Creating lightweight cross-platform Applications

REBOL Essentials

Viktor Pavlu
12-Dec-2002

I. REBOL language tutorial

What is REBOL?

REBOL is a free, cross platform, highly reflective, flexible, compact, interpreted language that optimally fits the needs of daily programming tasks – especially network/Internet related tasks. REBOL was designed by Carl Sassenrath, the software architect responsable for the Amiga OS. The first REBOL release was in 1997 and since it has experienced many improvements. This year REBOL is even listed as nominee for the Webby avards for technical achievement, nevertheless it's still rarely known.
REBOL stands for "Relative Expression Based Object Language". Let's look at the terms in this paragraph in more detail:

free

REBOL is not free in terms of "Free Software" (www.fsf.org), but it's free in that you don't have to pay for the interpreter as long as you don't want to sell your programs.

cross platform

Currently interpreters for 42 platforms exist. Scripts designed for Win32 can also be run on a UNIX platform (or on the other platforms for which an interpreter exists) without modification.

highly reflective

the specification of all functions (and other words) can be obtained and manipulated during run-time.

flexible

Everything in REBOL is a "word". There are no differences between control structures, functions, variables and so on like there are in most other languages. For example you could redefine the word IF that it no longer acts as the conditional expression we are used to.

compact

The interpreter for the /Core language weighs in at 250KB, the graphical interpreter /View is about 500KB in size and even more compact versions exist.

interpreted

REBOL programs are not compiled to binary instruction codes but rather remain in their source form. The interpreter takes this source code and executes it.
In recent times REBOL technologies (the company behind REBOL) developed a REBOL compiler. This is not a *real* compiler per definition in that it takes the source and translates it to binary instruction codes but rather a program that produces a standalone interpreter that includes a encapsulated version of your source which still remains interpreted.

optimally fits daily Internet programming tasks

Interacting with the Web is very easy:

```
page: read http://www.htl-tex.ac.at/
send vpavlu@plain.at page
```

This two line example reads a document from the WWW and sends it to the given email address.

relative expression

The words in REBOL (everything, as we already know (see flexible)) have special meanings depending on the context in which they are. `copy` used with a string, makes a copy of the string, whereas `copy` used with a port does not replicate the port but retrieves it's currently available data. More on the details of strings and ports later – just remember that there is no single defined meaning for a word but rather a unlimited set of things a word can stand for, depending on context.

Carl Sassenrath about REBOL

[...] REBOL is not a traditional computer language like C, BASIC, or Java. Instead, REBOL was designed to solve one of the fundamental problems in computing: the exchange and interpretation of information between distributed computer systems. REBOL accomplishes this through the concept of relative expressions (which is how REBOL got its name as the Relative Expression-Based Object Language). Relative expressions, also called "dialects", provide greater efficiency for representing code as well as data, and they are REBOL's greatest strength. For example, REBOL can not only create a graphical user interface in one line of code, but it can also send that line as data to be processed and displayed on other Internet computer systems around the world.

The ultimate goal of REBOL is to provide a new architecture for how information is stored, exchanged, and processed between all devices connected over the Internet. Unlike other approaches that require tens of megabytes of code, layers upon layers of complexity that run on only a single platform, and specialized programming tools, REBOL is small, portable, and easy to manage.[...]

-- Carl Sassenrath

Versions

Currently three versions of REBOL exist:

- **/Core** The core language. Console version, *free*
- **/View** Extends /Core with GUI abilities, *free*
- **/Command** "Server" edition. Provides access to the underlying System, offers database connectivity, FastCGI support and RSA encryption among other features.
- **/View/Pro** Adds sound to **/View**

In recent times there were so called REBOL kernels developed. That is smaller versions of the interpreter which only implement the most critical functions of the language. This results in reduced overhead and much faster startup times as you only include the words you know you are going to use.

- **/Base** Kernel that implements **/Core** functionality
- **/Pro** Adds command features to **/Base**
- **/Face** Adds graphics and sound to **/Pro**

Furthermore there is the **REBOL/SDK** to be released this week (12-Dec-2002). Not a real REBOL version, rather a kit of development tools comprising the kernels, the "compiler" (which is called **/Encap**) and **PREBOL**, REBOLs preprocessor.

REBOL/IOS is not part of the language tools but an application based on REBOL offered by REBOL technologies that enables its users to exchange data, co-work on projects and simultaneously use REBOL programs.

Read more about the REBOL language in general at
http://www.rebol.com/index-lang.html
http://www.rebolforces.com/
http://www.codeconscious.com/rebol/

Running your first program

Setup

In the first part of this text we only look at the core functionality until we get a reasonable grasp of REBOL. So the free /Core interpreter will suite perfectly for our needs. If you want to download /View instead of /Core, that's ok but you won't experience any advantages over /Core users.
Get a copy of the interpreter for you platform from www.rebol.com and start it. Answer the questions and we are done with setting up.
If you are experiencing problems with the /View setup because of limited access, close the application window with the button in the upper right corner – the installation will quit but leave you a REBOL console capable of /View commands.

Get the User Guide

Download the REBOL/Core User Guide (http://www.rebol.com/docs/core23/rebolcore.html). A great resource if you have to look something up. Reading the whole book takes a while – I know, i did. But to start working with REBOL you don't have to do it – this brief tutorial should suffice.

Try this...

Open the interpreter and try some REBOL snippets. >> is the console prompt and mustn't be entered.

```
>> print "Hello, world"

>> str1: "Hello,"
>> str2: "world"
>> print [str1 str2]

>> loop 10 [prin "*"]

>> loop 10 [print "no tv and no beer make homer go crazy"]
```

`prin` is not a typo. It does exactly what `print` does: printing a text to the console. But `prin` does not automatically append a line break.

```
>> help prin
>> help print

>> i: 20
>> proc: print ["i =" i]
```

Here we have seen that a word followed by a colon as `proc:` assigns the word the following value. But when we tried to assign `print` to `proc` it failed as the interpreter immediately executed `print` and as `print` does not return a value, there is nothing for `proc` to be set to.

To give `proc` the meaning we want it to have – being a procedure that prints the value of i – we have to prevent the interpreter from immediately executing the word `print` and rather return the value `print` to `proc`. This is done by enclosing the words with square brackets.

```
>> proc: [ print ["i =" i] ]
>> source proc
>> repeat i 10 proc
```

`SOURCE` show the code that created `proc`, so now we know that `proc` hold the right value. When we put `proc` in a loop that continuously incremets I, we get the result we've asked for. Putting REBOL code in brackets prevents the interpreter from immediately executing it.

REBOL Basics

Values

The REBOL language is built from three things: values, words and blocks. In this chapter we have a close look at the values.

A value is something that stands literally there. `42` for example. A number that has the value 42. Another example would be `"that's ok, my will is gone"`. This time it was a string. One last example: `$0.79`. Money as we would guess (and we are right).

```
>> type? $0.79
== money!
```

We have seen that there are many different types of entering values literally depending on the type of data. `42` is a number whereas `"42"` would be a string. So values have different types of data or datatypes. Similiar to other languages where you have datatypes like char, int, and float. In REBOL however not the variables have the datatypes but the values themselves. This is very important.

Datatypes

Datatype	**Example**
integer	`1234`
decimal	`12.34`
string	`"REBOL world!"`
time	`15:47:02`
date	`12-December-2002`
tuple	`192.168.0.16`
money	EUR`$0.79`
pair	640x`480`
char	`#"R"`
binary	`#{ab82408b}`
email	`vpavlu@plain.at`
issue	`#ISBN-020-1485-41-9`
tag	`<img src="cover.png">`
file	`%/c/rebol/rebol.exe`
url	`http://www.plain.at/vpavlu/`
block	`[good bad ugly]`

To convert between datatypes, use one of the existing `to-`*type*`!` functions. Type

```
>> help to-
```

in the console to get an overview of conversion functions.
For a more thorough examination of different datatypes and what you can do with them skim through the chapter *Values* in the Appendix A of REBOL/Core User Guide.

Words

The second important thing in REBOL are words. Words are like variables but they go a bit further. A variable can hold a value – words can, too. In C for example, *if*, *for* and *printf()* are not a variables; you can't change the "value" of an *if* in C. In REBOL everything not being a block or a value (which stand literally there) is a word and thus can be assigned a value.

```
>> num: 12
== 12
>> if: "some string"
== "some string"
```

You have just redefined the word `IF`. This is not a good idea unless you know exactly what you are doing because from now on, at every place where there is an `IF` it no longer checks the word immediately after it for being true and if so, executing the following block (that's what if usually does: conditional evaluation) but evaluates to "some string" which will change the behaviour of programs drastically.

Words do not have datatypes. Any word can hold any value and no declaration is required. Just assign a word a value. If you try to evaluate a word that has no value assigned (that has no meaning to REBOL), the interpreter will report an error.

```
>> print foobar
** Script Error: foobar has no value
** Near: print foobar
```

Though there a no datatypes for words, there do exist different types of words. (Don't get confused with that – it's easy)

Types of Words

Type	**Example**	**Purpose**
word	`var`	evaluate to it's value (interpret the word)
get-word	`:var`	get the value behind var
set-word	`var:`	set var to a new value
lit-word	`'var`	the word literally

*Word*s return the interpreted value behind the word. If the value is a number, this yields the number. If the value is a string, this yields the string. If the value is a function, this yields the result of the executed funtion.

*Get-word*s return the value behind the word. This is similiar to the previous paragraph in many cases, however with functions for example the result differs. Not the interpreted function but the function itself is returned.

```
>> func1:  now
== 12-Dec-2002/15:21:15+1:00
>> func2: :now
>> wait 0:01 ;1 minute
>> func1 ;holds interpreted 'now
== 12-Dec-2002/15:21:15+1:00
>> func2 ;holds 'now
== 12-Dec-2002/15:22:15+1:00
```

First we assigned `FUNC1` the value of now (`NOW` returns the current date/time value), secondly we assigned `FUNC2` the value behind now (`NOW` itself). This can be proven by the following lines:

```
>> source func1
func1: 12-Dec-2002/15:21:15+1:00
>> source func2
func2: native [
    "Returns the current local date and time."
    /year "Returns the year only."
    /month "Returns the month only."
    /day "Returns the day of the month only."
    /time "Returns the time only."
    /zone "Returns the time zone offset from GMT only."
    /date "Returns date only."
    /weekday {Returns day of the week as integer}
    /precise "Use nanosecond precision"
]
```

*Set-Word*s don't need any further explaination. A world followed by a colon sets it to the following value and returns this value.

```
>> print a: "REBOL"
REBOL
>> a
== "REBOL"
```

*Lit-Word*s are a way to literally specify a word. The words name itself is the value of a *lit-word*.

```
>> dump: func [ word ][
       either value? word [
           print [ word "is" get word ]
       ][
           print [ word "is undefined" ]
       ]
   ]

>> a: 42
== 42

>> dump 'a
a is 42
>> dump 'b
b is undefined
```

Here we passed the *lit-word*s to a function that tests whether a word is defined (has a value).

```
>> set 'name "REBOL"  ;same as name: "REBOL"
>> get 'name          ;same as :name
```

Unsetting a Word

By unsetting a word you take the previously assigned value from it. The value of the word is from then on undefined. Evaluating unset words yields an error.

```
>> word: $100
== $100.00
>> print word
$100.00
>> value? 'word
== true
>> unset 'word
>> value? 'word
== false
>> print word
** Script Error: word has no value
** Near: print word
```

Protecting a Word

If a word is protected, trying to assign it a new value produces an error. This can be used to prevent some words from being mistakenly redefined. It is, however, no guarantee that none of your functions can change it's value because a call to `UNPROTECT` makes the word accept values again.

```
>> chr: #"R"
== #"R"
>> protect 'chr
>> chr: #"A"
** Script Error: Word chr is protected, cannot modify
** Near: chr: #"A"
>> unprotect 'chr
>> chr: #"A"
== #"A"
```

Blocks

The third thing used in REBOL among values and words are blocks. This chapter introduces Blocks in a short manner – more detail follows in the chapter *Series!*.

As we already saw in the introductory example, blocks are made of square brackets with zero or more elements inside and the elements inside the block are prevented from evaluation. Blocks can be of any size and depth and their elements of any type.

```
>> colors: [red green blue]
== [red green blue]
>> data: [now/date colors [colors $12] 4]
== [now/date colors [colors $12.00] 4]
```

All of them are valid blocks. The first one consists of three (maybe undefined) words. That the words might be undefined is not a problem because the interpreter does not look inside the block until you tell to. This is sometimes required – as in the fourth line where we want to have the previously defined blocks as elements of this block, rather than the words.

```
>> do [now/date colors [colors $12] 4]
== 4
>> data: reduce [now/date colors [colors $12] 4]
== [12-Dec-2002 [red green blue] [colors $12.00] 4]
```

`DO` evaluates the block and returns the last resulting value. `REDUCE` also interprets the block but returns all results in a new block. This is often needed to pass complex arguments to functions.
Both words tell the interpreter to do evaluation inside the given block. If this block contains further blocks however, they are not evaluated. That's why the *colors* inside the inner block are still unevaluated.

```
>> compose [ now/date (now/date) ]
== [now/date 12-Dec-2002]
```

`compose` is a reduce limited to values inside parentheses which is sometimes useful to create blocks that contain code and data.

Word	**Example**	**Result**
`reduce`	`[1 2]`	evaluates block, returns block of results
`remold`	`"[1 2]"`	returns a string that looks the same as the result from reduce
`reform`	`"1 2"`	reduced block converted to a string
`rejoin`	`"12"`	a string containing all results joined together
`compose`	`[1 2]`	evaluates only words in parens inside a block

Conclusion

As there are only three types of information in REBOL (values, words and blocks) used for everything from variables, control structures, functions and data – there is no real difference between code and data for REBOL. All there is are words with a predefined meaning (value) that describe the language.
And this language is subject to rest of the first part.

Control Structures

As in (almost) every other programming language there are control structures in REBOL as well. Control structures are program statements that control the flow of the program.
The following lines compare REBOLs control statemenst with those known from C++ (or related languages)

```
do [...]                        {...}
```

DO evaluates the block. Or a string, or a file, …

```
if expr [...]                   if(expr) {...}
```

The block is only executed if the expression evaluates to something true.

```
either expr [...][...]          if(expr) {...} else {...}
```

If the expression evaluates to true, the first block is executed, the second block otherwise. Note that there is no *else* in REBOL.

```
while [expr][                   while(expr){
  ...                             ...
]                               }
```

While is the only control statement that has its condition inside a block. If more than one condition is found inside the condition block, all conditions must be met in order to have the loop executed.

```
for i 1 10 2 [                  for(i=1;i<=10;i+=2){
  ...                             ...
]                               }
```

For sets the given variable to the initial value (1 here) and executes the block. Then the increment (2 here) is repeatedly added to the variable and the block executed as long as the variables value is not greater than the limit (10 here). Note that i has no value after the execution of the loop.

```
until [                         do {
  ...                             ...
  expr                            ...
]                               } while(expr);
```

Until takes the following block and keeps evaluating it as long as the last word evaluates to true.

```
loop 10 [...]                   // N/A in C++
```

Repeats the passed block 10 times.

```
repeat i 10 [...]               for(i=1;i<=10;i++) {...}
```

Increments i from 1 to 10 and evaluates the block for every i.

```
forever [...]                   while(1){...}
```

A loop that never ends. Most times a BREAK is found inside this loop so that it is left again. BREAK can be used to exit all kinds of loops.

```
switch/default var [            switch(var){
  1 [...]                          case 1:  ... break;
  2 [...]                          case 2:  ... break;
][...]                             default: ...
                                }
```

Switch compares the observed value *var* with all its labels and if one matches, the code following the label is executed. If none matches and there is a default block, that block is executed. The `/default` refinement tells the interpreter that there will be a default block. In REBOL we would express this behaviour with some code similar to this:

```
switch: func [ var cases /default case ][
     either value: select cases var [do value][
          either default [do case][none]
     ]
]
```

By entering `source switch` we can verify this assumption. The process of creating own functions is explained in the chapter *function!* later in this text.

What is true?

Every word that evaluates to something different from *false* or *none* is considered true.

```
>> if 0 [ print "this is important!" ]
this is important!
```

Logical functions to make more complex conditions are

`NOT a`	inverts the result of a
`a AND b`	logic: true if both are true, false otherwise
`a OR b`	logic: false if both are false, true otherwise
`a XOR b`	logic: true if exact one is true, false otherwise

What *AND*, *OR* and *XOR* return their two values joined using the operator (bitwise). Shortcut functions for ORing or ANDing a list of words are as follows:

`all []`	*none* on the first word that evaluates to false, last value otherwise
`any []`	returns the first value that evaluates to true, *none* otherwise

Simple Math

Mathematic expressions are strictly evaluated from left to right. No operator priority is known, so you have to enclose the things you want to compute first in parentheses.

```
>> print 5 + 5 * 4
40
>> print 5 + (5 * 4)
25
```

Note that while there is no priority among the operators, operators take precedence over functions. That is the reason why `print 5` was not the first thing to be evaluated and the maths performed on the result (which would be kind of awkward)

Mathematical functions in REBOL can be applied to a wide range of numerical datatypes which consist of Integer! (32b numbers without decimal point), Decimal! and Money! (64b floating points), Time!, Date!, Pair! and Tuple!.

Mathematical Words

Operator	Word	Purpose
`+`	`add`	two words added
`-`	`subtract`	second subtracted from first
`*`	`multiply`	two words multiplied
`/`	`divide`	first divided by second
`**`	`power`	first raised to the power of second
`//`	`remainder`	remainder of first divided by second
	`exp` *value*	e^{value}
	`log-10` *value*	$\log_{10}$ value
	`log-2` *value*	$\log_{2}$ value
	`log-e` *value*	$\log_{e}$ value, ln value
	`square-root` *value*	vvalue
	`absolute`	returns absolute value
	`negate`	changes sign of value
	`min a b`	returns lesser of two values
	`max a b`	returns bigger of two values
	`sine`	trigonometric sine in degrees
	`cosine`	trigonometric cosine in degrees
	`tangent`	trigonometric tangent in degrees
	`arcsine`	trigonometric arcsine in degrees
	`arccosine`	trigonometric arccosine in degrees
	`arctangent`	trigonometric arctangent in degrees

Comparison Functions

Operator	Word	Purpose
`=`	`equal`	true if values are equal
`==`	`strict-equal`	true if equal (case-sensitive) and of same type
	`strict-not-equal`	true if not equal (case-sensitive) or different types
`=?`	`same?`	true if referencing the same value
`<>`		true if values are different
`>`	`greater`	true if left is greater
`<`	`lesser`	true if left is lesser
`>=`	`greater-or-equal`	true if left is greater or equal
`<=`	`lesser-or-equal`	true if left is lesser or equal

Strings

Strings in REBOL are a one of the series! datatypes which is covered later in more detail. To get a better grasp of what strings are about wait for the series! chapter. For now it's sufficient to know that strings are written enclosed in "double quotes" or {curly braces} and to have a look at these functions

`trim` *str*	remove surrounding whitespace
`uppercase` *str*	convert to UPPERCASE
`lowercase` *str*	convert to lowercase
`compress` *source*	compresses a string
`decompress` *source*	decompresses a compressed string
`append` *str value*	append to a string
`length?` *str*	returns lenght of string

Special Characters

`^"`	`"`
`^}`	`}`
`^^`	`^`
`^M`	carriage return
`^(line), ^/`	linefeed (=newline)
`^(tab), ^-`	tab
`^(page)`	new page
`^(back)`	backspace
`^(del)`	delete
`^(null), ^@`	\0, ASCII NULL character
`^(escape), ^(esc)`	escape character
`^(`*letter*`)`	control characters (`#"^A"` to `#"^Z"`)
`^(`*xx*`)`	ASCII char by hexadecimal number

Note also the predefined words `escape, newline, tab, crlf` and `cr`.

Exercise Programs I

This chapter offers you some easy problems you can solve with the REBOL knowledge you have acquired by now. Try to sovle some of the example problems. Source code of sample solutions for all programs can be found in the appendix or online at www.plain.at/vpavlu/REBOL/examples/.

Useful Functions

`read` *source*	returns the string read from source (file, url, …)
`write` *dest data*	writes data to destination (file, url, …)
`ask` *question*	prompts the user the question, returns entered string
`input`	read a line from the console
`to-integer` *value*	converts value to an integer
`to-date` *value*	converts value to a date
`to-file` *value*	converts value to a filename
`prin` *data*	prints data without line break
`print` *data*	prints data, appends line break
`foreach` *act list* `[...]`	executes the block for every element in list. act is set to the current element each time
`now`	returns current date/time

1. Save the source of http://www.rebol.com to a file named %rebol.html (%http-save.r)
2. Print the greatest of three numbers stored in a, b and c. (%abc-max.r)
3. Write a program that repeatedly asks the user for numbers and responds with the newly computed average value. (%avg-dlg.r)
4. Write a program that computes the average of a block of numbers. (%avg-blk.r)
5. Write a substring function that accepts a string and one paramater, the start offset inside the string. Provide an additional refinement called len to limit the length of the extracted substring. (%substr.r)
6. Compute the number of days since your birthday. (%age-days.r)
7. Scramble a string using ROT-13. Read the string from a textfile and print the scrambled result to the screen. Used in Newsgroups to prevent accidental reading of content. With ROT-13 characters from A to Z have numbers 1 to 26. When encrypting data, every character is replaced by the character that has its value plus 13 added. So A becomes N. If a value is beyond 26, start again at A. So N (14) plus 13 (27) would be A again. As we see, encryption and decryption is the same in ROT-13. (%rot13.r)

Working with REBOL

As REBOL is an interpreted language, programming with REBOL is somewhat different to programming in C++ or Java. It is more like a dialog with the console than constructing code which is then compiled. If you don't know how something worked, type a small example into the console to remind you or ask REBOL for help by typing `help` *word*.
Two methods of executing REBOL code exist

1. typing directly in the console – easy and best suited for one-liners
2. creating and executing scripts – use an editor to write a script and execute it from the interpreter

For the latter method you need to create a valid REBOL skript which consists of a REBOL header and some code.

```
REBOL []
;add code here
```

This is a minimalistic version of a REBOL script file with an empty header and no code. Open a new file, add the following lines and save as *hello.r*.

```
REBOL [
  title: "script example"
  author: "vpavlu"
  date: 12-Dec-2002
  version: 1.0.0
]
print "hello world"
```

Then, in the console enter

```
>> do %hello.r
Script: "script example" (12-Dec-2002)
hello world
```

and the script file is evaluated, assuming the interpreter runs in the same directory as the file was created, so it can read `%hello.r`.

Interpreter Startup

When the interpreter has finished startup, it tries to evaluate the files rebol.r and after that user.r. rebol.r is overwritten with every new release of REBOL so you shouldn't use it for your settings as they might get lost. User-defined settings can be stored in the user.r file. Your email settings for example.

```
>> set-net [ vpavlu@plain.at mail.plain.at ]
```

Information passed to Script

You can add information about a script to the header. View `probe system/standard/script` to see all valid fields for a header. If the script is run, the information from the header in the file can be accessed through `system/script/header`.

`system/script/args` arguments passed to a script via the commandline (or via drag'n drop, if a file gets dropped over your script) can be accessed through this string

`system/script/parent` holds the `system/script` object of the parent script (a script that called this one), if any

`system/script/path` the path the script is evaluated in

`system/options/home` home directory, the path where to find rebol.r and user.r

`system/options/script` the filename of initial script provided to interpreter when it was started

`system/options/path` current directory

`system/options/args` arguments passed initially to the interpreter via commandline

`system/options/do-arg` string provided by `--do` option on command line

Series!

A series is a set of values organized in a specific order. There are many series datatypes in REBOL which can all be processed with the same small set of functions. The simplest type of series is a block which we already used.

Every series in REBOL has an internal index pointing to the start of the series. When working with series this index is often changed. `find` for example searches for a given pattern and sets the index to point to the first element in the series that matches the pattern. Note that although the resulting series looks to be a completely new list as all elements before the internal index seem to be removed, it is still exactly the same series – only the actual start of the series is not longer at its head.

```
>> nums: copy [ 1 2 3 4 5 ]
== [1 2 3 4 5]
>> print nums
1 2 3 4 5
>> length? nums
== 5

>> nums: find nums 3
== [3 4 5]
>> print nums
3 4 5
>> length? nums
== 3

>> nums: head nums
== [1 2 3 4 5]
```

```
>> print nums
1 2 3 4 5
```

When saying the first value of the series you always talk of the value at the current index and not the one at the very head of the series.

Creating Series

```
>> a: "original"
>> b: a
>> append b " string"
>> print a
original string
```

Assigning series to a word is always done by reference. So the word *b* is in fact a new word pointing to the same data as *a*. If you want them to use different strings use `B: copy a`. Note that this applies to values, too. It the previous example the value *"original"* (in the first line) is changed to *"original string"* as well. To avoid unexpected behaviour, remember to use copy.

```
>> f: func [s][
  str: ""
  print append str join s ", "
]
>> loop 3 [ f "A" ]
A,
A, A,
A, A, A,

>> f: func [s][
  str: copy ""
  print append str join s ", "
]
>> loop 3 [ f "A" ]
A,
A,
A,
```

`copy` *series*	copies a series. don't forget to copy!
`array` size	creates a series with given size
`make block! l`*en*	creates a block! with given size

Retrieving Elements

`pick` *series index*	gets element at given index
`series/1`	gets element at given index
`first` *series*	gets first element (second, third, fourth, fifth as well)
`last` *series*	gets last element
`copy/part` *series nElem*	returns copy of first nElem elements

Modifying Elements

Be careful with modifying elements in a list that is referenced by more than one word as both words are pointing to the same data.

```
>> str: "this is a long string"
== "this is a long string"
>> pos: find str "long"
== "long string"
>> remove/part str 5
== "is a long string"
>> pos
== "string"
```

With `change` you can overwrite the element at the current index with a new value. If the new value is itself a series, all the elements are used to overwrite values in the list, starting at the current index.

```
>> nums: [1 2 3]
== [1 2 3]
>> print nums
1 2 3
>> change nums 3
== [2 3]
>> print nums
3 2 3
>> change nums [5 4]
== [3]
>> print nums
5 4 3
```

`insert` *series value*	inserts at current position
`append` *series value*	inserts at end
`change` *series value*	changes first value in series to given value
`poke` *series index value*	changes the element at (current index + index) to value
`replace` *series search replace*	searches for a value and replaces it
`remove` *series*	removes at current index
`clear` *series*	removes all elements

Traversing Series

Modify the internal index to traverse over a series. This is done with the following functions.

`next` *series*	returns series at next element
`back` *series*	returns series at previous element
`at` *series offset*	returns series at given offset (+/-) relative to index
`skip` *series offset*	returns series after given offset (+/-) relative to index

`head` *series*	returns series at very beginning
`tail` *series*	returns series at end (after last element)

```
>> nums: [1 2 3]
== [1 2 3]
>> while [not tail? nums][
     print nums/1
     nums: next nums
   ]
1
2
3
== []
>> empty? nums
== true
>> print nums

>> nums: head nums
== [1 2 3]
>> empty? nums
== false
>> print nums
1 2 3
```

Keep two things in mind when iterating over series: First, the functions listed above do not modify the internal index, they just return the series with modified index, so storing the result is required (see bold line). And second, after iterating over a series you are at the end and the series seems empty, so go back to the head.

There are also predefined words for this kind of loop

`forall` *series []*	does same as loop above
`forskip` *series nElem []*	iterates over a series, skipping nElem elements
`foreach` *word series []*	iterates over series, word holds current element

`Foreach` is different to the other two functions. The current element needn't be accessed through `series/1` but is stored in word each time the block executes and the internal index is not at the end after running a `foreach` loop.

Other Series! Functions

`join` *val1 val2*	returns the two values joined together
`form` *value*	returns value converted to a string
`mold` *value*	returns a REBOL readable form of value (easy to load)
`do` *block*	evalutates block, last value returned
`reduce` *block*	evaluates block, block returned
`rejoin, reform, remold`	evaluates block, join/form/mold applied to result
`sort` *series*	sorts a series
`reverse` *series*	reverses order of series
`find` *series value*	returns series at position of value or none
`select` *series value*	returns the value next to the given value
`switch` *series value*	does the value next to the given value
`length?` *series*	returns number of elements
`tail?, empty?` *series*	return true if series is at is empty (= is at its tail)
`index?` *series*	returns offset inside series
`unique` *series*	duplicates removed
`intersect` *seriesA seriesB*	values that occur in both series
`union` *seriesA seriesB*	series joined, duplicates removed
`exclude` *seriesA seriesB*	seriesA without values in seriesB
`difference` *seriesA seriesB*	values not in both series

Function!

A function is an optionally parametrized set of instructions that returns exactly one value. We already kept instructions in a block for later execution. This can be said to be a simple form of a function with no parameters

```
>> i: 7
>> dump-i: [ print ["i =" i] ]
>> do dump-i
i = 7
```

`dump-i` is not a real function, though as it still requires `do` to be evaluated.

```
>> dump-i: does [ print ["i =" i] ]
>> dump-i
i = 7

>> dump-i: func [][ print ["i =" i] ]
>> dump-i
i = 7
```

Here we have created real functions. The first one used `does` to produce a function value which is then assigned to `dump-i`, whereas the second snippet used `func` to do that. The difference between these words is the number of arguments they require. `FUNC` needs two blocks, the first to specify the arguments of the function and the second for the code. `does` is a shortcut for creating parameterless functions so the first block is omitted.
A third word for function creation exists: `function`, which accepts three blocks. The first for specifying arguments, the second to define local words and the third is for code.

Interface Specification Block

The first block `func` expects is called the *interface specification block*. A block that describes the parameters and refinements for the function and documents the function. In the simplest form its just a block of words representing parameters to the function.

```
>> dump: func [var][ print ["value =" var] ]
>> dump j
value = 7
>> dump 42
value = 42
```

By using parameters we can apply this function to all values we like to, not only `i` as in the previous example. We lose, however the additional information of the variables name in the output.

```
>> dump: func [name value][ print [name "=" value] ]
>> dump "j" j
j = 7
```

Though the function is not very useful any more and is kind of redundant, it does what we want it to.

Restricting Types

Sometimes it's required to limit the types of the arguments passed to a function. For example you can't do anything useful if you want to compute the area of a circle and instead of an integer reprsenting it's radius you get the current time.
You can restrict the valid types of an argument by writing a block of valid types behind the according parameter.

```
>> dump: func [
     name [string! word!]
     value
   ][
     print [name "=" value]
   ]
>> dump j "j"
** Script Error: dump expected name argument
   of type: string word
** Near: dump j "j"
```

If a argument of illegal type is passed, the interpreter will report an error.

Adding Documentation

Though it's not required for a function to perform correctly, it's good practice to document your functions inline, so that users can get information about them when typing `help` *funcname*. This is done by adding strings to the specification block. The first string describes the function itself. And after every parameter (or refinement) there can be a descriptive string as well.

```
>> dump: func [
     "Prints name and value of a word"
     name [string! word!] "name of word"
     value "value of the word"
   ][
     print [name "=" value]
   ]

>> help dump
USAGE:
    DUMP name value

DESCRIPTION:
     Prints name and value of a word
     DUMP is a function value.

ARGUMENTS:
     name -- name of word (Type: string word)
     value -- value of the word (Type: any)
```

Refinements

Refinements can be used to specify variation in the normal evaluation of a funciton as well as provide optional arguments. Refinements are added to the specification block as a word preceded by a slash (/).

Within the body of the function, the refinement word is used as logic value set to true, if the refinement was provided when the function was called.

```
>> dump: func [
     "Prints name and value of a word"
     name [string! word!] "name of word"
     value "value of the word"
     /hex "print output in hex format"
   ][
     if hex [
       either number? value [
         value: to-hex value
       ][
         value: enbase/base form value 16
       ]
     ]
     print [name "=" value]
   ]
>> dump/hex "k" k
k = 000000FF
>> dump/hex "str" str
str = 746861742773206F6B2C206D792077696C6C20697320676F6E65
```

A refinement can also have arguments. Parameter names after a refinement are only passed if the refinement was provided. Documenting strings can be provided to refinements as well as refinement parameters the same as they are written for "normal" parameters.

The order in which the refinements are provided to the function upon executing it need not match the order in which they were inside the specification block. The only thing you have to be careful with is that the order of refinement arguments matches the order of provided refinements.

```
>> dump: func [
     "Prints name and value of a word"
     name [string! word!] "name of word"
     value "value of the word"
     /hex "print output in hex format"
     /file "writes to a file"
      dest [file!] "file to write to"
   ][
     if hex [
       either number? value [
         value: to-hex value
       ][
         value: enbase/base form value 16
       ]
     ]
```

```
          either file [
            write/append dest rejoin [name " = " value "^/"]
          ][
            print [name "=" value]
          ]
        ]
     >> dump/hex/file "j" j %dump.log
```

Interaction with the Outside

Literal Arguments

Our `dump` function still has a weakness: We have to pass the words name and its value to the function.
When a function is executed, all its arguments are evaluated and passed to the function. So `dump` never got `j` as second argument but the value behind `j`. And while it's impossible to get the name of a variable if you only have its value, the other way is easy.
One way would be to pass `j` as lit-word so the evaluation of the literal j yields the word j, which is passed to the funtion. And there we could write

```
     >> dump: func [var][ print [ var "=" get var ] ]
     >> dump 'j
     j = 7
```

to get the desired result. But then every call to `dump` would require us to pass a literal which looks kind of strange.
Another way would be to prevent an argument from being evaluated and just passed as literal. This is done by making it a literal parameter.

```
     >> dump: func [ 'var ][ print [var "=" get var] ]
     >> dump j
     j = 7
```

Another benefit that comes with workig with the same word not only value is that the value can be changed inside the function affecting the word on the outside, too.

```
     >> zap: func [ 'v ][ set v 0 ]
     >> zap j
     >> dump j
     j = 0
```

Get Arguments

Get arguments are in the same way related to literal arguments as get-words are to lit-words. While the literal ones return the word without evaluating it, the gets return the value behind a word without evaluating it. For functions this would be their code instead of their return value.

```
     >> print-func-spec: func [ :f ][ print mold first :f]
```

Scope

Functions share the same scope as the environment that called them. That is, functions can access words on the outside without having them passed to them. And sometimes a function doesn't know what words are defined outside the function and must not be modified. The best thing to do is to define all words inside a function local to the function, unless you know that you want to modify something on the outside.
But in REBOL the only things really local to a function are its parameters and refinements. The trick used in REBOL is to define a refinement named `/local` and add all the words we want to be local variables as arguments to this refinement. The special thing about this refinement is, that it is not displayed by help.

```
>> f: func [ a /local b][ print [a "," b]]
>> f 23
23 , none
```

`/local` does not show up in the generated help, but it is still a normal refinement.

```
>> f/local 32 7
23 , 7
```

If you don't care about confusing help texts you can use other refinements as local variables as well.

```
>> swap: func ['a 'b /tmp ][
     tmp: get a
     set a get b
     set b tmp
   ]
>> set [a b][2 7]
>> swap a b
>> print [a b]
7 2
```

Returning Values

A function (as any other evaluated block) returns the last evaluated value. Some words however terminate the execution of a function before the end is reached

```
>> f0: func [][ 1 2 3 ]
>> f1: func [][ 1 return 2 3 ]
>> f2: func [][ 1 exit 2 3 ]
>> f3: func [][ 1 throw 2 3 ]
>> f0
== 3
>> f1
== 2
>> f2
>> f3
** Throw Error: No catch for throw: 2
** Where: f3
** Near: throw 2 3
```

Function Attributes

Function attributes provide control over the error handling behaviour of functions. They are written inside a block within the function specification body.

`catch`	errors raised inside the functions are caught automatically and returned to the point where the function was called. This is useful if you are providing a function library and don't want the error to be displayed within your function, but where it was called.
`throw`	causes a return or exit that has occured within this function to be thrown up to the previous level to return.

Errors

Whenever a certain irregular condition occurs, an error is raised. Errors are of type error! object. If such an object is evaluated, it prints an error message and halt.

```
>> either error? result: try [ ... ][
       probe disarm result
][
       print result
]
```

`try` evaluates a block and returns its last evaluated value or an object of type error!. `error?` returns true if an error! object is encountered and `disarm` prevents the object from being evaluated (which would result in an error message and a halt).

Error Object

`code`	error code number (should not be used)
`type`	identifies error category (`syntax`, `math`, `access`, `user`, `internal`)
`id`	name of the error. also provides block that will be printed by interpreter
`arg1…3`	arguments to error message
`near`	code fragment showing where error occured
`where`	field is reserved

Generating Errors

>> make error! "describe error here"
<miss>

Exercise Programs II

At the end of the first part of the book you should do even more practice in REBOL to use what you have learned. Write some example programs if you haven't yet. The more of these problems you solve yourself, the better you will be. Source code of sample solutions for all programs can be found in the appendix or online at www.plain.at/vpavlu/REBOL/examples/.

8. Write a substring function that accepts a string and one paramater, the start offset inside the string. Provide an additional refinement called len to limit the length of the extracted substring. (%substr.r)
9. Code the game hangman in REBOL. (%hangman.r)
10. Make a function that acts like `replace/all` buf for all files in a given directory and instead of accepting only one search/replacement pair this function should accept two blocks with search/replacement pairs. (%replace-in-dir.r)
11. Complete the function so that it takes all files in the current directory with the specified file-type as their extension, sorts them by date and renames them to name-prefix followed by a four digit index starting at 1. If the refinement /offset is given, this should be the starting index. (%name-files.r)

```
name-files: func [ file-type [file! string!]
                   name-prefix [file! string!]
                   /offset i [integer!] ][
  ...
]
name-files ".jpg" "vacation"
```

12. Add a `/recursive` refinement to `list-dir`. (%list-dir.r)
13. Write a script that recursively adds all files in a given directory to a compressed archive. Write an extraction program for this archive that requires the user to enter a password. Make sure the contents can not be read without the password and the password can not be obtained from the script. (%make-sfx.r)

Tiny Reference

This chapter concludes the first part of the book. The following chapters are self-contained and present a different aspect of REBOL programming each. Read them in no specific order – just start with the chapters you are interested in most.
At the end of part one we give you a short summary on most frequently used REBOL words already covered, to be able to cope with what follows. The exact types of arguments and refinements can be obtained from entering `help` *func*. It's not that important to know the functions in detail – this comes over time – but it's important to know what word to use what for.

Console I/O

ask ... prompt user for input
confirm ... user confirms
input ... read line of input
prin ... print (without newline)
print … print (trailing newline)
probe … print molded version

Files & Directories

read … read file,url,..
write … write to file,url,..
load … load REBOL code
save … save REBOL code
rename … renames file
delete … deletes file
dir? … is a directory?
exists? … does exists?
make-dir … creates directory
change-dir … changes current path
what-dir … current path
list-dir … prints directory contents
clean-path … cleans ./ and ../
split-path … returns [path target]

Help & Debug

help … displays help
source … displays source
trace … toggle trace mode

Evaluation

do … evaluates a block
try … like do. on error, returns *error!*
if … conditional evaluation
either … if with alternative
switch … multiple choices

Loops

while … test-first loop
until … test-after loop
loop … evaluate several times
repeat … increment a number
for … increment a number
forever … endless loop
foreach … execute for each element in series
forall … iterate a series
forskip … iterate a series in steps

Stopping evaluation

break … exit a loop
return … exit a function with value
exit … exit a function
halt … stop interpreter
quit … quit interpreter

Series

copy … copy a series
array … create series with initial size
reduce … evaluate inside block
compose … reduce values in () only
rejoin … reduce and join series
reform … reduce and form series
remold … reduce and mold series
pick … get element from series
first,..., fifth … get element
insert … insert at current index
append … insert at end
change … change first element
poke … change value at position
remove … remove first element
clear … remove all elements
next … series at next element
back … series at previous element
at … series at given element
skip … series after given element
head … very start of series
tail … end of series

length? … series' length
empty? … if empty
tail? … if empty
index? … value of current index
sort … sort a series
reverse … reverse a series
find … find an element
replace … replace an element
select … value after found element
unique … remove duplicates

intersect … sets: A ? B

union … sets: A ? B
exclude … sets: A - B

difference …sets: (A ? B) – (A ? B)

Strings

join … concatenate values
form … convert to string
mold … make REBOL readable
rejoin … join elements in block
reform, remold … see series
lowercase … convert to lowercase
uppercase … convert to uppercase
enbase … encode in given base
debase … decode from given base
dehex … decodes %xx url-strings
compress … compresses a string
decompress … decompresses a string

Misc

now … current date/time
random … random value
wait … delays execution

II. Selected REBOL Chapters

<miss>

Parsing

Objects

CGI & r80v5 embedded REBOL

Network Programming

Webserver

Instant Messenger

REBOL Idioms

Getting default values

```
>> load any [ system/options/cgi "" ]
```

Reducing common sub-expressions

```
>> data: [ name "viktor" email vpavlu@plain.at ]
>> either (flag) [
       print second find data 'name
   ][
       print second find data 'email
   ]
```

As we know `either` returns the last evaluated value in the block, we can take common sub-expressions out of the block which reduces typing effort, complexity and ease of maintaining. Searching for a label and then reducing the value immediately afterwards should be done with `select` instead of `second find`.

```
>> print select data either mode [ 'name ][ 'email ]
```

Third the `either expr [][]` is simply a `pick` with a logic! as argument (which returns the first block if true, the second otherwise).

```
>> print select data pick [name email] mode
```

III. REBOL/View

<miss>