}
```

2.7 String Spacing Functions

2.7.1 Trim

The **trim** function removes extra spaces from a string.

The default operation of **trim** is to remove extra spaces from the head and tail of a string:

```
str: "  line of text with spaces around it "
print trim str
line of text with spaces around it
```

Note that the string is modified in the process:

```
print str
line of text with spaces around it
```

To trim a copy of the string, write:

```
print trim copy str
line of text with spaces around it
```

Trim includes a number of refinements to specify where space is to be removed from a string:

/head	removes space from the head of the string
/tail	removes space from the tail of the string
/auto	removes space from each line, relative to the first line
/lines	removes newlines, replacing them with spaces
/all	- removes all whitespace
/with	removes all specified characters

Use the **/head** and **/tail** refinements to trim from either end of a string:

```
probe trim/head copy str
line of text with spaces around it
probe trim/tail copy str
line of text with spaces around it
```

Use the **/auto** refinement to trim leading spaces from multiple lines leaving indented spaces intact:

```
str: {
    indent text
        indent text
            indent text
        indent text
    indent text
}
print str
indent text
    indent text
        indent text
    indent text
indent text
probe trim/auto copy str
{indent text
    indent text
        indent text
    indent text
indent text
}
```

Use **/lines** to trim the head and tail and also convert newlines into spaces:

```
probe trim/lines copy str
{indent text indent text indent text indent text indent text}
```

Use **/all** to remove all whitespace:

```
probe trim/all copy str
indenttextindenttextindenttextindenttextindenttext
```

The **/with** refinement will remove all characters that you specify. In the following example, spaces, line breaks and the characters `e` and `t` are removed:

```
probe trim/with copy str " ^/et"
indnxindnxindnxindnxindnx
```

2.7.2 Detab and Entab

The **detab** and **entab** will convert tabs to spaces and spaces to tabs.

```
str:
{^(tab)line one
^(tab)^(tab)line two
^(tab)^(tab)^(tab)line three
^(tab)line^(tab)full^(tab)of^(tab)tabs}
print str
line one
        line two
```

```
            line three
    line    full    of  tabs
```

By default, the **detab** function converts tabs to four spaces (the REBOL standard spacing). All tabs in the string will be converted to spaces, regardless of where they are located.

```
probe detab str
{    line one
        line two
            line three
    line    full    of  tabs}
```

Note that the **detab** and **entab** functions affect the string that is provided as an argument. To change a copy of the source string, use the **copy** function.

The **entab** function converts spaces to tabs. Every four spaces will be converted to a single tab. Only spaces at the beginning of a line will be converted to tabs.

```
probe entab str
{^-line one
^-^-line two
^-^-^-line three
^-line^-full^-of^-tabs}
```

You can use the **/size** refinement to specify the size of tabs. For instance, if you want to convert each tab to eight spaces, or convert every eight spaces to a tab, you can use this example:

```
probe detab/size str 8
{        line one
                line two
                        line three
        line    full    of      tabs}
probe entab/size str 8
{^-line one
^-^-line two
^-^-^-line three
^-line^-full^-of^-tabs}
```

2.8 Uppercase and Lowercase

There are two functions for changing character casing: **uppercase** and **lowercase**. The **uppercase** function takes a string argument and converts its characters to uppercase:

```
print uppercase "SamPle TExT, tO test CASES"
SAMPLE TEXT, TO TEST CASES
```

The **lowercase** function converts characters to lowercase:

```
print lowercase "Sample TEXT, tO teST Cases"
```

```
sample text, to test cases
```

To convert only a portion of a string, use the **/part** refinement:

```
print upppercase/part "ukiah" 1
Ukiah
```

2.9 Checksum

The **checksum** returns the checksum of the string value. There are three types of checksum that can be computed:

CRC	24 bit circular redundancy checksum
TCP	standard Internet 16 bit checksum
Secure	a cryptographically secure checksum

By default, the CRC checksum is computed:

```
print checksum "hello"
52719
print checksum (read http://www.rebol.com/)
356358
```

To compute a TCP 16-bit checksum, use the **/tcp** refinement:

```
print checksum/tcp "hello"
10943
```

A secure checksum will return a binary value, not an integer. Use the **/secure** refinement to compute a secure checksum:

```
print checksum/secure "hello"
#{AAF4C61DDCC5E8A2DABEDE0F3B482CD9AEA9434D}
```

2.10 Compression and Decompression

The **compress** function will compress a string and return a binary datatype. In the following example, a small file is compressed by reading its contents, compressing them, then writing it back to disk:

```
Str:
{I wanted the gold, and I sought it,
  I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
  I hurled my youth into a grave.
I wanted the gold, and I got it --
  Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
  And somehow the gold isn't all.}
```

```
print [size? str "bytes"]
306 bytes
bin: compress str

print [size? bin "bytes"]
156 bytes
```

Note that the result of the compression is a binary data type.

The **decompress** function decompresses a previously compressed string.

```
print decompress bin
I wanted the gold, and I sought it,
  I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
  I hurled my youth into a grave.
I wanted the gold, and I got it --
  Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
  And somehow the gold isn't all.
```

Save Your Data

Always keep an uncompressed backup of compressed data. If you lose only one byte from a compressed binary, it can be difficult to recover the data. Do not store file archives in a compressed format unless you have copies that are not compressed.

2.11 Number Base Conversion

To be sent as text, binary strings must be converted to hexadecimal or base64 encoding. This is often done for email and newsgroup content.

The **enbase** function will encode a binary string:

```
line: "No! There's a land!"
print enbase line
Tm8hIFRoZXJlJ3MgYSBsYW5kIQ==
```

Encoded strings can be decoded with the **debase** function. Note that the result is a binary value. To convert it back to a string, use the **to-string** function.

```
b-line: debase e-line
print type? b-line
binary
probe b-line
#{4E6F212054686572652773206120616E6421}
print to-string b-line
No! There's a land!
```

The **/base** refinement may be used with **enbase** and **debase** to specify a base2 (binary), base16 (hexadecimal), or base64 encoding.

Here are some examples using base2:

```
e2-str: enbase/base str 2
print e2-str
01100001
b2-str: debase/base e2-str 2
print type? b2-str
binary
probe b2-str
#{61}
print to-string b2-str
a
```

Here are some examples using base16:

```
e16-line: enbase/base line 16
print e16-line
4E6F212054686572652773206120616E6421
b16-line: debase/base e16-line 16
print type? b16-line
binary
probe b16-line
#{4E6F212054686572652773206120616E6421}
print to-string b16-line
No! There's a land!
```

2.12 Internet Hexadecimal Decoding

The **dehex** function converts Internet URL and CGI style hexadecimal encoded characters to strings. Hexadecimal ASCII representations appear in a URL or CGI string as `%xx`, where `xx` is the hexadecimal value.

```
str: "there%20seem%20to%20be%20no%20spaces"
print dehex str
there seem to be no spaces
print dehex "%68%65%6C%6C%6F"
hello
```

9 - Functions

1. Overview
2. Evaluating Functions
    2.1 Arguments
    2.2 Argument Data Types
    2.3 Refinements
    2.4 Function Values
3. Defining Functions
    3.1 Interface Specifications
    3.2 Literal Arguments
    3.3 Get Arguments
    3.4 Defining Refinements
    3.5 Local Variables
    3.6 Local Variables Containing Series
    3.7 Returning a Value
    3.8 Returning Multiple Values
4. Nested Functions
5. Unnamed Functions
6. Conditional Functions
7. Function Attributes
8. Forward References
9. Scope of Variables
10. Reflective Properties
11. Online Function Help
12. Viewing Source Code

1. Overview

There are several kinds of functions provided by REBOL:

Native	A function that is evaluated directly by the processor. These are the lowest level functions of the language.
Function	A higher level function that is defined by a block and is evaluated by evaluating the functions within the block. Also called user-defined functions.
Mezzanine	A name for higher level functions that are a standard part of the language. These are not native functions.
Operator	A function that is used as an infix operator. Examples are +, -, * and /.
Routine	A function that is used to call external library functions (REBOL/Command feature).

2. Evaluating Functions

The Expressions Chapter covered the general details of evaluation. The way function arguments are evaluated dictates the general order of words and values in the language. This section goes into more detail on how functions are evaluated.

2.1 Arguments

Functions receive arguments and return results. Most functions require one or more arguments; although, some functions, such as **now** (current date and time), do not require any arguments.

The arguments that are supplied to a function are processed by the interpreter and then passed to the function. Arguments are processed in the same way, regardless of the type of function called, be it a native function, operator, user-defined function, or otherwise. For example, the **send** function expects two arguments:

```
friend: luke@rebol.com
message: "message in a bottle"

send friend message
```

The word friend is first evaluated and its value (luke@rebol.com) is provided as the first argument to **send**. Next, the word message is evaluated, and its value becomes the second argument. Think of the values of the friend and message variables as being substituted into the line before **send** is done:

```
send luke@rebol.com "message in a bottle"
```

If you provide too few arguments to a function, an error message is returned. For example, the send function expects two arguments and if you send one, an error is returned

```
send friend
** Script Error: send is missing its message argument.
** Where: send friend
```

If too many arguments are provided, the extra values are ignored.

```
send friend message "urgent"
```

In the previous example, **send** already has two arguments, so the string, which is the third argument, is ignored. Notice that no error message occurs. In this case, there were no functions expecting the third argument. However, in some cases the third argument may belong to another function that was evaluated before **send**.

Arguments to a function are evaluated from left to right. This order is followed even when the arguments themselves are functions. For example, if you write:

```
send friend detab copy message
```

the second argument must be computed by evaluating the **detab** function and the **copy** function. The result of the **copy** will be passed to **detab**, and the result of **detab** will be passed to **send**. In the previous example, the **copy** function is taking a single argument, the message, and returns a copy of it. The copied message is passed to the **detab** function, which removes the tab characters and returns the detabbed message, which is passed to the **send** function. Notice how the results of functions flow from right to left as the expression is evaluated.

The evaluation that is happening here can be shown by using parentheses to clarify what is evaluated first. (However, the parentheses are not required, and actually slow down the evaluation slightly.)

```
send friend (detab (copy message))
```

The cascading effect of results passed to functions is quite useful. Here is an example that uses **insert** twice within the same expression:

```
file: %image
insert tail insert file %graphics/ %.jpg
print file
graphics/image.jpg
```

In the following example, a directory name and a suffix are added to the base file name. Parentheses can be used to clarify the order of evaluation:

```
insert (tail (insert file %graphics/)) %.jpg
```

A Note About Parentheses

Parentheses make good "training wheels" to get started in writing REBOL. However, it won't take long before you can shed this aid and write the expressions directly without the parentheses. Not using parentheses lets the interpreter evaluate expressions quicker.

2.2 Argument Data Types

Functions usually require arguments of a specific data type. For example, the first argument to the **send** function can only be an email address or block of email addresses. Any other type of value will produce an error:

```
send 1234 "numbers"
** Script Error: send expected address argument of type: email block.
** Where: send 1234 "numbers"
```

In the previous example, the error message is telling you that the address argument of the **send** function needs to be either an email address or a block.

A quick way to find out what types of arguments are accepted by a function is to type the following at the console prompt:

```
help send
USAGE:
    SEND address message /only /header header-obj
DESCRIPTION:
    Send a message to an address (or block of addresses)
    SEND is a function value.
ARGUMENTS:
    address -- An address or block of addresses (Type: email block)
    message -- Text of message. First line is subject. (Type: any)
REFINEMENTS:
    /only -- Send only one message to multiple addresses
    /header -- Supply your own custom header
```

```
        header-obj -- The header to use (Type: object)
```

The ARGUMENTS section indicates the data type of each argument. Notice that the second argument can be of any data type. So, it is valid to write:

```
send luke@rebol.com $1000.00
```

2.3 Refinements

A refinement specifies a variation in the normal evaluation of a function. Refinements also allow optional arguments to be provided. Refinements are available for both native and user-defined functions.

Refinements are specified by following the function name with a forward slash (/) and a refinement name. For instance:

```
copy/part  (copy just part of a string)

find/tail  (return the tail of the match)

load/markup  (return XML/HTML tags and strings)
```

Functions can also include multiple refinements:

```
find/case/tail (match case and return tail)

insert/only/dup (insert entire block multiple times)
```

You have seen the **copy** function used to make a copy of a string. By default, **copy** returns a copy of its argument:

```
string: "no time like the present"
print copy string
no time like the present
```

Using the **/part** refinement, **copy** returns part of the string:

```
print copy/part string 7
no time
```

In the previous example, the **/part** refinement specifies that only seven characters of the string are copied.

To review what refinements are allowed on a function such as **copy**, use online help:

```
help copy
USAGE:
    COPY value /part range /deep
```

```
DESCRIPTION:
     Returns a copy of a value.
     COPY is an action value.
ARGUMENTS:
     value -- Usually a series (Type: series port bitset)
REFINEMENTS:
     /part -- Limits to a given length or position.
         range -- (Type: number series port)
     /deep -- Also copies series values within the block.
```

Notice that the **/part** refinement requires an additional argument. Not all refinements require additional arguments. For example, the **/deep** refinement specifies that **copy** make copies of all its sub-blocks. No other arguments are required.

When multiple refinements are used with a function, the order of the extra arguments is determined by the order in which the refinements are specified. For example:

```
str: "test"
insert/dup/part str "this one" 4 5
print str
this this this this test
```

Reversing the order of the **/dup** and **/part** refinement changes the order of the arguments. You can see the difference:

```
str: "test"
insert/part/dup str "this one" 4 5
print str
thisthisthisthisthistest
```

The refinements indicate the order of the arguments.

2.4 Function Values

The previous examples describe how functions return values when they are evaluated. Sometimes, however, you want to obtain the function as a value, not the value it returns. This can be done by preceding the function name with a colon or using the **get** function. For example, to set a word, pr, to the **print** function, you would write:

```
pr: :print
```

You could also write:

```
pr: get `print
```

Now **pr** is equivalent to the **print** function:

```
pr "this is a test"
this is a test
```

3. Defining Functions

You can define functions that work in the same way as native functions. These are called ***user-defined*** functions. User-defined functions are of the **function!** data type.

You can make simple functions that require no arguments with the **does** function. This example defines a new function that prints the current time:

```
print-time: does [print now/time]
print-time
10:30
```

The **does** function returns a value, which is the new function. In the example, the print-time word is set to the function. However, this function value can be set to a word, passed to another function, returned as the result of a function, saved in a block, or immediately evaluated.

Functions that require arguments are made with the **func** function, which accepts two arguments:

```
func spec body
```

The first argument is a block that specifies the interface to the function. It includes a description of the function, its arguments, the types allowed for arguments, descriptions of the arguments, and other items. The second argument is a block of code that is evaluated whenever the function is evaluated.

Here is an example of a new function called sum:

```
sum: func [arg1 arg2] [arg1 + arg2]
```

The newly defined function accepts two arguments, as specified in the first block. The second block is the body of the function, which, when evaluated, adds the two arguments together. The new function is returned as a value from **func** and the sum word is set to it. Here it is in use:

```
print sum 123 321
444
```

The result of arg1 being added to arg2 is returned and printed.

Func is Defined in REBOL

Func is a function that makes other functions. It performs a make on the **function!** data type. **Func** is defined as:

```
func: make function! [args body] [
    make function! args body
]
```

3.1 Interface Specifications

The first block of a function definition is called its ***interface specification***. This block includes a description of the function, its arguments, the data types allowed for arguments, descriptions of the arguments, and other items.

The interface specification is a dialect of REBOL (because it has different evaluation rules than normal code). The specification block has the format:

```
[
    "function description"
    [optional-attributes]

    argument-1 [optional-type]
    "argument description"

    argument-2 [optional-type]
    "argument description"

    ...

    /refinement
    "refinement description"

    refinement-argument-1 [optional-type]
    "refinement argument description"

    ...
]
```

The fields of the specification block are:

Description	A short description of the function. This is a string that can be accessed by other functions such as help to output descriptions of functions.
Attributes	A block that describes special properties of the function, such as its behavior on errors. It may be expanded in the future to include flags for optimizations.
Argument	A variable that is used to access an argument from within the body of the function.
Arg Type	A block that identifies the data types that are accepted by the function. If a data type not identified in this block is passed to the function, an error will occur.
Arg Description	A short description of the argument. Like the function description, this can be accessed by other functions such as help.
Refinement	A refinement word that indicates special behavior is required of the function.
Refinement Description	A short description of the refinement.
Refinement Argument	A variable that is used by the refinement.
Refinement Argument Type	A block that identifies the data types that are accepted by the refinement.
Refinement Argument Description	A short description of the refinement argument.

All of these fields are optional.

As an example, the argument block of the `sum` function (defined in a previous example) is expanded to

restrict the type of arguments accepted. It also includes a description of the function and its expected arguments.

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number!] "first number"
    arg2 [number!] "second number"
][
    arg1 + arg2
]
```

Now, the data type of the arguments is automatically checked, catching errors like:

```
print sum 1 "test"
** Script Error: sum expected arg2 argument of type: number.
** Where: print sum 1 "test"
```

To allow additional argument data types, more than one can be given:

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number! tuple! money!] "first number"
    arg2 [number! tuple! money!] "second number"
][
    arg1 + arg2
]

print sum 1.2.3 3.2.1
4.4.4
print sum $1234 100
$1334.00
```

Now the `sum` function accepts a number, tuple, or monetary value as arguments. If within the function you need to distinguish what data type was passed, you can use the data type test functions:

```
if tuple? arg1 [print arg1]

if money? arg2 [print arg2]
```

Because the `sum` function provided description strings, the **`help`** function now supplies useful information about it:

```
help sum
USAGE:
    SUM arg1 arg2
DESCRIPTION:
     Return the sum of two numbers.
     SUM is a function value.
ARGUMENTS:
     arg1 -- first number (Type: number tuple money)
     arg2 -- second number (Type: number tuple money)
```

3.2 Literal Arguments

As described earlier, the interpreter evaluates the arguments of functions and passes them to the function body. However, there are times when you do not want function arguments evaluated. For instance, if you need to pass a word and access it from the function body, you do not want it evaluated as an argument. The **help** function, which expects a word, is a good example:

```
help print
```

To prevent **print** from being evaluated, the **help** function must specify that its argument should not be evaluated.

To specify that an argument not be evaluated, precede the argument name with a single quote (indicates a literal word). For example:

```
zap: func [`var] [set var 0]

test: 10
zap test
print test
10
```

The var argument is preceded with a single quote, which instructs the interpreter to obtain the argument without evaluating it first. The argument is passed as the word. For example:

```
say: func [`var] [probe var]
say test
test
```

The example prints the word that is passed as an argument.

Another example is a function that increments a variable by one and returns its result (similar to the ++ increment function in C):

```
++: func ['word] [set word 1 + get word]

count: 0
++ count
print count
1
print ++ count
2
```

3.3 Get Arguments

Function arguments can also specify that a word's value be fetched but not evaluated. This is similar to the literal arguments described above, but rather than passing the word, the value of the word is passed without being evaluated.

To specify that an argument be fetched but not evaluated, precede the argument name with a colon. For example, the following function accepts functions as arguments:

```
print-body: func [:fun] [probe second :fun]
```

The sample function prints the body of a function that is passed to it. The argument is preceded by a colon, which indicates that the value of the word should be obtained, but not further evaluated.

```
print-body reform
[form reduce value]
print-body rejoin
[
    if empty? block: reduce block [return block]
    append either series? first block [copy first block] [
        form first block] next block
]
```

3.4 Defining Refinements

Refinements can be used to specify variation in the normal evaluation of a function as well as provide optional arguments. Refinements are added to the function specification block as a word preceded by a forward slash (/).

Within the body of the function, the refinement word is used as a logic value to determine if the refinement was provided when the function was called.

For example, the following code adds a refinement to the `sum` function, which was defined in a previous example:

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number!] "first number"
    arg2 [number!] "second number"
    /average "return the average of the numbers"
][
    either average [arg1 + arg2 / 2][arg1 + arg2]
]
```

The `sum` function specifies the `/average` refinement. In the body of the function, the word is tested with the either function, which returns true when the refinement is specified.

```
print sum/average 123 321
222
```

To specify a refinement that accepts additional arguments, follow the refinement with the arguments definitions:

```
sum: func [
    "Return the sum of two numbers."
    arg1 [number!] "first number"
    arg2 [number!] "second number"
    /times "multiply the result"
    amount [number!] "how many times"
][
    either times [arg1 + arg2 * amount][arg1 + arg2]
```

```
]
```

The amount is only valid when the times refinement is true. Here is an example:

```
print sum/times 123 321 10
4440
```

Do not forget to check the refinement word before using the additional arguments. If a refinement argument is used without the refinement being specified, it will have a none value.

3.5 Local Variables

A local variable is a word whose value is defined within the scope of a function. Changes to a local variable only affect the function in which the variable is defined. If the same word is used outside of the function, it will not be affected by the changes to the local variable of the same name.

Argument variables and refinements are local variables. Their values are defined within the scope of the function. By convention, additional local variables can be specified with the **/local** refinement. The **/local** refinement is followed by a list of words that are used as local variables within the function.

```
average: func [
    block "Block of numbers"
    /local total length
][
    total: 0
    length: length? block
    foreach num block [total: total + num]
    either length > 0 [total / length][0]
]
```

Here the total and length words are local to the function.

Another method of creating local words is to use the **function** function, which is identical to **func**, but accepts a separate block that contains the local words:

```
average: function [
    block "Block of numbers"
][
    total length
][
    total: 0
    length: length? block
    foreach num block [total: total + num]
    either length > 0 [total / length][0]
]
```

In this example, notice that the **/local** refinement is not used with the **function** function. The **function** function creates the refinements for you.

If a local variable is used before its value has been set within the body of its function, it will have a none value.

3.6 Local Variables Containing Series

Local variables that hold series need to be copied if the series is used multiple times. For example, if you want the `stars` string to be the same each time you call the `start-name` function, you should write:

```
star-name: func [name] [
    stars: copy "**"
    insert next stars name
    stars
]
```

Otherwise, if you write:

```
star-name: func [name] [
    stars: "**"
    insert next stars name
    stars
]
```

you will be using the same string each time and each time the function is used the pervious name will appear within the result.

```
print star-name "test"
*test*
print star-name "this"
*thistest*
```

This is Important

The concept described above is important to remember. If you forget it, you will observe odd results in your programs.

3.7 Returning a Value

As you know from the Expressions Chapter, blocks return their last value when they return from evaluation:

```
do [1 + 3  5 + 7]
12
```

This is also true for functions. The last value is returned as the value of the function:

```
sum: func [a b] [
    print a
    print b
    a + b
]
```

```
print sum 123 321
123
321
444
```

In addition, the **return** function can be used to stop the evaluation of a function at any point and return a value:

```
find-value: func [series value] [
    forall series [
        if (first series) = value [
            return series
        ]
    ]
    none
]

probe find-value [1 2 3 4] 3
[3 4]
```

In the example, if the value is found, the function returns the series at the position of the match. Otherwise, the function returns `none`.

To stop a function evaluation without returning a value, use the **exit** function:

```
source: func [
    "Print the source code for a word"
    'word [word!]
][
    prin join word ": "
    if not value? word [print "undefined" exit]
    either any [
        native? get word op? get word action? get word
    ][
        print ["native" mold third get word]
    ][print mold get word]
]
```

3.8 Returning Multiple Values

To return more than one value from a function, use a block. You can do this easily by returning a block that has been reduced.

For example:

```
find-value: func [series value /local count] [
    forall series [
        if (first series) = value [
            reduce [series  index? series]
        ]
    ]
    none
]
```

The function returns a block that holds the series and the index value where the value was found.

```
probe find-value [1 2 3 4] 3
[[3 4] 3]
```

The **reduce** is necessary to create a block of values from the block of words that it is given. Do not return the local variables themselves. That is not a supported mode of operation (currently).

To easily set variables to the return value of the function, use **set:**

```
set [block index] find-value [1 2 3 4] 3
print block
3 4
print index
3
```

4. Nested Functions

Functions can define other functions. The sub-functions can be global, local, or returned as a result, depending on their purpose.

For example, to create a global function from within a function, assign it to a global variable:

```
make-timer: func [code] [
    timer: func [time] code
]
make-timer [wait time]
timer 5
```

To make a local function, assign it to a local variable:

```
do-timer: func [code delay /local timer] [
    timer: func [time] code
    timer delay
    timer delay
]
do-timer [wait time] 5
```

The `timer` function only exists during the period when the `do-timer` function is being evaluated.

To return a function as a result:

```
make-timer: func [code] [
    func [time] code
]
timer: make-timer [wait time]
timer 5
```

Use Correct Local Variables

You should avoid using variables that are local to the top level function as an unevaluated part of the nested function. For example:

```
make-timer: func [code delay] [
    timer: func [time] [wait time + delay]
]
```

In the example, the `delay` word dynamically belongs to the `make-timer` function. This should be avoided, as the `delay` value will change in subsequent calls to `make-timer`.

5. Unnamed Functions

Function names are variables. In REBOL, a variable is a variable, regardless of what it holds. There is nothing special about function variables.

Furthermore, functions do not require names. You can create a function and immediately evaluate it, store it in a block, pass it as an argument to a function, or return it as a result from a function. Such functions are unnamed.

Here is an example that creates a block of unnamed functions:

```
funcs: []
repeat n 10 [
    append funcs func [t] compose [t + (n * 100)]
]
print funcs/1 10
110
print funcs/5 10
510
```

Functions can also be created and passed to other functions. For instance, when you use **`sort`** with your own comparison, you provide a function as an argument:

```
sort/compare data func [a b] [a > b]
```

6. Conditional Functions

Because functions are created dynamically by evaluation, you can determine how you want a function created, based on other information. This is a way to provide conditional code as is found in the macro or preprocessor sub-languages of other programming languages. Within the REBOL language this type of conditional code is done with normal REBOL code.

For instance, you may want to create a debugging version of a function that prints additional information:

```
test-mode: on

timer: either test-mode [
    func [delay] [
        print "delaying..."
        wait delay
```

```
        print "resuming"
    ]
][
    func [delay] [wait delay]
]
```

Here you will create one of two functions, based on the test-mode you are running. This can also be written shorter as:

```
timer: func [delay] either test-mode [[
    print "delaying..."
    wait delay
    print "resuming"
]][[wait delay]]
```

7. Function Attributes

Function attributes provide control over specific function behaviors, such as the method a function uses to handle errors or to exit. The attributes are an optional block of words within the interface specifications.

There are currently two function attributes: **catch** and **throw**.

Error messages typically are displayed when they occur within the function. If the **catch** attribute is specified, errors that are thrown within the function are caught automatically by the function. The errors are not displayed within the function but at the point where the function was used. This is useful if you are providing a function library (mezzanine functions) and don't want the error to be displayed within your function, but where it was called:

```
root: func [[catch] num [number!]] [
    if num < 0 [
        throw make error! "only positive numbers"
    ]
    square-root num
]

root 4
2
root -4
**User Error: only positive numbers
**Where: root -4
```

Notice that in this example, the error occurs where `root` was called even though the actual error was generated in the body of the function. This is because the **catch** attribute was used.

Without the **catch** attribute, the error would occur within the `root` function:

```
root: func [num [number!]] [
    square-root num
]
root -4
** Math Error: Positive number required.
** Where: square-root num
```

The user may not know anything about the internals of the `root` function. So the error message would be confusing. The user only knows about `root`, but the error was in **`square-root`**.

Do not get the **`catch`** attribute mixed up with the **`catch`** function. Although they are similar, the **`catch`** function can be applied to any block that is evaluated.

The **`throw`** attribute allows you to write your own control functions, such as **`for`**, **`foreach`**, **`if`**, **`loop`**, and **`forever`**, by allowing your functions to pass the **`return`** and **`exit`** operations. For example, this loop function:

```
loop-time: func [time block] [
    while [now/time < time] block
]
```

evaluates a block until a specific time has been reached or passed. This loop can then be used within a function:

```
do-job: func [job][
    loop-time 10:30 [
        if error? try [page: read http://www.rebol.com]
            [return none]
    ]
    page
]
```

Now, what happens when the `[return` none] block is evaluated? Because this block is evaluated by the `loop-time` function, the return occurs in that function, not in `do-job`.

This can be prevented with the **`throw`** attribute:

```
loop-time: func [[throw] time block] [
    while [now/time < time] block
]
```

The **`throw`** attribute causes a **`return`** or **`exit`** that has occurred within the block to be thrown up to the previous level, which is the next function causing `do-job` to return.

8. Forward References

Sometimes a script needs to refer to a function before it has been defined. This can be done as long as the variable for the function is not evaluated before it is defined.

```
buy: func [item] [
    append own item
    sell head item    ; appears before it is defined
]

sell: func [item] [
    remove find own item
]
```

9. Scope of Variables

The context of variables is called their `scope`. The broad scope of variables is that of global and local. REBOL uses a form of static scoping, which is called `definitional` scoping. The scope of a variable is determined when its context is defined. In the case of a function, it is determined by when the function is defined.

All of the local variables defined within a function are scoped relative to that function. Nested functions and objects are able to access their parent's words.

```
a-func: func [a] [
    print ["a:" a]
    b-func: func [b] [
        print ["b:" b]
        print ["a:" a]
        print a + b
    ]
    b-func 10
]
a-func 11
a: 11
b: 10
a: 11
21
```

Note here that the `b-func` has access to the `a-func` variable.

Words that are bound outside of a function maintain those bindings even when evaluated within a function. This is the result of static scoping, and it allows you to write your own block evaluation functions (like **`if`**, **`while`**, **`loop`**).

For example, here is a signed **`if`** function that evaluates one of three blocks based on the sign of a conditional value:

```
ifs: func [
    "If positive do block 1, zero do block 2, minus do 3"
    condition block1 block2 block3
][
    if positive? condition [return do block1]
    if negative? condition [return do block3]
    return do block2
]

print ifs 12:00 - now/time ["morning"]["noon"]["night"]
night
```

The blocks passed may contain the same words used within the `ifs` function without interfering with the words defined local to the function. This is because the words passed to the function are not bound to the function.

The next example passes the words `block1`, `block2` and `block3` to `ifs` as pre-defined words. The `ifs` function does not get confused between the words passed as arguments and the words of the same name defined locally:

```
block1: "morning right now"
block2: "just turned noon"
```

```
block3: "evening time"

print ifs (12:00 - now/time) [block1][block2][block3]
evening time
```

10. Reflective Properties

The specification of all functions can be obtained and manipulated during run-time. For example, you can print the specification block for a function with:

```
probe third :if
[
    "If condition is TRUE, evaluates the block."
    condition
    then-block [block!]
    /else "If not true, evaluate this block"
    else-block [block!]
]
```

The body code of functions can be obtained with:

```
probe second :append
[
    head either only [
        insert/only tail series :value
    ][
        insert tail series :value
    ]
]
```

Functions can be dynamically queried during evaluation. This is how the **help** and **source** functions work and how errors messages are formatted.

In addition, this feature is useful for creating your own unique versions of existing functions. For example, a user-defined print function can be created that has exactly the same specification as **print**, but sends its output to a string rather than the display:

```
output: make string! 1000

print-str: func third :print [
    repend output [reform :value newline]
]
```

The name of the argument used for print-str is obtained from the interface specification for print. You can examine that specification with:

```
probe third :print
[
    "Outputs a value followed by a line break."
    value "The value to print"
]
```

11. Online Function Help

Useful information about all functions of the system can be retrieved with the **help** function:

```
help send
USAGE:
    SEND address message /only /header header-obj
DESCRIPTION:
     Send a message to an address (or block of addresses)
     SEND is a function value.
ARGUMENTS:
     address -- An address or block of addresses (Type: email block)
     message -- Text of message. First line is subject. (Type: any)
REFINEMENTS:
     /only -- Send only one message to multiple addresses
     /header -- Supply your own custom header
         header-obj -- The header to use (Type: object)
```

All of this information comes from the definition of the function. Help can be obtained for all types of functions, not just natives or built-in functions. The help function can also be used for user-defined functions. The documentation that is displayed about a function is provided when the function is defined.

You can also search for help on functions that contain various patterns. For instance, at the command prompt, you could type

```
Help "path"
Found these words:
     clean-path      (function)
     lit-path!       (datatype)
     lit-path?       (action)
     path!           (datatype)
     path-thru       (function)
     path?           (action)
     set-path!       (datatype)
     set-path?       (action)
     split-path      (function)
     to-lit-path     (function)
     to-path         (function)
     to-set-path     (function)
```

to display all the words that contain the string `path`.

To view a list of all functions available in REBOL, type **what** at the command prompt.

```
what
* [value1 value2]
** [number exponent]
+ [value1 value2]
- [value1 value2]
/ [value1 value2]
// [value1 value2]
< [value1 value2]
<= [value1 value2]
<> [value1 value2]
= [value1 value2]
== [value1 value2]
```

```
=? [value1 value2]
> [value1 value2]
>= [value1 value2]
? ['word]
?? ['name]
about []
abs [value]
absolute [value]
...
```

12. Viewing Source Code

Another technique for learning about REBOL and for saving time in writing your own function is to look at how many of the REBOL mezzanine functions are defined. You can use the **source** function to do this.

```
source source
source: func [
    "Prints the source code for a word."
    'word [word!]
][
    prin join word ": "
    if not value? word [print "undefined" exit]
    either any [native? get word op? get word action? get word] [
        print ["native" mold third get word]
    ] [print mold get word]
]
```

Here the **source** function is used to print its own source code.

Note that you cannot see the source code for native functions because they exist only as machine code. However, the source function will display the native function interface specification. For example:

```
source add
add: native [
    "Returns the result of adding two values."
    value1 [number! pair! char! money! date! time! tuple!]
    value2 [number! pair! char! money! date! time! tuple!]
]
```

10 - Objects

1. Overview
2. Making Objects
3. Cloning Objects
4. Accessing Objects
5. Object Functions
6. Prototype Objects
7. Referring to Self
8. Encapsulation
9. Reflective Properties

1. Overview

Objects group values into a common context. An object can include scalar values, series, functions, and other objects. Objects are useful in dealing with complex structures as they allow related data and code to be encapsulated and passed as a single value to functions.

2. Making Objects

New objects are created with the **make** function. The **make** function requires two arguments and returns a new object. The format of the **make** function is:

```
new-object: make parent-object new-values
```

The first argument, `parent-object` , is the parent object from which the new object is made. If no parent object is available, as when defining an initial object, use the **object!** data type, as shown below:

```
new-object: make object! new-values
```

The second argument, `new-values` , is a block that defines additional variables and initial values for the new object. Each variable that is defined within the block is an `instance` variable of the object. For example, if the block contained two variable definitions, then they would be variables of the object:

```
example: make object! [
    var1: 10
    var2: 20
]
```

The `example` object has two variables that hold two integers.

The block is evaluated, so it can include any type of expression to compute the values of the variables:

```
example: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
]
```

Once an object has been made, it can serve as a prototype for creating new objects:

```
example2: make example []
```

The above example makes a second instance of the example object. The new object is a clone of the first object. New values for the second object are set in the block:

```
example2: make example [
    var1: 30
    var2: var1 + 10
]
```

In the example above, the `example2` object has different values than the original `example` object for two of its variables.

The `example2` object can also extend the object definition by adding new variables to it:

```
example2: make example [
    var4: now/date
    var5: "example"
]
```

The result is an object that has five variables: Three that came from the original object, `example` , and two new ones.

The process of extending the definition of an object can be repeated any number of times.

You can also create an object that contains variables that are initialized to some common value. This can be done using a cascaded set of word definitions:

```
example3: make object! [
    var1: var2: var3: var4: none
]
```

In the example above, the four variables are set to `none` within the new object.

To summarize, the process of creating an object involves these steps:

- Use make to create a new object based on a parent object or the object! data type.
- Add any new variables that are defined in the block to the new object.
- Evaluate the block, which causes the variables defined in the block to be set to the values in the new object.
- The new object is returned as a result.

3. Cloning Objects

When you use a parent object to make a new object, the parent object is cloned rather than inherited. This means that if the parent object is modified, it has no effect on the child object.

As an example, the following code creates a bank account object, whose variables are blank:

```
bank-account: make object! [
    first-name:
    last-name:
    account:
    balance: none
]
```

To use the new object, values can be provided to create an account for a customer:

```
luke: make bank-account [
    first-name: "Luke"
    last-name: "Lakeswimmer"
    account: 89431
    balance: $1204.52
]
```

Since new accounts are made on a regular basis, it helps to use a function and some global variables to create them:

```
last-account: 89431
bank-bonus: $10.00

make-account: func [
    "Returns a new account object"
    f-name [string!] "First name"
    l-name [string!] "Last name"
    start-balance [money!] "Starting balance"
][
    last-account: last-account + 1
    make bank-account [
        first-name: f-name
        last-name: l-name
        account: last-account
        balance: start-balance + bank-bonus
    ]
]
```

Now a new account object for `Fred` would only require:

```
fred: make-account "Fred" "Smith" $500.00
```

4. Accessing Objects

Variables within objects are accessed with paths. The path consists of the object name followed by the name of the variable. For example, the following code accesses the variables in the `example` object:

```
example/var1

example/var2
```

Here are examples using the `bank-account` object:

```
print luke/last-name
Lakeswimmer
print fred/balance
$510.00
```

Using a path, the variables of an object can also be modified:

```
fred/balance: $1000.00
print fred/balance
$1000.00
```

You can use the **`in`** function to access object variables by fetching their words from within their object context:

```
print in fred 'balance
balance
```

The `balance` word returned has the object `fred` as its context. You can get the value it holds by using **`get:`**

```
print get in fred 'balance
$1000.00
```

The second argument to the **`in`** function is a literal word. This allows you to dynamically change words depending on what is needed:

```
words: [first-name last-name balance]
foreach word words [print get in fred word]
FredSmith
$1000.00
```

Each word in the block is used to obtain its value in the object.

The **`in`** function can also be used to set object variables.

```
set in fred 'balance $20.00
print fred/balance
$20.00
```

If a word is not defined within an object, the **`in`** function returns `none` . This is useful for

detecting when a variable exists within an object.

```
if get in fred 'bank [print fred/bank]
```

5. Object Functions

An object can contain variables that refer to functions that are defined within the context of the object. This is useful because the functions are encapsulated within the context of the object, and can access the other variables of the object directly, without a need for a path.

As a simple example, the `example` object can include functions for computing new values within the object:

```
example: make object! [
    var1: 10
    var2: var1 + 10
    var3: now/time
    set-time: does [var3: now/time]
    calculate: func [value] [
        var1: value
        var2: value + 10
    ]
]
```

Notice in the example that the functions are able to refer to the variables of the object directly, rather than as paths. That is possible because the functions are defined within the same context as the variables they access.

To set a new time, use:

```
example/set-time
```

This example evaluates the function that sets `var3` to the current time.

To calculate new values for `var1` and `var2` , use:

```
example/calculate 100
print example/var2
110
```

In the case of the `bank-account` object, the functions for `deposit` and `withdraw` can be added to the current definition:

```
bank-account: make bank-account [
    deposit: func [amount [money!]] [
        balance: balance + amount
    ]
    withdraw: func [amount [money!]] [
        either negative? balance [
            print ["Denied. Account overdrawn by"
                absolute balance]
```

```
        ][balance: balance - amount]
    ]
]
```

In the example, notice that the functions are able to refer to the `balance` directly within the object. That's because the functions are part of the object's context.

Now if a new account is made, it will contain functions for depositing and withdrawing money. For example:

```
lily: make-account "Lily" "Lakeswimmer" $1000

print lily/balance
$1010.00
lily/deposit $100

print lily/balance
$1110.00
lily/withdraw $2000

print lily/balance
-$890.00
lily/withdraw $2.10
Denied. Account overdrawn by $890.00
```

6. Prototype Objects

Any object can serve as a prototype for making new objects. For instance, the `lily` account object previously defined can be used to make new objects with a line such as:

```
maya: make lily []
```

This makes an instance of an object. The object is a copy of the customer object and has identical values:

```
print lily/balance
-$890.00
print maya/balance
-$890.00
```

You can modify the new object while making it by providing the new values within the definition block:

```
maya: make lily [
    first-name: "Maya"
    balance: $10000
]

print maya/balance
$10000.00
maya/deposit $500

print maya/balance
```

```
$10500.00
print maya/first-name
Maya
```

The `lily` object serves as a prototype for creating the new object. Any words that are not redefined for the new object continue to have the values of the old object:

```
print maya/last-name
Lakeswimmer
```

New words are added to the object in a similar way:

```
maya: make lily [
    email: maya@example.com
    birthdate: 4-July-1977
]
```

7. Referring to Self

Every object includes a predefined variable called `self` . Within the context of an object, the `self` variable refers to the object itself. It can be used to pass the object to other functions or to return it as a result of a function.

In the following example, the `show-date` function requires an object as its argument and `self` is passed to it:

```
show-date: func [obj] [print obj/date]

example: make object! [
    date: now
    show: does [show-date self]
]

example/show
16-Jul-2000/11:08:37-7:00
```

Another example of using the `self` variable is a function that clones itself:

```
person: make object! [
    name: days-old: none
    new: func [name' birthday] [
        make self [
            name: name'
            days-old: now/date - birthday
        ]
    ]
]

lulu: person/new "Lulu Ulu" 17-May-1980

print lulu/days-old
7366
```

8. Encapsulation

An object provides a good way to encapsulate a group of variables that should not appear at the global level. When function variables are defined as globals, they can unintentionally be modified by other functions.

The solution to this problem of global variables is to wrap an object around both the variables and the function. When that is done, the function can still access the variables, but the variables cannot be accessed globally. For example:

```
Bank: make object! [

    last-account: 89431
    bank-bonus: $10.00

    set 'make-account func [
        "Returns a new account object"
        f-name [string!] "First name"
        l-name [string!] "Last name"
        start-balance [money!] "Starting balance"
    ][
        last-account: last-account + 1
        make bank-account [
            first-name: f-name
            last-name: l-name
            account: last-account
            balance: start-balance + bank-bonus
        ]
    ]
]
```

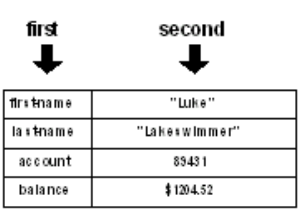
In this example, the variables are safe from accidental modification. Notice that the `make-account` function was set to a variable using the **`set`** function, rather than using a variable definition. This was done to make it a global function. The function can be used in the same way as functions set with a variable definition, but does not require an object path:

```
bob: make-account "Bob" "Baker" $4000
```

9. Reflective Properties

As with many other REBOL data types, you can access the components of objects in a manner that allows you to write useful tools and utilities for creating, monitoring, and debugging them.

The **`first`** and **`second`** functions allow you to access the components of an object. The **`first`** function returns the words defined for an object. The **`second`** function returns the values that the objects are set to. The following diagram shows the relationship between the return values of **`first`** and **`second:`**



The advantage to using **first** is that it allows you to obtain a list of the words for the function without knowing anything else about the function:

```
probe first luke
[self first-name last-name account balance]
```

In the above example, notice that the list contains the word, `self` , which is a reference to the object itself. You can exclude `self` when getting an object's word list by using **next:**

```
probe next first luke
[first-name last-name account balance]
```

Now you have a way to write a function that can probe the contents of an object:

```
probe-object: func [object][
    foreach word next first object [
        print rejoin [word ":" tab get in object word]
    ]
]

probe-object fred
first-name: Luke
last-name: Lakeswimmer
account: 89431
balance: $1204.52
```

When accessing objects in this fashion, care should be taken to avoid infinite loops. For instance, if you attempt to probe certain objects that contain references to themselves, your code may begin an endless loop. This is the reason why you cannot probe the **system** object directly. The **system** object contains many references to itself.

11 - Math

1. Overview
2. Scalar Data Types
3. Evaluation Order
4. Standard Functions and Operators
    4.1 absolute
    4.2 add
    4.3 complement
    4.4 divide
    4.5 multiply
    4.6 negate
    4.7 random
    4.8 remainder
    4.9 subtract
5. Type Conversion
6. Comparison Functions
    6.1 equal
    6.2 greater
    6.3 greater-or-equal
    6.4 lesser
    6.5 lesser-or-equal
    6.6 not equal to
    6.7 same
    6.8 strict-equal
    6.9 strict-not-equal
7. Logarithmic Functions
    7.1 exp
    7.2 log-10
    7.3 log-2
    7.4 log-e
    7.5 power
    7.6 square-root
8. Trigonometric Functions
    8.1 arccosine
    8.2 arcsine
    8.3 arctangent
    8.4 cosine
    8.5 sine
    8.6 tangent
9. Logic Functions
    9.1 and
    9.2 or
    9.3 xor
    9.4 complement
    9.5 not
10. Errors
    10.1 Attempt to divide by zero
    10.2 Math or number overflow
    10.3 Positive number required
    10.4 Cannot use operator on datatype! value

1. Overview

REBOL provides a comprehensive set of mathematical and trigonometric operations. Many of

these operators can handle multiple datatypes, including integer, decimal, money, tuple, time, and date. Some of these datatypes may even be mixed, or coerced.

2. Scalar Data Types

The mathematical functions of REBOL operate in a consistent manner over a wide range of scalar (numerical) data types. These data types include:

Datatype	Description
`Integer!`	32 bit numbers without decimal point
`Decimal!`	64 bit floating point numbers
`Money!`	currency with 64 bit floating point number
`Time!`	hours, minutes, seconds, and sub-seconds
`Date!`	day, month, year, time, time zone
`Pair!`	graphical position or size
`Tuple!`	versions, colors, network addresses

The following are a few examples that show a range of math operations over the scalar data types. Notice that operators produce useful results for each data type.

The `integer` and `decimal` data types:

```
print 2 + 1
3
print 2 - 1
1
print 2 * 10
20
print 20 / 10
2
print 21 // 10
1
print 2.2 + 1
3.2
print 2.2 - 1
1.2
print 2.2 * 10
22
print 2.2 / 10
0.22
print random 10
5
```

The `time` data type:

```
print 2:20 + 1:40
4:00
print 2:20 + 5
2:20:05
print 2:20 + 60
2:21
print 2:20 + 2.2
2:20:02.2
print 2:20 - 1:20
1:00
```

```
print 2:20 - 5
2:19:55
print 2:20 - 120
2:18
print 2:20 * 2
4:40
print 2:20 / 2
1:10
print 2:20:01 / 2
1:10:00.5
print 2:21 // 2
0:00
print - 2:20
-2:20
print random 10:00
5:30:52
```

The **date** data type:

```
print 1-Jan-2000 + 1
2-Jan-2000
print 1-Jan-2000 - 1
31-Dec-1999
print 1-Jan-2000 + 31
1-Feb-2000
print 1-Jan-2000 + 366
1-Jan-2001
birthday: 7-Dec-1944
print ["I've lived" (now/date - birthday) "days."]
I've lived 20305 days.
print random 1-1-2000
29-Apr-1695
```

The **money** data type:

```
print $2.20 + $1
$3.20
print $2.20 + 1
$3.20
print $2.20 + 1.1
$3.30
print $2.20 - $1
$1.20
print $2.20 * 3
$6.60
print $2.20 / 2
$1.10
print $2.20 / $1.10
2
print $2.21 // 2
$0.21
print random $10.00
$6.00
```

The **pair** data type:

```
print 100x200 + 10x20
```

```
110x220
print 10x10 + 3
13x13
print 10x20 * 2x4
20x80
print 100x100 * 3
300x300
print 100x30 / 10x3
10x10
print 100x30 / 10
10x3
print 101x32 // 10x3
1x2
print 101x32 // 10
1x2
print random 100x20
67x12
```

The **tuple** data type:

```
print 1.2.3 + 3.2.1
4.4.4
print 1.2.3 - 1.0.1
0.2.2
print 1.2.3 * 3
3.6.9
print 10.20.30 / 10
1.2.3
print 11.22.33 // 10
1.2.3
print 1.2.3 * 1.2.3
1.4.9
print 10.20.30 / 10.20.30
1.1.1
print 1.2.3 + 7
8.9.10
print 1.2.3 - 1
0.1.2
print random 10.20.30
8.18.12
```

3. Evaluation Order

There are two rules to remember when evaluating mathematical expressions:

- Expressions are evaluated from left to right.
- Operators take precedence over functions.

The evaluation of expressions from left to right is independent of the type of operator that is used. For example:

```
print 1 + 2 * 3
9
```

In the example above, notice that the result is not seven, as would be the case if multiplication took precedence over addition.

Important Note

The way mathematical expressions are evaluated from left to right regardless of the operator is different than many other computer languages. Many languages have rules of precedence that you must remember that determine the order of evaluation of operators. For example, a multiply is done before an add. Some languages have 10 or more such rules.

In REBOL, rather than requiring users to remember the precedence of operators, you only need to remember the left-to-right rule. More importantly, for advanced code such as expressions that handle expressions (in reflection, for example) you do not need to reorder terms based on precedence. The evaluation order is kept simple.

For most math expressions, the left-to-right evaluation rule works quite well and is easy to remember. However, because this rule is different from other languages, it can be a source of programming errors, so watch out.

The best solution is to check your work. You can also use parentheses if necessary to clarify your expression (see below), and you can always type your expression at the console to verify your result.

If you need to evaluate in some other order, reorder the expression or use parentheses:

```
print 2 * 3 + 1
7
print 1 + (2 * 3)
7
```

When functions are mixed with operators, the operators are evaluated first, then the functions:

```
print absolute -10 + 5
5
```

In the above example, the addition is performed first, and its result is provided to the **absolute** function.

In the next example:

```
print 10 + sine 30 + 60
11
```

the expression is evaluated in this order:

```
30 + 60 => 90
sine 90 => 1
10 + 1 => 11
print
```

To change the order such that the **sine** of 30 is done first, use parentheses:

```
print 10 + (sine 30) + 60
70.5
```

or reorder the expression:

```
print 10 + 60 + sine 30
70.5
```

4. Standard Functions and Operators

This section describes the standard math functions and operators used in REBOL.

4.1 absolute

The expressions:

```
absolute value

abs value
```

return the absolute value of `value`.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`pair`** data types.

```
print absolute -10
10
print absolute -1.2
1.2
print absolute -$1.2
$1.20
print absolute -10:20
10:20
print absolute -10x-20
10x20
```

4.2 add

The expressions:

```
value1 + value2

add value1 value2
```

return the result of adding `value1` to `value2`.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`tuple`**, **`pair`**, **`date`**, **`char`** data types.

```
print 1 + 2
```

```
3
print 1.2 + 3.4
4.6
print 1.2.3 + 3.4.5
4.6.8
print $1 + $2
$3.00
print 1:20 + 3:40
5:00
print 10x20 + 30x40
40x60
print #"A" + 10
K
print add 1 2
3
```

4.3 complement

The expression:

```
complement value
```

returns the numeric complement (bitwise complement) of a value.

Works with **integer**, **decimal**, **tuple** data types.

```
print complement 10
-11
print complement 10.5
-11
print complement 100.100.100
155.155.155
```

4.4 divide

The expressions:

```
value1 / value2

divide value1 value2
```

return the result of dividing `value1` by `value2`.

Works with **integer**, **decimal**, **money**, **time**, **tuple**, **pair**, **char** data types.

```
print 10 / 2
5
print 1.2 / 3
0.4
print 11.22.33 / 10
1.2.3
print $12.34 / 2
$6.17
```

```
print 1:20 / 2
0:40
print 10x20 / 2
5x10
print divide 10 2
5
```

4.5 multiply

The expressions:

```
value1 * value2

multiply value1 value2
```

return the result of multiplying `value1` by `value2`.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`tuple`**, **`pair`**, **`char`** data types.

```
print 10 * 2
20
print 1.2 * 3.4
4.08
print 1.2.3 * 3.4.5
3.8.15
print $10 * 2
$20.00
print 1:20 * 3
4:00
print 10x20 * 3
30x60
print multiply 10 2
20
```

4.6 negate

The expressions:

```
- value

negate value
```

change the sign of the `value`.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`pair`**, **`char`** data types.

```
<A name=_Toc487519923>print - 10
-10
print - 1.2
-1.2
print - $10
-$10.00
print - 1:20
```

```
-1:20
print - 10x20
-10x-20
print negate 10
-10
```

4.7 random

The expression:

```
random value
```

returns a random value that is less than or equal to value given.

Note that for integers **random** begins at 1, not 0, and is inclusive of the value given. This allows **random** to be used directly with functions like **pick**.

When a decimal is used the result is a decimal data type rounded to an integer.

The **/seed** refinement restarts the random generator. Use the **/seed** refinement with **random** first it if you want unique random number generation. You can use the current date and time to make the seed more random:

```
random/seed now
```

Works with **integer**, **decimal**, **money**, **time**, **tuple**, **pair**, **date**, **char**, **string**, **block** data types.

```
print random 10
5
print random 10.5
2
print random 100.100.100
79.95.66
print random $100
$32.00
print random 10:30
6:37:33
print random 10x20
2x4
print random 30-Jun-2000
27-Dec-1171
```

4.8 remainder

The expressions:

```
value1 // value2

remainder value1 value2
```

return the remainder of dividing `value1` by `value2`.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`tuple`**, **`pair`** data types.

```
print 11 // 2
1
print 11.22.33 // 10
1.2.3
print 11x22 // 2
1x0
print remainder 11 2
1
```

4.9 subtract

The expressions:

```
value1 - value2

subtract value1 value2
```

return the result of subtracting `value2` from `value1`.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`tuple`**, **`pair`**, **`date`**, **`char`** data types.

```
print 2 - 1
1
print 3.4 - 1.2
2.2
print 3.4.5 - 1.2.3
2.2.2
print $2 - $1
$1.00
print 3:40 - 1:20
2:20
print 30x40 - 10x20
20x20
print #"Z" - 1
Y
print subtract 2 1
1
```

5. Type Conversion

When math operations are performed between data types, normally the non-integer or non-decimal data type is returned. When integers are combined with decimals, a decimal data type is returned.

6. Comparison Functions

All comparison functions return either `true` or `false`.

6.1 equal

The expressions:

```
value1 = value2

equal? value1 value2
```

return `true` if the first and second values are equal.

Works with **integer**, **decimal**, **money**, **time**, **date**, **tuple**, **char** and **series** data types.

```
print 11-11-99 = 11-11-99
true
print equal? 111.112.111.111 111.112.111.111
true
print #"B" = #"B"
true
print equal? "a b c d" "A B C D"
true
```

6.2 greater

The expressions:

```
value1 > value2

greater? value1 value2
```

return `true` if the first value is greater than the second value.

Works with **integer**, **decimal**, **money**, **time**, **date**, **tuple**, **char** and **series** data types.

```
print 13-11-99 > 12-11-99
true
print greater? 113.111.111.111 111.112.111.111
true
print #"C" > #"B"
true
print greater? [12 23 34] [12 23 33]
true
```

6.3 greater-or-equal

The expressions:

```
value1 >= value2

greater-or-equal? value1 value2
```

return `true` if the first value is greater than or equal to the second value.

Works with **integer**, **decimal**, **money**, **time**, **date**, **tuple**, **char** and **series** data types.

```
print 11-12-99 >= 11-11-99
true
print greater-or-equal? 111.112.111.111 111.111.111.111
true
print #"B" >= #"A"
true
print greater-or-equal? [b c d e] [a b c d]
true
```

6.4 lesser

The expressions:

```
value1 < value2

lesser? value1 value2
```

return `true` if the first value is less than second value.

Works with **integer**, **decimal**, **money**, **time**, **date**, **tuple**, **char** and **series** data types:

```
print 25 < 50
true
print lesser? 25.3 25.5
true
print $2.00 < $2.30
true
print lesser? 00:10:11 00:11:11
true
```

6.5 lesser-or-equal

The expressions:

```
value1 <= value2

lesser-or-equal? value1 value2
```

return `true` if the first value is less than or equal to the second value.

Works with **integer**, **decimal**, **money**, **time**, **date**, **tuple**, **char** and **series** data types.

```
print 25 <= 25
true
print lesser-or-equal? 25.3 25.5
true
print $2.29 <= $2.30
true
print lesser-or-equal? 11:11:10 11:11:11
```

```
true
```

6.6 not equal to

The expressions:

```
value1 <> value2

not-equal? value1 value2
```

return `true` if the first and second values are not equal.

Works with **`integer`**, **`decimal`**, **`money`**, **`time`**, **`date`**, **`tuple`**, **`char`** and **`series`** data types.

```
print 26 <> 25
true
print not-equal? 25.3 25.5
true
print $2.29 <> $2.30
true
print not-equal? 11:11:10 11:11:11
true
```

6.7 same

The expressions:

```
value1 =? value2

same? value1 value2
```

return `true` if two words refer to the same value. For instance, when you want to see if two words are referencing the same index in a series.

Work with all data types.

```
reference-one: "abcdef"
reference-two: reference-one
print same? reference-one reference-two
true
reference-one: next reference-one
print same? reference-one reference-two
false
reference-two: next reference-two
print same? reference-one reference-two
true
reference-two: copy reference-one
print same? reference-one reference-two
false
```

6.8 strict-equal

The expressions:

```
value1 == value2

strict-equal? value1 value2
```

return `true` if the first and second values are strictly the same. Can be used as a case-sensitive version of the **`equal?`** (=) operator for strings and to differentiate between integers and decimals when their values are the same.

Works with all data types.

```
print strict-equal? "abc" "ABC"
false
print equal? "abc" "ABC"
true
print strict-equal? "abc" "abc"
true
print strict-equal? 1 1.0
false
print equal? 1 1.0
true
print strict-equal? 1.0 1.0
true
```

6.9 strict-not-equal

The expression:

```
strict-not-equal? value1 value2
```

returns `true` if the first and second values are strictly not the same. Can be used as a case-sensitive version of the **`not-equal?`** (<>) operator for strings and to differentiate between integers and decimals when their values are the same.

Works with all data types.

```
print strict-not-equal? "abc" "ABC"
true
print not-equal? "abc" "ABC"
false
print strict-not-equal? "abc" "abc"
false
print strict-not-equal? 1 1.0
true
print not-equal? 1 1.0
false
print strict-not-equal? 1.0 1.0
false
```

7. Logarithmic Functions

7.1 exp

The expression:

```
exp value
```

raises E (natural number) to the power of `value`.

7.2 log-10

The expression:

```
log-10 value
```

returns the base-10 logarithm of `value`.

7.3 log-2

The expression:

```
log-2 value
```

returns the base-2 logarithm of `value`.

7.4 log-e

The expression:

```
log-e value
```

returns the base-E (natural number) log. of `value`.

7.5 power

The expressions:

```
value1 ** value2

power value1 value2
```

return the result of raising `value1` to `value2` power.

7.6 square-root

The expression:

```
square-root value
```

returns the square root of `value`.

8. Trigonometric Functions

The trigonometric functions deal in degrees. Use the **`/radians`** refinement with any of the trigonometric functions to operate in and return radians.

8.1 arccosine

The expression:

```
arccosine value
```

returns the trigonometric arccosine of `value`.

8.2 arcsine

The expression:

```
arcsine value
```

returns the trigonometric arcsine of `value`.

8.3 arctangent

The expression:

```
arctangent value
```

returns the trigonometric arctangent of `value`.

8.4 cosine

The expression:

```
cosine value
```

returns the trigonometric cosine of `value`.

8.5 sine

The expression:

```
sine value
```

returns the trigonometric sine of `value`.

8.6 tangent

The expression:

```
tangent value
```

returns the trigonometric tangent of `value`.

9. Logic Functions

Logic functions can be performed on logic values and on some scalar values including **`integer`**, **`char`**, **`tuple`**, and **`bitset`**. When working with logic values, the logic functions return boolean values. When working with other types of values, the logic functions on the bits.

9.1 and

The **`and`** function compares two logic values and returns `true` if they are both true :

```
print (1 < 2) and (2 < 3)
true
print (1 < 2) and (4 < 3)
false
```

When used with integers, the **`and`** function compares bit for bit and returns `1` if both bits are `1`, or `0` if neither bit is `1` :

```
print 3 and 5
1
```

9.2 or

The **`or`** function compares two logic values and returns `true` if either of them are true or `false` or if both are false :

```
print (1 < 2) or (2 < 3)
true
print (1 < 2) or (4 < 3)
true
print (3 < 2) or (4 < 3)
false
```

When used with integers, **`or`** compares bit for bit and returns `1` if either bit is `1` or **`0`** if both bits are 0:

```
print 3 or 5
7
```

9.3 xor

The **xor** function compares two logic values and returns `true` if and only if one of the values is true and the other is false.

```
print (1 < 2) xor (2 < 3)
false
print (1 < 2) xor (4 < 3)
true
print (3 < 2) xor (4 < 3)
false
```

When used with integers, **xor** compares bit for bit and returns `1` if and only if one bit is `1` and the other `0` . Otherwise, it returns `0` :

```
print 3 xor 5
6
```

9.4 complement

The **complement** function returns the logic or bitwise complement of a value. It is used for bitmask integer numbers and inverting bitsets.

```
print complement true
false
print complement 3
-4
```

9.5 not

For a logic value **not** returns `true` if the value is false and `false` if the value is true. It does not perform numerical bitwise operations.

```
print not true
false
print not false
true
```

10. Errors

Math errors are reported when an illegal operation is performed or when an overflow or underflow occurs. The following errors may be encountered in math operations.

10.1 Attempt to divide by zero

An attempt was made to divide a number by 0.

```
1 / 0
** Math Error: Attempt to divide by zero
** Where: connect-to-link
** Near: 1 / 0
```

10.2 Math or number overflow

An attempt was made to process a number too large for REBOL to handle.

```
1E+300 + 1E+400
** Math Error: Math or number overflow
** Where: connect-to-link
** Near: 1 / 0
```

10.3 Positive number required

An attempt was made to process a negative number with a math operator that accepts only positive numbers.

```
log-10 -1
** Math Error: Positive number required
** Where: connect-to-link
** Near: log-10 -1
```

10.4 Cannot use operator on datatype! value

An attempt was made to process incompatible data types. The data type of the second argument in the operation is returned as listed.

```
10:30 + 1.2.3
** Script Error: Cannot use add on time! value
** Where: connect-to-link
** Near: 10:30 + 1.2.3
```

12 - Files

1. Overview
2. Names and Paths
    2.1 File Names
    2.2 Path Strings
    2.3 Case Sensitivity
    2.4 File Name Functions
3. Reading Files
    3.1 Reading Text Files
    3.2 Reading Binary Files
    3.3 Reading Over the Network
4. Writing Files
    4.1 Writing Text Files
    4.2 Writing Binary Files
    4.3 Writing Files to a Network
5. Line Conversion
6. Blocks of Lines
7. File and Directory Information
    7.1 Directory Check
    7.2 File Existence
    7.3 File Size
    7.4 File Modification Date
    7.5 Directory Information
8. Directories
    8.1 Reading a Directory
    8.2 Making a Directory
    8.3 Renaming Directories and Files
    8.4 Deleting Directories and Files
    8.5 Current Directory
    8.6 Changing the Current Directory
    8.7 Listing the Current Directory

1. Overview

An important aspect of REBOL's power is its ability to manipulate files and directories. REBOL provides a wide range of functions designed to allow operations ranging from simple file reads to direct access to files and directories. For more information on direct access to files and directories, see the Ports Chapter.

2. Names and Paths

REBOL provides a standard, machine independent file and path naming convention.

2.1 File Names

In scripts, file names and paths are written with a percent sign (%) followed by a sequence of characters:

```
%examples.r
%big-image.jpg
%graphics/amiga.jpg
```

```
%/c/plug-in/video.r
%//sound/goldfinger.mp3
```

The percent sign is necessary to prevent file names from being interpreted as words within the language.

Although it is not a good practice, spaces can be included in file names by enclosing the file name in double quotes (" "). The double quotes prevent the file name from being interpreted as multiple words:

```
%"this file.txt"
%"cool movie clip.mpg"
```

The standard Internet convention of using a percent sign (%) and a hex code is also allowed for character encoding. When this is done, quotes are not required. The above file names could also be written as:

```
%this%20file.txt
%cool%20movie%20clip.mpg
```

Note that the standard file suffix for REBOL scripts is ".r". On systems where this convention collides with another file type, a ".reb" suffix can be used instead.

2.2 Path Strings

File paths are written with a percent sign (%) followed by a sequence of directory names that are each separated by a forward slash (/).

```
%dir/file.txt
%/file.txt
%dir/
%/dir/
%/dir/subdir/
%../dir/file.txt
```

The standard character for separating directories is the forward slash (/), not the backslash (\). If backslashes are found they are converted to forward slashes:

```
probe %\some\cool\movie.mpg
%/some/cool/movie.mpg
```

REBOL provides a standard, operating system independent method for specifying directory paths. Paths can be relative to the current directory or absolute from the top-level file structure of the operating system.

File paths that do not begin with a forward slash (/) are relative paths.

```
%docs/intro.txt
%docs/new/notes.txt
```

```
%"new mail/inbox.mbx"
```

The standard convention of using double dots (..) to indicate a parent directory or a single dot (.) to refer to the current directory is also supported. For example:

```
%.
%./
%./file.txt
%..
%../
%../script.r
%../../plans/schedule.r
```

File paths use the standard Internet convention of beginning absolute paths with a forward slash (/). The forward slash indicates to start at the top level of the file system. (Generally, absolute paths should be avoided to ensure machine-independent scripts.) The example:

```
%/home/file.txt
```

would refer to a disk volume or partition named `home`. Other examples are:

```
%/ram/temp/test.r
%/cd0/scripts/test/files.r
```

To refer to the `C` volume that is often used by Windows, the notation is:

```
%/C/docs/file.txt
%"/c/program files/qualcomm/eudora mail/out.mbx"
```

Notice in the above lines that the disk volume, `C`, is not written as:

```
%c:/docs/file.txt
```

The above example is not a machine independent format and causes an error.

If the first directory name is absent, and the path begins with double forward slashes (//), then the file path is relative to the current volume:

```
%//docs/notes
```

2.3 Case Sensitivity

In REBOL, file names are ***not*** case-sensitive by default. However, when new files are created by the language, they keep the case they were typed in:

```
write %Script-File.r file-data
```

The above example creates the file name with the `S` and `F` in uppercase.

In addition, when file names are read from directories, the case is preserved:

```
print read %/home
```

For case-sensitive systems, such as UNIX, REBOL finds the closest case match to the specified file. For example, if a script asks to read `%test`.r, but only finds `%TEST`.r, the `%TEST`.r file is read. This behavior is necessary to allow machine-independent scripts.

2.4 File Name Functions

Various functions are provided to help you create file names and paths. These are listed below in File Name Functions.

to-file	Converts strings and blocks into a file name or file path.
split-path	Splits a path into its directory part and its file name.
clean-path	Returns the absolute path that is equivalent to any given path containing double dot (..) or dot (.).
what-dir	Returns the absolute path to the current directory.

3. Reading Files

Files are read as a series of text characters or as binary bytes. The source of the file is either a local file on your system or a file from the network.

3.1 Reading Text Files

To read a local text file, use the **read** function:

```
text: read %file.txt
```

The **read** function returns a string that holds the entire text of the file. In the above example, the variable `text` refers to that string.

Within the string returned by **read**, line terminators are converted to `newline` characters, regardless of what style of line termination is used on your operating system. This allows you to write scripts that search for `newline` without concern for what particular character or characters constitute a line termination.

```
next-line: next find text newline
```

A file can also be read as separate lines that are stored in a block:

```
lines: read/lines %file.txt
```

See the Line Conversion section for more information about `newline` and reading lines.

To read a file a piece at a time, use the **open** function as described in the Ports Chapter.

To view the contents of a text file, you can read it using **read** and print it using **print:**

```
print read %service.txt
I wanted the gold, and I sought it,I scrabbled and mucked like
a slave.
```

3.2 Reading Binary Files

To read a binary file such as an image, a program, or a sound, use **read/binary:**

```
data: read/binary %file.bin
```

The **read/binary** function returns a binary series that holds the entire contents of the file. In the above example, the variable `data` refers to the binary series. No conversion of any type is done to the file.

To read a binary file a piece at a time, use the **open** function as described in the Ports Chapter.

3.3 Reading Over the Network

Files can be read from a network. For example, to view a text file from a network using the HTTP protocol:

```
print read http://www.rebol.com/test.txt
Hellotherenewuser!
```

The file could be written locally with the line:

```
write %test.txt read http:/www.rebol.com/test.txt
```

In the write process the file will have its line termination converted to that which is used by your operating system.

To read and save a binary file, such as an image, use the following line:

```
write %image.jpg
read/binary http:/www.rebol.com/image.jpg
```

Refer to the chapter on Network Protocols for more information and examples of accessing files across networks.

4. Writing Files

You can write a file as a series of text characters or as binary bytes. The location of the file can be either a local file on your system or a file on a network.

4.1 Writing Text Files

To write a local text file, use the following line of code:

```
write %file.txt "sample text here"
```

This writes the entire text to the file.

If a file contains `newline` characters, they will be converted to those used by your local file system. This allows you to deal with files in a consistent **manner**, but write them out using the convention that is standard to your file system.

For instance, the following line converts any text file from one line termination format (UNIX, Macintosh, PC, Amiga) to that which is used by your local system:

```
write %newfile.txt read %file.txt
```

The above line reads the entire file while converting its line termination to the REBOL standard, then writes the file converting it to the local operating system format.

To append to the end of a file, use the **/append** refinement:

```
write/append %file.txt "more text"
```

A file can also be written from separate lines that are stored in a block.

```
write/lines %file.txt lines
```

To write a file a piece at a time, use the **open** function as described in the Ports Chapter.

4.2 Writing Binary Files

To write a binary file such as an image, a program, a sound, use **write/binary:**

```
write/binary %file.bin data
```

The **write/binary** function creates the file if it does not exist or overwrites the file if it already exists. No conversion of any type is done to the file.

To write a binary file a piece at a time, use the **open** function as described in the Ports Chapter.

4.3 Writing Files to a Network

Files can also be written to a network. For example, to write a text file to a network using FTP, use:

```
write ftp://ftp.domain.com/file.txt "save this text"
```

The file can be read locally and written to the net with a line such as:

```
write ftp://ftp.domain.com/file.txt read %file.txt
```

In the process, the file has its line termination converted to the standard CRLF format.

To write a binary file, such as an image, to the network, use the following lines of code:

```
write/binary ftp://ftp.domain.com/file.txt/image.jpg
    read/binary %image.jpg
```

Refer to the chapter on Network Protocols for more information and examples of accessing files from networks.

5. Line Conversion

When a file is read as text, all line terminators are converted to `newline` (line feed) characters. Line feeds (used as line terminators on Amiga, Linux, and UNIX systems), carriage returns (used as line terminators on Macintosh), and the CR/LF combination (PC and Internet) are all converted to the equivalent `newline` characters.

Using a standard line terminator within scripts allows them to operate in a machine-independent fashion. For example, to search for and count all `newline` characters within a text file:

```
text: read %file.txt
count: 0
while [spot: find text newline][
    count: count + 1
    text: next spot
]
```

The line conversion is also useful for reading network files:

```
text: read ftp://ftp.rebol.com/test.txt
```

When a file is written, the `newline` character is converted to the line termination style standard for the operating system being used. For instance, the `newline` character is converted to a CRLF on the PC, LF on UNIX or Amiga, or CR for a Macintosh. Network files are written with CRLF.

The following function converts any text file with any terminator style to that used by the local operating system:

```
convert-lines: func [file] [write file read file]
```

The file is read and all line terminators are converted, then the file is written and `newline` characters are converted to the local operating system style.

Line conversion can be disabled by reading the file as binary. For instance, the following line:

```
write/binary %newfile.txt read/binary %oldfile.txt
```

preserves the line terminators of the text file.

6. Blocks of Lines

Text files can be easily accessed and managed as individual lines of text, rather than as a single series of characters. For example, to read a file as a block of lines:

```
lines: read/lines %service.txt
```

The above example returns a block containing a series of strings (one for each line) without line terminators. Empty lines are represented by empty strings.

To print a specific line you use the following code:

```
print first lines
print last lines
print pick lines 100
print lines/500
```

To print all of the lines of a file, use the following line of code:

```
foreach line lines [print line]
I wanted the gold, and I sought it,
I scrabbled and mucked like a slave.
Was it famine or scurvy -- I fought it;
I hurled my youth into a grave.
I wanted the gold, and I got it --
Came out with a fortune last fall, --
Yet somehow life's not what I thought it,
And somehow the gold isn't all.
```

To print all of the lines that contain the string `gold`, use the following line of code:

```
foreach line lines [
   if find line "gold" [print line]
]
I wanted the gold, and I sought it,
I wanted the gold, and I got it --
And somehow the gold isn't all.
```

You can write the text file out as lines using the **write** function:

```
write/lines %output.txt lines
```

To write out specific lines from a block, use:

```
write/lines %output.txt [
    "line one"
    "line two"
    "line three"
]
```

In fact, the functions **read/lines** and **write/lines** can be combined to process files one line at a time. For example the following code removes all of the comments from a REBOL script:

```
script: read/lines %script.r
foreach line script [
    where: find line ";"
    if where [clear where]
]
write/lines %script1.r script
```

The sample script in the above example is for demonstration purposes only. In addition to removing comments, the sample script would also remove valid semicolons in quoted strings.

Files can be read as lines from a network as well:

```
data: read/lines http://www.rebol.com

print pick (read/lines ftp://ftp.rebol.com/test.txt) 3
new
```

The **/lines** refinement can be used with the **open** function to read a line at a time from console input. See the chapter on Ports for more information.

In addition **/lines** can be used with **/append** to append lines from a block to a file.

7. File and Directory Information

There are a number of functions that provide useful information about a file, such as whether it exists, its file size in bytes, when it was last modified, and whether it is a directory.

7.1 Directory Check

To determine if a file name is that of a directory, use the **dir?** function:

```
print dir? %file.txt
false
print dir? %.
```

```
true
```

The **dir?** function works with some network protocols as well:

```
print dir? ftp://www.rebol.com/pub/
true
```

7.2 File Existence

To determine if a file exists, use the **exists?** function:

```
print exists? %file.txt
```

To determine if a file exists before you read it, use:

```
if exists? file [text: read file]
```

To avoid overwriting a file you can check it with, use:

```
if not exists? file [write file data]
```

The **exists?** function also works with some network protocols:

```
print exists? ftp://www.rebol.com/file.txt
```

7.3 File Size

To obtain the byte size of a file, use the **size?** function:

```
print size? %file.txt
```

The **size?** function also works with some network protocols:

```
print size? ftp://www.rebol.com/file.txt
```

7.4 File Modification Date

To obtain the last modification date of a file, use the **modified?** function:

```
print modified? %file.txt
30-Jun-2000/14:41:55-7:00
```

Not all operating systems keep track of the creation date of a file, so to keep REBOL scripts operating system independent only the last modification date is accessible.

The **modified?** function also works with some network protocols:

```
print modified? ftp://www.rebol.com/file.txt
```

7.5 Directory Information

The **info?** function obtains all file directory information at the same time. The information is returned as an object:

```
probe info? %file.txt
make object! [
    size: 306
    date: 30-Jun-2000/14:41:55-7:00
    type: 'file
]
```

To print information about all files in the current directory, use:

```
foreach file read %. [
    info: info? file
    print [file info/size info/date info/type]
]
build-guide.r 22334 30-Jun-2000/14:24:43-7:00 file
code/ 11 11-Oct-1999/18:37:04-7:00 directory
data.r 41 30-Jun-2000/14:41:36-7:00 file
file.txt 306 30-Jun-2000/14:41:55-7:00 file
```

The **info?** function also works with some network protocols:

```
probe info? ftp://www.rebol.com/file.txt
```

8. Directories

There are several easy-to-use functions for reading directories, managing subdirectories, making new directories, renaming files, and deleting files. In addition, there are standard functions for getting, changing, and listing the current directory. For more information on direct access to directories, see the Ports Chapter.

8.1 Reading a Directory

Directories are read in the same manner as files. The **read** function returns a block of file names rather than text or binary data.

To read all the file names from the current directory, use the following line of code:

```
read %.
```

The above example reads the entire directory and returns a block of file names.

To print the names of all files in a directory, use the following line of code:

```
print read %intro/
CVS/ history.t intro.t overview.t quick.t
```

Within the returned block, names of directories are indicated with a trailing forward slash. To print each file name on a separate line, use:

```
foreach file read %intro/ [print file]
CVS/
history.t
intro.t
overview.t
quick.t
```

Here is an easy way to print just the directories that were found:

```
foreach file read %intro/ [
    if #"/" = last file [print file]
]
CVS/
```

If you want to read a directory from the network, be sure to include a forward slash at the end of the URL to indicate to the protocol that you are referring to a directory:

```
print read ftp://ftp.rebol.com/
```

8.2 Making a Directory

The **make-dir** function makes a new directory. The new name for the directory can be relative to either the current directory or an absolute path.

```
make-dir %new-dir
make-dir %local-dir/
make-dir %/work/docs/old-docs/
```

The trailing slash is optional for this function.

Internally, the **make-dir** function calls **open** with the **/new** refinement. The line:

```
close open/new %local-dir/
```

also creates a new directory. The trailing slash is important in this example, indicating that a directory is to be created rather than a file.

If you use the **make-dir** function to create a directory that already exists, an error will occur.

The error can be caught with the **try** function. The directory can be checked in advance with the **exists?** function.

8.3 Renaming Directories and Files

To rename a file, use the **rename** function:

```
rename %old-file %new-file
```

The old file name may include a complete path to the file, but the new file name must not include a path. This is because the **rename** function is not intended to move files between directories (various operating systems do not provide this function).

```
rename %../docs/intro.txt %conclusion.txt
```

If the old file name is a directory (indicated by a trailing slash), the **rename** function renames the directory:

```
rename %../docs/ %manual/
```

If the file cannot be renamed, an error will occur. The error can be caught with the **try** function.

8.4 Deleting Directories and Files

Files can be deleted using the **delete** function:

```
delete %file
```

The file to delete can include a path:

```
delete %source/docs/file.txt
```

A block of files within the same directory can also be deleted:

```
delete [%file1 %file2 %file3]
```

A group of files can be deleted using a wildcard character and the **/any** refinement:

```
delete/any %file*
delete/any %secret.?
```

The asterisk (*) wildcard character matches all characters, and the question mark (?) wildcard character matches a single character.

To delete a directory, provide a trailing forward slash:

```
delete %dir/
delete %../docs/old/
```

If the file cannot be deleted, an error will occur. The error can be caught with the **try** function.

8.5 Current Directory

Use the **what-dir** function to determine the current directory:

```
print what-dir
/work/REBOL/
```

The **what-dir** function refers to the current script's directory path, as found in `system/script/path`.

8.6 Changing the Current Directory

To change the current directory, use:

```
change-dir %new-path/to-dir/
```

If the trailing slash is not included, the function adds it.

8.7 Listing the Current Directory

To list the contents of the current directory, use:

```
list-dir
```

The number of columns used to show the directory is dependent on the console window size and the maximum file name length.

13 - Network Protocols

Contents:

1. Overview
2. REBOL Networking Basics
    2.1 Modes of Operation
    2.2 Specifying Network Resources
    2.3 Schemes, Handlers, and Protocols
    2.4 Monitoring Handlers
3. Initial Setup
    3.1 Basic Network Settings
    3.2 Proxy Settings
    3.3 Other Settings
    3.4 Access to Settings
4. DNS - Domain Name Service
5. Whois Protocol
6. Finger Protocol
7. Daytime - Network Time Protocol
8. HTTP - Hyper Text Transfer Protocol
    8.1 Reading a Web Page
    8.2 Scripts on Web Sites
    8.3 Loading Markup Pages
    8.4 Other Functions
    8.5 Acting Like a Browser
    8.6 Posting CGI Requests
9. SMTP - Simple Mail Transport Protocol
    9.1 Sending Email
    9.2 Multiple Recipients
    9.3 Bulk Mail
    9.4 Subject Line and Headers
    9.5 Debug Your Scripts
10. POP - Post Office Protocol
    10.1 Reading Email
    10.2 Removing Email
    10.3 Handling Email Headers
11. FTP - File Transfer Protocol
    11.1 Using FTP
    11.2 FTP URLs
    11.3 Transferring Text Files
    11.4 Transferring Binary Files
    11.5 Appending to Files
    11.6 Reading Directories
    11.7 File Information
    11.8 Making Directories
    11.9 Deleting Files
    11.10 Renaming Files
    11.11 About Passwords
    11.12 Transferring Large Files
12. NNTP - Network News Transfer Protocol
    12.1 Reading the Newsgroup List
    12.2 Reading All Messages
    12.3 Reading Single Messages
    12.4 Handling News Headers
    12.5 Sending a News Message
13. CGI - Common Gateway Interface
    13.1 CGI Server Setup
    13.2 CGI Scripts
    13.3 Generating HTML Content
    13.4 CGI Environment
    13.5 CGI Requests
    13.6 Processing HTML Forms
14. TCP - Transmission Control Protocol

14.1 Creating Clients
14.2 Creating Servers
14.3 A Tiny Server
14.4 Testing TCP Code
15. UDP - User Datagram Protocol

1. Overview

REBOL includes several of the primary Internet service protocols built-in. These protocols are easy to use within your scripts; they require no extra libraries or include files, and many useful operations can be done with only a single line of source code.

The protocols listed in Network Protocols are supported:

DNS	Domain Name Service: translates computer names into addresses and addresses into names.
Finger	Obtains information about a user from their profile.
Whois	Obtains information about domain registration.
Daytime	Network Time Protocol. Gets the time from a server.
HTTP	Hypertext Transfer Protocol. Used for the Web.
SMTP	Simple Mail Transfer Protocol. Used for sending email.
POP	Post Office Protocol. Used for fetching email.
FTP	File Transfer Protocol. Exchanges files with a server.
NNTP	Network News Transfer Protocol. Posts or reads Usenet news.
TCP	Transmission Control Protocol. Basic Internet protocol.
UDP	User Datagram Protocol. Packet-based protocol.

In addition, you can create handlers for other Internet protocols or make your own custom protocols.

2. REBOL Networking Basics

2.1 Modes of Operation

There are two basic modes of network operation: atomic and port-based.

`Atomic` network operations are those that are accomplished in a single function. For instance, you can read an entire Web page with a single call to the read function. There is no need to separately open a connection or set up the read. All of that is done automatically as part of the read. For example, you can type:

```
print read http://www.rebol.com
```

The host is found and opened, its Web page transferred, and the connection closed.

The `port-based` mode of operation is one that uses a more traditional programming approach. It involves opening a port and performing various series operations on the port. For instance, if you want to read your email from a POP server one message at a time, you would use this method. Here is an example that reads and displays all of your email:

```
pop: open pop://user:pass@mail.example.com
forall pop [print first pop]
close pop
```

The atomic method of operation is easier, but it is also more limited. The port-based method allows more types of operations, but also requires a greater understanding of networking.

2.2 Specifying Network Resources

REBOL provides two approaches for specifying network resources: URLs and port specifications.

Uniform Resource Locators (URL) are used on the Internet to identify a network resource, such as a Web page, FTP site, email address, file, or other resource or service. URLs are integral to the operation of REBOL, and they can be expressed directly in the language.

The standard notation for URLs consists of a scheme followed by a specification:

```
scheme:specification
```

The scheme is often the name of a protocol, such as HTTP, FTP, SMTP, and POP; however, that is not a requirement. A scheme can be any name that identifies the method used to access a resource.

The format of a scheme's specification depends on the scheme; however, most schemes share a common format for identifying network hosts, user names, passwords, port numbers, and file paths. Here are a few commonly used formats:

```
scheme://host

scheme://host:port

scheme://user@host

scheme://user:pass@host

scheme://user:pass@host:port

scheme://host/path

scheme://host:port/path

scheme://user@host/path

scheme://user:pass@host/path

scheme://user:pass@host:port/path
```

Network Resource Specification lists the fields used in the above formats.

scheme	The name used to identify the type of resource, often the same as the protocol. For example, HTTP, FTP, and POP.
host	The network name or address for a machine. For example, www.rebol.com, cnn.com, accounting.
port	Port number on the host machine for the scheme being used. Normally there is a default for this, so it is not required most of the time. Examples: 21, 23, 80, 8000.
user	A user name to access the resource.
pass	A password to verify the user name.
path	A file path or some other method for referencing the resource. This is scheme dependent. Some schemes include patterns and script arguments (such as CGI).

Another way to identify a resource is with a REBOL port specification. In fact, when a URL is used, it is automatically converted into a port specification. A port specification can accept many more arguments than a URL, but it requires multiple lines to express.

A port specification is written as an object block definition that provides each of the parameters necessary to access the network resource. For instance, the URL to access a Web site is:

```
read http://www.rebol.com/developer.html
```

but, it can also be written as:

```
read [
    scheme: 'HTTP
    host: "www.rebol.com"
    target: %/developer.html
]
```

The URL for an FTP read can be:

```
read ftp://bill:vbs@ftp.example.com:8000/file.txt
```

but, it can also be written as:

```
read [
    scheme: 'FTP
    host: "ftp.example.com"
    port-id: 8000
    target: %/file.txt
    user: "bill"
    pass: "vbs"
]
```

In addition, there are many other port fields that can be specified, such as timeout, type of access, and security.

2.3 Schemes, Handlers, and Protocols

REBOL networking operates by using `schemes` to identify `handlers` that communicate with `protocols`.

In REBOL a `scheme` is used to identify the method of accessing a resource. That method uses a code object that is called a `handler`. Each of the URL schemes that are supported by REBOL (such as HTTP, FTP) has a handler. The list of schemes can be obtained with:

```
probe next first system/schemes
[default Finger Whois Daytime SMTP POP HTTP FTP NNTP]
```

In addition, there are lower level scheme names that are not shown here. For instance, the TCP and UDP schemes are used for direct, lower level communication.

New schemes can be added to this list. For instance, you can define your own scheme, called FTP2, that provides special features for FTP access, such as automatically supplying your username and password so it does not need to be included in every FTP URL.

Most handlers are used to provide an interface to a network protocol. A `protocol` is used to communicate between various devices, including clients and servers.

Although each protocol is quite different in how it communicates, it does have some things in common with other protocols. For instance, most protocols require a network connection to be opened, read, written, and closed. These common operations are performed by a default handler in REBOL. This handler makes protocols like finger, whois, and daytime almost trivial to implement.

Scheme handlers are written as objects. The default handler serves as the root object for all the other

handlers. When a handler requires a particular field, such as a timeout value to use for reading data, if the value is not defined in the specific handler, it will be provided by the default handler. Hence, handlers overlay one another with their fields and value. You can also create handlers that use other handlers for default values. For instance, you can create an FTP2 handler that looks for missing fields first in the FTP handler, then in the default handler.

When a port is used to access network resources, it is linked to a specific handler. The handler and the port together form the unit that is used to provide the data, code, and state information to process all protocols.

The source code to handlers can be obtained from the system/scheme object. This can be useful if you want to modify the behavior of a handler or build your own handler. For instance, to view the code for the whois handler, type:

```
probe get in system/schemes 'whois
```

Note that what you are seeing is a composite of the default handler with the whois handler. The actual source code that is used to create the whois handler is only a few lines:

```
make Root-Protocol [
    open-check:  [[any [port/user ""]] none]
    net-utils/net-install Whois make self [] 43
]
```

2.4 Monitoring Handlers

For debugging purposes, you can monitor the actions of any handler. Each handler has its own debugging output to indicate what operations are being performed. To enable network debugging, turn network tracing on with the line:

```
trace/net on
```

To turn network debugging off, use:

```
trace/net off
```

Here is an example:

```
read pop://carl:poof@zen.example.com
URL Parse: carl poof zen.example.com none none none
Net-log: ["Opening tcp for" POP]
connecting to: zen.example.com
Net-log: [none "+OK"]
Net-log: {+OK QPOP (version 2.53) at zen.example.com starting.}
Net-log: [["USER" port/user] "+OK"]
Net-log: "+OK Password required for carl."
Net-log: [["PASS" port/pass] "+OK"]
** User Error: Server error: tcp -ERR Password supplied for "carl"
is incorrect.
** Where: read pop://carl:poof@zen.example.com
```

3. Initial Setup

REBOL networking is built-in. To create scripts that use the network protocols you do not need any special include files or libraries. The only requirement is that you provide the basic information necessary to enable protocols to connect to servers or through firewalls and proxies. For instance, to send an email, the SMTP

protocol needs an SMTP server name and a reply email address.

3.1 Basic Network Settings

When you run REBOL the first time, you re prompted for the necessary network settings, which is stored in the `user`.r file. REBOL uses this file to load the required network settings each time it is started. If a `user`.r is not created and REBOL cannot find an existing `user`.r file in its paths, no settings are loaded. See the chapter on Operation for more information.

To change the network settings, type **`set-user`** at the prompt. This runs the same network configuration script that ran when REBOL first started. This script is loaded from the `rebol`.r file. If that file cannot be found, or if you want to edit the setting directly, you can use a text editor on the `user`.r file.

Within the `user`.r file the network settings are found in a block that follows the **`set-net`** function. At a minimum the block should contain two items:

- Your email address for use in the from and reply fields of email and for anonymous FTP login
- Your default server; this is also your primary email server

In addition, you can specify a few other items:

- A different incoming email server (for POP)
- A proxy server (for connecting to the network)
- A proxy port number
- A proxy type (see Proxy Settings below).

You can also add lines after the **`set-net`** function to configure other protocol values. For instance you can set the timeout values for protocols, set the FTP passive mode, set the HTTP user-agent identifier, set up separate proxies for different protocols, and more.

An example of **`set-net`** is:

```
set-net [user@domain.dom mail.server.dom]
```

The first field specifies your email from address, and the second field indicates your default server (notice that it does not need quotes here). For most networks, this is enough and no other settings are necessary (unless you require a proxy). Also your default server is used whenever a specific server is not provided.

In addition, if you use a POP server (for incoming email) that is different from your SMTP server (for outgoing email), you can specify that as well:

```
set-net [
    user@domain.dom
    mail.server.dom
    pop.server.dom
]
```

However, if your SMTP and POP servers are the same, then this is not necessary.

3.2 Proxy Settings

If you use a proxy or firewall, you can provide the set-net function with your proxy settings. This can include the proxy server name or address, a proxy port number to access the server, and an optional proxy type. For example:

```
set-net [
    email@addr
    mail.example.com
    pop.example.com
    proxy.example.com
```

```
    1080
    socks
]
```

This example would use a proxy called `proxy`.example.com on its TCP port 1080 with the socks proxy method. To use a socks4 proxy server, use the word `socks4` rather than `socks`. To use the generic CERN server, use the word `generic`.

You can also set the proxy to be different machines for different schemes (protocols). Each protocol has its own proxy object where you can set the proxy values for just that scheme. Here is an example of setting a proxy for FTP:

```
system/schemes/ftp/proxy/host: "proxy2.example.com"

system/schemes/ftp/proxy/port-id: 1080

system/schemes/ftp/proxy/type: 'socks
```

In this case, only FTP uses a special proxy server. Notice that the machine name must be a string and the proxy type must be a literal word.

Here are two more examples. The first example sets the proxy for HTTP to be the generic (CERN) proxy method:

```
system/schemes/http/proxy/host: "wp.example.com"

system/schemes/http/proxy/port-id: 8080

system/schemes/http/proxy/type: 'generic
```

In the above example, all HTTP requests go through a generic proxy on `wp`.example.com using TCP port 8080.

If you want to disable the proxy settings for a particular scheme, you can set the proxy fields to false.

```
system/schemes/smtp/proxy/host: false

system/schemes/smtp/proxy/port-id: false

system/schemes/smtp/proxy/type: false
```

In the above example, all outgoing email does not go through a proxy. The `false` value prevents even the default proxy from being used. If you set these fields to `none`, then the default proxy is used if it is configured.

If you want to bypass the proxy settings for particular machines, such as those on your local network, you can provide a bypass list. Here is a bypass list for the default proxy:

```
system/schemes/default/proxy/bypass:
    ["host.example.net" "*.example.com"]
```

Note that the asterisk (*) and question mark (?) characters can be used for pattern matching. The asterisk (*) as used in the example above bypasses any machine that ends with `example`.com.

To set a bypass list for only the HTTP scheme, type:

```
system/schemes/http/proxy/bypass:
    ["host.example.net" "*.example.com"]
```

3.3 Other Settings

In addition to proxy settings, you can set network timeout values for all of the schemes (in the default) or for specific schemes. For instance, to increase the timeout for all schemes, you can write:

```
system/schemes/default/timeout: 0:05
```

This sets the network timeout for 5 minutes.

If you want to increase the timeout just for SMTP, you would write:

```
system/schemes/smtp/timeout: 0:10
```

Some schemes have custom fields. For instance, the FTP scheme allows you to set passive mode for all transfers:

```
system/schemes/ftp/passive: on
```

FTP passive mode is useful because FTP servers that are set to passive mode do not attempt to connect back through your firewall.

When making HTTP accesses to Web sites, you may want to use a different user-agent field in the HTTP request to get better results on a few sites that detect the browser type:

```
system/schemes/http/user-agent: "Mozilla/4.0"
```

3.4 Access to Settings

Each time REBOL is started, it reads the `user`.r file to find its network settings. These settings are made with the **`set-net`** function. Scripts have access to these settings through the `system/schemes` object.

```
system/user/email ; used for email from and reply
system/schemes/default/host - your primary server
system/schemes/pop/host - your POP server
system/schemes/default/proxy/host - proxy server
system/schemes/default/proxy/port-id - proxy port
system/schemes/default/proxy/type - proxy type
```

Below is a function that returns a block containing the network settings in the same order as **`set-net`** accepts them:

```
get-net: func [][
    reduce [
        system/user/email
        system/schemes/default/host
        system/schemes/pop/host
        system/schemes/default/proxy/host
        system/schemes/default/proxy/port-id
```

```
        system/schemes/default/proxy/type
    ]
]

probe get-net
```

4. DNS - Domain Name Service

DNS is the network service that translates domain names to their associated IP address. In addition, you can use DNS to find a machine and domain name from an IP address.

The DNS protocol can be used in three ways: you can lookup the primary IP address of a machine name, you can lookup the domain name for an IP address, and you can find the name and IP address of your local system.

To lookup the primary IP address of a specific machine within a specific domain, type:

```
print read dns://www.rebol.com
207.69.132.8
```

You can also obtain the domain name that is associated with a particular IP address:

```
print read dns://207.69.132.8
rebol.com
```

Note that it is not unusual for this reverse DNS lookup to return a none. There are machines that do not have host names.

```
print read dns://11.22.33.44
none
```

To find your system's host name, read an empty DNS URL of the form:

```
print read dns://
crackerjack
```

The data returned here depends on the type of machine. It may be the unqualified host name, as shown above, but it can also be the fully-qualified host name, `crackerjack`.example.com. This depends on the operating system and the network configuration in the operating system.

Here's an example that looks up and prints the IP addresses for a number of Web sites:

```
domains: [
    www.rebol.com
    www.rebol.org
    www.mochinet.com
    www.sirius.com
]

foreach domain domains [
    print ["address for" domain "is:"
        read join dns:// domain]
]
address for www.rebol.com is: 207.69.132.8
address for www.rebol.org is: 207.66.107.61
```

```
address for www.mochinet.com is: 216.127.92.70
address for www.sirius.com is: 205.134.224.1
```

5. Whois Protocol

The whois protocol retrieves information about domain names from a central registry. The whois service is provided by the organizations that run the Internet. Whois is often used to retrieve registration information about an Internet domain or server. It can tell you who owns the domain, how their technical contact can be reached, along with other information.

To obtain information, use the **read** function with a whois URL. This URL should contain the domain name and a whois server name separated by an at sign (@). For example to obtain information about example.com from the Internic registry:

```
print read whois://example.com@rs.internic.net
connecting to: rs.internic.net
Whois Server Version 1.1
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
Domain Name: EXAMPLE.COM
Registrar: NETWORK SOLUTIONS, INC.
Whois Server: whois.networksolutions.com
Referral URL: www.networksolutions.com
Name Server: NS.ISI.EDU
Name Server: VENERA.ISI.EDU
Updated Date: 17-aug-1999
>>> Last update of whois database: Sun, 16 Jul 00 03:16:34 EDT <<<
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

The above code is only an example. The details of the information being returned and the servers that support whois change over time.

If instead of a domain name you provide a word, all entries that match that word are returned:

```
print read whois://example@rs.internic.net
connecting to: rs.internic.net
Whois Server Version 1.1
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
EXAMPLE.512BIT.ORG
EXAMPLE.ORG
EXAMPLE.NET
EXAMPLE.EDU
EXAMPLE.COM
To single out one record, look it up with "xxx", where xxx is one
of the of the records displayed above. If the records are the same, look them
up with "=xxx" to receive a full display for each record.
>>> Last update of whois database: Sun, 16 Jul 00 03:16:34 EDT <<<
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

The whois protocol does not accept URLs, such as www.example.com, unless the URL is part of the registrant's company name.

6. Finger Protocol

The finger protocol retrieves user-specific information stored in the user log file.

To request user information from a server it must be running the finger protocol. The information is requested by reading a finger URL that contains a username and a domain name in an email style format:

```
print read finger://username@example.com
```

The above example retrieves information about the user at `username@example`.com. The information returned depends on the information provided by the user and the settings of the finger server. Also, the details of the information being returned are up to each server; the examples below only describe typical servers. Many servers can have non-standard behaviors on their finger ports.

For instance, the following information may be returned:

```
Login: username
Name: Firstname Lastname
Directory: /home/user
Shell: /usr/local/bin/tcsh
Office: City, State +1 555 555 5555
Last login Wed Jul 28 01:10 (PDT) on ttyp0 from some.example.com
No Mail.
No Plan.
```

Notice that finger reports when the user last logged in from a machine, and whether the user has mail waiting. If the user reads email from this account, finger sometimes reports when mail was received and when the user last retrieved email:

```
New mail received Sun Sep 26 11:39 1999 (PDT)
Unread since Tue Sep 21 04:45 1999 (PDT)
```

The finger server can also report the contents of a `plan` file and a `project` file if they exist. Users can include any information they want in a `plan` or `project` file.

It is also possible to retrieve information about users using their real first name or their last name. Some finger servers require that you capitalize the names exactly as they appear in the login file or in the file used by the online finger server, to retrieve user information. Other finger servers are more liberal about capitalization. A finger server will respond to real name queries by returning all listings that match the query criteria. For instance, if there are several users on a host that have the first name `zaphod`, then entering the query

```
print read finger://Zaphod@main.example.com
```

will retrieve all such users whose first or last name is `Zaphod`.

Some finger servers return a listing of users when the user name is omitted. For example,

```
print read finger://main.example.com
```

retrieves a list of all users who are logged onto the machine, if the finger service installed on the hosting machine allows it.

Some host machines limit finger services for security reasons. They may require a valid username and only return information regarding that user. If you finger such a server without providing user information, the server will report that it requires specific user information.

If a system does not support the finger protocol, REBOL reports an access error:

```
print read finger://host.dom
connecting to: host.dom
Access Error: Cannot connect to host.dom.
Where: print read finger://host.dom
```

7. Daytime - Network Time Protocol

The daytime protocol retrieves the current day and time. To connect to a daytime server use **read** with a daytime URL. The URL contains the name of the server to read the date from:

```
print read daytime://everest.cclabs.missouri.edu
Fri Jun 30 16:40:46 2000
```

The format of the information returned by servers may vary, depending on the server. Notice that the time zone may not be present.

If the server you choose does not support daytime, REBOL returns an error:

```
print read daytime://www.example.com
connecting to: www.example.com
** Access Error: Cannot connect to www.example.com.
** Where: print read daytime://www.example.com
```

8. HTTP - Hyper Text Transfer Protocol

The world wide Web is driven by two fundamental technologies: HTTP and HTML. HTTP is the Hypertext Transfer Protocol that controls how Web servers and Web browsers communicate with each other. HTML is the Hypertext Markup Language that defines the structure and contents of a Web page.

To retrieve a Web page, the browser sends a request to a Web server using HTTP. On receiving the request, the server interprets it, sometimes using a CGI script (see CGI - Common Gateway Interface), and sends back data. This data can be just about anything, including HTML, text, images, programs, and sound.

8.1 Reading a Web Page

To read a Web page, use the **read** function with an HTTP URL. For example:

```
page: read http://www.rebol.com
```

This returns the Web page for www.rebol.com. Note that a string that contains the HTML code for the page is returned by the read. No graphics or other information are fetched. To do so you would need to provide additional reads. The page can be displayed as HTML code using **print**, it can be written to a file with **write**, or it can be sent as email using **send**.

```
print page

write %index.html page

send zaphod@example.com page
```

The page can be processed in a variety of ways by using a variety of REBOL functions, such as **parse**, **find**, and **load**.

For instance, to search a Web page for all occurrences of the word REBOL, you can write:

```
parse read http://www.rebol.com [
    any [to "REBOL" copy line to newline (print line)]
]
```

8.2 Scripts on Web Sites

A Web server can provide more than just HTML scripts. Web servers are quite useful for supplying REBOL scripts as well.

You can load REBOL scripts directly from a Web server with **load:**

```
data: load http://www.rebol.com/data.r
```

You can also evaluate scripts directly from a server with **do:**

```
data: do http://www.rebol.com/code.r
```

Warning

Do this with care. Evaluating arbitrary scripts on open Internet servers is asking for trouble. Evaluate a script only if you completely trust its source, have fully inspected its source, or have kept your REBOL security settings on maximum.

In addition, Web pages that contain HTML can contain embedded REBOL scripts, and they can be run with:

```
data: do http://www.rebol.com/example.html
```

To determine if a script exists on a page before evaluating it, use the **script?** function.

```
if page: script? http://www.rebol.com [do page]
```

The **script?** function reads the page from the Web site and returns the page at its REBOL header position.

8.3 Loading Markup Pages

HTML and XML pages can be quickly converted to a REBOL block with the **load/markup** function. This function returns a block that consists of all the tags and strings found within the page. All spacing and line breaks are left intact.

To filter out all of the tags for a Web page and just print its text, type:

```
tag-text: load/markup http://www.rebol.com
text: make string! 2000

foreach item tag-text [
    if string? item [append text item]
]

print text
```

You could then search this text for string patterns. It will contain all of the spaces and line breaks of the original HTML file.

Here's another example that checks all links found on a Web page to make sure that the pages they reference exist:

```
REBOL []

page: http://www.rebol.com/developer.html
set [path target] split-path page
system/options/quiet: true  ; turn off connection msgs
tag-text: load/markup page
links: make block! 100

foreach tag tag-text [  ; find all anchor href tags
    if tag? tag [
        if parse tag [
            "A" thru "HREF="
            [{"} copy link to {"} | copy link to ">"]
            to end
        ][
            append links link
        ]
    ]
]

print links

foreach link unique links [  ; try each link
    if all [
        link/1 <> #"#"
        any [flag: not find link ":"
             find/match link "http:"]
    ][
        link: either flag [path/:link][to-url link]
        prin [link "... "]
        print either error? try [read link]
            ["failed"]["OK"]
    ]
]
```

8.4 Other Functions

To check if a Web page exists, use the **exists?** function, which returns `true` if the page exists.

```
if exists? http://www.rebol.com [
    print "page still there"
]
```

Note: It is usually faster in many cases to just read the page rather than checking first to see if it exists. Otherwise the script must contact the server twice, and that can be time consuming.

To request the date on which a Web page was last modified, use the **modified?** function:

```
print modified? http://www.rebol.com/developer.html
```

However, note that not all Web servers provide modification date information. Dynamically generated Web pages typically do not return a modification date.

Another way to determine if a Web page has changed is poll it every so often and check it. A handy way to verify that a Web page has changed is by using the **checksum** function. If the previously calculated checksum

of a Web page differs from its current value, then the Web page has been modified since it was last checked. Here is an example that uses this technique. It checks a page every eight hours.

```
forever [
    page: read http://www.rebol.com
    page-sum: checksum page
    if any [
        not exists? %page-sum
        page-sum <> (load %page-sum)
    ][
        print ["Page changed" now]
        save %page-sum page-sum
        send luke@rebol.com page
    ]
    wait 8:00
]
```

Whenever the page changes, it is sent to Luke via email.

8.5 Acting Like a Browser

Normally, REBOL identifies itself to a server when it reads from a Web site. However, some servers are programmed to respond to particular browsers only. If a request to a server does not produce the correct Web page, you can change the request to make it look like it came from some other type of Web browser. Pretending to be a Web browser is done by many programs to get Web sites to respond correctly. However, this practice does end up defeating the purpose behind the browser identification.

To change HTTP requests to look as though they are being sent by Netscape 4.0, you can modify the user-agent within the HTTP handler:

```
system/options/http/user-agent: "Mozilla/4.0"
```

Setting this variable affects all HTTP requests that follow.

8.6 Posting CGI Requests

HTTP CGI requests can be posted in two ways. You can include the CGI request data in the URL or you can provide the request data through an HTTP post operation.

The URL CGI request uses a normal URL. The example below sends the CGI script `test`.r the data value of 10.

```
read http://www.example.com/cgi-bin/test.r?data=10
```

The post CGI request requires that you supply the CGI data as part of a custom refinement to the **`read`** function. The example below shows how data is posted to CGI:

```
read/custom http://www.example.com/cgi-bin/test.r [
    post "data: 10"
]
```

In this example, the **`/custom`** refinement is used to provide additional information to the read. The second argument is a block that begins with the word **`post`** and is followed by the string to send.

The post method is useful for easily sending REBOL code and data to a web server that runs CGI. The following example illustrates this:

```
data: [sell 10 shares of "ACME" at $123.45]

read/custom http://www.example.com/cgi-bin/test.r reduce [
    `post mold data
]
```

The mold function will produce the proper REBOL string to be sent to the server.

9. SMTP - Simple Mail Transport Protocol

The Simple Mail Transport Protocol (SMTP) controls the transfer of email messages on the Internet. SMTP defines the interaction between Internet hosts that participate in forwarding email from a sender to its destination.

9.1 Sending Email

Email is sent through SMTP by using the **send** function. This function can send an email message to one or more email addresses.

For **send** to operate correctly, your networking must be set up. The **send** function requires that you specify your email From address and your default email server. See Initial Setup above.

The **send** function takes two arguments: an email address and a message. For example:

```
send user@example.com "Hi from REBOL"
```

The first argument must be an email or block data type. The second argument can be any data type.

```
send luke@rebol.com $1000.00

send luke@rebol.com 10:30:40

send luke@rebol.com bill@ms.dom

send luke@rebol.com [Today 9-Apr-99 10:30]
```

Each of these simple email messages can be interpreted on the receiver's side (with REBOL) or viewed with a normal email program.

You can send an entire file by reading the file and passing it as the second argument to the send function:

```
send luke@rebol.com read %task.txt
```

Binary data, such as an image or executable file, can also be sent:

```
send luke@rebol.com read/binary %rebol
```

The binary data is encoded to allow it to be transferred as text.

To send a self-extracting binary message you can write:

```
send luke@rebol.com join "REBOL for the job" [
    newline "REBOL []" newline
```

```
    "write/binary %rebol decompress "
    compress read/binary %rebol
]
```

When the message is received, the file can be extracted by using the **do** function.

9.2 Multiple Recipients

To send to multiple recipients, you can provide a block of email names:

```
send [luke@rebol.com ben@example.com] message
```

In this case, each message is individually addressed with only the recipient's email name appearing in the `To` field (similar to BCC addressing).

The block of email addresses can be any size or even a file that you load. Just be sure that they are valid addresses, not strings. Strings are ignored.

```
friends: [
    bob@cnn.dom
    betty@cnet.dom
    kirby@hooya.dom
    belle@apple.dom
    ...
]

send friends read %newsletter.txt
```

9.3 Bulk Mail

If you are sending email to a large group, you can reduce the load on your server by delivering everyone in the group a single message. This is the purpose of the **/only** refinement. It uses a feature of SMTP to send only one message to multiple email addresses. Using the friends list from the previous example:

```
send/only friends message
```

The messages are not individually addressed. You may have seen this mode in some of the bulk email that you receive. When you receive bulk email, your address does not appear in the `To` field.

The bulk email mode of SMTP should be used for email lists and not for sending spam. Spam email is not proper network etiquette, it is illegal in some countries and states, and spam will get you banned from your ISP and from other sites.

9.4 Subject Line and Headers

By default the **send** function uses the first line of a message as the subject line. To provide your own subject line, you need to supply an email header to the **send** function. In addition to a subject line, you can provide an organization, date, CC, and even your own custom fields.

To include a header, use the **/header** refinement of the **send** function and include a header object. The header object must be made from the `system/standard/email` object. For example:

```
header: make system/standard/email [
    Subject: "Seen REBOL yet?"
    Organization: "Freedom Fighters"
```

```
]
```

Notice that the standard fields, such as the `From` address, are not required and are supplied automatically by the **send** function.

The header is then provided as an argument to **send/header:**

```
send/header friends message header
```

The email above is sent using the custom header for each message.

9.5 Debug Your Scripts

When testing email scripts, it is advised that you send email to yourself first, before sending it to others. Examine your test email carefully to make sure that it is what you want. It is common to have errors such as sending a file name rather than the file contents. For instance, you might write:

```
send person %the-data-file.txt
```

This sends the name of the file, not the file itself.

10. POP - Post Office Protocol

The Post Office Protocol (POP) allows you to fetch email that is waiting in a mail server mailbox. POP defines a number of operations for how to access and store email on your server.

10.1 Reading Email

You can read all of your email in a single line without removing it from the email server: This is done by reading from a POP URL in which you provided your username, password, and email host.

```
mail: read pop://user:pass@mail.example.com
```

The messages are returned as a block of strings which you can handle one message at a time using code such as:

```
foreach message mail [print message]
```

To read individual email messages from the server, you need to open a port connection to the server and handle each message one at a time. To open the POP port:

```
mailbox: open pop://user:pass@mail.example.com
```

In the example, `mailbox` can be accessed as a series. It responds to many of the standard series functions, such as **length?**, **first**, **second**, **third**, **pick**, **next**, **back**, **head**, **tail**, **head?**, **tail?**, **remove**, and **clear**.

To determine the number of mail messages residing on the server, use the **length?** function.

```
print length? mailbox
```

```
37
```

In addition, you can find out the total size of all messages and the individual sizes of messages with:

```
print mailbox/locals/total-size

print mailbox/locals/sizes
```

To display the first, second, and last messages, you can write:

```
print first mailbox

print second mailbox

print last mailbox
```

You can also use **pick** to fetch a specific message:

```
print pick mailbox 27
```

You can fetch and display each message from the oldest to the newest using a loop that is identical to that used for other types of series:

```
while [not tail? mailbox] [
    print first mailbox
    mailbox: next mailbox
]
```

You can also read your email from newest to oldest with a loop such as:

```
mailbox: tail mailbox

while [not head? mailbox] [
    mailbox: back mailbox
    print first mailbox
]
```

When you are done, be sure to **close** the mailbox. This can be done with a line such as:

```
close mailbox
```

10.2 Removing Email

As with series, the **remove** function can be used to delete a single message, and the **clear** function can be used to delete all of the messages from the current position to the end of the mailbox.

For example, to read a message, save it to a file, and remove it from the server:

```
mailbox: open pop://user:pass@mail.example.com
write %mail.txt first mailbox
```

```
remove mailbox
close mailbox
```

The message is removed from the server when the **close** is done.

To remove the 22nd email message from the server, you can write:

```
user:pass@mail.example.com
remove at mailbox 22
close mailbox
```

You can remove a number of messages by using the **/part** refinement with the remove function:

```
remove/part mailbox 5
```

To remove all of the messages in your mailbox, use the **clear** function:

```
mailbox: open pop://user:pass@example.com
clear mailbox
close mailbox
```

The **clear** function can also be used at different positions within the mailbox to remove messages to the end of the mailbox.

10.3 Handling Email Headers

Email messages always include a header. The header holds information such as the sender, recipient, subject, date, and other fields.

In REBOL email headers are handled as objects that contain all of the necessary fields. To convert email message to a header object you can use the **import-email** function. For example:

```
msg: import-email first mailbox

print first msg/from  ; the email address
print msg/date
print msg/subject
print msg/content
```

You can easily write a filter that scans your email for messages that begin with a particular subject line:

```
mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    msg: import-email first mailbox
    if find/match msg/subject "[REBOL]" [
        print msg/subject
    ]
    mailbox: next mailbox
]

close mailbox
```

Here is another example that informs you when email is received from a group of friends:

```
friends: [orson@rebol.com hans@rebol.com]

messages: read pop://user:pass@example.com

foreach message messages [
    msg: import-email message
    if find friends first msg/from [
        print [msg/from newline msg/content]
        send first msg/from "Got your email!"
    ]
]
```

This spam filter removes all messages from the server that do not contain your email name anywhere within the message:

```
mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    mailbox: either find first mailbox user@example.com
        [next mailbox][remove mailbox]
]

close mailbox
```

Here is a simple email list server that receives messages and sends them to a group. The server only accepts email from people in the group.

```
group: [orson@rebol.com hans@rebol.com]

mailbox: open pop://user:pass@example.com

while [not tail? mailbox] [
    message: import-email first mailbox
    mailbox: either find group first message/from [
        send/only group first mailbox
        remove mailbox
    ][next mailbox]
]

close mailbox
```

11. FTP - File Transfer Protocol

The File Transfer Protocol (FTP) is used widely on the Internet for transferring files to and from a remote host. FTP is commonly used for uploading pages to a Web site and for providing online file archives.

11.1 Using FTP

In REBOL FTP file operations are handled in much the same way as local file operations. Functions such as **read**, **write**, **load**, **save**, **do**, **open**, **close**, **exists?**, **size?**, **modified?**, and others are used with FTP. REBOL distinguishes between local files and files accessible by FTP through the use of an FTP URL.

Access to FTP servers can be open or closed. Open access allows anyone to login to the site and download files. This is called anonymous access and it is used frequently for public file archives. Closed access requires that you provide a username and password to download and upload files. This is the mode of operation for uploading Web pages to a Web site.

Although FTP does not require your REBOL networking to be configured, if you wish to use anonymous access, an email address is required. This address is found in the `system/user/email` object. Normally, when you boot REBOL, this field is set from your `user`.r file. See Initial Setup for more detail.

If you are using FTP through a proxy server or firewall, FTP may need to operate in passive mode. Passive mode does not require reverse connections from the FTP server to the client for data transfers. This mode only makes outgoing connections from your machine and allows a greater level of security. To enable passive mode you need to set a flag in the FTP protocol handler:

```
system/schemes/ftp/passive: true
```

If you do not know if it is necessary, try FTP first without it. If that does not work, try setting the passive flag.

11.2 FTP URLs

An FTP URL has the basic form:

```
ftp://user:pass@host/directory/file
```

For anonymous access the username and password can be left out:

```
ftp://host/directory/file
```

Most of the examples in this section use this form for simplicity; however, they also work with a username and password.

To access a remote directory, end the URL with a slash, such as:

```
ftp://user:pass@host/directory/

ftp://host/directory/

ftp://host/
```

More about directory access is shown below.

It is convenient to put the URL in a variable and use paths to provide the file names. This allows you to refer to the URL with just a word. For example:

```
site: ftp://ftp.rebol.com/pub/

read site/readme.txt
```

This technique is used in some of the sections that follow.

11.3 Transferring Text Files

FTP distinguishes between text files and binary files. When transferring text files, FTP converts the line break characters. This is not desirable for binary files.

To read a text file, supply the **read** function with an FTP URL:

```
file: read ftp://ftp.site.com/file.r
```

This puts the contents of the file into a string. To write the file locally, use this line:

```
write %file.r read ftp://ftp.site.com/file.r
```

Many of the refinements of **read** can also be used. For instance, you can use **read/lines** with:

```
data: read/lines ftp://ftp.site.com/file.r
```

This example returns a block of lines for the file. See the Files chapter for more information about the refinements to the **read** function.

To write a text file to the server, use the **write** function:

```
write ftp://ftp.site.com/file.r read %file.r
```

The **write** function can also include refinements. See the Files chapter.

As with normal text file transfers, all line termination will be properly converted during FTP transfers.

Here is a simple script that updates files to your Web site:

```
site: ftp://wwwuser:secret@www.site.dom/pages

files: [%index.html %home.html %info.html]

foreach file files [write site/:file read file]
```

This should not be used for transferring graphics or sound files, as they are binary. Use the technique shown in Transferring Binary Files.

In addition to the **read** and **write** functions, you can use the **load**, **save**, and **do** functions with FTP.

```
data: load ftp://ftp.site.com/database.r

save ftp://ftp.site.com/data.r data-block

do ftp://ftp.site.com/scripts/test.r
```

11.4 Transferring Binary Files

To avoid the line termination conversion when transferring binary files (images, archives, executable files), use the **/binary** refinement. For instance, to read a binary file from an FTP server:

```
data: read/binary ftp://ftp.site.com/file
```

To make a local copy of the file:

```
write/binary %file read/binary ftp://ftp.site.com/file
```

To write a binary file to a server:

```
write/binary ftp://ftp.site.com/file read/binary %file
```

No line termination conversions are performed.

To transfer a set of graphics files to a Web site, use this script:

```
site: ftp://user:pass@ftp.site.com/www/graphics

files: [%icon.gif %logo.gif %photo.jpg]

foreach file files [
    write/binary site/:file read/binary file
]
```

11.5 Appending to Files

FTP also allows you to append text and data to an existing file. To do so, use the **write/append** refinement as described in the Files chapter.

```
write/append ftp://ftp.site.com/pub/log.txt reform
    ["Log entry date:" now newline]
```

This can also be used with binary files.

```
write/binary/append ftp://ftp.site.com/pub/log.txt
    read/binary %datafile
```

11.6 Reading Directories

To read the file names of an FTP directory, follow the directory name with a forward slash:

```
print read ftp://ftp.site.com/

pub-files: read ftp://ftp.site.com/pub/
```

The ending forward slash (/) indicates that this is a directory access not a file access. The forward slash is not always required, but it is recommended when you know you are accessing a directory.

The block of files that is returned includes all of the files in the directory. Within that block, directory names are indicated with a forward slash following their names. For example:

```
foreach file read ftp://ftp.site.com/pub/ [
    print file
]
readme.txt
rebol.r
rebol.exe
library/docs/
```

You can also use the **dir?** function on a file to determine if it is a directory.

11.7 File Information

The same functions that provide information about files also provide information about FTP files. This includes the **modified?**, **size?**, **exists?**, **dir?**, and **info?** functions.

You can use the **exists?** function to determine if a file exists:

```
if exists? ftp://ftp.site.com/pub/log.txt [
    print "Log file is there"
]
```

This works for directories too, but include the forward slash at the end of the directory name:

```
if exists? ftp://ftp.site.com/pub/rebol/ [
    print read ftp://ftp.site.com/pub/rebol/
]
```

To get the size or modification date for a file:

```
print size? ftp://ftp.site.com/pub/log.txt

print modified? ftp://ftp.site.com/pub/log.txt
```

To determine if the file is actually a directory:

```
if dir? ftp://ftp.site.com/pub/text [
    print "It's a directory"
]
```

You can obtain all this information in a single access by using the **info?** function:

```
file-info: info? ftp://ftp.site.com/pub/log.txt

probe file-info

print file-info/size
```

To perform the same operation on a directory:

```
probe info? ftp://ftp.site.com/pub/
```

To print a directory listing:

```
files: open ftp://ftp.site.com/pub/

forall files [
    file: first files
    info: info? file
    print [file info/date info/size info/type]
]
```

11.8 Making Directories

New FTP directories can be created with the **make-dir** function:

```
make-dir ftp://user:pass@ftp.site.com/newdir/
```

11.9 Deleting Files

With appropriate permission settings, files can be deleted from a remote FTP server by using the **delete** function:

```
delete ftp://user:pass@ftp.site.com/upload.txt
```

You can also delete directories:

```
delete ftp://user:pass@ftp.site.com/newdir/
```

Note that a directory must be empty for this to succeed.

11.10 Renaming Files

You can **rename** a file with the line:

```
rename ftp://user:pass@ftp.site.com/foo.r %bar.r
```

The new name for the file will be `bar`.r.

FTP also allows you to move a file to a different directory with:

```
rename ftp://user:pass@ftp.site.com/foo.r %pub/bar.r
```

To rename a directory on an FTP site be sure to follow the directory name with a slash:

```
rename ftp://user:pass@ftp.site.com/rebol/ rebol-old/
```

11.11 About Passwords

The above examples include the password within their URLs, but if you plan on sharing your script, you probably don't want that information to be known. Here's a simple way to prompt for a password and build the correct URL:

```
pass: ask "Password? "

data: read join ftp://user: [pass "@ftp.site.com/file"]
```

Or, you can ask for both the username and password:

```
user: ask "Username? "
pass: ask "Password? "
data: read join ftp:// [
```

```
    user ":" pass "@ftp.site.com/file"
]
```

You can also open FTP connections by using a port specification rather than a URL. This allows you to use any password, even ones containing special characters that are not easily written in URLs. An example of a port specification to open an FTP connection is:

```
ftp-port: open [
    scheme: `ftp
    host: "ftp.site.com"
    user: ask "Username? "
    pass: ask "Password? "
]
```

See Specifying Network Resources above for more detail.

11.12 Transferring Large Files

Transferring large files requires special considerations. You may want to transfer the file in chunks to reduce the memory required by your computer and to provide user feedback while the transfer is happening.

Here is an example that downloads a very large binary file in chunks.

```
inp: open/binary/direct ftp://ftp.site.com/big-file.bmp
out: open/binary/new/direct %big-file.bmp
buf-size: 200000
buffer: make binary! buf-size + 2

while [not zero? size: read-io inp buffer buf-size][
    write-io out buffer size
    total: total + size
    print ["transferred:" total]
]
```

Be sure to use the **/direct** refinement, otherwise the entire file will be buffered internally by REBOL. The **read-io** and **write-io** functions allow reuse of the buffer memory that has already allocated. Other functions such as **copy** would allocate additional memory.

If the transfer fails, you can restart FTP from where it left off. To do so, examine the output file or the size variable to determine where to restart the transfer. Open the file again with a custom refinement that specifies **restart** and the location from which to start the read. Here is an example of the **open** function to use when the `total` variable indicates the length already read:

```
inp: open/binary/direct/custom
    ftp://ftp.site.com/big-file.bmp
    reduce ['restart total]
```

You should note that restart only works for binary transfers. It cannot be used with text transfers because the line terminator conversion that takes place will cause incorrect offsets.

12. NNTP - Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) is the basis for tens of thousands of newsgroups that provide a public forum for millions of Internet users. REBOL includes two levels of support for NNTP.

The built-in support for NNTP that provides very limited functionality and access. This is the NTTP scheme.

An extended level of functionality that is provided by the news scheme that is implemented in the file

distributed as nntp.r.

12.1 Reading the Newsgroup List

NNTP consists of two components: a list of newsgroups supported by a specific newsgroup server (newsgroups are typically selected by an Internet service provider); and, a database of messages that are currently available for any particular newsgroup.

To retrieve the list of all newsgroups from a specific news server, use the **read** function with an NNTP URL such as:

```
groups: read nntp://news.example.com
```

This may take a while, depending on your connection; there are thousands of newsgroups.

12.2 Reading All Messages

If you are using a fast connection, you can read all of the pending messages for a newsgroup with:

```
messages: read nntp://news.example.com/alt.test
```

However, caution is advised. Some newsgroups can have thousands of messages. It can take a long time to download all the messages, and you may run out of memory to hold them.

12.3 Reading Single Messages

To read single messages, open NNTP as a port and use series functions to access messages. This is similar to how you read email from a POP port. For example:

```
group: open nntp://news.example.com/alt.test
```

You can use the **length?** function to determine the number of messages that are available in the newsgroup:

```
print length? group
```

To read the first message available in the newsgroup, use **first:**

```
message: first group
```

To select a specific message in the group by index, use **pick:**

```
message: pick group 37
```

To create a simple loop that scans all messages for a keyword, use:

```
forall group [
    if find msg: first first group "REBOL" [
        print msg
    ]
```

```
]
```

Remember that when the loop returns, the group series is positioned to the tail. If you need to return to the head of the group:

```
group: head group
```

Be sure to **close** a port once you are done using it:

```
close group
```

12.4 Handling News Headers

News messages always include a header. The header holds information such as the sender, summary, keywords, subject, date, and other fields.

Headers are handled as objects. To convert a news message to a news header object you can use the **import-email** function. For example:

```
message: first first group
header: import-email message
```

You can now access the fields of the news message:

```
print [header/from header/subject header/date]
```

Different newsgroups and newsgroup clients use different fields in their header. To view the fields available for a specific message display the first item of the header object:

```
print first header
```

12.5 Sending a News Message

Before you can send a news message, you need to create a header for it. Here is a generic header that can be used for news:

```
news-header: make object! [
    Path: "not-for-mail"
    Sender: Reply-to: From: system/user/email
    Subject: "Test message"
    Newsgroups: "alt.test"
    Message-ID: none
    Organization: "Docs For All"
    Keywords: "Test"
    Summary: "A test message"
]
```

Before you can send it, you need to create a unique global identification number for it. Here is a function that does that:

```
make-id: does [
    rejoin [
        "<"
        system/user/email/user
        "."
        checksum form now
        "."
        random 999999
        "@"
        read dns://
        ">"
    ]
]

print news-header/message-id: make-id
<carl.4959961.534798@fred.example.com>
```

Now you can combine the header with the message. They must be separated by at least one blank line. The content of the message is read from a file.

```
write nntp://news.example.net/alt.test rejoin [
    net-utils/export news-header
    newline newline
    read %message.txt
    newline
]
```

13. CGI - Common Gateway Interface

The common gateway interface is used with many Web servers to provide processing beyond the normal HTTP Web interface. CGI requests are submitted from Web browsers to Web servers. When a server receives a CGI request, it typically executes a script to process the request and return a result to the browser. These CGI scripts can be written in a variety of languages, and REBOL provides one of the easier ways of handling CGI.

13.1 CGI Server Setup

Setting up CGI access is different for every Web server. See the instructions provided with your server.

Typically a server has an option for enabling CGI operation. You need to enable this option and provide a path to the directory where your CGI scripts reside. A common directory for CGI scripts is in `cgi-bin`.

On Apache servers, the `ExecCGI` option enables CGI scripts, and you can provide a directory (`cgi-bin`) for your scripts. This is normally set up by default installation of Apache.

To configure CGI for Microsoft IIS, go to the `properties` for cgi-bin and click on the `configuration` button. On the configuration panel click `add` and enter the path to your `rebol`.exe file. The format for this is:

```
C:\rebol\rebol.exe -cs %s %s
```

The two `%s` symbols are required for correctly passing the script and command line arguments to REBOL. Add the extension for REBOL files (.r) and set the last field to `PUT`, `DELETE`. The script engine does not need to be selected.

The `-cs` option that is provided to REBOL enables CGI operation and allows the script to access all files. (!! See notes below on how scripts can limit file access to selected directories).

With Web servers other than those described above, the server requires configuration to execute the REBOL executable for .r extension files and run REBOL with the required option `-cs`.

13.2 CGI Scripts

Before a script can be executed on most CGI servers, it needs to have the correct file permissions. On UNIX-type systems or those that use the Apache server you need to change the permissions to enable the script to be readable and executable by all users. This can be done with the **chmod** function. If you are new to this concept, you should read your operating system manual or talk with your system administrator before changing file permissions.

For Apache and various other Web servers to run REBOL scripts, you need to provide the correct header at the top of each script file. The header specifies the path to the REBOL executable file and the `-cs` option. This can be followed by the normal REBOL script header. Here is a simple CGI script that prints the string, `hello!`.

```
#!/path/to/rebol -cs

REBOL [Title: "CGI Test Script"]

print "Content-Type: text/plain"

print ""  ; required

print "Hello!"
```

There are many things that prevent a CGI script from running correctly. Get this simple script working first before you try more complex scripts. If your script does not work, here are a few items to check:

- You have CGI enabled on your Web server.
- The first line begins with a #! and the correct path to REBOL.
- The -cs option is supplied to REBOL.
- The script begins with "Content-Type:" being printed. (!!see below)
- The script is in the correct directory. (normally the cgi-bin directory)
- The script has the correct file permissions (readable and executable by all).
- The script contains the correct line break characters. Some servers do not run scripts that contain the CR character for line breaks. You may need to convert the file. (Use REBOL to do this in one line: write file, read file).
- The script does not contain errors. Test it without CGI to make sure that the script loads (does not have syntax errors) and functions properly. Provide some sample data and test it.
- All files that are accessed by the script have the correct file permissions.

Often one or more of the above items is wrong and prevent your script from running. You may see an error when viewing the Web page. If it says "Server Error" or "CGI Error" then it is typically something to do with the permissions or setup of the script. If it shows a REBOL error message, then the script is running, but you have an error within the script.

In the example script shown above, the `Content-Type` line is critical. It is part of the HTTP header that is returned to the browser, and it tells the browser the type of content being delivered. This is followed by a blank line to separate it from the actual content.

Many different types of content can be delivered. The previous example was plain text, but you can also deliver HTML as is shown in the next example. (See your Web server manual for more information about content types.)

The content type and blank line can be combined into a single line. The caret forward slash `(^/)` symbol provides an additional line break to separate it from the content.

```
print "Content-Type: text/plain^/"
```

It is a good practice to always print this line immediately from your script. This allows error messages to be seen by the browser if your script encounters an error.

Here is a simple CGI script that prints the time:

```
#!/path/to/rebol -cs

REBOL [Title: "Time Script"]

print "Content-Type: text/plain^/"

print ["The time is now" now/time]
```

13.3 Generating HTML Content

There are as many ways to create HTML content as there are ways to create strings. This page creates a page that displays a page hit counter:

```
#!/path/to/rebol -cs

REBOL [Title: "HTML Example"]

print "Content-Type: text/html^/"

count: either exists? %counter [load %counter][0]
save %counter count: count + 1

print [
    {<HTML><BODY><H2>Web Counter Page</H2>
    You are visitor} count {to this page!<P>
    </BODY></HTML>}
]
```

The script in the example above loads and saves to a counter text file. For this file to be accessible, it will require the appropriate permissions be set to allow access by all users.

13.4 CGI Environment

When a CGI script is run the server provides information to REBOL about the CGI request and its arguments. All of this information is provided as an object within the `system/options` object. To view the fields of the object, type:

```
probe system/options/cgi
make object! [
    server-software: none
    server-name: none
    gateway-interface: none
    server-protocol: none
    server-port: none
    request-method: none
    path-info: none
    path-translated: none
    script-name: none
    query-string: none
    remote-host: none
    remote-addr: none
    auth-type: none
    remote-user: none
    remote-ident: none
    Content-Type: none
    content-length: none
    other-headers: []
]
```

Of course, your script will ignore most of this information, but some of it could be of use. For instance, you may want to create a log file that records the network address of the system that made the request, or check

the type of browser being used.

To generate a CGI page that displays this content in your browser:

```
#!/path/to/rebol -cs

REBOL [Title: "Dump CGI Server Variables"]

print "Content-Type: text/plain^/"

print "Server Variables:"

probe system/options/cgi
```

If you want to use this information in a log, you can write it to a file. For example, to log the addresses of visitors to your CGI page you could write:

```
write/append/lines %cgi.log
    system/options/cgi/remote-addr
```

The **/append** and **/lines** refinements causes the write to be at the tail of the file and include a line-break. Here's another approach that puts multiple items on the same line:

```
write/append %cgi.log reform [
    system/options/cgi/remote-addr
    system/options/cgi/remote-ident
    system/options/cgi/content-type
    newline
]
```

13.5 CGI Requests

There are two methods for CGI to provide request data to your scripts: GET and POST.

The GET method encodes CGI data into the URL. This is used to provide information to the server. You may have noticed before that some URLs look like this:

```
http://www.example.com/cgi-bin/test.r?&data=test
```

The string that follows the question mark (?) provides the arguments to CGI. At times they can be quite long. This string is provided to your script when it is run. It can be obtained from the `cgi/query-string` field. For instance, to print the string from a script:

```
print system/options/cgi/query-string
```

The data within the string can include whatever data you require. However, because the string is part of a URL, data must be encoded. There are restrictions on the characters that are allowed.

In addition, when the data is created by HTML forms, it is encoded in a standard way. This data can be decoded and placed within an object with the code:

```
cgi: make object! decode-cgi-query
    system/options/cgi/query-string
```

The `decode-cgi-query` function returns a block that contains variable names and their values. See the HTML form example in the next section.

The POST method provides the CGI data as a string. The data does not need to be encoded. It can be in any format you desire and can even be binary. Post data is read from the standard input device. You will need to read it from the input with a line such as:

```
data: make string! 2002
read-io system/ports/input data 2000
```

This would read up to the first 2000 bytes of POST data and put it in a string.

A good format for POST data is to use a REBOL dialect and create a simple parser. The POST data can be loaded and parsed as a block. See the Parsing chapter.

Warning About Blocks

It is not a good idea to pass REBOL blocks to be directly evaluated because this can present a security risk. For instance, someone could POST a block that reads or deletes files on the server. However, it is safe to pass blocks that are interpreted by your script (a dialect).

Here is an example script that displays the post data in your browser:

```
#!/path/to/rebol -cs

REBOL [Title: "Show POST data"]

print "Content-Type: text/html^/"
data: make string! 10000
foreach line copy system/ports/input [
    repend data [line newline]
]

print [
    <HTML><BODY>
    {Here is the posted data.}
    <HR><PRE>data</PRE>
    </BODY></HTML>
]
```

13.6 Processing HTML Forms

CGI is often used for processing HTML forms. The forms accept input from various fields and submit them to the Web server as an HTML get or post method.

Here is an example that uses the CGI get to process a form and send an email as the result. There are two parts to this: the HTML page and the CGI script.

Here is an HTML page that includes a form:

```
<HTML><BODY>

<FORM ACTION="http://example.com/cgi-bin/send.r" METHOD="GET">

<H1>CGI Emailer</H1><HR>

Enter your email address:<P>

<INPUT TYPE="TEXT" NAME="email" SIZE="30"><P>
```

```
<TEXTAREA NAME="message" ROWS="7" COLS="35">
Enter message here.
</TEXTAREA><P>

<INPUT TYPE="SUBMIT" VALUE="Submit">

</FORM>
</BODY></HTML>
```

When the above script is submitted, it needs a CGI script to handle its results. Here is an example of such a script. This example script decodes the form data and sends the email. It returns a confirmation page.

```
#!/path/to/rebol -cs

REBOL [Title: "Send CGI Email"]

print "Content-Type: text/html^/"

cgi: make object! decode-cgi-query
    system/options/cgi/query-string

print {<HTML><BODY><H1>Email Status</H1><HR><P>}

failed: error? try [send to-email cgi/email cgi/message]

print either failed [
    {The email could not be sent.}
][
    [{The email to} cgi/email {was sent.}]
]

print {</BODY><HTML>}
```

This script should be named `send`.r and stored in the `cgi-bin` directory. It's permissions must be set to being readable and executable by all.

When the form has been submitted by a browser, this script will run. It decodes the CGI query string into a cgi object. The object now has email and message variables that are used for the **`send`** function. Before **`send`** is done, the email field is converted from a string to an `email` datatype.

The **`send`** function is placed within a try block to catch errors if they occur while sending the email. The failed variable is set to true if an error occurred, and the appropriate message is generated.

Other CGI examples can be found in the REBOL Script Library at http://www.rebol.com/library/library.html.

14. TCP - Transmission Control Protocol

In addition to those protocols previously described, you can create your own network servers and clients with the transmission control protocol, TCP.

14.1 Creating Clients

TCP ports can be opened in the same way as other REBOL protocols, using the TCP URL. To open a TCP connection to an HTTP (Web) server on TCP port number 80:

```
http-port: open tcp://www.example.com:80
```

Another way of opening a TCP connection is to provide the port specification directly. This is a substitute for using a URL and is often quite useful:

```
http-port: open [
    scheme: 'tcp
    host: "www.example.com"
    port-id: 80
]
```

Since ports are series, you can use the same series functions for sending and receiving data. The example below queries the HTTP server opened in the previous example. It uses the **insert** function to put data into the port series which sends it to the server:

```
insert http-port "GET / HTTP/1.0^/^/"
```

The two newline characters are used to tell the server that the header has been sent.

The newline characters are automatically converted to CR LF sequences because the port was opened in text mode.

The server processes the HTTP request and returns a result to the port series. To read the result, use the **copy** function:

```
while [data: copy http-port] [prin data]
```

This loop will continue to fetch data until a `none` is returned from copy. This behavior differs between protocols. A none is returned because the server closes the connection. Other protocols may send a special character to indicate the end of the transfer.

Now that all the data has been received, HTTP port should be closed:

```
close http-port
```

Here is another example that connects to a POP port on a server:

```
pop: open/lines tcp://fred.example.com:110
```

This example uses the **/lines** refinement. The connection will now be line oriented. Data will be written and read as lines. To read the first line from the server:

```
print first pop
+OK QPOP (version 2.53) at fred.example.com starting.
```

To send the server a username for POP login:

```
insert pop "user carl"
```

Because the port is operating in line mode, a line terminator is sent after the insert. The server response can be read with with:

```
print first pop
```

```
+OK Password required for carl.
```

And the rest of the communication would proceed as:

```
insert pop "pass secret"

print first pop
+OK carl has 0 messages (0 octets).
insert pop "quit"

first pop
+OK Pop server at fred.example.com signing off.
```

The connection should now be closed:

```
close pop
```

14.2 Creating Servers

To create a server you need to wait for connections and respond to them as they occur. To set up a port on your machine that can be used to wait for incoming connections:

```
listen: open tcp://:8001
```

Notice that you do not supply a host name, only a port number. This type of port is called a `listen` port. The system now accepts connections on port number 8001.

To wait for a connection from another machine, you **`wait`** on the listen port.

```
wait listen
```

This function does not return until a connection has been made.

NOTE: There are other options available for **`wait`** *. For instance, you can wait on multiple ports or for a timeout as well.*

You can now open the connection port from the machine that has contacted your system:

```
connection: first listen
```

This returns the connection that has been made to the listen port. It is a port like all others and can now be used to receive and send data using the **`insert`**, **`copy`**, **`first`**, and other series functions:

```
insert connection "you are connected^/"

while [newline <> char: first connection] [
    print char
]
```

When the communications is complete, the connection should be closed:

```
close connection
```

You are now ready for the next connection on the listen port. You can **wait** again and use **first** again to get the connection.

When you are done with serving, you can close the listen port with:

```
close listen
```

14.3 A Tiny Server

Here is a useful REBOL server that only requires a few lines of code. This server evaluates whatever REBOL code is sent to it. Lines of REBOL are read from the client until an error occurs. Each line must be a complete REBOL expression. They can be of any length but must be a single line.

```
server-port: open/lines tcp://:4321

forever [
    connection-port: first server-port
    until [
        wait connection-port
        error? try [do first connection-port]
    ]
    close connection-port
]
close server-port
```

If an error occurs, the connection is closed and the server waits for the next connection.

Here is an example of a client script that allows you to enter REBOL command lines remotely:

```
server: open/lines tcp://localhost:4321
until [error? try [insert server ask "R> "]]
close server
```

Here the query is used to determine if the connection was been closed due to an error.

14.4 Testing TCP Code

To test your server code, connect from your own machine, rather than requiring both a server and a client. This can be done from two separate REBOL processes or even from the same process.

To connect to your local machine, you can use a line such as:

```
port: open tcp://localhost:8001
```

Here is an example that makes two ports connect to each other in line mode. This is a sort of `echo` port since you're sending data to yourself. It provides a good test of your code and networking:

```
listen: open/lines tcp://:8001
remote: open/lines tcp://localhost:8001
local: first listen
insert local "How are you?"
print first remote  ; response
```

```
close local
close remote
close listen
```

15. UDP - User Datagram Protocol

The User Datagram Protocol is another transport layer protocol that provides a connectionless method of communicating between machines. It allows you to send datagrams, packets, between machines.

The operation of UDP is much different than TCP. UDP is simpler, but it is essentially unreliable. There is no guarantee that a packet will ever reach its destination. In addition, UDP has no flow control. If you send messages too quickly, packets may be lost.

Like TCP, the **wait** function can be used to wait for the next packet to arrive and the **copy** function is used to return the data. If there is no data, **copy** waits until there is. Note, however, that **insert** never waits.

Here is an example of a simple UDP server script:

```
udp: open udp://:9999
wait udp
print copy udp
insert udp "response"
close udp
```

The messages inserted here by the server are sent to the client the server last received a message from. This allows responses to be sent for incoming messages. However, unlike TCP you do not have a continuous connection between the machines. Each packet transfer is a separate exchange.

The client script to communicate with the above server would be:

```
udp: open udp://localhost:9999
insert udp "Test"
wait udp
print copy udp
close udp
```

You should know that the maximum UDP packet size depends on the operating system. 32 KB and 64 KB are common values. In order to send larger amounts of data, you will need to buffer the data, chopping it into smaller pieces. However, careful programming is required to make sure that each piece of the data is received. Remember that with UDP, there are no guarantees.

14 - Ports

1. Overview
2. Opening a Port
    2.1 The Open Function
    2.2 Open Refinements
3. Closing a Port
4. Reading from a Port
5. Writing to a Port
6. Updating a Port
7. Waiting for a Port
8. Other Port Modes
    8.1 Line Mode
    8.2 Read and Write Only
    8.3 Direct Port Access
    8.4 Skipping Data
9. File Permissions
10. Directory Ports

1. Overview

Ports access ***external*** series such as files, networks, consoles, events, databases, data encoders, and data decoders. Port data is processed using the standard REBOL series functions as described in the Series Chapter.

Ports are used for both input and output. The type of data a port handles depends on how the port is opened. Three types of data are possible:

String	a series of bytes, converts line breaks (default)
Binary	a series of bytes, no conversion of the data
Block	a series of REBOL values

A port can be opened in one of two buffering modes:

Buffered	all of the data is held in memory (default)
Direct	data is not held in memory

In addition, a port can be opened with:

Wait	port will wait for data (default)
No-wait	port will not wait for data

2. Opening a Port

2.1 The Open Function

The `open` function initializes access to a port according to specified parameters. The function can be

supplied with a filename, a URL, or an object. In addition, there are several refinements that will affect the open operation or the access to the port's data.

The simplest method of using **open** is to provide it with a filename or URL as its argument. In the example below, a file port is opened:

```
fp: open %file.txt
```

The fp variable refers to the port. If the port did not open, an error will occur. If necessary, the error can be caught with the **try** function.

By default the file is opened as buffered. This means that the file is accessed and modified in memory and changes to the file are not written out until the port is closed or updated.

For files, the **open** function will automatically create the file if it does not already exist.

```
close open %somefile.txt
if exists? %somefile.txt [print "somefile exists"]
somefile exists
```

The **/new** refinement can be used to overwrite an existing file.

```
write %somefile.txt "text in some file"
print read %somefile.txt
text in some file
close insert open/new %somefile.txt "new data"
print read %somefile.txt
new data
```

Once a port is open, the series operations such as **copy**, **insert**, **remove**, **clear**, **first**, **next**, and **length?** can be used to access and change the contents the port.

2.2 Open Refinements

The open function accepts a number of refinements that can be used to modify its operation:

/binary	port data is binary
/string	port data is text, translate all line terminators
/with	specify an alternate line termination
/lines	handle data a line at a time or as a block of lines
/direct	do not buffer the port
/new	create or recreate the target of the port
/read	open for read only operation
/write	open for write only operation
/no-wait	do not wait for data
/skip	skip part of the data
/allow	specify protection attributes of files
/custom	allow special refinements

3. Closing a Port

Access to a port is terminated with the **close** function. All buffered data that has not been saved will be written to the target file. The example below will close a port opened earlier:

```
close fp
```

If you attempt to close a port that is not open, an error will occur.

A port that is closed can be reopened again with the **open** function:

```
open fp
```

4. Reading from a Port

The series **copy** function is used to read data from an open port:

```
print copy fp
I wanted the gold, and I sought it,I scrabbled and mucked like
a slave....
```

This function will wait for the port data. If you don't want to wait for the data, open the port with the /no-wait refinement.

To read only a portion of the port data, use **copy/part:**

```
print copy/part fp 35
I wanted the gold, and I sought it,
```

Note that the second argument to copy can be a length or a position within the port.

You can use the series **find** and **copy** functions to read just part of the port's data:

```
a: find fp "famine"
print copy/part a find a newline
famine or scurvy -- I fought it;
```

The first, next, and other positional series functions can also be used on the port:

```
print first fp
I
print first next next fp
w
```

The **copy** function will return **none** when all data have been read from a port. When running in **/no-wait** mode, the **copy** function will return an empty string if no data is available for the

port.

```
tp: open/direct/binary/no-wait tcp://system:8000
content: make binary! 1000
while [wait tp  data: copy tp] [append content data]
close tp
```

5. Writing to a Port

The **insert** function is used to write to a port.

```
insert fp "I was a fool to seek it."
```

If the port is buffered, the change will occur externally when the port is closed or updated (with the **update** function). If the port is opened with **/direct**, then the change will occur immediately.

All of the **insert** refinements can be used on the port. For example, to write 20 spaces into a port:

```
insert/dup fp " " 20
```

You can also use the **remove**, **clear**, **change**, **append**, **replace**, and other series modifying functions on the port.

For example, to remove a single character or a number of characters:

```
remove fp

remove/part fp 20
```

and to remove all remaining characters, write:

```
clear fp
```

6. Updating a Port

The **update** function forces a port to update its status with respect to the external device. For example, when writing a buffered file, the **update** function can be used to force the data buffer out to the file. When reading, the **update** function can be used to be certain that any pending data has been read into memory.

```
update fp
```

7. Waiting for a Port

The **wait** function is essential to programs that need to handle asynchronous data transfers. With **wait**, you can wait for data on one or more ports, or for a timeout to occur.

The **wait** function will accept a single port:

```
wait port
```

or, an entire block of ports can be provided:

```
wait [port1 port2 port3]
```

In addition, a timeout value can be provided as a number of seconds or as a time value:

```
wait [port1 port2 10]

wait [port1 port2 0:00:05]
```

The first example will time out in ten seconds. The second example will timeout in five seconds.

The **wait** function will return the port that is ready or `none` if the timeout occurred.

```
ready: wait [port1 port2 10]
if ready [data: copy ready]
```

The above example will read data from the first ready port if a timeout did not occur.

To obtain a block of all ports that are ready, use the /all refinement.

```
ready: wait/all [port1 port2 10]
if ready [
    foreach port ready [
        append data copy port
    ]
]
```

This example would append data from all ready ports into a single series.

You can also use the **dispatch** function to evaluate a block or function based on the results of a **wait** on multiple ports.

```
dispatch [
    port1 [print "port1 awake"]
    port2 [print "port2 awake"]
    10 [print "timeout!"]
]
```

Use /No-wait and /Direct

To use **wait** with most ports, you will need to specify the **/no-wait** and **/direct** refinements as part of the open. This indicates that the normal data access functions should not block and that data is not buffered.

```
port1: open/no-wait/direct tcp://system:8000
```

8. Other Port Modes

8.1 Line Mode

The **open** function allows ports to be opened for line access. In line mode, the first function will return a line of text, rather than a character. The example below reads a file one line at a time:

```
fp: open/lines %file.txt
print first fp
I wanted the gold, and I got it --
print third fp
Yet somehow life's not what I thought it,
```

The /lines refinement is also useful for Internet protocols that are line oriented.

```
tp: open/lines tcp://server:8000
print first tp
```

8.2 Read and Write Only

You can use the **/read** refinement to open a port as read only:

```
fp: open/read %file.txt
```

Changes made to the port's buffer, are not written back to the file.

To open for write only, use the **/write** refinement:

```
fp: open/write %file.txt
```

File ports opened with the **/write** refinement will not read the current data upon opening the port.

Closing, or updating a write only file port will cause existing data in the file to be overwritten:

```
insert fp "This is the law of the Yukon..."
close fp
```

```
print read %file.txt
This is the law of the Yukon...
```

8.3 Direct Port Access

The **/direct** refinement opens an unbuffered port. This is useful to access files a portion at a time, such as when a file is too large to be held in memory.

```
fp: open/direct %file.txt
```

Reading the data with a **copy** function will move the port's head forward:

```
print copy/part fp 40
I wanted the gold, and I sought it,^/ I
print copy/part fp 40
scrabbled and mucked like a slave.^/Was i
```

In direct mode, the port will always be at its head position:

```
print head? fp
true
```

The **copy** function will return none when the port has reached its end.

Here is an example that uses direct ports to copy a file of any size:

```
from-port: open/direct %a-file.jpg
to-port: open/direct %a-file.jpg
while [data: copy/part from-port 100000 ][
    append to-port data
]
close from-port
close to-port
```

8.4 Skipping Data

There are two ways to skip data that exists in a port. First, you can open the port with the **/skip** refinement. This **open** function will automatically skip to a point in the port. For example:

```
fp: open/direct/skip %file.big 1000000

fp: open/skip http://www.example.com/bigfile.dat 100000
```

You can also use the **skip** function on the port. For files that are opened with **/direct** and **/binary** the skip operation is identical to a file system seek operation. Data is not read into memory. This is not possible in **/string** mode because the line breaks interfere with the skip size.

```
fp: open/direct/binary %file.dat
fp: skip fp 100000
```

9. File Permissions

When files are created by REBOL, default access permissions are set. On Windows and Macintosh systems files are created with full access privileges. On UNIX systems files are created with the permissions set to the current umask setting.

When using **open** or **write** to access a file the **/allow** refinement is used to set file access permissions.

The **/allow** refinement takes a block as an argument. This block can consist of any or all of the three words **read**, **write** and **execute**.

Operating System Restrictions

The **/allow** refinement will only set permissions on operating systems supporting the specified permission setting. If the operating system does not support a permission setting used, the setting will be ignored. For instance, files on UNIX systems may be set as executable (execute), but the Windows and Macintosh operating systems don't support this. When dealing with UNIX systems, permissions set using **/allow** will only set the user permissions. Using **/** allow will cause all access permissions to be removed for users and others.

To make a file read only, use **open/allow**, or **write/allow** with a read block.

```
write/allow %file.txt [read]
```

To make a file readable and executable:

```
open/allow %file.txt [read execute]
```

You can set similar permissions for write access:

```
write/allow %file.txt [read write]
```

To prevent any access to a file (for operating systems where this would make a difference) provide an empty permissions block:

```
write/allow %file.txt []
```

To permit full access:

```
write/allow %file [read write execute]
```

10. Directory Ports

Directory ports allow you to open direct access to file directories. Within the system, this is how most other directory functions are created.

When you open a directory, you gain direct access to the directory as a block of filenames:

```
mydir: open %intro/
forall mydir [print first mydir]
CVS/
history.t
intro.t
overview.t
quick.t
close mydir
```

You can advance to a specific position within a directory series and remove a file with code such as:

```
dir: open %.
remove next dir
close dir
```

This deletes the second file in the current directory. Similarly,

```
remove at dir 5
```

would delete the fifth file in the directory, and:

```
clear dir
```

would delete all of the files in the directory.

To delete all files that contain with the word "junk", you can write:

```
dir: open %intro/
while [not tail? dir] [
    either find first dir "junk" [remove dir][
        dir: next dir
    ]
]
close dir
```

The changes made to a directory are made when the directory is closed or when it is updated. To force the action to occur immediately use a line such as:

```
update dir
```

The method of directory access can also be used for changing the names of files. After the **open**, the line:

```
change at dir 3 %newname.txt
```

will rename the third file in the directory. Similarly, the names of any of the files in the directory can be changed.

Here is an example that renames all of the files in a directory by adding the word REBOL to their names:

```
dir: open %intro/
forall dir [insert first dir "REBOL"]
close dir
```

15 - Parsing

1. Overview
2. Simple Splitting
3. Grammar Rules
4. Skipping Input
5. Match Types
6. Recursive Rules
7. Evaluation
    7.1 Return Value
    7.2 Expressions in Rules
    7.3 Copying the Input
    7.4 Marking the Input
    7.5 Modifying the String
    7.6 Using Objects
    7.7 Debugging
8. Dealing with Spaces
9. Parsing Blocks and Dialects
    9.1 Matching Words
    9.2 Matching Datatypes
    9.3 Characters Not Allowed
    9.4 Dialect Examples
    9.5 Parsing Sub-blocks
10. Summary of Parse Operations
    10.1 General Forms
    10.2 Specifying Quantity
    10.3 Skipping Values
    10.4 Getting Values
    10.5 Using Words
    10.6 Value Matches (block parsing only)
    10.7 Datatype Words

1. Overview

Parsing splits a sequence of characters or values into smaller parts. It can be used for recognizing characters or values that occur in a specific order. In addition to providing a powerful, readable, and maintainable approach to regular expression pattern matching, parsing enables you to create your own custom languages for specific purposes.

The **parse** function has the general form:

```
parse series rules
```

The series argument is the input that is parsed and can be a string or a block. If the argument is a string, it is parsed by character. If the argument is a block, it is parsed by value.

The rules argument specifies how the series argument is parsed. The rules argument can be a string for simple types of parsing or a block for sophisticated parsing.

The **parse** function also accepts two refinements: **/all** and **/case**. The **/all** refinement parses all the characters within a string, including all delimiters, such as space, tab, newline,

comma, and semicolon. The **/case** refinement parses a string based on case. When **/case** is not specified, upper and lower cases are treated the same.

2. Simple Splitting

A simple form of **parse** is for splitting strings:

```
parse string none
```

The **parse** function splits the input argument, `string`, into a block of multiple strings, breaking each string wherever it encounters a delimiter, such as a space, tab, newline, comma, or semicolon. The `none` argument indicates that no other delimiters other than these. For example:

```
probe parse "The trip will take 21 days" none
["The" "trip" "will" "take" "21" "days"]
```

Similarly,

```
probe parse "here there,everywhere; ok" none
["here" "there" "everywhere" "ok"]
```

In the example above, notice that the commas and semicolons have been removed from the resulting strings.

You can specify your own delimiters in the second argument to **parse**. For example, the following code parses a telephone number with dash (-) delimiters:

```
probe parse "707-467-8000" "-"
["707" "467" "8000"]
```

The next example uses equal (=) and double quote (") as the delimiters:

```
probe parse <IMG SRC="test.gif" WIDTH="123"> {="}
["IMG" "SRC" "test.gif" "WIDTH" "123"]
```

The next example parses a string based on commas only; any other delimiters are ignored. Consequently, the spaces within the strings are not removed:

Normally when you parse strings, any whitespace (space, tab, lines) are automatically processed as delimiters. To avoid that action, you can use the `/any` refinement. Compare these two examples:

```
parse "Test This" ""
["Test" "This"]
parse/all "Test This" ""
["Test This"]
```

In the second, you can see that the space was not treated as a delimiter.

Here is another example:

```
probe parse/all "Harry, 1011 Main St., Ukiah" ","
["Harry" " 1011 Main St." " Ukiah"]
```

You can also parse strings that contain null characters as separators (such as certain types of data files):

```
parse/all nulled-string "^(null)"
```

3. Grammar Rules

The **parse** function accepts grammar rules that are written in a ***dialect*** of REBOL. Dialects are sub-languages of REBOL that use the same lexical form for all datatypes, but allow a different ordering of the values within a block. Within this dialect the grammar and vocabulary of REBOL is altered to make it similar in structure to the well known BNF (Backus-Naur Form) which is commonly used to specify language grammars, network protocols, header formats, etc.

To define rules, use a block to specify the sequence of the inputs. For instance, if you want to parse a string and return the characters "the phone", you can use a rule:

```
parse string ["the phone"]
```

To allow any number of spaces or no spaces between the words, write the rule like this:

```
parse string ["the" "phone"]
```

You can indicate alternate rules with a vertical bar (|). For example:

```
["the" "phone" | "a" "radio"]
```

accepts strings that match any of the following:

```
the phone
a radio
```

A rule can contain blocks that are treated as sub-rules. The following line:

```
[ ["a" | "the"] ["phone" | "radio"] ]
```

accepts strings that match any of the following:

```
a phone
a radio
the phone
the radio
```

For increased readability, write the sub-rules as a separate block and give them a name to help indicate their purpose:

```
article: ["a" | "the"]
device: ["phone" | "radio"]
parse string [article device]
```

In addition to matching a single instance of a string, you can provide a count or a range that repeats the match. The following example provides a count:

```
[3 "a" 2 "b"]
```

which accepts strings that match:

```
aaabb
```

The next example provides a range:

```
[1 3 "a" "b"]
```

which accepts strings that match any of the following:

```
ab aab aaab
```

The starting point of a range can be zero, meaning that it is optional.

```
[0 3 "a" "b"]
```

accepts strings that match any of the following:

```
b ab aab aaab
```

Use **some** to specify that one or more characters are matched. Use **any** to specify that zero or more characters are matched. For example, **some** used in the following line:

```
[some "a" "b"]
```

accepts strings that contain one or more characters `a` and `b`:

```
ab aab aaab aaaab
```

The next example uses **`any`**:

```
[any "a" "b"]
```

which accepts strings that contain zero or more characters `a` or `b`:

```
b ab aab aaab aaaab
```

The words **`some`** and **`any`** can also be used on blocks. For example:

```
[some ["a" | "b"]]
```

accepts strings that contain any combination of the characters `a` and `b`.

Another way to express that a character is optional is to provide an alternate choice of `none`:

```
["a" | "b" | none]
```

This example accepts strings that contain `a` or `b` or `none`.

The `none` is useful for specifying optional patterns or for catching error cases when no pattern matches.

4. Skipping Input

The **`skip`**, **`to`**, and **`thru`** words allow input to be skipped.

Use **`skip`** to skip a single character, or use it with a **`repeat`** to skip over multiple characters:

```
["a" skip "b"]
["a" 10 skip "b"]
["a" 1 10 skip]
```

To skip until a specific character is found, use **`to`**:

```
["a" to "b"]
```

The previous example starts parsing at `a` and ends at `b` but does not include `b`.

To include the ending character, use **thru:**

```
["a" thru "b"]
```

The previous example starts parsing at a, ends at b, and includes b.

The following rule finds the title of an HTML page and prints it:

```
page: read http://www.rebol.com/
parse page [thru <title> copy text to </title>]
print text
REBOL Technologies
```

The first **thru** finds the title tag and goes immediately past it. Next, the input string is copied into a variable called text until the ending tag is found (but it doesn't go past it, or the text would include the tag).

5. Match Types

When parsing strings, these datatypes and words can be used to match characters in the input string:

Match Type	Description
"abc"	match the entire string
#"c"	match a single character
tag	match a tag string
end	match to the end of the input
(bitset)	match any specified char in the set

To use all of these words (except bitset, which is explained below) in a single rule, use:

```
[<B> ["excellent" | "incredible"] #"!" </B> end]
```

This example parses the input strings:

```
<B>excellent!</B>
<B>incredible!</B>
```

The **end** specifies that nothing follows in the input stream. The entire input has been parsed. It is optional depending on whether the **parse** function's return value is to be checked. Refer to the Evaluation section below for more information.

The **bitset** datatype deserves more explanation. Bitsets are used to specify collections of characters in an efficient manner. The **charset** function enables you to specify individual characters or ranges of characters. For example, the line:

```
digit: charset "0123456789"
```

defines a character set that contains digits. This allows rules like:

```
[3 digit "-" 3 digit "-" 4 digit]
```

which can parse phone numbers of the form:

```
707-467-8000
```

To accept any number of digits, it is common to write the rule:

```
digits: [some digit]
```

A character set can also specify ranges of characters. For instance, the `digit` character set could have be written as:

```
digit: charset [#"0" - #"9"]
```

Alternatively, you can combine specific characters and ranges of characters:

```
the-set: charset ["+-." #"0" - #"9"]
```

To expand on this, here is the alphanumeric set of characters:

```
alphanum: charset [#"0" - #"9" #"A" - #"Z" #"a" - #"z"]
```

Character sets can also be modified with the **`insert`** and **`remove`** functions, or combinations of sets can be created with the **`union`** and **`intersect`** functions. This line copies the digit character set and adds a dot to it:

```
digit-dot: insert copy digit "."
```

The following lines define useful character sets for parsing:

```
digit: charset [#"0" - #"9"]
alpha: charset [#"A" - #"Z" #"a" - #"z"]
alphanum: union alpha digit
```

6. Recursive Rules

Here is an example of rule set that parses mathematical expressions and gives a precedence (a priority) to the math operators used:

```
expr:     [term ["+" | "-"] expr | term]
term:     [factor ["*" | "/"] term | factor]
factor:   [primary "**" factor | primary]
primary:  [some digit | "(" expr ")"]
digit:    charset "0123456789"
```

Now we can parse many types of math expressions. The following examples return `true`, indicating that the expressions were valid:

```
probe parse "1 + 2 * ( 3 - 2 ) / 4" expr
true
probe parse "4/5+3**2-(5*6+1)" expr
true
```

Notice in the examples that some of the rules refer to themselves. For instance, the `expr` rule includes `expr`. This is a useful technique for defining repeating sequences and combinations. The rule is ***recursive*** --it refers to itself.

When using recursive rules, care is required to prevent endless recursion. For instance:

```
expr: [expr ["+" | "-"] term]
```

creates an infinite loop because the first thing `expr` does is use `expr` again.

7. Evaluation

Normally, you parse a string to produce some result. You want to do more than just verify that the string is valid, you want to do something as it is parsed. For instance, you may want to pick out substrings from various parts of the string, create blocks of related values, or compute a value.

7.1 Return Value

The examples in previous chapters showed how to parse strings, but no results were produced. This is only done to verify that a string has the specified grammar; the value returned from **`parse`** indicates its success. The following examples show this:

```
probe parse "a b c" ["a" "b" "c"]
true
probe parse "a b" ["a" "c"]
false
```

The **`parse`** function returns `true` only if it reaches the end of the input string. An unsuccessful match stops the parse of the series. If **`parse`** runs out of values to search for before reaching the end of the series, it does not traverse the series and returns `false` **:**

```
probe parse "a b c d" ["a" "b" "c"]
false
```

```
probe parse "a b c d" [to "b" thru "d"]
true
probe parse "a b c d" [to "b" to end]
true
```

7.2 Expressions in Rules

Within a rule, you can include a REBOL expression to be evaluated when **parse** reaches that point in the rule. Parentheses are used to indicate such expressions:

```
string: "there is a phone in this sentence"
probe parse string [
    to "a"
    to "phone" (print "found phone")
    to end
]
found phone
true
```

The example above parses the string `a phone` and prints the message `found phone` after the match is complete. If the strings `a` or `phone` are missing and the parse can not be done, the expression is not evaluated.

Expressions can appear anywhere within a rule, and multiple expressions can occur in different parts of a rule. For instance, the following code prints different strings depending on what inputs were found:

```
parse string [
    "a" | "the"
    to "phone" (print "answer") |
    to "radio" (print "listen") |
    to "tv"    (print "watch")
]
answer
string: "there is the radio on the shelf"

parse string [
    "a" | "the"
    to "phone" (print "answer") |
    to "radio" (print "listen") |
    to "tv"    (print "watch")
]
listen
```

Here is an example that counts the number of times the HTML pre-format tag appears in a text string:

```
count: 0
page: read http://www.rebol.com/docs/dictionary.html
parse page [any [thru <pre> (count: count + 1)]]
print count
777
```

7.3 Copying the Input

The most common action done with **parse** is to pick up parts of the string being parsed. This is done with **copy**, and it is followed by the name of a variable to which you want to copy the string. The following example parses the title of a web page:

```
parse page [thru <title> copy text to </title>]
print text
REBOL/Core Dictionary
```

The example works by skipping over text until it finds the <title> tag. That's where it starts making a copy of the input stream and setting a variable called `text` to hold it. The copy operation continues until the closing <title> tag is found.

The copy action also can be used with entire rule blocks. For instance, for the rule:

```
[copy heading ["H" ["1" | "2" | "3"]]
```

the heading string contains the entire `H1`, `H2`, or `H3` string. This also works for large multi-block rules.

7.4 Marking the Input

The **copy** action makes a copy of the substring that it finds, but that is not always desirable. In some cases, it is better to save the current position of the input stream in a variable.

NOTE: The **copy** word as used in parse is different from the **copy** function used in REBOL expressions. Parse uses a dialect of REBOL, and **copy** has a different meaning within that dialect.

In the following example, the `begin` variable holds a reference to the `page` input string just after <title>. The `ending` refers to the `page` string just before . These variables can be used in the same way as they would be used with any other series.

```
parse page [
    thru <title> begin: to </title> ending:
    (change/part begin "Word Reference Guide" ending)
]
```

You can see the above parse expression actually changed the contents of the title:

```
parse page [thru <title> copy text to </title>]
print text
Word Reference Guide
```

Here is another example that marks the position of every table tag in an HTML file:

```
page: read http://www.rebol.com/index.html
tables: make block! 20
parse page [
    any [to "<table" mark: thru ">"
        (append tables index? mark)
    ]
```

```
]
```

The `tables` block now contains the position of every tag:

```
foreach table tables [
    print ["table found at index:" table]
]
table found at index: 836
table found at index: 2076
table found at index: 3747
table found at index: 3815
table found at index: 4027
table found at index: 4415
table found at index: 6050
table found at index: 6556
table found at index: 7229
table found at index: 8268
```

NOTE: The current position in the input string can also be modified. The next section explains how this is done.

7.5 Modifying the String

Now that you know how to obtain the position of the input series, you also can use other series functions on it, including **`insert`**, **`remove`**, and **`change`**. To write a script that replaces all question marks (?) with exclamation marks (!), write:

```
str: "Where is the turkey? Have you seen the turkey?"
parse str [some [to "?" mark: (change mark "!") skip]]
print str
Where is the turkey! Have you seen the turkey!
```

The **`skip`** at the tail advances the input over the new character, which is not necessary in this case, but it is a good practice.

As another example, to insert the current time everywhere the word `time` appears in some text, write:

```
str: "at this time, I'd like to see the time change"
parse str [
    some [to "time"
        mark:
        (remove/part mark 4  mark: insert mark now/time)
        :mark
    ]
]
print str
at this 14:42:12, I'd like to see the 14:42:12 change
```

Notice the `:mark` word used above. It sets the input to a new position. The **`insert`** function returns the new position just past the insert of the current time. The word `:mark` is used to set the input to that position.

7.6 Using Objects

When parsing large grammar from a set of rules, variables are used to make the grammar more readable. However, the variables are global and may become confused with other variables that have the same name somewhere else in the program.

The solution to this problem is to use an object to make all the rule words local to a context. For instance:

```
tag-parser: make object! [
    tags: make block! 100
    text: make string! 8000
    html-code: [
        copy tag ["<" thru ">"] (append tags tag) |
        copy txt to "<" (append text txt)
    ]
    parse-tags: func [site [url!]] [
        clear tags clear text
        parse read site [to "<" some html-code]
        foreach tag tags [print tag]
        print text
    ]
]
tag-parser/parse-tags http://www.rebol.com
```

7.7 Debugging

As rules are written, there are times debugging is needed. Specifically, you may want to know how far you got in the parsing of a rule.

The `trace` function can be used to watch the parse operation progress, but this can output thousands of lines that are difficult to review.

A better way is to insert debugging expressions into the parse rules. As an example, to debug the rule:

```
[to "<IMG" "SRC" "=" filename ">"]
```

insert a the `print` function after key sections to monitor your progress through the rule:

```
[to "<IMG" (print 1) "SRC" "=" (print 2)
    filename (print 3) ">"]
```

This example prints `1`, `2`, and `3` as the rule is processed.

Another approach is to print out part of the input string as the parse happens:

```
[
   to "<IMG" here: (print here)
   "SRC" "=" here: (print here)
    filename here: (print here) ">"
]
```

If this is done often, you can create a rule for it:

```
here: [where: (print where)]

[
   to "<IMG" here
   "SRC" "=" here
    filename here ">"
]
```

The **copy** function can also be used to indicate what substrings were parsed as the rule was handled.

8. Dealing with Spaces

The **parse** function normally ignores all intervening whitespace between patterns that it scans. For instance, the rule:

```
["a" "b" "c"]
```

returns strings that match:

```
abc
a bc
ab c
a b c
a  b  c
```

and other similarly spaced combinations.

To enforce a specific spacing convention, use **parse** with the **/all** refinement. In the preceeding example, this refinement causes **parse** to only match the first case (abc).

```
parse/all "abc" ["a" "b" "c"]
```

Specifying the / all refinement forces every character in the input stream to be dealt with, including the default delimiters, such as space, tab, newline.

To handle spaces in your rules, create a character set that specifies the valid space characters:

```
spacer: charset reduce [tab newline #" "]
```

If you want a single space character between each letter write:

```
["a" spacer "b" spacer "c"]
```

To allow multiple space characters, write:

```
spaces: [some spacer]
["a" spaces "b" spaces "c"]
```

For more sophisticated grammars, create a character set that lets you scan a string up to a space character.

```
non-space: complement spacer
to-space: [some non-space | end]
words: make block! 20
parse/all text [
    some [copy word to-space (append words word) spacer]
]
```

The preceding example builds a block of all of its words. The `complement` function inverts the character set. Now it contains everything ***except*** the spacing characters you defined earlier. The `non-space` character set contains all characters except space characters. The `to-space` rule accepts one or more characters up to a space character or the end of the input stream. The main rule expects to begin with a word, copy that word up to a space, then skip the space character and begin the next word.

9. Parsing Blocks and Dialects

Blocks are parsed similar to strings. A set of rules specify the order of expected values. However, unlike the parsing of strings, the parsing of blocks is not concerned with characters or delimiters. Parsing of blocks is done at the value level, making the grammar rules easier to specify and operation many times faster.

Block parsing is the easiest way to create REBOL ***dialects***. Dialects are sub-languages of REBOL that use the same lexical form for all data types but allow a different ordering of the values within a block. The values do not need to conform to the normal order required by REBOL function arguments. Dialects are able to provide greater expressive power for specific domains of use. For instance, the parser rules themselves are specified as a dialect.

9.1 Matching Words

When parsing a block, to match against a word specify the word as a literal:

```
'name
'when
'empty
```

9.2 Matching Datatypes

You can match a value of any datatype by specifying the data type word. See Datatype Matches below.

Datatype Word	Description
string!	matches any quoted string
time!	matches any time
date!	matches any date
tuple!	matches any tuple

NOTE: Don't forget the "!" that is part of the name or an error will be generated.

9.3 Characters Not Allowed

The **parse** operations allowed for blocks are those that deal with specific characters. For instance, a match cannot be specified to the first letter of a word or string, nor to spacing or newline characters.

9.4 Dialect Examples

A few concise examples help illustrate the parsing of blocks:

```
block: [when 10:30]
print parse block ['when 10:30]
print parse block ['when time!]
parse block ['when set time time! (print time)]
```

Notice that a specific word can be matched by using its literal word in the rule (as in the case of `'when`). A datatype can be specified rather than a value, as in the lines above containing **time!**. In addition, a variable can be set to a value with the **set** operation.

As with strings, alternate rules can be specified when parsing blocks:

```
rule: [some [
    'when set time time! |
    'where set place string! |
    'who set persons [word! | block!]
]]
```

These rules allow information to be entered in any order:

```
parse [
    who Fred
    where "Downtown Center"
    when 9:30
] rule
print [time place persons]
```

This example could have used variable assignment, but it illustrates how to provide alternate input ordering.

Here's another example that evaluates the results of the parse:

```
rule: [
    set count integer!
    set str string!
    (loop count [print str])
]
parse [3 "great job"] rule
parse [3 "hut" 1 "hike"] [some rule]
```

Finally, here is a more advanced example:

```
rule: [
    set action ['buy | 'sell]
    set number integer!
    'shares 'at
    set price money!
    (either action = 'sell [
            print ["income" price * number]
            total: total + (price * number)
        ][
            print ["cost" price * number]
            total: total - (price * number)
        ]
    )
]

total: 0
parse [sell 100 shares at $123.45] rule
print ["total:" total]

total: 0
parse [
    sell 300 shares at $89.08
    buy  100 shares at $120.45
    sell 400 shares at $270.89
] [some rule]
print ["total:" total]
```

It should be noted that this is one way how expressions that use the ***dialect*** concept first described in Chapter 4 can be ***evaluated***.

9.5 Parsing Sub-blocks

When parsing a block, if a sub-block is found, it is treated as a single value that is of the **`block!`** datatype. However, to parse a sub-block, you must invoke the parser recursively on the sub-block. The **`into`** word provides this capability. It expects that the next value in the input block is a sub-block to be parsed. This is as if a **`block!`** datatype had been provided. If the next value is not a **`block!`** datatype, the match fails and **`into`** looks for alternates or exits the rule. If the next value is a block, the parser rule that follows the **`into`** word is used to begin parsing the sub-block. It is processed in the same way as a sub-rule.

```
rule: [date! into [string! time!]]
data: [10-Jan-2000 ["Ukiah" 10:30]]
print parse data rule
```

All of the normal parser operations can be applied to **`into`**.

```
rule: [
    set date date!
    set info into [string! time!]]
]
data: [10-Jan-2000 ["Ukiah" 10:30]]
print parse data rule

print info

rule: [date! copy items 2 into [string! time!]]
data: [10-Jan-2000 ["Ukiah" 10:30] ["Rome" 2:45]]
print parse data rule
```

```
probe items
```

10. Summary of Parse Operations

10.1 General Forms

Operator	Description
|	alternate rule
[block]	sub-rule
(paren)	evaluate a REBOL expression

10.2 Specifying Quantity

Operator	Description
none	match nothing
opt	zero or one time
some	one or more times
any	zero or more times
12	repeat pattern 12 times
1 12	repeat pattern 1 to 12 times
0 12	repeat pattern 0 to 12 times

10.3 Skipping Values

Operator	Description
skip	skip a value (or multiple if repeat given)
to	advance input to a value or datatype
thru	advance input thru a value or datatype

10.4 Getting Values

Operator	Description
set	set the next value to a variable
copy	copy the next match sequence to a variable

10.5 Using Words

Operator	Description
word	look-up value of a word
word:	mark the current input series position
:word	set the current input series position
'word	matches the word literally (parse block)

10.6 Value Matches (block parsing only)

Operator	Description
"fred"	matches the string "fred"
%data	matches the file name %data
10:30	matches the time 10:30
1.2.3	matches the tuple 1.2.3

10.7 Datatype Words

Word	Description
type!	matches anything of a given datatype

Values

1. Number Values
    1.1 Decimal
    1.2 Integer
2. Series Values
    2.1 Binary
    2.2 Block
    2.3 Email
    2.4 File
    2.5 Hash
    2.6 Image
    2.7 Issue
    2.8 List
    2.9 Paren
    2.10 Path
    2.11 String
    2.12 Tag
    2.13 URL
3. Other Values
    3.1 Character
    3.2 Date
    3.3 Logic
    3.4 Money
    3.5 None
    3.6 Pair
    3.7 Refinement
    3.8 Time
    3.9 Tuple
    3.10 Words

1. Number Values

1.1 Decimal

1.1.1 Concept

The **decimal!** datatype is based on 64-bit standard IEEE floating point numbers. They are distinguished from integer numbers by a decimal point (a period or a comma is allowed for international usage, see the notes below).

1.1.2 Format

Decimal values are a sequence of numeric digits, followed by a decimal point, which can be a period (.) or a comma (,), followed by more digits. A plus (+) or minus (-) immediately before the first digit indicates sign. Leading zeros before the decimal point are ignored. Extra spaces, commas, and periods are not allowed.

```
1.23
123.
123.0
0.321
0.123
```

```
1234.5678
```

A comma can be used in place of a period to represent the decimal point (which is the standard in many countries):

```
1,23
0,321
1234,5678
```

Use a single quote (`) to separate the digits in long decimals. Single quotes can appear anywhere after the first digit in the number, but not before the first digit.

```
100'234'562.3782
100'234'562,3782
```

Do not use commas or periods separate the digits in a decimal value.

Scientific notation can be used to specify the exponent of a number by appending the number with the letter **E** or **e** followed by a sequence of digits. The exponent can be a positive or negative number.

```
1.23E10
1.2e007
123.45e-42
56,72E300
-0,34e-12
0.0001e-001
```

Decimal numbers span from 2.2250738585072e-308 up to 1.7976931348623e+308 and can contain up to 15 digits of precision.

1.1.3 Creation

Use the **to-decimal** function to convert **string!**, **integer!**, **block!**, or **decimal!** datatypes to a decimal number:

```
probe to-decimal "123.45"
123.45
probe to-decimal 123
123
probe to-decimal [-123 45]
-1.23E+47
probe to-decimal [123 -45]
1.23E-43
probe to-decimal -123.8
-123.8
probe to-decimal 12.3
12.3
```

If a decimal and integer are combined in an expression, the integer is converted to a decimal number:

```
probe 1.2 + 2
3.2
probe 2 + 1.2
3.2
probe 1.01 > 1
true
probe 1 > 1.01
false
```

1.1.4 Related

Use **decimal?** to determine whether a value is an **decimal!** datatype.

```
print decimal? 0.123
true
```

Use the **form**, **print**, and **mold** functions with an integer argument to print a decimal value in its simplest form:

- integer. If it can be represented as one.
- decimal without exponent. If it's not too big or too small.
- scientific notation. If it's too big or small.

For example,

```
probe mold 123.4
123.4
probe form 2222222222222222
2.22222222222222E+15
print 1.00001E+5
100001
```

Single quotes (`) and a leading plus sign (+) do not appear in decimal output:

```
print +1'100'200.222'112
1100200.222112
```

1.2 Integer

1.2.1 Concept

The **integer!** datatype includes 32-bit positive and negative numbers and zero. Unlike decimal numbers, integers do not contain a decimal point.

1.2.2 Format

Integer values consist of a sequence of numeric digits. A plus (+) or minus (-) immediately before the first digit indicates sign. (There cannot be a space between the sign and the first digit.) Leading zeros are ignored.

```
0 1234 +1234 -1234 00012 -0123
```

Do not use commas or periods in integers. If a comma or period is found within an integer it is interpreted as a decimal value. However, you can use a single quote (`) to separate the digits in long integers. Single quotes can appear anywhere after the first digit in the number, but not before the first digit.

```
2'147'483'647
```

Integers span a range from -2147483648 to 2147483647.

1.2.3 Creation

Use the **to-integer** function to convert a **string!**, **logic!**, **decimal!**, or **integer!** datatype to an integer:

```
probe to-integer "123"
123
probe to-integer false
0
probe to-integer true
1
probe to-integer 123.4
123
probe to-integer 123.8
123
probe to-integer -123.8
-123
```

If a decimal and integer are combined in an expression, the integer is converted to a decimal:

```
probe 1.2 + 2
3.2
probe 2 + 1.2
3.2
probe 1.01 > 1
true
probe 0 < .001
true
```

1.2.4 Related

Use **integer?** to determine whether a value is an **integer!** datatype.

```
probe integer? -1234
true
```

Use the **form**, **print**, and **mold** functions with an integer argument to print a integer value as a string:

```
probe mold 123
123
probe form 123
123
print 123
123
```

Integers that are out of range or cannot be represented in 32 bits are flagged as an error.

2. Series Values

2.1 Binary

2.1.1 Concept

Binary values hold binary data of any arbitrary type. Any sequence of bytes can be stored, such as an image, audio, executable file, compressed data, and encrypted data. The source format for binary data can be base-2 (binary), base-16 (hex), and base-64. The default base for binary data in REBOL is base-16.

2.1.2 Format

Binary strings are written as a number sign (#) followed by a string enclosed in braces. The characters within the string are encoded in one of several formats as specified by an optional number prior to the number sign. Base-16 is the default format.

```
#{3A18427F 899AEFD8} ; default base-16
2#{10010110110010101001011011001011} ; base-2
64#{LmNvbSA8yw9CB0aGvXmgUkVCu2Uz934b} ; base-64
```

Spaces, tabs and newlines are permitted within the string. Binary data can span multiple lines.

```
probe #{
    3A
    18
    92
    56
}
#{3A189256}
```

Strings that are missing the correct number of characters to create a correct binary result are padded on the right.

2.1.3 Creation

The **to-binary** function converts data to the **binary!** datatype at the default base set in **system/options/binary-base:**

```
probe to-binary "123"
#{313233}
probe to-binary "today is the day..."
#{746F6461792069732074686520646179 2E2E2E}
```

To convert an integer into its binary value, pass it in a block:

```
probe to-binary [1]
#{01}
probe to-binary [11]
#{0B}
```

Converting a series of integers into a binary, returns the bit conversion for each integer concatenated into a single binary value:

```
probe to-binary [1 1 1 1]
#{01010101}
```

2.1.4 Related

Use **binary?** determine whether a value is an **binary!** datatype.

```
probe binary? #{616263}
true
```

Binary values are a type of series:

```
probe series? #{616263}
true
probe length? #{616263} ; three hex values in this binary
3
```

Closely related to working with **binary!** datatypes are the functions **enbase** and **debase**. The **enbase** function converts strings to their base-2, base-16 or base-64 representations as strings. The **debase** function converts enbased strings to a binary value of the base specified in **system/options/binary-base**.

2.2 Block

2.2.1 Concept

Blocks are groups of values and words. Blocks are used everywhere, from a script itself to blocks of data and code provided in a script.

Block values are indicated by opening and closing square brackets ([ ]) with any amount of data contained between them.

```
[123 data "hi"]   ; block with data
[]                ; empty block
```

Blocks can hold records of information:

```
woodsmen: [
    "Paul" "Bunyan" paul@bunyan.dom
```

```
    "Grizzly" "Adams" grizzly@adams.dom
    "Davy" "Crocket" davy@crocket.dom
]
```

Blocks can contain code:

```
[print "this is a segment of code"]
```

Blocks are also a type of series, and thus anything that can be done with a series can be done with a block value.

Blocks can be searched:

```
probe copy/part (find woodsmen "Grizzly") 3
[
    "Grizzly" "Adams" grizzly@adams.dom]
```

Blocks can be modified:

```
append woodsmen [
    "John" "Muir" john@muir.dom
]
probe woodsmen
[
    "Paul" "Bunyan" paul@bunyan.dom
    "Grizzly" "Adams" grizzly@adams.dom
    "Davy" "Crocket" davy@crocket.dom
    "John" "Muir" john@muir.dom
]
```

Blocks can be evaluated:

```
blk: [print "data in a block"]
do blk
data in a block
```

Blocks can contain blocks:

```
blks: [
    [print "block one"]
    [print "block two"]
    [print "block three"]
]
foreach blk blks [do blk]
block one
block two
block three
```

2.2.2 Format

Blocks can contain any number of values or no values at all. They can extend over multiple lines and can include any type of value, including other blocks.

An empty block:

```
[ ]
```

A block of integers:

```
[24 37 108]
```

A REBOL header:

```
REBOL [
    Title: "Test Script"
    Date: 31-Dec-1998
    Author: "Ima User"
]
```

The condition and evaluation block of a function:

```
while [time < 10:00] [
    print time
    time: time + 0:10
]
```

Words in a block need not be defined:

```
blk: [undefined words in a block]
probe value? pick blk 1
false
```

Blocks allow any number of lines, spaces, or tabs. Lines and spaces can be placed anywhere within the block, so long as they do not divide a single value.

2.2.3 Creation

The **to-block** function converts data to the **block!** datatype:

```
probe to-block luke@rebol.com
[luke@rebol.com]
probe to-block {123 10:30 "string" luke@rebol.com}
[123 10:30 "string" luke@rebol.com]
```

2.2.4 Related

Use **block?** to determine whether a value is an **block!** datatype.

```
probe block? [123 10:30]
true
```

As blocks are a subset of the **series!** pseudotype, use **series?** to check this:

```
probe series? [123 10:30]
true
```

Using **form** on a block value creates a string from the contents contained in the block:

```
probe form [123 10:30]
123 10:30
```

Using **mold** on a block value creates a string from the block value and it's contents, thus allowing it to be reloaded as a REBOL block value:

```
probe mold [123 10:30]
[123 10:30]
```

Closely related datatypes are **hash!** and **list!**. They are used in much the same way as block values, but have special capabilities. List values are designed to handle modification of lists more quickly than block values, and hash values are designed handle data lookup and hash indexing of data. These are useful when dealing with large data sets.

2.3 Email

2.3.1 Concept

An email address is a datatype. The **email!** datatype allows for easy expression of email addresses:

```
send luke@rebol.com {some message}

emails: [
    john@keats.dom
    lord@byron.dom
    edger@guest.dom
    alfred@tennyson.dom
]
mesg: {poetry reading at 8:00pm!}
foreach email emails [send email mesg]
```

Email is also one of the **series!** datatypes, so the same rules that apply to series apply to emails:

```
probe head change/part jane@doe.dom "john" 4
john@doe.dom
```

2.3.2 Format

The standard format of an email address is a name, followed by an at sign (@), followed by a domain. An email address can be of any length, but must not include any of restricted characters, such as square brackets, quotes, braces, spaces, newlines, etc..

The following **email!** datatype formats are valid:

```
info@rebol.com
123@number-mail.org
my-name.here@an.example-domain.com
```

Upper and lower cases are preserved in email addresses.

2.3.3 Access

Refinements can be used with an email value to get the user name or domain. The refinements are:

- /user - Get the user name.
- /host - Get the domain.

Here's how these refinements work:

```
email: luke@rebol.com
probe email/user
luke
probe email/host
rebol.com
```

2.3.4 Creation

The **to-email** function converts data to the **email!** datatype:

```
probe to-email "info@rebol.com"
info@rebol.com
probe to-email [info rebol.com]
info@rebol.com
probe to-email [info rebol com]
info@rebol.com
probe to-email [user some long domain name out there dom]
user@some.long.domain.name.out.there.dom
```

2.3.5 Related

Use **email?** to determine whether a value is an **email!** datatype.

```
probe email? luke@rebol.com
true
```

As emails are a subset of the **series!** pseudotype, use **series?** to determine whether the value is a series:

```
probe series? luke@rebol.com
true
probe pick luke@rebol.com 5
#"@"
```

2.4 File

2.4.1 Concept

The **file!** datatype can be a file name, directory name, or directory path.

```
%file.txt
%directory/
%directory/path/to/some/file.txt
```

File values are a subset of series, and thus can be manipulated as a series:

```
probe find %dir/path1/path2/file.txt "path2"
%path2/file.txt
f: %dir/path/file.txt
probe head remove/part (find f "path/") (length? "path/")
%dir/file.txt
```

2.4.2 Format

Files are designated with a percent sign (%)followed by a sequence of characters:

```
load %image.jpg
prog: load %examples.r
save %this-file.txt "This file has few words."
files: load %../programs/
```

Unusual characters in file names must be encoded with a % hexadecimal number, which is an Internet convention. A file name with a space (hexadecimal 20) would look like:

```
probe %cool%20movie%20clip.mpg
%cool%20movie%20clip.mpg
print %cool%20movie%20clip.mpg
cool movie clip.mpg
```

Another format is to enclose the file name in quotes:

```
probe %"cool movie clip.mpg"
%cool%20movie%20clip.mpg
print %"cool movie clip.mpg"
cool movie clip.mpg
```

The standard character for separating directories in a path is the forward slash (/), not the

backslash (\). However, the REBOL language automatically converts backslashes found in file names to forward slashes:

```
probe %\some\path\to\some\where\movieclip.mpg
%/some/path/to/some/where/movieclip.mpg
```

2.4.3 Creation

The **to-file** function converts data to the **file!** datatype:

```
probe to-file "testfile"
%testfile
```

When passed a block, elements in the block are concatenated into a file path with the final element used as the file name:

```
probe to-file [some path to a file the-file.txt]
%some/path/to/a/file/the-file.txt
```

2.4.4 Related

Use **file?** to determine whether a value is an **file!** datatype.

```
probe file? %rebol.r
true
```

As files are a subset of the **series!** pseudotype, use **series?** to check this:

```
probe series? %rebol.r
true
```

2.5 Hash

2.5.1 Concept

Hash is a block that is specially organized to make finding data faster. When searching is performed on a hash block, the search is performed by using a hash table for lookup. For large blocks, this can speed searches by hundreds of times.

2.5.2 Format

Hash blocks must be constructed by using **make** or **to-hash**. They have no lexical format.

2.5.3 Creation

Use **make** to initialize a hash block:

```
hsh: make hash! 10 ; allocating space for 10 elements
```

The **to-hash** function converts data to the hash datatype.

Convert a block:

```
blk: [1 "one" 2 "two" 3 "three" 4 "four"]
probe hash: to-hash blk
make hash! [1 "one" 2 "two" 3 "three" 4 "four"]
print select hash 2
two
```

Convert various values:

```
probe to-hash luke@rebol.com

probe to-hash 123.5

probe to-hash {123 10:30 "string" luke@rebol.com}
```

2.5.4 Related

Use **hash?** to test the datatype.

```
hsh: to-hash [1 "one" 2 "two" 3 "three" 4 "four"]
probe hash? Hsh
true
```

As hashes are a subset of the series! pseudotype, use **series?** to check this:

```
probe series? hsh
true
```

Forming a hash value creates a string from the contents contained in the hash:

```
probe form hsh
"1 one 2 two 3 three 4 four"
```

Molding a hash value creates a string of the hash value itself and its contents, thus allowing it to be reloaded as a REBOL hash value:

```
probe mold hsh
make hash! [1 "one" 2 "two" 3 "three" 4 "four"]
```

2.6 Image

2.6.1 Concept

The **image!** datatype is a series that holds RGB images. This datatype is used with REBOL/View.

The image formats supported are GIF, JPEG, and BMP. The loaded image can be manipulated as a series.

2.6.2 Format

Images are normally loaded from a file. However, they can be expressed in source code as well by making an image. The block provided includes the image size and its RGB data.

```
image: make image! [192x144 #{
    B34533B44634B44634B54735B7473
    84836B84836B84836BA4837BA4837
    BC4837BC4837BC4837BC4837BC483 ...
}
```

2.6.3 Creation

Empty images can be created using **make** or **to-image:**

```
empty-img: make image! 300x300
empty-img: to-image 150x300
```

The size of the image is provided.

Images can also be made from snapshots of a face object. This is also done using **make** or **to-image:**

```
face-shot: make image! face
face-shot: to-image face
```

Use **load** to load an image file. If the image's format is not supported, it will fail to load.

Loading an image:

```
img: load %bay.jpg
```

2.6.4 Related

Use **image?** to determine whether a value is the **image!** datatype:

```
probe image? img
```

Images are a subset of the **series!** pseudotype:

```
probe series? img
```

Use the **/size** refinement to return the pixel size of an image as a pair value:

```
probe img/size
```

The pixel values of an image are obtained using **pick** and changed using **poke**. The value returned by **pick** is an RGB tuple value. The value replaced with **poke** also should be a tuple value.

Picking specific pixels:

```
probe pick img 1

probe pick img 1500
```

Poking specific pixels:

```
poke img 1 255.255.255
probe pick img 1

poke img 1500 0.0.0
probe pick img 1500
```

2.7 Issue

2.7.1 Concept

An **issue!** is a series of characters used to sequence symbols or identifiers for things like telephone numbers, model numbers, serial numbers, and credit card numbers.

Issue values are a subset of series, and thus can be manipulated as series:

```
probe copy/part find #888-555-1212 "555" 3
#555
```

2.7.2 Format

Issues start with a number sign (#) and continue until the first delimiting character (such as a space) is reached.

```
#707-467-8000
#A-0987654321-CD-09876
#1234-5678-4321-8765
#MG82/32-7
```

Values that contain delimiting characters should be written as strings rather than issues.

2.7.3 Creation

The **to-issue** function converts data to the **issue!** datatype:

```
probe to-issue "1234-56-7890"
#1234-56-7890
```

2.7.4 Related

Use **issue?** to determine whether a value is an **issue!** datatype.

```
probe issue? #1234-56-7890
true
```

As issues are a subset of the series pseudotype, use **series?** to check this:

```
probe series? #1234-56-7890
true
```

The **form** function returns an issue as a string without the number sign (#):

```
probe form #1234-56-7890
1234-56-7890
```

The **mold** function returns an issue as a string that can be read by REBOL as an issue value:

```
probe mold #1234-56-7890
#1234-56-7890
```

The **print** function prints an issue to standard output after doing a **reform** on it:

```
print #1234-56-7890
1234-56-7890
```

2.8 List

2.8.1 Concept

Lists are linked list blocks that allow for faster and more efficient insertion and removal of their values. They can be used in cases where a large number of insertions or removals are being performed on large blocks.

2.8.2 Format

List blocks must be constructed by using **make** or **to-list**. They have no lexical format.

Lists values are not a direct substitute for blocks. There are a couple of differences between

blocks and lists:

- Inserting into a list modifies its reference to just after the point of insertion.
- Removing the element currently referenced in a list causes the reference to reset to the tail of the list

The following examples show the difference in behavior between inserting into a list and a block.

Initializing a block and list:

```
blk: [1 2 3]

lst: to-list [1 2 3]
```

Inserting into a block and list:

```
insert blk 0

insert lst 0
```

Looking at the word after the block and list after insertion. Notice `blk` points to the head, as before the insertion of `0`, but `lst` points to just after the point of insertion:

```
print blk
0 1 2 3
print lst
1 2 3
print head lst
0 1 2 3
```

The following examples show the difference in behavior between removing an element from a list and a block.

Initializing a block and a list:

```
blk: [1 2 3]

lst: to-list [1 2 3]
```

Removing from the block and list:

```
remove blk

remove lst
```

Looking at the word after removal of the value. Notice `lst` now points to the tail of the series:

```
print blk
2 3
```

```
print tail? lst
true
print head lst
2 3
```

If you don't want the word to be at the tail after removing a value, step forward and remove the value behind the current index. The following examples depicts this.

Initializing a list:

```
lst: to-list [1 2 3]
```

Stepping forward and removing the value behind the current index:

```
remove back (lst: next lst)
```

Looking at the word after removing the value:

```
probe lst
make list! [2 3]
```

2.8.3 Creation

Use **make** to initialize a list value:

```
lst: make list! 10 ; allocating space for 10 elements
```

The **to-list** function converts data to the **list!** datatype:

Convert a block:

```
blk: [1 "one" 2 "two" 3 "three" 4 "four"]
probe to-list blk
```

2.8.4 Related

Use **list?** to determine whether a value is an **list!** datatype.

```
lst: to-list [1 "one" 2 "two" 3 "three" 4 "four"]
probe list? Lst
true
```

Since lists are a subset of the **series!** datatype, use **series?** to check whether a list is a series:

```
probe series? lst
true
```

Using **form** on a list value creates a string from the contents contained in the list:

```
probe form lst
"1 one 2 two 3 three 4 four"
```

Using **mold** on a list value creates a string of the list value itself and it's contents, thus allowing it to be reloaded as a REBOL list value:

```
probe mold lst
make list! [1 "one" 2 "two" 3 "three" 4 "four"]
```

2.9 Paren

2.9.1 Concept

A **paren!** datatype is a block that is immediately evaluated. It is identical to a block in every way, except that it is evaluated when it is encountered and its result is returned.

When used within an evaluated expression, a **paren!** allows you to control the order of evaluation:

```
print 1 + (2 * 3)
7
print 1 + 2 * 3
9
```

The value of a **paren!** can be accessed and modified in the same way as any block. However, when referring to a **paren!**, care must be taken to prevent if from being evaluated. If you store a paren in a variable, you will need to use a get-word form (:word) to prevent it from being evaluated.

Parens are a type of series, thus anything that can be done with a series can be done with paren values.

```
paren: first [(1 + 2 * 3 / 4)]

print type? :paren
paren!
print length :paren
7
print first :paren
1
print last :paren
4
insert :paren [10 + 5 *]
probe :paren
(10 + 5 * 1 + 2 * 3 / 4)
print paren
```

```
12.75
```

2.9.2 Format

Parens are identified by their open and closing parenthesis. They can span multiple lines and contain any data, including other paren values.

2.9.3 Creation

The **make** function can be used to allocate a paren value:

```
paren: make paren! 10
insert :paren 10
insert :paren `+
insert :paren 20

print :paren
20 + 10
print paren
30
```

The **to-paren** function converts data to the **paren!** datatype:

```
probe to-paren "123 456"
(123 456)
probe to-paren [123 456]
(123 456)
```

2.9.4 Related

Use **paren?** to test the datatype.

```
blk: [(3 + 3)]
probe pick blk 1
(3 + 3)
probe paren? pick blk 1
true
```

As parens are a subset of the **series!** pseudotype, use **series?** to check this:

```
probe series? pick blk 1
true
```

Using **form** on a paren value creates a string from the contents contained in the paren:

```
probe form pick blk 1
3 + 3
```

2.10 Path

2.10.1 Concept

Paths are a collection of words and values delineated with forward slashes (/). Paths are used to navigate to or find something. The words and values of a path are called refinements, and they are combined to provide a means of navigating through a value or function.

Paths can be used on blocks, files, strings, lists, hashes, functions, and objects. How a path operates depends on the datatype being used.

Paths can be used to select values from blocks, pick characters from strings, access variables in objects, refine the operation of a function:

```
USA/CA/Ukiah/size  (block selection)

names/12           (string position)

account/balance    (object function)

match/any          (function option)
```

The example below shows the simplicity of using a path to access a mini-database created from a few blocks:

```
towns: [
    Hopland [
        phone #555-1234
        web   http://www.hopland.ca.gov
    ]

    Ukiah [
        phone #555-4321
        web   http://www.ukiah.com
        email info@ukiah.com
    ]
]

print towns/ukiah/web
http://www.ukiah.com
```

Summary of path constructs:

Action	Type Word	Type Test	Conversion
path/word:	set-path!	set-path?	to-set-path
path/word	path!	path?	to-path
'path/word	lit-path!	lit-path?	to-lit-path

Examples of paths:

Evaluate an object's function:

```
obj: make object! [
    hello: func [] [print "hello! hello!"]
]
obj/hello
```

```
hello! hello!
```

Evaluate an object's word:

```
obj: make object! [
    text: "do you believe in magic?"
]
probe obj/text
do you believe in magic?
```

Function refinements:

```
hello: func [/again] [
    print either again ["hello again!"]["hello"]
]
hello/again
hello again!
```

Select from blocks, or multiple blocks:

```
USA: [
    CA [
        Ukiah [
            population 15050
            elevation [610 feet]
        ]
        Willits [
            population 5073
            elevation [1350 feet]
        ]
    ]
]
print USA/CA/Ukiah/population
15050
print form USA/CA/Willits/elevation
1350 feet
```

Pick elements from series and embedded series by their numeric position:

```
string-series: "abcdefg"
block-series: ["John" 21 "Jake" 32 "Jackson" 43 "Joe" 52]
block-with-sub-series: [ "abc" [4 5 6 [7 8 9]]]
probe string-series/4
#"d"
probe block-series/3
Jake
probe block-series/6
43
probe block-with-sub-series/1/2
#"b"
probe block-with-sub-series/2/2
5
probe block-with-sub-series/2/4/2
```

```
8
```

The words supplied as paths are symbolic and therefore unevaluated. This is necessary to allow the most intuitive form for object referencing. To use a word's reference, an explicit word value reference is required:

```
city: 'Ukiah
probe USA/CA/:city
[
    population 15050
    elevation "610 feet"
]
```

Paths in blocks, hashes, or objects are evaluated by matching the word at the top level of the path, and verifying the word as a **block!**, **hash!** or **object!** value. Then the next word in the path is sought as a word expressed in the block, hash or object and an implicit **select** is performed. The value following the word matched is returned. When the returned value is a block, hash, or object, the path can be extended:

Getting the value associated with **CA** in **USA:**

```
probe USA/CA
[
    Ukiah [
        population 15050
        elevation "610 feet"
    ]
    Willits [
        population 9935
        elevation "1350 feet"
    ]
]
```

Getting the value associated with **Willits** in **USA/CA:**

```
probe USA/CA/Willits
[
    population 9935
    elevation "1350 feet"
]
```

Getting the value associated with **population** in **USA/CA/Willits:**

```
probe USA/CA/Willits/population
9935
```

When a word is used in a path that does not exist at the given point in the structure, an error is produced:

```
probe USA/CA/Mendocino
** Script Error: Invalid path value: Mendocino.
```

```
** Where: probe USA/CA/Mendocino
```

Paths can be used to change values in blocks and objects:

```
USA/CA/Willits/elevation: "1 foot, after the earthquake"
probe USA/CA/Willits
[
    population 9935
    elevation "1 foot, after the earthquake"
]
obj/text: "yes, I do believe in magic."
probe obj
make object! [
    text: "yes, I do believe in magic."
]
```

Blocks, hashes, functions, and objects can be mixed in paths.

Selecting from elements in a block inside an object:

```
obj: make object! [
    USA: [
        CA [
            population "too many"
        ]
    ]
]
probe obj/USA/CA/population
too many
```

Using function refinements within an object:

```
obj: make object! [
    hello: func [/again] [
        print either again [
            "hello again"
        ] [
            "oh, hello"
        ]
    ]
]
obj/hello/again
hello again
```

Paths are a type of series, thus anything that can be done with a series can be done with path values:

```
root: [sub1 [sub2 [
    word "a word at the end of the path"
    num 55
]   ]   ]
path: 'root/sub1/sub2/word
probe :path
```

```
root/sub1/sub2/word
```

In the previous example, the **:path** notation was used to get the path itself, not the path's value:

```
probe path
a word at the end of the path
```

Looking at how long a path is:

```
probe length? :path
4
```

Finding a word within a path:

```
probe find :path 'sub2
sub2/word
```

Changing a word in a path:

```
change find :path 'word 'num
probe :path
root/sub1/sub2/num
probe path
55
```

2.10.2 Format

Paths are expressed relative to a root word by providing a number of refinements, each separated by a forward slash (/). These refinements can be words or values. Their specific interpretation vary depending on the datatype of the root value.

The words supplied as refinements in paths are symbolic and are not evaluated. This is necessary to allow the most intuitive form for object referencing. To use a word's reference, an explicit word value reference is required:

```
root/:word
```

This example uses the value of the variable, rather than it name.

2.10.3 Creation

You can **make** an empty path of a given size with:

```
path: make path! 10
insert :path `test
insert tail :path `this
print :path
```

```
test/this
```

The **to-path** function converts data to the **path!** datatype:

```
probe to-path [root sub]
root/sub
probe to-path "root sub"
root/sub
```

The **to-set-word** function converts other values to the **set-word** datatype.

```
probe to-set-path "root sub"
root/sub:
```

The **to-lit-word** function converts other values to the **lit-word** datatype.

```
probe to-lit-path "root sub"
'root/sub
```

2.10.4 Related

Use **path?**, **set-path?**, and **lit-path?** to determine the datatype of a value.

```
probe path? second [1 two "3"]
false
blk: [sub1 [sub2 [word 1]]]
blk2: [blk/sub1/sub2/word: 2]
if set-path? (pick blk2 1) [print "it is set"]
it is set
probe lit-path? first ['root/sub]
true
```

As paths are a subset of the **series!** pseudotype, use **series?** to check this:

```
probe series? pick [root/sub] 1
true
```

Use **form** on a path value creates a string from the path:

```
probe form pick [root/sub] 1
root/sub
```

Use **mold** on a path value creates a string of the path value itself, thus allowing it to be reloaded as a REBOL path value:

```
probe mold pick [root/sub] 1
root/sub
```

2.11 String

2.11.1 Concept

Strings are a series of characters. All operations performable on series values can be performed on strings.

2.11.2 Format

String values are written as a sequence of characters surrounded by double quotes " " or braces {}. Strings enclosed in double quotes are restricted to a single line and must not contain unprintable characters.

```
"This is a short string of characters."
```

Strings enclosed in braces are used for larger sections of text that span multiple lines. All of the characters of the string, including spaces, tabs, quotes, and newlines are part of the string.

```
{This is a long string of text that will
not easily fit on a single line of source.
These are often used for documentation
purposes.}
```

Braces are counted within the string, so a string can include other braces as long as the number of closing braces matches the number of opening braces.

```
{
This is another long string of text that would
never fit on a single line. This string also
includes braces { a few layers deep { and is
valid because there are as many closing braces }
as there are open braces } in the string.
}
```

You can include special characters and operations in strings by prefixing them with a caret (^). Special characters include:

Character	Definition
^"	Inserts a double quote (").
^}	Inserts a closing brace (}).
^^	Inserts a caret (^).
^/	Starts a new line.
^(line)	Starts a new line.
^-	Inserts a tab.
^(tab)	Inserts a tab.

^(page)	Starts a new page.
^(back)	Erases one character to the left of the insertion point.
^(null)	Inserts a null character.
^(escape)	Inserts an escape character.
^(letter)	Inserts control-letter (A-Z).
^(xx)	Inserts an ASCII character by hexidecimal (xx) number. his format allows for expansion into unicode characters in the future.

2.11.3 Creation

Use **make** to create a pre-allocated amount of space for an empty string:

```
make string! 40'000 ; space for 40k characters
```

The **to-string** function converts data of other datatypes to a **string!** datatype:

```
probe to-string 29-2-2000
"29-Feb-2000"
probe to-string 123456.789
"123456.789"
probe to-string #888-555-2341
"888-555-2341"
```

Converting a block of data to a string with **to-string** has the effect of doing a **rejoin**, but without evaluating the block's contents:

```
probe to-string [123 456]
"123456"
probe to-string [225.225.225.0 none true 'word]
"225.225.225.0nonetrueword"
```

2.11.4 Related

Use **string?** or **series?** to determine whether a value is an **string!** datatype:

```
print string? "123"
true
print series? "123"
true
```

The functions **form** and **mold** are closely related to strings, as they create strings from other datatypes. The **form** function makes a human readable version of a specified datatype, while **mold** makes a REBOL readable version.

```
probe form "111 222 333"
"111 222 333"
probe mold "111 222 333"
```

```
{"111 222 333"}
```

2.12 Tag

2.12.1 Concept

Tags are used in HTML and other markup languages to indicate how text fields are to be treated. For example, the tag **<HTML>** at the beginning of a file indicates that it should be parsed by the rules of the Hypertext Markup Language. A tag with a forward slash (/), such as **</HTML>**, indicates the closing of the tag.

Tags are a subset of series, and thus can be manipulated as such:

```
a-tag: <img src="mypic.jpg">
probe a-tag
<img src="mypic.jpg">
append a-tag { alt="My Picture!"}
probe a-tag
<img src="mypic.jpg" alt="My Picture!">
```

2.12.2 Format

Valid tags begin with an open angle bracket (<) and end with a closing bracket (>). For example:

```
<a href="index.html">
<img src="mypic.jpg" width="150" height="200">
```

2.12.3 Creation

The **to-tag** function converts data to the **tag!** datatype:

```
probe to-tag "title"
<title>
```

Use **build-tag** to construct tags, including their attributes. The **build-tag** function takes one argument, a block. In this block, the first word is used as the tag name and the remaining words are processed as attribute value pairs:

```
probe build-tag [a href http://www.rebol.com/]
<a href="http://www.rebol.com/">
probe build-tag [
    img src %mypic.jpg width 150 alt "My Picture!"
]
<img src="mypic.jpg" width="150" alt="My Picture!">
```

2.12.4 Related

Use **tag?** to determine whether a value is an **tag!** datatype.

```
probe tag? <a href="http://www.rebol.com/">
```

```
true
```

As tags are a subset of the series pseudotype, use **series?** to check this:

```
probe series? <a href="http://www.rebol.com/">
true
```

The **form** function returns a tag as a string:

```
probe form <a href="http://www.rebol.com/">
{<a href="http://www.rebol.com/">}
```

The **mold** function returns a tag as a string:

```
probe mold <a href="http://www.rebol.com/">
{<a href="http://www.rebol.com/">}
```

The **print** function prints a tag to standard output after doing a **reform** on it:

```
print <a href="http://www.rebol.com/">
<a href="http://www.rebol.com/">
```

2.13 URL

2.13.1 Concept

URL is an acronym for Uniform Resource Locator, an Internet standard used to access resources such as web pages, images, files, and email across the network. The best known URL scheme is that used for web locations such as **http://www**.REBOL.com.

URL values are a subset of series, and thus can be manipulated as series:

```
url: http://www.rebol.com/reboldoc.html
probe to-file find/reverse (tail url) "rebol"
%reboldoc.html
```

2.13.2 Format

The first part of a URL indicates its communications protocol, called a scheme. The language supports several schemes, including: web pages (**HTTP:**), file transfer (**FTP:**), newsgroups (**NNTP:**), email (**MAILTO:**), files (**FILE:**), finger (**FINGER:**), whois (**WHOIS:**), small network time (**DAYTIME:**), post office (**POP:**), transmission control (**TCP:**) and domain name service (**DNS:**). These scheme names are followed by characters that are dependent on which scheme being used.

```
http://host.dom/path/file
ftp://host.dom/path/file
```

```
nntp://news.some-isp.net/some.news.group
mailto:name@domain
file://host/path/file
finger://user@host.dom
whois://rebol@rs.internic.net
daytime://everest.cclabs.missouri.edu
pop://user:passwd@host.dom/
tcp://host.dom:21
dns://host.dom
```

Some fields are optional. For instance, the host can be followed by a port number if it differs from the default. An FTP URL supplies a default password if one is not specified:

```
ftp://user:password@host.dom/path/file
```

Characters in a URL must conform to Internet standards. Restricted characters must be encoded in hexadecimal by preceding them with the escape character %:

```
probe http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
print http://www.somesite.dom/odd%28dir%29/odd%7Bfile%7D.txt
http://www.somesite.dom/odd(dir)/odd{file}.txt
```

2.13.3 Creation

The **to-url** function converts blocks to the **url!** datatype, the first element in the block is the scheme, the second element is the domain (with or without **user:pass** and port) the remaining elements are the path and file:

```
probe to-url [http www.rebol.com reboldoc.html]
http://www.rebol.com/reboldoc.html
probe to-url [http www.rebol.com %examples "websend.r"]
http://www.rebol.com/examples/websend.r
probe to-url [http usr:pass@host.com:80 "(path)" %index.html]
http://usr:pass@host.com:80/%28path%29/index.html
```

2.13.4 Related

The datatype word is **url!**.

Use **url?** to test the datatype.

```
probe url? ftp://ftp.rebol.com/
true
```

As urls are a subset of the series pseudotype, use **series?** to check this:

```
probe series? http://www.rebol.com/
true
```

```
 
```

3. Other Values

3.1 Character

3.1.1 Concept

Characters are not strings; they are the individual values from which strings are constructed. A character can be a printable, unprintable, or a control character.

3.1.2 Format

A **char!** value is written as a number sign (#) followed by a string enclosed in double quotes. The number sign is necessary to distinguish a character from a string:

```
#"R"     ; the single character: R
"R"      ; a string with the character: R
```

Characters can include escape sequences that begin with a caret(^)and are followed by one or more characters of encoding. This encoding can include the characters **#"^A"** to **#"^Z"** for **control A** to **control Z** (upper and lower case are the same):

```
#"^A" #"^Z"
```

In addition, if parens are used within the character, they specify a special value. For example, null can be written as:

```
"^@"
"^(null)"
"^(00)"
```

Thes last line is written in hex format (base 16). The parens around the value allow for expansion into 16 bit unicode characters in the future.

Following is a table of control characters that can be used in REBOL.

Character	Definition
#"(null)" or #"@"	null (zero)
#"(line)", #"/" or, #"."	end of line
#"(tab)" or #"-"	horizontal tab
#"(page)"	new page (and page eject)
#"(esc)"	escape
#"(back)"	backspace
#"(del)"	delete
#"^"	caret character
#"^""	quotation mark

`#"(00)" to #"(FF)"`	hex forms of characters

3.1.3 Creation

Characters can be converted to and from other datatypes with the **`to-char`** function:

```
probe to-char "a"
#"a"
probe to-char "z"
#"z"
```

Characters follow the ASCII standard and can be constructed by specifying a character's numeric equivalent:

```
probe to-char 65
#"A"
probe to-char 52
#"4"
probe to-char 52.3
#"4"
```

Another method of obtaining a character is to get the first character from a string:

```
probe first "ABC"
#"A"
```

While characters in strings are not case sensitive, comparison between individual characters is case sensitive:

```
probe "a" = "A"
true
probe #"a" = #"A"
false
```

However, when used in many types of functions, the comparison is not case sensitive unless you specify that option. Examples are:

```
select [#"A" 1] #"a"
1
select/case [#"A" 1] #"a"
none
find "abcde" #"B"
"bcde"
find/case "abcde" #"B"
none
switch #"A" [#"a" [print true]]
true
```

3.1.4 Related

Use **char?** to determine whether a value is a **char!** datatype.

```
probe char? "a"
false
probe char? #"a"
true
```

Use the **form** function to print a character without the number sign:

```
probe form #"A"
"A"
```

Use **mold** on to print a character with the number sign and double quotes (and escape sequences for those characters that require it.):

```
probe mold #"A"
{#"A"}
```

3.2 Date

3.2.1 Concept

Around the world, dates are written in a variety of formats. However, most countries use the **day-month-year** format. One of the few exceptions is the United States, which commonly uses a **month-day-year** format. For example, a date written numerically as 2/1/1999 is ambiguous. The month could be interpreted as either February or January. Some countries use a dash (-), some use a forward slash (/), and others use a period (.) as a separator. Finally, computer people often prefer dates in the year-month-day (ISO) format so they can be easily sorted.

3.2.2 Format

The REBOL language is flexible, allowing **date!** datatypes to be expressed in a variety of formats. For example, the first day of March can be expressed in any of the following formats:

```
probe 1/3/1999
1-Mar-1999
probe 1+++1999
1-Mar-1999
probe 1999+++1   ;ISO format
1-Mar-1999
```

The year can span up to 9999 and down to 1. Leap days (February 29) can only be written for leap years:

```
probe 29-2-2000
29-Feb-2000
```

The fields of dates can be separated with forward slashes (/) or dashes (-). Dates can be written in either a year-month-day format or a day-month-year format:

```
probe 1999-10-5
5-Oct-1999
probe 1999/10/5
5-Oct-1999
probe 5-10-1999
5-Oct-1999
probe 5/10/1999
5-Oct-1999
```

Because the international date formats that are not widely used in the USA, a month name or month abbreviation can also be used:

```
probe 5/Oct/1999
5-Oct-1999
probe 5-October-1999
5-Oct-1999
probe 1999/oct/5
5-Oct-1999
```

When the year is the last field, it can be written as either a four digit or two digit number:

```
probe 5/oct/99
5-Oct-1999
probe 5/oct/1999
5-Oct-1999
```

However, it is preferred to write the year in full. Otherwise, problems occur with date comparison and sorting operations. While two digits can be used to express a year, the interpretation of a two-digit year is relative to the current year and is only valid for 50 years in the future or in the past:

```
probe 28-2-66   ; refers to 1966
28-Feb-1966
probe 12-Mar-20 ; refers to 2020
12-Mar-2020
probe 11+++45   ; refers to 2045, not 1945
11-Mar-2045
```

It is recommended to use a four-digit year to avoid potential problems.

To represent dates in the first century (which is rarely done because the Gregorian calendar did not exist), use leading zeros to represent the century (as in **9-4-0029**).

Dates can also include an optional time field and an optional time zone. The time is separated from the date with a forward slash (/). The time zone is appended using a plus (+) or minus (-), and no spaces are allowed. Time zones are written as a time shift (plus or minus) from GMT. The resolution of the time zone is to the half hour. If the time shift is an integer, it is assumed to be hours:

```
probe 4/Apr/2000/6:00+8:00
4-Apr-2000/6:00+8:00
probe 1999-10-2/2:00-4:00
2-Oct-1999/2:00-4:00
```

```
probe 1/1/1990/12:20:25-6
1-Jan-1990/12:20:25
```

There can be no spaces within the date. For example:

```
10 - 5 - 99
```

would be interpreted as a subtraction expression, not a date.

3.2.3 Access

Refinements can be used with a date value to get any of its defined fields:

Refinement	Description
/day	Gets the day.
/month	Gets the month.
/year	Gets the year.
/julian	Gets the day of the year.
/weekday	Gets the weekday (1-7/Mon-Sun).
/time	Gets the time (if present).
/hour	Gets the time's hour (if present)
/minute	Gets the time's minute (if present).
/second	Gets the time's second (if present).
/zone	Gets the time zone (if present).

Here's how these refinements work:

```
some-date: 29-Feb-2000
probe some-date/day
29
probe some-date/month
2
probe some-date/year
2000
days: ["Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"]
probe pick days some-date/weekday
Tue
```

When a time is present, the time related refinements can be used. The **/hour**, **/minute** and **/second** refinements are used with the **/time** refinement that isolates the time segment of the date value for them to work on:

```
lost-time: 29-Feb-2000/11:33:22.14-8:00
probe lost-time/time
11:33:22.14
probe lost-time/time/hour
11
probe lost-time/time/minute
33
```

```
probe lost-time/time/second
22.14
probe lost-time/zone
-8:00
```

3.2.4 Creation

Use the **to-date** function to convert values to a **date!**:

```
probe to-date "5-10-1999"
5-Oct-1999
probe to-date "5 10 1999 10:30"
5-Oct-1999/10:30
probe to-date [1999 10 5]
5-Oct-1999
probe to-date [5 10 1999 10:30 -8:00]
5-Oct-1999/10:30-8:00
```

[!Note When converting to a **date!**, the year must be specified as four digits.

Conversions can be applied to various math operations on dates:

```
probe 5-Oct-1999 + 1
6-Oct-1999
probe 5-10-1999 - 10
25-Sep-1999
probe 5-Oct-1999/23:00 + 5:00
6-Oct-1999/4:00
```

3.2.5 Related

Use **date?** to determine whether a value is a **date!** datatype.

```
probe date? 5/1/1999
true
```

The related function **to-idate** returns a standard Internet date string. The Internet date format is day, date, month, year, time (24-hour clock), and time zone offset from GMT.

```
probe to-idate now
Fri, 30 Jun 2000 14:42:26 -0700
```

The **now** function returns the current date and time in full format including the time zone offset:

```
probe now
30-Jun-2000/14:42:26-7:00
```

3.3 Logic

3.3.1 Concept

The **logic!** datatype consists of two states representing **true** and **false**. They are often returned from comparisons such as:

```
age: 100
probe age = 100
true
time: 10:31:00
probe time < 10:30
false
str: "this is a string"
probe (length? str) > 10
true
```

The logic! datatype is most commonly used as parameters to conditional functions such as **if**, **while**, and **until:**

```
if age = 100 [print "Centennial human"]
Centennial human
while [time > 6:30] [
    send person "Wake up!"
    wait [0:10]
]
```

The complement of a logic value is obtained from the **not** function:

```
there: place = "Ukiah"
if not there [...]
```

3.3.2 Format

Normally, logic values are retrieved from the evaluation of comparison expressions. However, words can be set to a logic value and used to turn the word **on** or **off:**

```
print-me: false
print either print-me ["turned on"]["turned off"]
turned off
print-me: true
print either print-me ["turned on"]["turned off"]
turned on
```

The **false** value is not equivalent to integer zero or **none**. However, in conditional expressions **false** and **none** have the same effect:

```
print-me: none
print either print-me ["turned on"]["turned off"]
turned off
```

Just about any value assigned to a word has the same effect as **true:**

```
print-me: "just a string"
print either print-me ["turned on"]["turned off"]
turned on
print-me: 11-11-1999
print either print-me ["turned on"]["turned off"]
turned on
```

The following words are predefined to hold logic values:

```
true
on      ;same as true
yes     ;same as true
false
off     ;same as false
no      ;same as false
```

So, instead of **true** and **false**, when it makes sense, the words **on** and **off**, or **yes** and **no** can be used instead:

```
print-me: yes
print either print-me ["turned on"]["turned off"]
turned on
print-me: no
print either print-me ["turned on"]["turned off"]
turned off
print-me: on
print either print-me ["turned on"]["turned off"]
turned on
print-me: off
print either print-me ["turned on"]["turned off"]
turned off
```

3.3.3 Creation

The **to-logic** function converts **integer!** or **none!** values to the **logic!** datatype:

```
probe to-logic 0
false
probe to-logic 200
true
probe to-logic none
false
probe to-logic []
true
probe to-logic "a"
true
probe to-logic none
false
```

3.3.4 Related

Use **logic?** to determine whether a value is a **logic!** datatype.

```
probe logic? 1
false
probe logic? on
true
probe logic? false
true
```

Use the functions **form**, **print**, and **mold** to print a logic value:

```
probe form true
true
probe mold false
false
print true
true
```

3.4 Money

3.4.1 Concept

There is a wide variety of international symbols for monetary denominations. Some symbols are used before the amount and some after. As a standard for representing international monetary values, the REBOL language uses the United States monetary format, but allows the inclusion of specific denominations.

3.4.2 Format

The **money!** datatype uses standard IEEE floating point numbers allowing up to 15 digits of precision including cents.

The language limits the length to 64 characters. Values that are out of range or cannot be represented in 64 characters are flagged as an error.

Monetary values are prefixed with an optional currency designator, followed by a dollar sign ($). A plus (+) or minus (-) can appear immediately before the first character (currency designator or dollar sign) to indicate sign.

```
$123
-$123
$123.45
US$12
US$12.34
-US$12.34
$12,34
-$12,34
DEM$12,34
```

To break long numbers into readable segments, a single quote (`) can be placed anywhere between two digits within the amount, but not before the amount.

```
probe $1'234.56
$1234.56
probe $1'234'567,89
```

```
$1234567.89
```

Do not use commas and periods to break up large amounts, as both these characters represent decimal points.

The money! datatype is a hybrid datatype. Conceptually money is scalar--an amount of money. However, because the currency designation is stored as a string, the **money!** datatype has two elements:

- string! - The currency designator string, which can have 3 characters maximum.
- decimal! - The money amount.

To demonstrate this, the following money is specified with the USD prefix:

```
my-money: USD$12345.67
```

Here are the two components:

```
probe first my-money
USD
probe second my-money
12345.67
probe pick my-money 3        ; only two components
none
```

If no currency designator is used, the currency designator string is empty:

```
my-money: $12345.67

probe first my-money
""
probe second my-money
12345.67
```

Various international currencies can be specified in the currency designator, such as:

```
my-money: DKM$12'345,67

probe first my-money
DKM
probe second my-money
12345.67
```

3.4.3 Creation

Use the **to-money** function to convert money from a **string!**, **integer!**, **decimal!**, or **block!**.

```
probe to-money 123
$123.00
```

```
probe to-money "123"
$123.00
probe to-money 12.34
$12.34
probe to-money [DEM 12.34]
DEM$12.34
probe to-money [USA 12 34]
USA$12.34
```

Money can be added, subtracted, and compared with other money of the same currency. An error occurs if a different currency is used for such operations (automatic conversions are not currently supplied).

```
probe $100 + $10
$110.00
probe $100 - $50
$50.00
probe equal? DEM$100.11 DEM$100.11
true
```

Money can be multiplied and divided by integers and decimals. Money can also be divided by money, resulting in an integer or decimal.

```
probe $100 + 11
$111.00
probe $100 / 4
$25.00
probe $100 * 5
$500.00
probe $100 - 20.50
$79.50
probe 10 + $1.20
$11.20
probe 10 - $0.25
$9.75
probe $10 / .50
$20.00
probe 10 * $0.75
$7.50
```

3.4.4 Related

Use **money?** to determine whether a value is an **money!** datatype.

```
probe money? USD$12.34
true
```

Use the **form**, **print**, and **mold** functions with a money argument to print a money value with the currency designator and dollar sign ($), as a decimal number with two digits of decimal precision.

```
probe form USD$12.34
USD$12.34
probe mold USD$12.34
```

```
USD$12.34
print USD$12.34
USD$12.34
```

3.5 None

3.5.1 Concept

The **none!** datatype contains a single value that represents nothing or no value.

The concept of none is not the same as an empty block, empty string, or null character. It is an actual value that represents non-existence.

A **none!** value can be returned from various functions, primarily those involving series (for example, **pick** and **find**).

The REBOL word **none** is defined as a **none!** datatype and contains a **none!** value. The word **none** is not equivalent to **zero** or **false**. However, **none** is interpreted as **false** by many functions.

A **none!** value has many uses such as a return value from series functions like **pick**, **find** and **select:**

```
if (pick series 30) = none [...]
```

In databases, a **none** can be a placeholder for missing values:

```
email-database: [
    "Bobby" bob@rebol.com 40
    "Linda" none 23
    "Sara"  sara@rebol.net 33
]
```

It also can be used as a logic value:

```
secure none
```

3.5.2 Format

The word **none** is predefined to hold a **none** value.

Although **none** is not equivalent to **zero** or **false**, it is valid within conditional expressions and has the same effect as **false:**

```
probe find "abcd" "e"
none
if find "abcd" "e" [print "found"]
```

3.5.3 Creation

The **to-none** function always returns **none**.

3.5.4 Related

Use **none?** to determine whether a value is a **none!** datatype.

```
print none? 1
false
print none? find [1 2 3] 4
true
```

The **form**, **print**, and **mold** functions print the value **none** when passed a **none** argument.

```
probe form none
none
probe mold none
none
print none
none
```

3.6 Pair

3.6.1 Concept

A pair! datatype is used to indicate spatial coordinates, such as positions on a display. They are used for both positions and sizes. Pairs are used primarily in REBOL/View.

3.6.2 Format

A pair is specified as integers separated by an **x** character.

```
100x50

1024x800

-50x200
```

3.6.3 Creation

Use **to-pair** to convert block or string values into a pair datatype:

```
p: to-pair "640x480"
probe p
640x480
p: to-pair [800 600]
probe p
800x600
```

3.6.4 Related

Use **pair?** to determine whether a value is a **pair!** datatype:

```
probe pair? 400x200
true
probe pair? pair
true
```

Pairs can be used with most integer math operators:

```
100x200 + 10x20

10x20 * 2x4

100x30 / 10x3

100x100 * 3

10x10 + 3
```

Pairs can be viewed by their individual coordinates:

```
pair: 640x480
probe first pair
640
probe second pair
480
```

All pair values support the **/x** and **/y** refinements. These refinements allow the viewing and manipulation of individual pair coordinates.

Viewing individual coordinates:

```
probe pair/x
640
probe pair/y
480
```

Modifying individual coordinates:

```
pair/x: 800
pair/y: 600
probe pair
800x600
```

3.7 Refinement

3.7.1 Concept

Refinements are modifiers, similar to adjectives used in natural (human) languages. A refinement indicates a variation in the use of, or extension in the meaning of, a function, object, filename, URL, or path. Refinements are always symbolic in their value.

Refinements are used for functions:

```
block: [1 2]
append/only block [3 4]
```

objects:

```
print system/version
```

files:

```
dir: %docs/core
print read dir/file.txt
```

urls:

```
site: http://www.rebol.com
print read site/index.html
```

3.7.2 Format

Refinements are composed with a slash followed by a valid REBOL word (see the words section below for definition). Examples are:

```
/only
/test1
/save-it
```

Refinements are usually joined to other words, such as in the case of:

```
port: open/binary file
```

But refinements can also be written alone, as is done when specifying refinements to a function:

```
save-data: function [file data /limit /reload] ...
```

3.7.3 Creation

Refinements can be created literally in source code:

```
/test
```

or can be composed from the `to-refinement` word:

```
probe to-refinement "test"
/test
```

3.7.4 Related

To test for a refinement, use the refinement? function:

```
probe refinement? /test
true
probe refinement? 'word
false
```

3.8 Time

3.8.1 Concept

The REBOL language supports the standard expression of time in hours, minutes, seconds, and subseconds. Both positive and negative times are permitted.

The `time!` datatype uses relative rather than absolute time. For example, `10:30` is 10 hours and 30 minutes rather than the time of 10:30 A.M. or P.M.

3.8.2 Format

Times are expressed as a set of integers separated by colons (:).. Hours and minutes are required, but seconds are optional. Within each field, leading zeros are ignored:

```
10:30
0:00
18:59
23:59:50
8:6:20
8:6:2
```

The minutes and seconds fields can contain values greater than 60. Values greater than 60 are automatically converted. For instance `0:120:00` is the same as `2:00`.

```
probe 00:120:00
2:00
```

Subseconds are specified using a decimal in the seconds field. Use either a period or a comma as the decimal point. The hours and minutes fields become optional when the decimal is present. Subseconds are encoded to the nano-second, or one billionth of a second:

```
probe 32:59:29.5
32:59:29.5
probe 1:10,25
0:01:10.25
probe 0:0.000000001
0:00:00.000000001
probe 0:325.2
```

```
0:05:25.2
```

Times can be followed by **AM** or **PM**, but no space is permitted. **PM** adds 12 hours to the time:

```
probe 10:20PM
22:20
probe 3:32:20AM
3:32:20
```

Times are output in a standard hours, minutes, seconds, and subseconds format, regardless of how they are entered:

```
probe 0:87363.21
24:16:03.21
```

3.8.3 Access

Time values have three refinements that can be used to return specific information about the value:

Refinement	Description
/hour	Gets the value's hour.
/minute	Gets the value's minute.
/second	Gets the value's second.

Here's how to use a time value's refinements:

```
lapsed-time: 91:32:12.14
probe lapsed-time/hour
91
probe lapsed-time/minute
32
probe lapsed-time/second
12.14
```

Times with time zones can only be used with the **date!**.

3.8.4 Creation

Times can be converted using the **to-time** function:

```
probe to-time "10:30"
10:30
probe to-time [10 30]
10:30
probe to-time [0 10 30]
0:10:30
probe to-time [10 30 20.5]
10:30:20.5
```

In the previous examples, the values are not evaluated. To evaluate values as mathematical expressions, use the reduce function:

```
probe to-time reduce [10 30 + 5]
10:35
```

In various math operations involving time values, the time values, integers, or decimals are converted as shown below:

```
probe 10:30 + 1
10:30:01
probe 10:00 - 10
9:59:50
probe 0:00 - 10
-0:00:10
probe 5:10 * 3
15:30
probe 0:0:0.000000001 * 1'500'600
0:00:00.0015006
probe 8:40:20 / 4
2:10:05
probe 8:40:20 / 2:20:05
3
probe 8:40:20 // 4:20
0:00:20
```

3.8.5 Related

Use **time?** to determine whether a value is a **time!** datatype:

```
probe time? 10:30
true
probe time? 10.30
false
```

Use the **now** function with the **/time** refinement to return the current local date and time:

```
print now/time
14:42:15
```

Use the **wait** function to wait for a duration, port, or both.

If a value is a **time!** datatype, **wait** delays for that period of time. If a value is a **date!/time!**, **wait** waits until the indicated date and time. If the value is an **integer!** or **decimal!**, the function waits the indicated number of seconds. If the value is a **port**, the function will wait for an event from that port. If a block is specified, it will wait for any of the times or ports to occur. It returns the port that caused the wait to complete or returns **none** if the timeout occurred. For example,

```
probe now/time
14:42:16
wait 0:00:10
```

```
probe now/time
14:42:26
```

3.9 Tuple

3.9.1 Concept

It is common to represent version numbers, Internet addresses, and RGB color values as a sequence of three or four integers. These types of numbers are called a **tuple!** (as in quintuple) and are represented as a set of integers separated by periods.

```
1.3.0 2.1.120 1.0.2.32      ; version
199.4.80.250 255.255.255.0 ; net addresses/masks
0.80.255 200.200.60         ; RGB colors
```

3.9.2 Format

Each integer field of a **tuple!** datatype can range between 0 and 255. Negative integers produce an error.

Three to ten integers can be specified in a tuple. In the case where only two integers are given, there must be at least two periods, otherwise the value is treated as a decimal.

```
probe 1.2      ; is decimal
1.2
probe type? 1.2
decimal!
probe 1.2.3    ; is tuple
1.2.3
probe 1.2.     ; is tuple
1.2.0
probe type? 1.2.
tuple!
```

3.9.3 Creation

Use the **to-tuple** function to convert data to the **tuple!** datatype:

```
probe to-tuple "12.34.56"
12.34.56
probe to-tuple [12 34 56]
12.34.56
```

3.9.4 Related

Use **tuple?** to determine whether a value is a **tuple!** datatype.

```
probe tuple? 1.2.3.4
true
```

Use the **form** function to print a tuple as a string:

```
probe form 1.2.3.4
1.2.3.4
```

Use the **mold** function to convert a tuple into a string that can be read back into REBOL as a tuple:

```
probe mold 1.2.3.4
1.2.3.4
```

Use the **print** function to print a tuple to standard output after using the **reform** function:

```
print 1.2.3.4
1.2.3.4
```

3.10 Words

3.10.1 Concept

Words are the symbols used by REBOL. A word may or may not be a variable, depending on how it is used. Words are often used directly as symbols.

REBOL has no keywords, there are no restrictions on what words are used or how they are used. For instance, you can define your own function called **print** and use it instead of the predefined function for printing values.

There are four different formats for using words, depending on the operation required.

Action	Type Word	Type Test	Conversion
word:	set-word!	set-word?	to-set-word
:word	get-word!	get-word?	to-get-word
word	word!	word?	to-word
'word	lit-word!	lit-word?	to-lit-word

3.10.2 Format

Words are composed of alphabetic characters, numbers, and any of the following characters:

```
? ! . ' + - * & | = _ &#126;
```

A word cannot begin with a number, and there are also some restrictions on words that could be interpreted as numbers. For instance, -1 and +1 are numbers, not words.

The end of a word is marked by a space, a newline, or one of the following characters:

```
[ ] ( ) { } " : ; /
```

Thus, the square brackets of a block are not part of a word:

```
[test]
```

The following characters are not allowed in words:

```
@ # $ % ^ ,
```

Words can be of any length, but cannot extend past the end of a line.

```
this-is-a-very-long-word-used-as-an-example
```

Sample words are:

```
Copy print test

number?  time?  date!

image-files  l'image

++ -- == +-

***** *new-line*

left&right left|right
```

The REBOL language is not case-sensitive. The words following words:

```
blue

Blue

BLUE
```

all refer to the same word. The case of the word is preserved when it is printed.

Words can be reused. The meaning of a word is dependent on its context, so words can be reused in different contexts. You can reuse any word, even predefined REBOL words. For instance, the REBOL word `if` can be used in your code differently than how it is used by the REBOL interpreter.

3.10.3 Creation

The `to-word` function converts values to the `word!` datatype.

```
probe to-word "test"
test
```

The `to-set-word` function converts values to the `set-word!` datatype.

```
probe make set-word! "test"
test:
```

The **to-get-word** function converts values to the **get-word!** datatype.

```
probe to-get-word "test"
:test
```

The **to-lit-word** function converts values to the **lit-word!** datatype.

```
probe to-lit-word "test"
'test
```

3.10.4 Related

Use **word?**, **set-word?**, **get-word?**, and **lit-word?** to test the datatype.

```
probe word? second [1 two "3"]
true
if set-word? first [word: 10] [print "it is set"]
it is set
probe get-word? second [pr: :print]
true
probe lit-word? first ['foo bar]
true
```

Errors

1. Overview
2. Error Categories
    2.1 Syntax Errors
    2.2 Script Errors
    2.3 Math Errors
    2.4 Access Errors
    2.5 User Errors
    2.6 Internal Errors
3. Catching Errors
4. Error Object
5. Generating Errors
6. Error Messages
    6.1 Syntax Errors
    6.2 Script Errors
    6.3 Access Errors
    6.4 Internal Errors

1. Overview

Errors are exceptions that occur when certain irregular conditions occur. These conditions range from syntax errors to file or network access errors. Here are a few examples:

```
12-30
** Syntax Error: Invalid date -- 12-30.
** Where: (line 1) 12-30
1 / 0
** Math Error: Attempt to divide by zero.
** Where: 1 / 0
read %nofile.r
** Access Error: Cannot open /example/nofile.r.
** Where: read %nofile.r
```

Errors are processed within the system as values of the **error!** datatype. An error is an object that, if evaluated, will print an error message and halt. You can also catch errors and handle them in your script. Errors can be passed to functions, returned from functions, and assigned to variables.

2. Error Categories

There are several categories of errors.

2.1 Syntax Errors

Syntax errors occur when a script uses REBOL syntax incorrectly. For instance, if a closing bracket is missing or a string is missing its closing quote, a syntax error will occur. These errors only occur during the load or evaluation of a file or string.

2.2 Script Errors

Script errors are general run-time errors. For instance, an invalid argument to a function will cause a script error.

2.3 Math Errors

Math errors occur when a math operation cannot be processed. For instance, when attempting to divide by zero an error will occur.

2.4 Access Errors

Access errors occur when a problem occurs with a file, port or network access. For example, an access error will occur when attempting to read a file that does not exist.

2.5 User Errors

User errors are generated explicitly by a script by creating an error value and returning it.

2.6 Internal Errors

Internal errors occur within the REBOL interpreter.

3. Catching Errors

You can catch errors with the **try** function. The **try** function is similar to the **do** function. It evaluates a block, but always returns a value, even when an error occurs.

When no error occurs, **try** returns the value of a block. For example:

```
print try [100 / 10]
10
```

When an error occurs, try returns the error. If you write:

```
print try [100 / 0]
** Math Error: Attempt to divide by zero.
** Where: 100 / 0
```

the error is returned from the **try** and the **print** function cannot handle it.

To handle errors in a script, you must prevent REBOL from evaluating the error. You can prevent an error from being evaluated by passing it to a function. For instance, the **error?** function will return true when it is passed an error:

```
print error? try [100 / 0]
true
```

You can also print the data type of the value returned from a try:

```
print type? try [100 / 0]
error!
```

The **disarm** function converts an error to an error object that can be examined. In the example below, the error variable holds an error object:

```
error: disarm try [100 / 0]
```

When an error is disarmed, it will be an object! data type, not an error! datatype. Evaluating the disarmed object will not cause an error:

```
probe disarm try [100 / 0]
make object! [
    code: 400
    type: 'math
    id: 'zero-divide
    arg1: none
    arg2: none
    arg3: none
    near: [100 / 0]
    where: none
]
```

Error values can be set to a word before they are disarmed. To set a word to an error, it must be preceded by a function that prevents the error from propagating further. For example:

```
disarm err: try [100 / 0]
```

Setting a variable enables you to access the value of the block later. The example below will print an error or non-error value:

```
either error? result: try [100 / 0] [
    probe disarm result
][
    print result
]
```

4. Error Object

The error object shown above has the structure:

```
make object! [
    code: 400
    type: 'math
    id: 'zero-divide
    arg1: none
    arg2: none
    arg3: none
    near: [100 / 0]
    where: none
]
```

Where the fields are:

code	The error code number. These are obsolete and should not be used.
type	The type field identifies the error category. It is always a word data type of syntax, script, math, access, user, and internal.
id	The id field is the name for the error as a word. It identifies the specific error that occurred within the error category.
arg1	This field holds the first argument to the error message. For instance, it may include the data type of the value that caused the error.
arg2	This field holds the second argument to the error message.
arg3	This field holds the third argument to the error message.
near	The near field is a code fragment that shows where the error occurred.
where	The where field is reserved.

You can write code that checks any of the error object fields. In this example, the error is printed only when the error id indicates a divide by zero error:

```
error: disarm try [1 / 0]
if error/id = 'zero-divide [
    print {It is a Divide by Zero error}
]
It is a Divide by Zero error
```

The error id word also provides the error block that will be printed by the interpreter. For example:

```
error: disarm try [print zap]
probe get error/id
[:arg1 "has no value"]
```

This block is defined by the system/errors object.

5. Generating Errors

User errors can be generated. The simplest way to generate an error is **make** it. Here is an example:

```
make error! "this is an error"
** User Error: this is an error.
** Where: make error! "this is an error"
```

Any of the existing errors can be generated by making the error with a block argument. This block contains the error category name and the error message id name. If the error requires arguments, the arguments follow the message id name. The arguments are what define the `arg1`, `arg2` and `arg3` values in the error object. Here is an example:

```
make error! [script expect-set series! number!]
** Script Error: Expected one of: series! - not: number!.
** Where: make error! [script expect-set series! number!]
```

Custom errors can be entered into the `system/error` object's `user` category. This is done by making a new user category with new entries. These entries are used when generating errors. For instance, the

following example enters an error into the user category:

```
system/error/user: make system/error/user [
    my-error: "a simple error"
]
```

Now an error can be generated using the `my-error` message id:

```
if error? err: try [
    make error! [user my-error]
] [probe disarm err]
make object! [
    code: 803
    type: 'user
    id: 'my-error
    arg1: none
    arg2: none
    arg3: none
    near: [make error! [user my-error]]
    where: none
]
```

To create more informative errors, define an error that uses data available when it is generated. This data is included in the disarmed error object and printed as part of the error message. For instance, to use all three argument spaces in an error object:

```
system/error/user: make system/error/user [
    my-error: [:arg1 "doesn't go into" :arg2 "using" :arg3]
]

if error? err: try [
    make error! [user my-error [this] "that" my-function]
] [probe disarm err]
make object! [
    code: 803
    type: 'user
    id: 'my-error
    arg1: [this]
    arg2: "that"
    arg3: 'my-function
    near: [make error! [user my-error [this] "that" my-function]]
    where: none
]
```

The error message generated for `my-error` can be printed without stopping the script:

```
disarmed: disarm err
print bind (get disarmed/id) (in disarmed 'id)
this doesn't go into that using my-function
```

A new library category may be created if there is a need to group a series of errors together by making a new category in `system/error:`

```
system/error: make system/error [
    my-errors: make object! [
        code: 1000
        type: "My Error Category"
        error1: "a simple error"
        error2: [:arg1 "doesn't go into" :arg2 "using" :arg3]
    ]
]
```

The **type** defined in the error object will be the error type printed when the error is generated. The following example illustrates generating an error from both `error1` and `error2` in the `my-error` category.

Generating an error from `error1`. This error requires no arguments:

```
disarmed: disarm try [make error! [my-errors error1]]
print get disarmed/id
a simple error
```

Generating an error from `error2` requires three arguments:

```
disarmed: disarm try [
make error! [my-errors error2 [this] "that" my-function]]
print bind (get disarmed/id) (in disarmed 'id)
this doesn't go into that using my-function
```

Finally, the description that returns the errors defined in `my-errors` may be obtained with:

```
probe get in get disarmed/type 'type
My Error Category
```

6. Error Messages

Listed below is a list of all errors defined in the system/error object error catalog.

6.1 Syntax Errors

6.1.1 invalid

Data could not be translated into a valid REBOL datatype. In other words, a malformed value was evaluated.

Message:

```
["Invalid" :arg1 "--" :arg2]
```

Example:

```
filter-error try [load "1024AD"]
** Syntax Error: Invalid integer -- 1024AD
** Where: (line 1) 1024AD
```

6.1.2 missing

A block, string or paren expression was left unclosed.

Message:

```
["Missing" :arg2 "at" :arg1]
```

Example:

```
filter-error try [load "("]
** Syntax Error: Missing ) at end-of-script
** Where: (line 1) (
```

6.1.3 header

An attempt was made to evaluate a file as a REBOL script and the file did not have a REBOL header.

Message:

```
Script is missing a REBOL header
```

Example:

```
write %no-header.r {print "data"}
filter-error try [do %no-header.r]
** Syntax Error: Script is missing a REBOL header
** Where: do %no-header.r
```

6.2 Script Errors

6.2.1 no-value

An attempt was made to evaluate an undefined word.

Message:

```
[:arg1 "has no value"]
```

Example:

```
filter-error try [undefined-word]
** Script Error: undefined-word has no value
** Where: undefined-word
```

6.2.2 need-value

An attempt was made to define a word to nothing. A set-word was used without an argument.

Message:

```
[:arg1 "needs a value"]
```

Example:

```
filter-error try [set-to-nothing:]
** Script Error: set-to-nothing needs a value
** Where: set-to-nothing:
```

6.2.3 no-arg

A function was evaluated without providing it with all the arguments it was expecting.

Message:

```
[:arg1 "is missing its" :arg2 "argument"]
```

Example:

```
f: func [b][probe b]
filter-error try [f]
** Script Error: f is missing its b argument
** Where: f
```

6.2.4 expect-arg

A function was provided an argument of a datatype it wasn't expecting.

Message:

```
[:arg1 "expected" :arg2 "argument of type:" :arg3]
```

Example:

```
f: func [b [block!]][probe b]
filter-error try [f "string"]
** Script Error: f expected b argument of type: block
```

```
** Where: f "string"
```

6.2.5 expect-set

Two series values were used together in a way that was not compatible. For instance, when trying to do a **union** between a string and a block.

Message:

```
["Expected one of:" :arg1 "- not:" :arg2]
```

Example:

```
filter-error try [union [a b c] "a b c"]
** Script Error: Expected one of: block! - not: string!
** Where: union [a b c] "a b c"
```

6.2.6 invalid-arg

This is a generic error for handling values that were used improperly. For instance, when a set-word is used inside of a function's specification block.

Message:

```
["Invalid argument:" :arg1]
```

Example:

```
filter-error try [f: func [word:][probe word]]
** Script Error: Invalid argument: word
** Where: func [word:] [probe word]
```

6.2.7 invalid-op

An attempt was made to use an operator that had been redefined. The operator used is no longer a valid operator.

Message:

```
["Invalid operator:" :arg1]
```

Example:

```
*: "operator redefined to a string"
filter-error try [5 * 10]
** Script Error: Invalid operator: *
```

```
** Where: 5 * 10
```

6.2.8 no-op-arg

A math or comparison operator was used without providing the second argument.

Message:

```
Operator is missing an argument
```

Example:

```
filter-error try [1 +]
** Script Error: Operator is missing an argument
** Where: 1 +
```

6.2.9 no-return

A function expecting a block to return a value did not return anything. For instance, when using the **while** or **until** function.

Message:

```
Block did not return a value
```

Examples:

```
filter-error try [ ; first block returns nothing
    while [print 10][probe "ten"]
]
10
** Script Error: Block did not return a value
** Where: while [print 10] [probe "ten"]
filter-error try [
    until [print 10] ; block returns nothing
]
10
** Script Error: Block did not return a value
** Where: until [print 10]
```

6.2.10 not-defined

A word used was not defined within any context.

Message:

```
[:arg1 "is not defined in this context"]
```

6.2.11 no-refine

An attempt was made to use a function refinement that didn't exist for that function.

Message:

```
[:arg1 "has no refinement called" :arg2]
```

Example:

```
f: func [/a] [if a [print "a"]]
filter-error try [f/b]
** Script Error: f has no refinement called b
** Where: f/b
```

6.2.12 invalid-path

An attempt was made to access a block or object value using a path that did not exist within that block or object.

Message:

```
["Invalid path value:" :arg1]
```

Example:

```
blk: [a "a" b "b"]
filter-error try [print blk/c]
** Script Error: Invalid path value: c
** Where: print blk/c
obj: make object! [a: "a" b: "b"]
filter-error try [print obj/d]
** Script Error: Invalid path value: d
** Where: print obj/d
```

6.2.13 cannot-use

An attempt was made to perform an operation on a value of an incompatible datatype. For instance, when attempting to add a string to a number.

Message:

```
["Cannot use" :arg1 "on" :arg2 "value"]
```

Example:

```
filter-error try [1 + "1"]
** Script Error: Cannot use add on string! value
** Where: 1 + "1"
```

6.2.14 already-used

An attempt was made to **alias** a word that had already been aliased.

Message:

```
["Alias word is already in use:" :arg1]
```

Example:

```
alias 'print "prink"
filter-error try [alias 'probe "prink"]
** Script Error: Alias word is already in use: prink
** Where: alias 'probe "prink"
```

6.2.15 out-of-range

An attempt was made to modify an invalid series index.

Message:

```
["Value out of range:" :arg1]
```

Example:

```
blk: [1 2 3]
filter-error try [poke blk 5 "five"]
** Script Error: Value out of range: 5
** Where: poke blk 5 "five"
```

6.2.16 past-end

An attempt was made to access series data beyond the length of the series.

Message:

```
Out of range or past end
```

Example:

```
blk: [1 2 3]
filter-error try [print fourth blk]
** Script Error: Out of range or past end
** Where: print fourth blk
```

6.2.17 no-memory

The system ran out of memory while trying to complete an operation.

Message:

```
Not enough memory
```

6.2.18 wrong-denom

A math operation was performed on money values of two different denominations. For instance, when trying to add USD$1.00 to DEN$1.50.

Message:

```
[:arg1 "not same denomination as" :arg2]
```

Example:

```
filter-error try [US$1.50 + DM$1.50]
** Script Error: US$1.50 not same denomination as DM$1.50
** Where: US$1.50 + DM$1.50
```

6.2.19 bad-press

An attempt was made to decompress a binary value that was corrupt or not a compressed format.

Message:

```
["Invalid compressed data - problem:" :arg1]
```

Example:

```
compressed: compress {some data}
change compressed "1"
filter-error try [decompress compressed]
** Script Error: Invalid compressed data - problem: -3
** Where: decompress compressed
```

6.2.20 bad-port-action

An attempt was made to perform an unsupported action on a port. For instance, when trying to use `find` on a TCP port.

Message:

```
["Cannot use" :arg1 "on this type port"]
```

6.2.21 needs

An attempt was made to run a script that needed either a new version of REBOL or a file that couldn't be found. This information will be found in the script's REBOL header.

Message:

```
["Script needs:" :arg1]
```

6.2.22 locked-word

An attempt was made to modify a protected word. The word will have been protected with the `protect` function.

Message:

```
["Word" :arg1 "is protected, cannot modify"]
```

Example:

```
my-word: "data"
protect 'my-word
filter-error try [my-word: "new data"]
** Script Error: Word my-word is protected, cannot modify
** Where: my-word: "new data"
```

6.2.23 dup-vars

A function was evaluated that had multiple occurrences of a word defined in its specification block. For instance, if the word `arg` was defined as both argument one and two.

Message:

```
["Duplicate function value:" :arg1]
```

Example:

```
filter-error try [f: func [a /local a][print a]]
** Script Error: Duplicate function value: a
** Where: func [a /local a] [print a]
```

6.3 Access Errors

6.3.1 cannot-open

A file could not be accessed. This could be a local or network file. Most common reason for this error is a nonexistent directory.

Message:

```
["Cannot open" :arg1]
```

Example:

```
filter-error try [read %/c/path-not-here]
** Access Error: Cannot open /c/path-not-here
** Where: read %/c/path-not-here
```

6.3.2 not-open

An attempt was made to use a port that was closed.

Message:

```
["Port" :arg1 "not open"]
```

Example:

```
p: open %file.txt
close p
filter-error try [copy p]
** Access Error: Port file.txt not open
** Where: copy p
```

6.3.3 already-open

An attempt was made to **open** a port that was already open.

Message:

```
["Port" :arg1 "already open"]
```

Example:

```
p: open %file.txt
filter-error try [open p]
** Access Error: Port file.txt already open
** Where: open p
```

6.3.4 already-closed

An attempt was made to **close** a port that had already been closed.

Message:

```
["Port" :arg1 "already closed"]
```

Example:

```
p: open %file.txt
close p
filter-error try [close p]
** Access Error: Port file.txt not open
** Where: close p
```

6.3.5 invalid-spec

An attempt was made to create a port with **make** using a specification that a port could not be built from.

Message:

```
["Invalid port spec:" :arg1]
```

Example:

```
filter-error try [p: make port! [scheme: 'naughta]]
** Access Error: Invalid port spec: scheme naughta
** Where: p: make port! [scheme: 'naughta]
```

6.3.6 socket-open

The operating system ran out of sockets to allocate.

Message:

```
["Error opening socket" :arg1]
```

6.3.7 no-connect

A connection to another host failed. This is generic error covering a range of reasons for the connection failure. When more information is known about the reason for the connection failure, a more specific error will be thrown.

Message:

```
["Cannot connect to" :arg1]
```

Example:

```
filter-error try [read http://www.host.dom/]
```

```
** Access Error: Cannot connect to www.host.dom
** Where: read http://www.host.dom/
```

6.3.8 no-delete

An attempt was made to **delete** a file that was either locked or protected.

Message:

```
["Cannot delete" :arg1]
```

Example:

```
p: open %file.txt
filter-error try [delete %file.txt]
** Access Error: Cannot delete file.txt
** Where: delete %file.txt
```

6.3.9 no-rename

An attempt was made to **rename** a file that was either locked or protected.

Message:

```
["Cannot rename" :arg1]
```

Example:

```
p: open %file.txt
filter-error try [rename %file.txt %new-name.txt]
** Access Error: Cannot rename file.txt
** Where: rename %file.txt %new-name.txt
```

6.3.10 no-make-dir

An attempt was made to create a directory in a file path that did not exist or was write protected.

Message:

```
["Cannot make directory" :arg1]
```

Example:

```
filter-error try [make-dir %/c/no-path/dir]
** Access Error: Cannot make directory /c/no-path/dir/
** Where: m-dir path return path
```

6.3.11 timeout

The timeout period elapsed while waiting to for a response from another host. This timeout is set in the port's `timeout` attribute.

Message:

```
Network timeout
```

6.3.12 new-level

An attempt was made within a script to change the security to a lower level of security that was denied. This is to say, whenever a script requests a lower security setting and the user denies the request, this error is thrown.

Message:

```
["Attempt to change security level to" :arg1]
```

Example:

```
secure quit
filter-error try [secure none] ; denied request

secure none
```

6.3.13 security

A security violation occurred. This will happen when an attempt is made to access a file or the network when the **`secure`** setting is set to `throw`.

Message:

```
REBOL - Security Violation
```

Example:

```
secure throw
filter-error try [open %file.txt]
** Access Error: REBOL - Security Violation
** Where: open %file.txt
secure none
```

6.3.14 invalid-path

A malformed file path was used.

Message:

```
["Bad file path:" :arg1]
```

Example:

```
filter-error try [read %/]
```

6.4 Internal Errors

6.4.1 bad-path

A path was evaluated that began with an invalid word.

Message:

```
["Bad path:" arg1]
```

Example:

```
path: make path! [1 2 3]
filter-error try [path]
** Internal Error: Bad path: 1
** Where: path
```

6.4.2 not-here

An attempt was made to use a REBOL/Command or REBOL/View feature from REBOL/Core.

Message:

```
[arg1 "not supported on your system"]
```

6.4.3 stack-overflow

The system's memory stack overflowed while trying to perform an operation.

Message:

```
["Stack overflow"]
```

Example:

```
call-self: func [][call-self]
filter-error try [call-self]
** Internal Error: Stack overflow
** Where: call-self
```

6.4.4 globals-full

The maximum allowable number of defined global words has been exceeded.

Message:

```
["No more global variable space"]
```

Console

1. Command Prompt
2. Result Indicator
3. History Recall
4. Busy Indicator
5. Advanced Console Operations
    5.1 Keyboard Input Sequences
    5.2 Terminal Output Sequences

1. Command Prompt

The default command line prompt is ">>". You can change the prompt with code such as:

```
system/console/prompt: "Input: "
```

The prompt then becomes:

```
Input:
```

The prompt can be a block that is evaluated each time. This line prints the current time:

```
system/console/prompt: [reform [now/time " >> "]]
```

This would result in a prompt of:

```
10:30 >>
```

2. Result Indicator

The default result indicator is "==" and can be modified with a line such as:

```
system/console/result: "Result: "
```

These settings can be placed in the `user.r` file to make them permanent.

3. History Recall

Each line typed into REBOL at the prompt is stored in a history block, and it can be recalled later using the up and down arrow keys. For instance, pressing the up arrow once recalls the prior

input line.

The history block containing all input lines is accessed from the system console object:

```
probe system/console/history
```

You can save the history block as a file:

```
save %history.r system/console/history
```

and it can be reloaded later with:

```
system/console/history: load %history.r
```

These lines can be put in the `user.r` file to save and reload your history between REBOL sessions.

4. Busy Indicator

When REBOL waits for a network operation to complete, a busy indicator appears on screen to indicate that something is happening. You can change the indicator with a line like:

```
system/console/busy: "123456789-"
```

Whe REBOL is running in quiet mode, te busy indicator will not be displayed.

5. Advanced Console Operations

The console provides "virtual terminal" capability that allows you to perform operations such as cursor movement, cursor addressing, line editing, screen clearing, control key input, and cursor position querying.

The console control sequences follow the ANSI standard. These features provide you with the capability to write your own platform-independent terminal programs such as text editors, email clients, or telnet emulators.

The console features apply to both input and output. On input, function keys will be converted to multiple-character escape sequences. On output, multiple-character escape sequences can be used to control the display of text in the console window. Both the input and output sequences begin with the ANSI escape character, 27 decimal (1B hex). The next character in the sequence indicates the control keys on input or the terminal control operation on output.

The ANSI control characters are case-sensitive and normally require an upper case character.

5.1 Keyboard Input Sequences

The input sequences for function keys are listed in the table below (on systems and shells that support them, such as Linux, BSD, etc.)

To receive these sequences as a stream of unprocessed characters, disable the input port line handling mode:

```
set-modes system/ports/input [lines: false]
```

Now you can get input from the port (with COPY or READ-IO) or use a function like INPUT to get each character:

```
while [
    code: input
    code <> 13   ; ENTER
][
    probe code
]
```

Here are a few of the common ANSI input function codes:

Function Key	As Escape Code	As REBOL Block
F1	ESC O P	[27 79 80]
F2	ESC O Q	[27 79 81]
F3	ESC O R	[27 79 82]
F4	ESC O S	[27 79 83]
F5	ESC [1 5 ~	[27 91 49 53 126]
F6	ESC [1 7 ~	[27 91 49 55 126]
F7	ESC [1 8 ~	[27 91 49 56 126]
F8	ESC [1 9 ~	[27 91 49 57 126]
F9	ESC [2 0 ~	[27 91 50 48 126]
F10	ESC [2 1 ~	[27 91 50 49 126]
F11	ESC [2 2 ~	[27 91 50 50 126]
F12	ESC [2 3 ~	[27 91 50 51 126]
Home	ESC [1 ~	[27 91 49 126]
End	ESC [4 ~	[27 91 52 126]
Page-up	ESC [5 ~	[27 91 53 126]
Page-down	ESC [6 ~	[27 91 54 126]
Insert	ESC [2 ~	[27 91 50 126]
Up	ESC [A	[27 91 65]
Down	ESC [B	[27 91 66]
Left	ESC [D	[27 91 68]
Right	ESC [C	[27 91 67]

5.2 Terminal Output Sequences

There are several variations in the terminal control output character sequences. Some command codes are preceded by a number (sent in ASCII) indicating that the operation is to be performed the specified number of times. For example the cursor motion command may be preceded by two numbers separated by a semicolon to indicate the row and column position to move to. The cursor command characters (upper case required) are included in the following table:

Output Sequence	Description
(1B)	Use this escape code prior to the following codes
D	Moves cursor one space left
C	Moves cursor one space right
A	Moves cursor one space up
B	Moves cursor one space down
n D	Moves cursor n spaces left
n C	Moves cursor n spaces right
n A	Moves cursor n spaces up
n B	Moves cursor n spaces down
r ; c H	Moves cursor to row r, column c*
H	Moves cursor to top left corner (home)*
P	Deletes one character to the right at current location
n P	Deletes n characters to the right at current location
@	Inserts one blank space at current location
n @	Inserts n blank spaces at current location
J	Clears screen and moves cursor to top left corner (home)*
K	Clears from current position to end of current line
6n	Places the current cursor position in the input buffer
7n	Places screen dimensions in the input buffer

The top left corner is defined as row 1, column 1

The following example moves the cursor to the right ten spaces:

```
print "^(1B)[10CHi!"
Hi
```

This example moves the cursor to the left seven spaces and clears the remainder of the line:

```
cursor: func [parm [string!]][join "^(1B)[" parm]
print ["How are you" cursor "7D" cursor "K"]
How a
```

To find the current console window size, you can use this example:

```
cons: open/binary [scheme: 'console]

print cursor "7n"
screen-dimensions: next next to-string copy cons
33;105R
close cons
```

The above example opens the console, sends a control character to the input buffer and copies the return value. It reads the value (screen dimensions) that is returned after the control

character and closes the console. The return value is the height and width separated by a semicolon (;) and followed by an R. In the above example, the screen is 33 high by 105 wide.

Autoscroll

Printing a character to the bottom-right corner of some terminals will cause a new line, which will scroll the screen. Others will not. This inconsistency between console terminal types must be considered when writing REBOL scripts intended to be cross-platform.