Creating lightweight cross-platform Applications

REBOL Essentials

Viktor Pavlu
9-Jan-2003

Foreword

This is the accompanying tutorial to the REBOL course I held during 2002/2003 at the technical college *HTL Spengergasse* in Vienna. As class time was very short I had to put as much useful information in this book to make it possible for the students to follow the fast pace of my lessons by studying at home. At the same time it should comprise all essential information on REBOL into a single document.

Resources and Examples

During the text you will often find references to files like %filename.r. These point to scripts that can be found online at http://plain.at/vpavlu/REBOL/examples and are not included in the printed tutorial.
Source code of examples and sample solutions for all exercise programs can also be found online at http://plain.at/vpavlu/REBOL/examples/.

Source code throughout the tutorial that has a >> prompt in front can be directly entered into the console. If the prompt is missing, the code is some specific kind of dialect and thus needs to be passed to a function which understands that dialect (ie. VID code must be passed to `layout`). What to do with the code is pointed out directly in the chapters.

PART I. REBOL language tutorial

The first part makes you familiar with REBOL concepts and terms, summarizes all language elements and provides a profound starting ground for own programs and the following specialized chapters.

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 responsible for the Amiga OS. REBOL was first released in 1997 and since then there have been many improvements. In 2002 REBOL was even listed as nominee for the Webby awards for technical achievement, nevertheless it's still rarely known.
REBOL stands for "Relative Expression Based Object Language". Let's look at some 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

What others say

This, like the Amiga and BeOS, could be another doomed computer language that should have ruled the field. It probably came along five years too late. REBOL is a fully network-aware relative expression based object language. Take a dash of PERL, mix with the cross platform compatibilty of a Java, and make it extremely easy for beginners to start coding, and you get the general idea. REBOL has all kinds of cool potential, but until a deep and wide developer/user community gets built, and until it finds its niche in an already crowded language marketplace, it's probably doomed to obscurity. As a startup, finding the funding is going to be problematic in an environment where instant results are called for.

-- turksheadreview.com

Versions

Currently three versions of REBOL exist:

- **/Core** The core language. Console version, *free*
- **/View** Extends /Core with GUI features, *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 "compilers" (**/Enbase, /Enface** and **/Enpro**) 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/
http://www.rebol.com/bio-carl.html

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. The free /Core interpreter is suited 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` shows 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. Similar 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	640x480
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://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 function.

*Get-word*s return the value behind the word. This is similar 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 in REBOL. All there is are words with a predefined meaning (value) that describe the language.
And this language is the subject of the 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 statements 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 [
  1 [...]
  2 [...]
][...]
```

```
switch(var){
   case 1:  ... break;
   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! (32bit numbers without decimal point), Decimal! and Money! (64bit 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 length of string
`parse` *str delim*	splits a string into tokens, delimited by delim

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 solve some of the example problems. Source code of sample solutions for all programs can be found online at http://lain.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 parameter, 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 script 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
`remove-each` *word series []*	like foreach, removes curent element if block is true

`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. `remove-each` acts similar but also removes the current element from the list if the block evaluates true for this iteration.

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*	evaluates 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 representing 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 function 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 function. 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 working with the same word and not only the 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 occurred 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 halts.

```
>> 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 occurred
`where`	field is reserved

Generating Errors

```
>> make error! "describe error here"
>> make error! [ category id arg1 arg2 arg3 ]
```

The first line creates a user error with the default id 'message. It will print the message unless the error is handled with a catch.
The second line creates a predefined error. *category* and *id* are required and may be followed by up to three arguments. To see all predefined errors have a look at the `system/error` object where an object containing templates for the error messages lives for every category.

To create a new predefined error, just add a new id and error-message to the `system/error/user` object.

```
>> system/error/user: make system/error/user [
     my-error: [:arg1 "doesn't match" :arg2]
]
>> make error! [ user my-error "foo" "bar" ]
```

You can also group a series of errors together by adding a new category to `system/error`

```
>> system/error: make system/error [
     my-cat: make object! [
       code: 1000
       type: "My Errors"
       my-error: [:arg1 "doesn't match" :arg2]
       too-late: ["it's too late"]
     ]
   ]
>> make error! [ my-cat too-late ]
** My Errors: it's too late
** Near: make error! [my-cat too-late]
```

To just print the error message without halting execution of the script, use these lines

```
>> disarmed: disarm try [ make error! [my-cat too-late] ]
>> print bind (get disarmed/id) (in disarmed 'id)
it's too late
```

More about `bind` and `in` can be found in the object! chapter.

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.

8. Code the game hangman in REBOL. (%hangman.r)
9. Make a function that acts like `replace/all` but 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)
10. 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"
```

11. Add a `/recursive` refinement to `list-dir`. (%list-dir.r)
12. 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)
13. Write a script that downloads a whole website for offline browsing. Be careful to follow only *href* and *src* attributes that point to locations on the same server. (%get-site.r) Hint:

```
get-hrefs: func [ markup /local urls url][
  urls: copy []
  parse markup [ any [
    thru "href=^"" copy url to "^"" (append urls url)
  ]]
  urls
]
```

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

PART II. Selected REBOL Chapters

The following chapters are self-contained texts on various interesting REBOL topics collected from the REBOL/Core User Guide, the mailing list, various resources from other people and of course, my experience with programming in REBOL. It is recommended that you read the chapters you are interested most at the beginning, in order to be able to write programs you can use and the other chapters when there is time, in order to get a decent understanding of the REBOL universe.

Parsing

Parsing is the process of structuring a linear representation in accordance with a given grammar. This definition has been kept abstract on purpose, to allow as wide an interpretation as possible. The "linear representation" may be a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behaviour, in short any linear sequence in which the preceding elements in some way restrict the next element. (If there is no restriction, the sequence still has a grammar, but this grammar is trivial and uninformative.) For some of the examples the grammar is well-known, for some it is an object of research and for some our notion of a grammar is only just beginning to take shape.
For each grammar, there are generally an infinite number of linear representations ("sentences") that can be structured with it. That is, a finite-size grammar can supply structure to an infinite number of sentences. This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they summarize succinctly the structure of an infinite number of objects of a certain class. -- [Grune, Jacobs: *Parsing Techniques, a practical guide*]

```
<even-number> ::= <num>* [0 | 2 | 4 | 6 | 8]
<num>         ::= [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ]
```

An example for a simple grammar in BNF notation for the infinite-size set of even numbers.
For more information on parsing in general and different parsing techniques have a look at the execellent book on parsing techniques by Dick Grune and Ceriel Jacobs published by Ellis Horwood, Chichester, England, 1990;

```
Parsing Techniques, a practical guide
Dick Grune, Ceriel Jacobs
ISBN 0-13-651431-6
http://www.cs.vu.nl/~dick/PTAPG.html
```

REBOL features a own BNF-like (backus-naur form) parsing dialect for this subject.

Quick Introduction to BNF-like Grammars

A grammar is a set of rules that describes a language, that is describes all correct assemblies of characters to words (or words to sentences) within that language. A context-free grammar is a formalism consisting of a set of terminal symbols T (constant, literal values), non-terminal symbols N (placeholders for other non-terminal- or terminal symbols), a set of production rules how to transform N to T and a special non-terminal symbol S to start the production. The Backus-Naur Form is a notation to describe such grammars.

Again, two types of symbols exist: terminal symbols and non-terminal symbols. The terminal symbols have a fixed, literal value. Non-terminal symbols are placeholders for other terminal- or non-terminal symbols. If you now want to create a regular word within this defined language, you start with a special non-terminal symbol that is defined as the entry point for all words and continually replace the variable non-terminal symbols with values according to the grammar rules which tell which symbol consists of what other symbols. An Example will clarify this:

```
<signed_number> ::= <sign> <digits> [ "." <digits> ]
<sign>          ::= "+" | "-" | e
<digits>        ::= <digit> | <digit> <digits>
<digit>         ::= "0" | "1" | "2" | ... | "8" | "9"
```

A simple grammar for a number consisting of a sign, some digits and an optional decimal part. The sign can be one of +, - or e, the empty symbol. The square brackets denote that the symbols inside are optional. Digits is either a single digit or a single digit followed by other digits. By this recursion we get numbers of arbitrary lengths but at least one digit. The symbols enclosed in quotes are the terminal symbols T.

```
-2.2.2    <sign> ... "-"
          <digits> ... <digit> ... "2"
          "."
          <digits> ... <digit> ... "2"
          "." not found in production rules; not a valid <signed_number>
13        <sign> ... e(emtpy)
          <digits> ... <digit> <digits> ... "1" <digits>
          <digits> ... <digit> ... "3"
          end reached, all rules obeyed; a valid  <signed_number>
```

BNF Symbols

Non-terminal symbols	`<non-terminal>`
Terminal symbols	`"terminal"` or **`terminal`**
Make something optional	`[` optional `]`
Repeat Zero to n times	`{` repeated `}`
Repeat min to max times	`{` repeated `}`$_{min..max}$
Alternative	`<a> | <b>`
Grouped alternative	`(<a> <b>) | <c>`

Parsing in REBOL

In REBOL parsing is done with the function `parse` which takes two arguments: the subject to parse and a parsing rule. The simplest method form parsing is to split a string into tokens of information.

`parse` *subject* `none`	split at whitespace
`parse` *subject delim-string*	split at delim-string
`parse` *subject rule-block*	parse according to rules

`parse` with `none` as rule does in fact no splitting. The reason the string is split after a `parse` with `none` is, that parse per default treats whitespace as delimiter and splits. If you call parse with the `/all` refinement (treat whitespace as normal characters) and `none` as param, you get the string unmodified.

```
>> str: "1,234,220.4 56,322.0 99,118.43"
>> parse str none
== ["1" "234" "220.4" "56" "322.0" "99" "118.43"]
>> parse str ",."
== ["1" "234" "220" "4" "56" "322" "0" "99" "118" "43"]
>> parse/all str ",."
== ["1" "234" "220" "4 56" "322" "0 99" "118" "43"]
```

Real parsing (not splitting as we did until now) is a bit more complex. The second parameter is a block of BNF like parsing rules. Then parse does not return the split tokens (there will be none) but a boolean value telling whether the string completely matches the rules. That is if the string can be built from start to its end according to the rules.
Be sure to know the basic BNF terms before continuing.

REBOLs BNF dialect

A dialect is an extension to the REBOL language for a particular task that makes it easier to express what you want for that given problem, in this case: parsing.

Non-Terminal symbols
are just plain REBOL words that hold a block with a production rule.

Terminal symbols
are strings, characters, tags, bitsets and the special symbol *end*.

`"string"`	matches this string
`#"c"`	matches this character
`<tag>`	matches this tag
`end`	matches the end of parsed input ($ in regex)

Bitsets are used to specify a range of allowed characters:

```
>> numeric: charset [ "012" #"3" - #"9" ]
>> alphanum: union numeric charset
                    [ #"a" - #"z" #"A" - #"Z" ]
>> white-space: charset reduce [ tab newline #" " ]
>> no-space: complement white-space
>> parse "parse is powerful" [ some alphanum ]
== true
```

Note that whitespace is ignored unless you specify `/all`.

Production rules
are any combination of terminal- and non-terminal symbols inside a block.

`[` *pat1* `|` *pat2* `]`	pat1 or pat2
`[` *pat1 pat2* `]`	pat1 then pat2
`[ 4` *pat* `]`	4 times the pattern
`[ 2 5` *pat* `]`	2 to 5 times the pattern pat1
`[ some` *pat* `]`	1 to n times the pattern (pattern+ in regex)
`[ any` *pat* `]`	0 to n times the pattern (pattern* in regex)
`[ opt` *pat* `]`	0 or 1 times the pattern (pattern? in regex)
`[ none ]`	e (match nothing)

Grouping of values or words is done with square brackets.

Special words

`skip`	skips exactly one character
`to` *pat*	skips until pattern; (.....)pat
`thru` *pat*	skips until after pattern; (.....pat)

Production

The process of continually replacing non-terminals with values according to the production rules while moving over the text that is to be parsed. If we successfully reach the end, the string is a regular word in the grammar. Fine.
But what we actually wanted to do, is *parse* the string not just test it. We have to somehow get and modify the input so we can do something with it.

`(` *code* `)`	the code is interpreted upon reaching this point
`copy` *target*	copies text of next match to target
var`:`	gets string into var
`:`*var*	sets string to var

By combining grammar rules with executable REBOL code you can do powerful parsers.

Object!

CGI & r80v5 embedded REBOL

Network Programming

Webserver

Instant Messenger

XML-RPC

XML remote procedure calls – a simple way to communicate with the outside world through the use of standard protocols. Remote procedure calls are encoded in xml and transported over http which makes it possible for two or more programs written in different languages, running on different systems to communicate and co-work.

REBOL Idioms

Getting default values

Sometimes you want to use a default value if something is none. To avoid constructs like

```
>> either none? system/options/cgi [][
  load system/options/cgi
]
```

use any to have the first value that is not *false* or *none* returned.

```
>> 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
```

PART III. REBOL/View

In the third part of the book, the graphical elements of REBOL are covered. For this we have to download REBOL/View or purchase any other of the GUI aware versions of REBOL.

The version of View and VID used in this tutorial is 1.155.2.3 (check at startup). Some new styles have been introduced since View 1.155.0 (used in /View 1.2.1) which are discussed. In order to have access to the same styles and words as in this tutorial, you should get the latest version of the free /View interpreter from http://www.reboltech.com/downloads/.
The interpreter that was used for this tutorial was REBOL/View 1.2.8.3.1 (where the *3.1* stands for the Win32 platform)
To tell which version you are currently working with, type `system/version` in the console.
Everything in this tutorial should work in future releases of /View as well.

All graphical elements in REBOL are made of faces. A face is an rectangular area that can be displayed on the screen and is described by various pieces of information such as size, color, offset, text in a specific font, an image to be displayed, entry points for event handling functions, …
To view the basic face from which all other faces are derived type `probe face` in the console. If you get an error like face has no value, you should remember to download a graphics enabled version of REBOL. As stated before, all graphical user interfaces are made of such faces. Fortunately REBOL provides us with an dialect for easy creation of predefined and customized faces so we don't have to reinvent buttons and the like. So we start with examining the visual interface dialect before diving deeper into /view.

VID

VID (Visual Interface Dialect) is an extension of the REBOL language that makes it easier to express user interfaces.
`layout` is the function that does the VID processing. It returns a construct of faces which can then be displayed with `view`.
Note: All sample code in this chapter is visual interface dialect only. The samples have to be written inside a `layout []` block which then has to be displayed.

Styles

With styles you express *what* to display. A `field` for text input or a `button` are examples of styles. Every style can be customized with parameters written after them (called facets). The order of the params does not matter as VID differentiates them by their datatype. If a string! follows, it is interpreted as the text for the specific widget, if a pair! (20x10) follows, it is taken as the size and so on. A complete list of styles and what params will have what effect on them can be found in the *Style Reference* later in this chapter.

Using Styles

It's time that we create our first dialog. (*%first-vid.r*)

```
view layout [
  across
  label italic font [ color: brick ] "To:"
  tab
  inp-to: field
  return
  label italic font [ color: brick ] "Subject:"
  tab
  inp-subj: field
  return
  inp-text: area
  return
  button "Send" [
    foreach addr parse inp-to/text ",;" [
      send to-email addr
        rejoin [inp-subj/text newline inp-text/text]
    ]
    quit
  ]
]
```

The words inside the block are parsed by `layout` for valid VID words and then interpreted to create a set of faces which themselves are displayed with `view`. `label`, `area` and `button` are the styles in this example. The "To:" after the first label is a facet that tells the label what text to display. `inp-to` (and the other inp- words) are normal REBOL words that hold a reference to the style after them. So `inp-to/text` can be used to access the text attribute of the input field right after `to`. Much the same way as a string after a style sets the text to be displayed, a block of REBOL code sets the action that should be performed if the style is clicked. We see that adding styles to a layout is very easy and customizing these styles with facets is easy, too as long as we know what facets can be applied to which styles.

Fortunately most of the facets can be applied to all styles. A complete list of styles and applicable facets follows, again, in the reference at the end of this chapter.
`across`, `return` and `tab` are keywords rather than styles that affect the placement (or something different) of the styles.

Custom Styles

If you see yourself writing the same attributes for your styles again and again like

```
label italic font [ color: brick ] ...
```

in the previous example, it's time to define a custom style that already has these attributes to reduce redundancy. Use `style` to define a new style based on the characteristics of an existing one plus additional attributes.

```
style red-lbl label italic font [ color: brick ]
red-lbl "To:"
red-lbl "Subject:"
```

By doing so it's possible to change the appearance of the whole gui without problems, too.

Positioning

VID offers auto-layout functionality, that is we just add elements to a pane without specifying where and VID takes care of the positioning itself. By default subsequent styles are placed *below* each other but this behaviour can be changed to being placed *across* the GUI. Either way the word `return` changes to the next column or row.

```
across
text "1"
text "2"
text "3"
return
text "A"
text "B"
text "C"

below
text "1"
text "2"
text "3"
return
text "A"
text "B"
text "C"
```

Style Reference

document text (dark text on light background)

`title`	title
`body`	normal text
`text`	normal text
`txt`	normal text
`h1, ..., h5`	headers 1 through 4
`code`	source code (bold, nonproportional)
`tt`	typewriter like text
`lbl`	small label

video text (light text on dark background)

`banner`	title
`vtext`	normal text
`vh1, ..., vh4`	headers 1 through 4
`label`	small label

text input

`field`	single-line text input
`area`	multi-line text input
`info`	read-only field

buttons

`button`	pushbutton
`btn`	rounded button*
`btn-help`	rounded help button (displays help messagebox)*
`btn-enter`	rounded enter button*
`btn-cancel`	rounded cancel button*
`toggle`	on/off button
`tog`	rounded toggle*
`rotary`	switch through 1/2/../n
`choice`	popup selector
`drop-down`	dropdown*
`check`	checkbox
`radio`	radio button
`arrow`	clickable arrow

visuals

`image`	image
`anim`	animated image
`icon`	thumbsize image with text
`logo-bar`	vertical REBOL logo*
`led`	indicator light

areas

`backdrop`	scaled background
`backtile`	tiled background
`box`	rectangular box in the foreground

other items

`progress`	status indicator
`slider`	sliding bar

`scroller`	sliding bar with arrow buttons*
lists & sublayouts	
`panel []`	simple sublayout
`list []`	repeated sublayout
`text-list`	list of text lines
event listeners	
`sensor`	listens for mouse events
`key`	listens for keyboard events

Note: Styles marked with an asterisk have been newly introduced in /View 1.2.8.
For a complete list of styles in your current version of /view, query the system object. Also, have a look at %vid-inspect.r

```
>> styles: skip system/view/vid/vid-styles 2
>> forskip styles 2 [ print styles/1 ]
```

Exercise Programs III

As we've now covered almost every aspect of creating GUIs with VID, it is time for you to see what you have learned by creating own GUIs.

14. Write a REBLET that displays a box for every predefined color. If a box is clicked, the color should be printed in decimal tuple (rrr.ggg.bbb) and hex (#RRGGBB) form inside a big box that has the selected color as background. (%color-select.r)

```
colors: copy []
words: first system/words
forall words [
  if tuple? get/any in system/words words/1 [
    append colors words/1
  ]
]
```

15. Create a dialog with which you can send emails. (%mail-dlg.r)
16. Display the number of days between now and your birthday. (%days-to-go.r)

Draw

This chapter covers rendering of graphics primitives such as lines, polygons and circles. It is a good idea to read the chapters on REBOL/View in general and the layout description dialect VID first as these are very good sources for getting a decent grasp of the details behind /View.

REBOL/view was designed for displaying user interfaces and presentations and was optimized for combining multiple graphical elements such as images, text, buttons, and effects. Although it was not intended for low level graphics, such as rendering bitmaps, lines, or polygons, it is capable of drawing basic shapes with a variety of attributes. As already mentioned earlier, all visual elements in /view are made up of faces. One important attribute of those faces is their *effect* block, which holds effect words. A very important and powerful word in the effect dialect is `draw`.
Draw is another dialect of its own which is capable of rendering basic shapes and will be examined in detail in this chapter.

The draw dialect is very similar to VID which we already know. It consists of words that specify how the following values should be interpreted (ie. `line`) and the values that actually specify the primitive (ie. coordinates).

```
line 10x90 50x10 90x90
line 25x50 75x50
```

Here we draw two linestrips that look like an ‘A’.
The examples in this chapter are `draw` code only and must be encapsulated within a `draw` block inside `effect` of a face.
The most important thing you need to know from VID is that you can't draw just anywhere. The draw dialect inside faces' effect blocks is currently the only method of drawing primitives in /view. So the first thing you need to create is a face as your drawing canvas.

```
>> view layout [
  origin 0x0
  box white 400x300
  effect [
    draw [
      ;draw code goes here
    ]
  ]
]
```

In the previous example the white box is your drawing canvas with 0x0 at the upper left corner and 400x300 at the lower right.
If you are not familiar with VID concepts, use this function to conveniently test the draw examples

```
>> draw: func [ "draws basic shapes with draw dialect"
draw-code [block!] /size s][
  if not size [ s: 100x100 ]
  view layout compose/deep [
    origin 0x0
    box white s
    effect [
      draw [ (draw-code) ]
    ]
  ]
]

>> draw [ pen black line 100x100 0x0 ]
```

Draw Dialect Words

line	draw a line
polygon	draw a polygon
box	draw a rectangle
circle	draw a circle
pen	set foreground and background color
fill-pen	set foreground and background fill colors
line-pattern	set the line pattern
flood	fill from a point outward
image	insert an image
text	draw text
font	specify text font attributes

Drawing in Detail

Lines

The line command draws a line between two given points. Points are given as *pair!* values where 0x0 is the upper left corner and values are increasing to the lower right (fourth quarter in standard cartesian coodinate space). If more points are given, multiple lines are drawn connected but the linestrip is not closed (as with polygon).
The line is drawn in the current pen color using the current line-pattern.

```
line 0x0 20x20 40x0
```

Polygons

For simple examples drawing polygons seems to be equal to drawing lines except that the last point is connected back to the first. With filling however, the difference becomes apparent: Polygons describe areas rather than linestrips.
Polygons are always filled with the current fill-pen.

```
fill-pen gold
polygon 10x100 50x10 90x100 50x75
```

Bugnote: In version 1.155.2.3 of View and VID (check at startup), polygons lose their edge colors when a fill-pen is specified. This is going to be fixed.
And in some cases, rendering of polygons that extend outside the bounding face may crash during rendering the draw. This is hopefully going to be fixed, too.

Rectangles

`box` provides a shortcut to drawing rectangular shapes. Only the upper left and lower right coordinates are required.

```
box 20x20 80x80
```

Circles

With `circle` you can draw circles by specifying the center point and the radius (which must be an integer).

```
circle 50x50 40
```

Specifying Colors

We have already used this in the preceding examples. `pen` sets the outline color (affects `line`, `polygon`, `box`, and `circle`) while `fill-pen` sets the color with which areas are filled (affects `polygon`, `box`, `circle`, and `flood`).
Both pens have a foreground and a background part. The foreground color is the one which is visible. The background color for `pen` is used with line-patterns as the secondary color. The `fill-pen` background color has no meaning (View 1.155.2.3).

```
pen red blue
fill-pen none
line-pattern 2 4 1
line 10x10 90x90 90x10 10x90 10x10
```

A color set to *none* means transparent.
The `pen` foreground defaults to inverse face color. All other colors default to none.

Line-patterns

Line-patterns affect the rendering of outlines for line, polygon and box (not circle!). The parameters are lengths of line segments that are alternately drawn in foreground color or omitted (that is, drawn in background color). Up to eight lengths may be specified. After that the pattern repeats itself.

```
pen navy none
line-pattern 4 4 1 4 ;dash-dot
box 10x10 80x80
line-pattern         ;solid again
box 20x20 90x90
```

`line-pattern` without parameters sets the style back to a solid line.
Bugnote: In version 1.155.2.3 of View and VID (check at startup), a bug with background colors and line-patterns was discovered.

Filling areas

With `flood` you flood fill the area around a given point with the fill-pen foreground color until any other color is reached.

```
pen sky
fill-pen white
line 20x30 50x80 80x30 20x30
line 20x65 50x15 80x65 20x65
flood 50x50
```

You can also specify a border color. Then the flood fill stops at no color but the specified one.

```
pen sky
fill-pen white
line 20x30 50x80 80x30 20x30
pen leaf
line 20x65 50x15 80x65 20x65
flood 50x50 leaf
```

Adding Images

Adding images to faces can also be done in the draw block. (if you want the face to be completely covered by an image, it's better done directly in VID). The syntax for adding images is as follows

image pos [*pair!*] image [*image!*] ?transparent-key [*tuple! integer!*]?

which adds the image at the given position. The optional third parameter specifies the color that should be rendered as transparent. If the transparent-key is a tuple! this color is considered transparent. If it’s an integer!, all colors with a lower luma value are drawn transparently.

Adding Text

Adding text is similiar to adding images.

text pos [*pair!*] text [*string!*]

inserts the text at the given position with the font settings of the face. The position resembles the upper left corner of the text.
With the `font` word you can set the faces' font from inside the draw block the same way as you would do in VID. The color however, is affected by the pen foreground.

```
pen black
text 10x10 “Standard”
font make face/font [ name: “Verdana” ]
text 10x30 “Verdana”
font make face/font [ name: “Trebuchet MS” ]
text 10x50 “Trebuchet MS”
font make face/font [ name: “Sans-serif” ]
text 10x70 “Sans-serif”
```

The font-obj must be made of a valid `face/font` object. With the `name` field you can select an font available on the system. If a font is not found, the standard font (Arial) is used.

```
>> probe face/font

make object! [
    name: "arial"
    style: none
    size: 12
    color: 0.0.0
    offset: 2x2
    space: 0x0
    align: 'center
    valign: 'center
    shadow: none
]
```

Working with Images

The draw part of this chapter is now finished. This last paragraph shows how to store drawn things in offscreen images or files. This technique is not restricted to draw but rather can be applied to all types of faces.
After creating a set of faces with layout or make face we usually viewed them, but we can also convert all those styles, facets, font information, effect- and draw-instructions to a plain image.

```
>> lay: layout [
  origin 0x0
  box white 30x30
  effect [
    grid 10x10 32.128.32
    draw [
      pen brick
      line  0x0  30x30
      line 10x0  30x20
      line 20x0  30x10
    ]
  ]
]
>> img: to-image lay
>> length? mold lay
== 744821
>> length? mold img
== 5514
```

`img` now holds a pixel representation of the previously created layout. With `save` and the appropriate refinement create a valid .png or .bmp header, you can create image files.

```
save/png %layout.png img
```

You can also create an image from scratch. For example a 2x2 image with pixels set to red, green, blue, and yellow.

```
>> squares: make image! 2x2
>> squares/1: red
>> squares/2: green
>> poke squares 3 blue
>> poke squares 4 yellow

>> view layout [ image squares 20x20 ]
```

The size of 20x20 is specified to stretch the image to be 20x20 pixels in size in order to make it more easily visible. The image, though, is still 2x2 pixels.

Single pixels can be accessed as if they were in a normal block holding the color values line by line. That's what we did with `poke`. Use these functions to access the pixels in a more common way:

```
>> getpixel: func [
  "returns color at given pos (0x0 is upper left)"
  img [image!] pos [pair!]
][
  pick img img/size/x * pos/y + pos/x + 1
]

>> setpixel: func [
  "sets pixel at given pos to color (0x0 is upper left)"
  img [image!] pos [pair!] color [tuple!]
][
  poke img img/size/x * pos/y + pos/x + 1 color
]
```

Exercise Programs IV

Again, it’s time to get some hands-on experience by solving simple exercises.

17. Write a simple REBLET that draws a scaled graph of values inside a block. The block can be of any length and the values of any size - the graph should always be 400x300 in size. (%draw-graph.r)
18. Extend %draw-graph.r to be able to render multiple graphs. The ingoing block consists of a color followed by a block of values for each graph. All value blocks are of the same length. (%draw-graph2.r)
19. Write a REBLET that draws a pie chart from [color number] values inside a block. The block can be of any length and the values of any size - the graph should always be 400x400 in size. (%pie-chart.r)
20. Write a script that creates .png thumbnails for all .jpg files in a given directory. The size of the thumbnails is 120 pixels in width or height, which is smaller. The other coordinate should be resized accordingly. Also write the size (KB) of the original image in the thumbnails. (%thumbs-make.r)

Effects

The `effect` block is an attribute that every face has. Inside this block various effects can be specified to be applied to the face. We already discussed the `draw` command inside the `effect` block with which you can draw lines etc. But the effect block offers many more commands that affect the view of a face besides draw.

Applying effects to a face is very easy - just append the word effect followed by a block of effect dialect words that will be applied to the face in the entered order.

```
>> view layout [
  origin 0x0
  box black 100x100 effect [
    draw [
      fill-pen red
      circle 50x50 45
      pen black
      line 50x5 50x95
      line 5x50 95x50
    ]
  ]
]
```

What follows is a list of all available effects plus a short description what the effect does and how it is to be applied.

Scaling

fit	face is resized to fit in parent face
aspect	same as fit, but aspect ratio is preserved
extend extend-offset[pair!] pixels-to-extend[pair!]	stretched without affecting scale

Tiling

tile	image is tiled over face
tile-view	image is tiled over face; tile offset is relative to window face

Subimages

clip	clips image to size of face (speeds up effects)
crop position[pair!] dimension[pair!]	extracts specified image from face

Translation

flip direction[pair!]	flips image in given direction
rotate degrees[integer!]	rotates number of degrees in clockwise direction (90,180,270,360 are supported)

reflect direction[pair!] reflects an image in X,Y or both directions. positive values to reflect upper/left part, negative for lower/right

Image processing

invert inverts colors in rgb color space
luma val[integer!] modifies brightness of image (positive lightens, negative darkens image)
contrast degree[integer!] modifies contrast (positive inreases contrast)
brighten degree[integer!] modifies brightness of image
tint color-phase[integer!] modifies tint of image with given color-phase
grayscale converts image to grayscale
colorize color[tuple!] colors an image with given color
multiply value[integer! tuple! image!] multiplies each pixel with give value
difference [integer! tuple! image!] difference to each pixel is computed
blur blurs image (use multiple times)
sharpen sharpens image (use multiple times)
emboss applies emboss effect

Gradients

gradient direction[pair!] [color-from[tuple!]] [color-to[tuple!]] produces gradient effect in given direction with optional colors
gradcol like gradient, colorizes image
gradmul liek gradient, multiplies color values

Keys

key [tuple! integer!] all values with lower luma value as given integer! or with a color equal to given tuple! are considered transparent
shadow equal to key, but additionally generates 50% drop shadow

Algorithmic Shapes

colors can be specified, edge color is used otherwise

arrow creates a equally-sided triangular shape pointing upwards
cross lays a X over the face
oval leaves a oval hole over face, rest is overlayed
tab [edge-to-round[pair!]] [radius[integer!]] [thickness[integer!]] [color[tuple!]] generates button with rounded corners
grid space[pair!] draws a grid

Handling Events

As we already know how to create GUIs, it's time that we get to know how to let them do something useful, that is make them respond to the users' actions. Until know we did this by appending a block of REBOL code to styles we added to our layout. This code was executed when the user clicked the button, text, ... or triggered something we can think of as the *main* event of the control, somehow different. (ie. dragging a slider or pressing a key that was defined as shortcut). But there are also other events that a face can react on which we are now going to examine.

Every face has a feel object that defines how the face behaves on events. Whether the user is pressing a key, moving the mouse or the face needs to be redrawn etc – it's the feel of the object that determines how the event is handled.

The Feel Object

All events in /View are handled by only four functions which are in the feel object of every face. These functions are:

`engage:` `func [face action event][...]`
: This is the real event handler for the face. It gets called when the user presses or releases a mouse button or key on the keyboard, when a timer exceeds, …

`detect:` `func [face event][...]`
: Is called for every event that is intended for this face or faces that lie inside this face enabling you to intercept certain events to keep your GUI free of unnecessary event processing.

`redraw:` `func [face action position][...]`
: Redraw is called whenever something in the displaying of a face changes. Each time the face refreshes, is shown or hidden this function gets called. Note that `position` is not the position of the cursor but the position of the face. Only interesting with iterated layouts like `list`.

`over:` `func [face action position][...]`
: Over is called whenever the cursor is moved over a face. As this happens very often, this function should be set to *none* unless you really need it to not unnecessarily slow down /View. Position tells the position of the mouse cursor relative to the upper left corner of the window.

Every face may implement their own bodies for these functions or just set them to *none*. The `face` parameter always holds the face for which the event occured. `action` is a word that identifies the type of event like `down` if the mouse button has been pressed. `position` is a *pair!* value giving coordinates. The last parameter is the event object which is of type *event!*

Event!

All events in /View are stored as special datatypes called *event!*. In order to be able to write your own event handlers, you need to know about the event that occured. Usually, if we don't know what fields exist in an REBOL object, we try something like `help` or `mold` to see the objects' fields – with *event!*s however, this does not work.
Here is the information I gathered from the mailling list and various examples:

face	The face in which the event occured (root pane)
type	A word that describes the type of event – same as `action`
offset	Current position of the cursor relative to the root pane
key	A character representing the key that was pressed (if it was a key event). If the it was a special key (ie F1) a word representing the keys' name is stored instead. *None* with non-key events.
shift	a logic!, true if the shift-key was pressed during the event
control	a logic!, true if the ctrl-key was pressed during the event

(if something is missing or I'm plain wrong, please tell me via vpavlu@plain.at)

Engage

As the main event handler, `engage` is called for all events that are not handled by `redraw` or `over`. That is mouse events where a button is pressed or released, keyboard events and timers.

```
engage: func [face action event][...]
```

Try this small example to get familiar with engage. It will create a box, just as we did in the previous /View chapters but this time we add our own `feel`.

```
>> view/new layout [
  canvas: box ivory rate 1 feel [
    engage: func [face action event][
      print rejoin [ "action = '" action ]
      if action = 'key [
          print join event/key " was pressed."
      ]
    ]
  ]
]
>> focus canvas ;key events need focus
>> do-events
```

Some actions we see: (alt-)up, (alt-)down, over, away, key, time, scroll-line, scroll-page

Timers

Also note the word `rate` – It specifies how many time events should be triggered per second. Or instead of time events per second you can also specify the intervals between two time events by passing a *time!* value like 0:00:10 for every 10 seconds.
To stop a timer, the `face/rate` is set to *none* and `show` is called to update internal timer settings.

Detect

`detect` is similar to `engage` in terms of what events trigger it. But it also has the ability to swallow events so the subfaces never get notified of events filtered by `detect`.

```
detect: func [face event][...]
```

Either *none* (the event is swallowed) or the event (passed to subfaces for further processing) must be returned.

Redraw

The `redraw` function is called immediately before the face is drawn.

```
redraw: func [face action position][...]
```

It listens for three types of actions: `show`, `hide` and `draw`.
If the GUI is displayed for the first time or the `show` command is applied to a face, the `redraw` function of that face is called twice – first with a `show`, then with a `draw` message. If the face is hidden (ie. `hide` command), it receives the appropriate `hide` action.

Over

Last but not least, the `over` function which is called whenever the mouse cursor enters or leaves a face. You can also force `view` to call `over` for all cursor move events by passing the `all-over` option to `view`. Note that this can drastically reduce performance of your GUI.

```
>> view/options layout [
  box ivory feel [
    over: func [face action position][
      print join either action [
        "entered at"
      ][
        "exited at"
      ] position
    ]
  ]
][all-over]
```

In `over, action` is a *logic!* that expresses whether the cursor entered or left the face.

Exercise Programs V

This chapter ends our expedition into /View. We have learned the basics of VID and how this dialect integrates into /Views system of faces, created and modified styles, discussed the details of effect and draw and finally put life into our interfaces. Now it's time to bring all this together.
After some simple feel-only exercises, more complex ones follow for which you'll need knowledge about /View in general and especially /Views VID.

21. Create a style for a drag-able box (%drag-box.r)
22. Create a digital clock (%clock.r)
23. Create an analog clock (%clock-draw.r)
24. Write a REBLET that can be used to change a password providing two text entry fields that display stars instead of the entered characters. The REBLET should furthermore contain an information field that displays "the password is too short" (less than 5 characters), "passwords do not match" or "passwords ok" depending on the input in the fields. Third there has to be an OK button that is only clickable, if the passwords are equal and long enough. (%passwd-dlg.r)
25. Write an application with which the user can draw lines. If the right mouse button is pressed, an color selection dialog (`request-color`) should pop up to set the color of the lines. (%paint.r)
26. Extend %paint.r to let the user select the type of primitive to draw via a context-menu that pops up when the right mouse button is pressed. Also provide a menu entry for color selection in the context-menu. Primitives the user might want to draw are lines, rectangles and circles.
    F12 saves the currently created image as .png to disk (`request-file`) (%paint+.r)
27. Create an application that lets the user play around with an neuronet consisting of simple threshold-based mccullogh-pitts cells.
    A cell fires if the sum of all inputs ($x_{i\ (0..n)}$) times their weights ($w_{i\ (0..n)}$) is not lesser than the given threshold value.

    $$\text{sum}\ (x_i * w_i)_{0..n} >= th$$

    Nodes are added via left mouse button click, can be dragged around and their threshold value can be edited if a node is double-clicked. Assume a value of 1 for all weights. The inputs come from other cells that are connected to them and are either 0 or 1, if the cell fires or not. The connections between nodes should be created by dragging the lower left corner of one cell over another. If this happens, a line should be drawn indicating the connection. Initial nodes (the ones with no inputs) can be switched on or off.