

---

# SQUIRREL Developer's Guide

Programming Reference

---

**Version 5.3**  
March 11, 2001





*SQUIRREL Developer's Guide*

Copyright ©1999-2001 *Kirilla* . All Rights Reserved

No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of *Kirilla* .

The contents of this document are furnished for informational use only; they are subject to change without notice and should not be understood as a commitment by *Kirilla* . *Kirilla* has tried to make the information in this document as accurate and reliable as possible, but assume no liability for errors or omissions.

*Kirilla* will revise often the software described in this document and reserves the right to make such changes without notification.

Author: Jean-Louis Villecroze

Email: [jl@kirilla.com](mailto:jl@kirilla.com)

Web site: <http://www.kirilla.com>

This document was prepared with  $\text{\LaTeX}$  2 $\epsilon$ .

$\text{\TeX}$  is a trademark of the American Mathematical Society

# Contents

<b>About this Developer's Guide</b>	<b>1</b>
<b>1 Getting Started</b>	<b>3</b>
1.1 What is SQUIRREL ? . . . . .	3
1.2 Requirements for SQUIRREL . . . . .	4
1.3 Installation . . . . .	4
<b>2 SQUIRREL Basics</b>	<b>5</b>
2.1 Console . . . . .	5
2.1.1 The menus . . . . .	6
2.1.2 Activity status . . . . .	7
2.2 Programming fundamentals . . . . .	7
2.2.1 Commands - an introduction . . . . .	7
2.2.2 More about words and variables . . . . .	10
2.2.3 Commands as input to another command . . . . .	11
2.2.4 Mathematical expressions . . . . .	12
2.2.5 Boolean expressions . . . . .	13
2.3 Strings and words . . . . .	14
2.4 Lists . . . . .	15
2.5 Control structures . . . . .	17
2.5.1 Decision structures . . . . .	18
2.5.2 Loops . . . . .	21
2.5.3 Error handling . . . . .	24
2.6 Procedures . . . . .	25
2.7 Local and global variables . . . . .	27
2.8 Using objects . . . . .	28
2.9 Variable Binding . . . . .	29
2.9.1 Get event . . . . .	30
2.9.2 erase event . . . . .	31
2.9.3 set event . . . . .	31
2.10 Adding comments to scripts . . . . .	32
2.11 Including script . . . . .	33
2.12 System global variables . . . . .	34
2.13 Using Add-Ons . . . . .	34

<b>3</b>	<b>Standard Objects</b>	<b>36</b>
3.1	Application Messaging . . . . .	36
3.2	Image object . . . . .	41
<b>4</b>	<b>Primitives</b>	<b>44</b>
4.1	Add-ons management . . . . .	44
4.2	Communication . . . . .	46
4.3	Control . . . . .	52
4.4	Data processing . . . . .	54
4.5	Data structures . . . . .	59
4.6	Exec . . . . .	71
4.7	File Input/Output . . . . .	72
4.8	Image processing . . . . .	74
4.9	Inspector . . . . .	75
4.10	List processing . . . . .	78
4.11	Mail . . . . .	82
	4.11.1 Primitives . . . . .	82
	4.11.2 Mail Object . . . . .	82
4.12	Mathematics . . . . .	85
4.13	String processing . . . . .	94
4.14	Storage . . . . .	100
4.15	Threading . . . . .	108
4.16	Time . . . . .	112
4.17	Workspace . . . . .	114
<b>5</b>	<b>Release notes</b>	<b>117</b>
5.1	Release 5.3 . . . . .	117
	5.1.1 Changes . . . . .	117
	5.1.2 Additions . . . . .	117
	5.1.3 Bugs fixed . . . . .	117
5.2	Release 5.2b . . . . .	118
	5.2.1 Changes . . . . .	118
	5.2.2 Additions . . . . .	118
	5.2.3 Bugs fixed . . . . .	118
5.3	Release 5.2 . . . . .	118
	5.3.1 Changes . . . . .	118
	5.3.2 Additions . . . . .	118
	5.3.3 Bugs fixed . . . . .	118
5.4	Release 5.1b and 5.1c . . . . .	118
	5.4.1 Notes . . . . .	118
	5.4.2 Changes . . . . .	119
	5.4.3 Additions . . . . .	119
	5.4.4 Bugs fixed . . . . .	119
5.5	Release 5.1 . . . . .	119
	5.5.1 Notes . . . . .	119
	5.5.2 Changes . . . . .	119
	5.5.3 Additions . . . . .	119
	5.5.4 Bugs fixed . . . . .	120

5.6	Developer Release 5.0	120
5.6.1	Notes	120
5.6.2	Changes	120
5.6.3	Additions	120
5.6.4	Bugs fixed	120
5.7	Developer Release 4.9	120
5.7.1	Notes	120
5.7.2	Changes	121
5.7.3	Additions	121
5.7.4	Bugs fixed	121
5.8	Developer Release 4.8	121
5.8.1	Notes	121
5.8.2	Changes	121
5.8.3	Additions	121
5.8.4	Bugs fixed	121
5.9	Developer Release 4.7	122
5.9.1	Notes	122
5.9.2	Changes	122
5.9.3	Additions	122
5.9.4	Bugs fixed	122
5.10	Developer Release 4.5	122
5.10.1	Notes	122
5.10.2	Changes	122
5.10.3	Additions	122
5.10.4	Bugs fixed	123
5.11	Developer Release 4	123
5.11.1	Notes	123
5.11.2	Changes	123
5.11.3	Additions	123
5.11.4	Bugs fixed	123
5.12	Developer Release 3	123
5.12.1	Notes	123
5.12.2	Changes	124
5.12.3	Additions	124
5.12.4	Bugs fixed	124
5.13	Developer Release 2	125
5.13.1	Notes	125
5.13.2	Changes	125
5.13.3	Additions	125
5.13.4	Bugs fixed	126
5.14	Developer Release 1	126

# About this document

Squirrel is a programming language in the Logo family. There are some distinct differences between SQUIRREL and Logo, some stemming from SQUIRREL taking advantage of advanced features of the Be operating system (BeOS).

At this time, neither SQUIRREL nor this document is perfect. We would appreciate notification of any errors found.

This guide is divided into four parts:

**Getting started** introduces SQUIRREL and describes how to install it

**Squirrel basics** shows, mostly by examples, how to use the SQUIRREL language

**Primitives** lists and describes all the primitives

**Release notes** contains pertinent information on the releases

It should be understood that several additional features will be included in upcoming releases. These features include advanced scripting capabilities and other additions for the Add-ons. Some Add-ons will be complete while others will not. An Add-on API in the near future will allow third party additions, which will enhance SQUIRREL's usefulness.

We have included several documentation conventions in this document. These are:

- All code elements are presented in a distinct font like `print "foo`
- Primitive syntax is often a combination of code element and italic font. The part in italic is always the input to the primitive.
- Primitive inputs use special kind of symbols :
  - **(word)** indicate that the input is optional
  - **word | number** indicate that the input can be either a word or a number
  - **(word)+** indicate that the primitive can take on several words as input, but at least one is required.
  - **(word)\*** indicate that the primitive can take on several words as input, but one is optional.

A bar in the right or left margin helps to locate an update or addition (from the previous revision) in this document.

A big *Mahalo*<sup>1</sup> to (in chronological order) :

- Henry van Eyken
- Susan Banh<sup>2</sup>
- Ulrich "scholly" Scholz
- Guido Soranzio
- Jonas Sundström

for their much appreciated contributions towards rewriting and editing this document !

Please enjoy reading this manual and have fun with SQUIRREL !

Jean-Louis, March 11, 2001

---

<sup>1</sup>"Thank you" in Hawaiian

<sup>2</sup>and all my love

# Chapter 1

## Getting Started

### 1.1 What is SQUIRREL ?

The initial intention of the author of SQUIRREL was to design a programming language simple, yet powerful as a scripting language. While contemplating this idea, it dawned on the author to build a language based on an existing one with a track record of being easy to learn. He then thought of Logo. Logo itself is a descendant of LISP, a *List Processing* language designed to be a tool for the development of artificial intelligence.

Incorporated in SQUIRREL is Turtle Graphics. Turtle Graphics is perhaps Logo's most well known feature. It allows children to explore geometry by issuing commands to a little device called *turtle*. In SQUIRREL, it is *Skippy*, the squirrel, who takes on turtle's role.

Those who are familiar with Logo are aware of the two things it offers: a philosophy of education and a computer language designed to realize that philosophy. It provides children, as well as adults, control over computers. Unlike developer programming languages like C++, Logo permits immediate visualization and exploration. Although it was designed to be a tool for the mental development of children, people of all ages can doodle with it in a way people have made preliminary sketches for envisaging ideas they intend to embody for centuries now. Some of the works may have included scripts for controlling a computer and for giving better insights through a learning process called *concretization*.

A script is a series of instructions written to direct the computer to accomplish one or more task. Unlike an application, a script is not compiled; that is to say, it is not converted by a compiler into executable code which can then be run and is understood by the computer hardware. Instead, when a script runs, a program called an *interpreter* translates each command line it encounters into machine code and immediately executes it. Hence, scripts are easier to write than applications, but they do execute slower. Writing programs for compilation requires detail and painstaking preparation in designing. Scripts, on the other hand, are easily written and modified. This distinguishes compiled languages from interpreted ones. The source coding of a compiled language must begin with a complete understanding of the problem, followed by a detailed solution. The source coding of an interpreted language permits



trial and error, allowing one to explore better ideas along the way.

Scripting languages differ in detailed objectives and in readability from compiled languages. SQUIRREL combines the simplicity and power of Logo with convenient scripting for work such as file manipulation or GUI design. Written for BeOS, SQUIRREL provides easy access to key features such as multi-threading. Furthermore, SQUIRREL is readily extensible by third party add-ons. SQUIRREL has been designed to be a perfect tool for beginners, as well as for experienced programmers. It is fun to learn and use!

## 1.2 Requirements for SQUIRREL

SQUIRREL can be installed and run on a PC clone, also referred to as an Intel system even though the main processing chip is not necessarily of Intel manufacture. The required operating system is BeOS R5.x. At least 4 Mb of RAM and 7 Mb of hard-drive space is needed.

## 1.3 Installation

This release is distributed as a zip file. Unzip the file and execute the *Squirrel installer* to install SQUIRREL. This places all of SQUIRREL's decompressed files, except the Add-ons, in a directory Squirrel you will have chose, the most common place is in `/boot/apps`. The Add-ons are placed in `/boot/home/config/add-ons/Squirrel`.

There are two executables in SQUIRREL :

`Squirrel.5.3` used primarily as the *Preferred Application*  
`Squirrel's Console.5.3` to interact with the interpreter

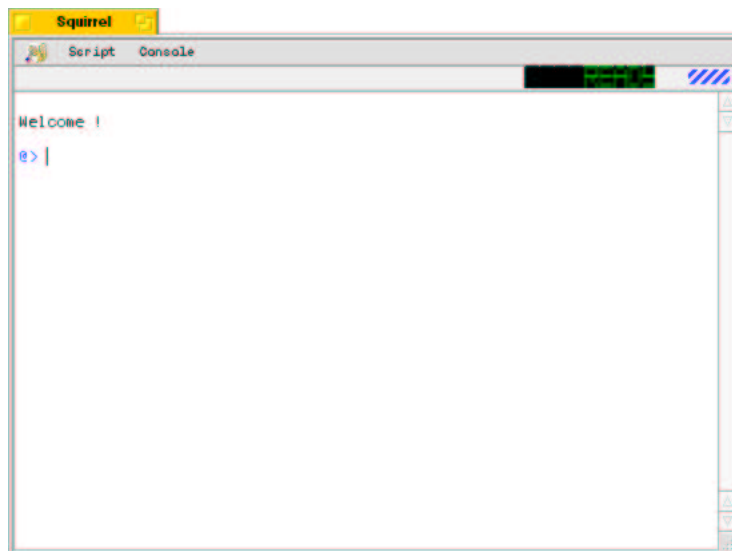
## Chapter 2

# SQUIRREL Basics

This guide concentrates on showing how to program in SQUIRREL . Demonstrations of the capabilities of SQUIRREL are not provided here. However, the user may find some demo files that may be run by opening them from SQUIRREL .

### 2.1 Console

When SQUIRREL is running on its console version, the first thing one notices is the main window or console. The BeOS Terminal may also be used; details will be given at the end of this section.



Console and *BeOS Terminal* are used in the same way. Prompts like @>, invite the user to enter their statements containing SQUIRREL commands. These statements are also known as command lines.

Once an entry has been made, pressing the <entry> key causes SQUIRREL's interpreter to examine (or parse) the command line for immediate execution. The interpreter will print out a warning in red lettering if it does not understand the request. This will occur if the user made an improper entry or if an error occurred during execution.

The keyboard's left and right arrow keys permit one to move the cursor along a command line. Pressing the up arrow copies the previous entry. Entered text may be copied and pasted using the Alt-C and Alt-V key combinations. The keyboard's function keys may be configured to handle a sequence of tasks. This will be described in the chapter *Squirrel menu* under *Preferences*.

Experienced users wishing to perform computation-intensive work may wish to use the *BeOS Terminal* program rather than SQUIRREL's console. To start, they should enter the following line to the first line of their script file:

```
#!/boot/apps/Squirrel/Squirrel.5.3
```

Adapt this line to wherever you have installed SQUIRREL.

An alternative approach is to set `Squirrel.5.3` as the preferred application of their script file.

This is treated like the console, except that this does not accept interactive commands.

### 2.1.1 The menus

- SQUIRREL menu

The SQUIRREL menu provides a number of options related to SQUIRREL itself.

**About** displays a window with information about the current version of SQUIRREL.

**Help** starts your favorite browser and loads the HTML version of this document.

**Preferences** SQUIRREL behaves according to a number of default settings that may be altered by the user.

The Preferences window has three tabs:

**Start-up** lets the user enter the name of the folder containing the SQUIRREL script files (extension `.sqi`), which will be loaded at start-up. This option works for the SQUIRREL console, and not the *BeOS Terminal*.

**Console** shows three panels. One panel lets the user select the type and size of font used in the console. Another permits the user to save a record entered in the console. *Size* is the number of most recent lines saved. A third panel serves to bound macros to each of the keyboard's twelve function keys.

**Interpreter** gives the user control over certain behavior of the interpreter, such as automatic garbage collection and thread priority. (Refer to the release notes for more detail.)

**Quit** closes Squirrel.

- Script menu

**Open a script file** loads the code into the interpreter from one or more SQUIRREL script file.

- Console menu

**Clear** clears the console of all text and displays a new prompt.

**Save text as ...** This option permits one to save the console's content as a text file.

### 2.1.2 Activity status

Below the menu bar is a bar with two widgets showing the activity status of the interpreter. The left widget is a *LED-style* widget which shows the elapsed time of the last command executed (in seconds). The right widget indicates whether or not the interpreter is active.

## 2.2 Programming fundamentals

The following pages cover SQUIRREL's syntax, by providing lots of simple examples. Users with Logo experience are also encouraged to pursue this material since the syntax in SQUIRREL is not entirely identical to that of Logo!

### 2.2.1 Commands - an introduction

In Logo, a command is either a *primitive* or a *procedure*. Primitives are predefined and are immediately available to SQUIRREL's interpreter. We demonstrate this with some following examples. Chapter 3, *Primitives* lists and describes these commands. Procedures are created by combining commands and using parameters. Thus, they become commands for SQUIRREL to do more complicated things. Procedures may call on already existing procedures. Typically, procedures are created by the user themselves. Users may store them in a library<sup>1</sup> for future use.

Users may give procedures any name they wish, provided that name has not already been assigned to an existing primitive or procedure<sup>2</sup>. The name can take on any length, but it must be an uninterrupted string of characters. Characters which are permitted include dots (.), underscores (\_), and question marks (?). SQUIRREL is case-sensitive, hence `Myfunc` and `myfunc` identify different procedures..

Here are some examples of SQUIRREL statements that show how primitives and data are combined into instructions. The reader is encouraged to enter these examples in the console. People familiar with Logo will notice some of SQUIRREL's departures from Logo. These have been deliberately incorporated by SQUIRREL's author.

The syntax of a SQUIRREL's command is:

```
order input1 input2 input3 ....
```

#### Example 1

```
@> print 6+4
10
```

Note. – The interpreter also accepts spaces between numbers and mathematical symbols, eg. :

#### Example 2

---

<sup>1</sup> SQUIRREL script files

<sup>2</sup> If the procedure already exists, the new procedure will replace the old one

```
@> print 6 + 4
10
```

#### Example 3

```
@> print sum 6 4
10
```

Two primitives are invoked in Example 3, `print` and `sum`. This is an instance in which the argument to `print` is a command itself. The section *Commands as input to another command* discusses this kind of situation. Observe that it is obviously necessary here to put spaces between the numbers entered.

**Rule: All data on a statement line must be separated from each other by spaces.**

#### Example 4

```
@> print "Hello "world
Hello world
```

SQUIRREL and Logo attaches a special meaning to *word*. That special meaning is brought out by using a double-quote as shown here. Typically, a word may store a variable. Variables may be other words and numbers. In SQUIRREL, a variable may also be a string, which is defined as anything that is enclosed between single quotes. This will be shown in the following examples.

#### Example 5

```
@> make "a "Hello
@> make "b "world
@> print :a :b
Hello world
```

Here the words `"a` and `"b` are made to store values. In this case, these values are words. The colons in the `print` statement causes the interpreter to print what has been stored.

#### Example 6

```
@> make "a 'Hello world!'
@> print :a
Hello world!
```

This example shows an important difference between how SQUIRREL and Logo handles their strings.

#### Example 7

```
@> make "a 12
@> make "b 4
@> print :a*:b
48
```

Here the labels "a and "b are made to store or assign numerical values. The `print` command displays the product of these values. A double quote defines a container created to hold a variable. A colon points to the content of that container.

If we had used double quotes instead of colons in the `print` command, would have obtained the names of the containers, a b:

#### *Example 8*

```
@> make "a
Wrong number of arguments made
```

When one runs the last line in SQUIRREL 's console, the result appears in red.

#### *Example 9*

```
@> print :a
Unknown variable [] a
```

In other words, the container is empty.

#### *Example 10*

```
@> make "a "Squirrel
@> print slength :a
8
```

The interpreter printed out the number of characters stored in a. In this example, the double quote identified a word; it was not counted as part of that word. We emphasize this point because we have found literature on Logo which erroneously states that the double quote is part of the word. This may cause mental confusion. The double quote merely states that what is attached is to be treated as a word, or a container that holds a value.

#### *Example 11*

```
@> make "b 'Skippy'
@> print slength :b
6
```

The interpreter printed out the number of characters transferred to it, and then stored in the word "b. This example shows how single quotes are used to identify a string; they are not part of that string. The term `slength` is a primitive.

The small set examples just given, was to simply get the user's feet wet. The following pages go into more detail.

### 2.2.2 More about words and variables

The primitive `make` is used to:

- a. create a word if it does not already exist and then
- b. assign a value to it

That value may be a number, a word (a string of characters preceded by a double quote), a string (anything contained between two single quotes) or a list. A string may be empty, but there does not exist an empty word.

#### *Example 12*

```
@> make "word "apples, make "string 'We have', make "number 12
@> print :string :number :word
We have 12 apples
```

There may be more than one command on a command line, but these must be separated by commas. Clearly, the value stored by a word may be a number, a word, or a string.

#### *Example 13*

```
@> make "my_list ["man "wife "son "daughter]
@> show :my_list
["man "wife "son "daughter]
```

Logo works with lists of data. These lists are recognized by placing them within square brackets. SQUIRREL's string allows a list to be stored in a word.

#### *Example 14*

```
@> make "print 'letters'
@> print :print
letters
```

This example merely demonstrates how words may have the same name as primitives – or procedures, for that matter.

#### *Example 15*

```
@> make "number '12'
@> print :number * 3
Invalid operation [String * Object]
```

Reason: The interpreter did not recognize `'12'` to be a number. Multiplication was impossible since it was perceived as a string.

#### *Example 16*

```
@> make "empty ' '
@> make "fruit "oranges
@> print :empty :fruit
oranges
```

:empty had nothing to offer, so only the contents of :fruit was printed.

Note. – Programmers with experience in Pascal, C, or C++ will recognize that *data typing* is not required here. Data typing is a procedure which specifies the particular kind of variable that is set aside in the computer's memory, e.g. strings, integers or floating point numbers. A lot of hassle is avoided when this is not required. This permits users to explore ideas without the constraint of data typing.

### 2.2.3 Commands as input to another command

When SQUIRREL's interpreter parses a command line, it analyzes the line from left to right and then attempts to execute the given instructions in that order.

#### Example 17

```
@> print sum 4 5 3
12
```

The interpreter first sees the command, "print". It expects that something is to be printed. That something is called the input to the command print. In this instance, the input is another command, the primitive sum. Because sum is another command, the interpreter expects sum's inputs to follow. The inputs to sum are 4, 5 and 3. Another way of putting it is that the entire command:

```
sum 4 5 3
```

is seen as the input to print.

If the sum of the numbers in the previous example were followed by something else to be printed, it becomes necessary to separate the sum from that something else. We use parentheses to do this:

#### Example 18

```
@> print (sum 4 5) 'is the value of 4 + 5'
9 is the value of 4 + 5
```

Without the parentheses, the command sum would attempt to add the string but, of course, it cannot! If this seems a little tricky, remember that parentheses are always needed on a command line where there is more than one command with inputs.

What will happen if we do not use parentheses? Let's see:

#### Example 19



```
@> print sum 4 5 'is the value of 4+5'
9
```

In this instance, the interpreter attempts to evaluate the sum of three entities, the last of which has no numerical value. Hence, the primitive `sum` will ignore the third input.

Note. – The reader may wish to compare this result with that in Example 15 of the previous section, *More about words and variables*:

#### Example 20

```
make "number '12'
print :number * 3
Invalid operation [String * Object]
```

Why did it not print 0 ? This is because `:number` has no numerical value whereas 0 is a numerical value. Thus, all the interpreter interprets is the instruction:

```
* 3
```

The interpreter then asks the question, "how much times three?" Since there is no answer to the conundrum, it prints a warning in red.

### 2.2.4 Mathematical expressions

SQUIRREL evaluates mathematical expressions perfectly e.g.  $2.34567 * 10^{**3} = 2345.67$ . It also accepts scientific notation. Here, SQUIRREL surpasses ordinary Logo written for children.

SQUIRREL recognizes the following operators:

<b>**</b>	Power	<b>:x**4</b>
<b>*</b> <b>/</b> <b>//</b> <b>%</b>	Multiply, divide, integer division, remainder	<b>:x*4</b>
<b>+</b> <b>-</b>	Add, subtract	<b>:x-4</b>

Table 2.1: Arithmetic operators from highest to lowest precedence

#### Example 21

```
@> print 'solutions:' 10+4 3-2 'and' 12+5*2
14 1 and 22
```

Here, the numbers 10 and 4 are related by an operator.  $3-2$  and  $12+5*2$  are also related by an operator. The mathematical operations are therefore executed. The three mathematical expressions are independent from one another; hence, the interpreter's response is a list of data separated by spaces.

In the third expression of this example, multiplication takes precedence over addition. Commonly accepted rules for operator precedence are adhered to – see Table 2.1.

#### Example 22

```
@> print ((2.3+ 3.7) * 2**3)**2
2304
```

This example contains no curly brackets nor square brackets. SQUIRREL requires these brackets for other computing tasks which will be discussed later. Only parentheses are used in mathematical operations.

#### *Example 23*

```
@> print (sum 12 8) / 5
4
```

This example demonstrates that primitives may be placed within a mathematical expression. For assurance, parentheses should be used to clarify the command line for the interpreter. The same applies to procedures.

#### *Example 24*

```
@> print 12.3 * -.5
-6.150
```

This demonstrates that the absence of a leading zero presents no problem. The interpreter will correct for that in the calculation.

#### *Example 25*

```
@> print (rsin .2345)**2
0.054
```

This example corresponds to what is traditionally written as:  $\sin(0.2345)^2$ . The *r* in *rsin* refers to an angle expressed in radians.

Further examples of mathematical functions are given in Chapter 3. *Primitives*.

### **2.2.5 Boolean expressions**

Boolean expressions are used mostly for testing how values compare. We will encounter them in the chapter on control structures. The following table summarizes the operators now available in SQUIRREL.

The boolean value *true* and *false* are part of Squirrel grammar and are recognized as such. For example :

```
@> do_something 45.5 true "hello false
```

=	equality test	:x = 5
<>	non-equality test	:x <> 5
<	less than test	:x < 3
<=	less than or equal test	:x <= 3
>	greater than test	:x > 3
>=	greater than or equal test	:x >= 3
not	negation test	not :x<3

Table 2.2: Booleans Operators

## 2.3 Strings and words

Let's recapitulate. SQUIRREL's syntax distinguishes between words and strings. Words are strings of characters whereas strings may be composed of more than one word.

Like Logo, SQUIRREL identifies a word with a leading double quote. For those unfamiliar with Logo, the term *word* is likely to be synonymous with the term *identifier*. A string is identified by two single quotes, marking the beginning and end of the string. SQUIRREL's strings serve to store text which may be manipulated. Here are some examples of words:

```
"computer
"this_is_a_word
"<<..>>
```

Most commands that accept a string will also accept a word, but the inverse is not true. Commands designed to accept a word will not accept a string.

*Example 26*

```
@> print slength 'this is a test'
14
```

The string may contain any number of words. The answer is the string's character count.

*Example 27*

```
@> make "a "horse
@> make "b 'feathers'
@> print :a + :b
horsefeathers
```

This is an example of concatenation. Concatenation is one way of manipulating text.

Why does Squirrel have strings if words themselves are enough? To answer this question, consider this example:

*Example 28*

```
@> print "this "is "my "long "string
this is my long string
```

To provide that answer, the machine had to operate on five distinct words. By joining all these words into a single string, the computer's work is reduced to a single operation. Thus, working with strings tends to be faster.

Some special characters are required to initiate important computer operations that cannot otherwise be taken care of. Backslashes in a string serve to identify those special characters that control aspect of the computer's behavior. Here is a list of them and their actions:

<code>\n</code>	line feed (newline)
<code>\t</code>	horizontal tabulation
<code>\v</code>	vertical tabulation
<code>\b</code>	backspace
<code>\r</code>	carriage return
<code>\f</code>	form feed
<code>\'</code>	single quote
<code>\\</code>	backslash

Table 2.3: Backslashed characters

Here's how they are used:

```
@> make "a 'this is a test\n'
@> make "b '\t Squirrel is cool !!!!!\n'
```

It is best to copy and paste these lines into the console to observe the effects.

– **Note:** Logo is not familiar with strings which are made up of separate words. When the word string is used in Logo, it is interpreted as the string of characters that make a word. A word is one or more characters, usually offset by blank spaces or by a bracket before or after it.

## 2.4 Lists

The basic list elements in SQUIRREL are: numbers, strings, words, and other lists. It is possible<sup>3</sup> to create list with references to variables or primitive/function calls. In this case, when the list is evaluated, all such elements will be processed and a new list is created.

To introduce lists, we again present Example 28 from the page headed *Strings and Words*:

```
@> print "this "is "my "long "string
```

---

<sup>3</sup>since release 5.1

The interpreter is instructed to print out a list of words that are separated by spaces. Alternately, the list might have been prepared before hand, like in this sequence of statements:

*Example 29*

```
@> make "a ["this "is "my "long "string]
@> print :a
this is my long string
```

The first statement contains a list with five elements, all of them being words. That list is stored in the variable `a`. The name of the variable is given as a word. As shown on the second statement, accessing the variable value is done by using the semicolon. Typically, a list is enclosed by square brackets.

*Example 30*

```
@> make "mylist ['this is a list' 3.45 'with' 56.6
["various "things "inside]]
@> print :mylist
this is a list 3.45 with 56.6 various things inside
```

*Example 31*

```
@> make "element_a [10]
@> make "element_b "element_a
@> print :element_b
element_a
```

One element has been substituted by another.

*Example 32*

```
@> make "element_a [10]
@> make "element_b :element_a
@> show :element_b
[10]
```

Here, one element has been given the value of another.

The next example show how to use variable and function call in a list:

*Example 33*

```
@> make "a 34
@> make "b 45
@> make "lst [:a :b :a+:b (max :a :b)]
@> show :lst
[34 45 79 45]
```

A concluding note of some historical interest. – As mentioned before, LISP is a language of artificial intelligence on which Logo is based on. In LISP, individual units are atoms. Atoms are letters, numbers, or any string composed of alphanumeric characters. Both atoms and lists are referred to as s-expressions (symbolic expressions). Logo's list elements are similar. A big difference is that the atoms of LISP cannot be decomposed (the word atom comes from classical Greek meaning indivisible). Atoms are no longer atoms in Logo. Here, they can be broken down. The string 'horsefeathers' may be split into 'horse' and 'feathers'. However, the meaning of a list is no different in the Logo family of languages from that of LISP. There is a difference in notation. LISP uses parentheses to identify lists, whereas Logo and Squirrel uses square brackets. Lists are used for building elements for programming.

## 2.5 Control structures

A control structure determines the order in which the interpreter does its work. Programmers recognize sequence structures, decision structures, and iteration structures.

A sequence of commands is called a sequence structure. Example:

```
@> make "a 2 + 3
@> make "b 'apples'
@> print :a :b
```

A structure that lets the interpreter decide which action to take is called a *decision structure*. A decision involves a criterion, or a basis for deciding whether to do *this* or *that*. Example:

```
@> if :a > 10 {print 'ok'}
```

Here, if the value of :a <= 10, nothing will be printed.

A structure that causes the interpreter to go through a repeated cycle of instructions is called an *iteration structure*. This structure is commonly referred to as a *loop*. Typically, a loop involves decision making with a base case for ending the iteration. Example:

```
@> for ["i 0 10] {print :i}
```

The result will print the numbers from 0 to 10. Once 10 has been printed, the interpreter terminates the iteration since decision has been made to stop printing. The interpreter will then act on the next statement.

In SQUIRREL, for various reasons, lists containing primitives or procedures are enclosed within curly braces { }. These lists are referred to as *blocks*. The block notation also improves the readability of a program as well as the efficiency of execution.

One may endow variables with the value of a block, e.g.

```
@> make "ablock {print 'hello world'}
@> run :ablock
```

The answer supplied is: `hello world`

Sadly, a block may be empty. It will then be denoted by `{ }`.

Commands for decision structures:

```
if
ifelse
test
iftrue
iffalse
```

Commands for iteration structures (loops):

```
for
repeat
do.while
while
do.until
until
foreach
```

### 2.5.1 Decision structures

#### **if**

An `if` statement tests whether an expression meets a criteria and if so, executes a command. There are, therefore, two parts to the statement: the test and the command. What follows the word "if" is a Boolean expression. The Boolean expression can either return TRUE or FALSE. The command to be executed is held between curly brackets. The curly brackets identify a block.

*if expression block*

*Example 34*

```
@> make "a 78
@> if :a>10 {print 'ok!!'}
ok!!
```

`if` supports a third input which must be a block. If this input is specified, the `if` will act like an `ifelse`

**ifelse**

An `ifelse` statement contains two blocks. If the Boolean expression returns `TRUE`, then the first block is executed. If it returns `FALSE`, then the second block is executed.

`ifelse expression block block`

*Example 35*

```
@> make "a 3
@> ifelse :a>10 {print 'ok'} {print 'not ok'}
not ok
```

**test**

Execution of this test will assign a value that is either `TRUE` or `FALSE` to `test`. Typically, such a test statement is followed by an `iftrue` or an `iffalse` statement. The syntax is:

`test expression`

A call to this command will always replace the previous value, so only the last `test` is valid.

**iftrue**

`iftrue` will execute the associated block if `test` returned a Boolean `TRUE`. The syntax is:

`iftrue block`

*Example 36*

```
@> make "a 10
@> make "b random 10
@> test :a<:b
@> type 'here'
@> iftrue {print 'my random number was bigger than 10'}
```

In this example, the output could either be `here` or `my random number was greater than 10`.

**iffalse**

`iffalse` will execute the associated block if `test` returned a Boolean `FALSE`. The syntax is:

`iffalse block`

*Example 37*



```
@> make "a 10
@> make "b random 10
@> test :a<:b
@> type 'here'
@> iffalse {print 'my random number was smaller than 10'}
```

In this example, the output could either be `here` or `my random number was smaller than 10`.

### **switch**

Occasionally, a script may contain a series of decisions in which the same variable or expression is tested. SQUIRREL provides the `switch` multiple-selection structure for this special purpose.

A `switch` is composed of a series of `case` and `range` labels, and an optional `other` label. The variable or expression to be tested could be of any basic type (number, string, word, list).

The following example tests the value of the variable `var`:

#### *Example 38*

```
switch :var {
    case 2 {
        print 'var is 2'
    }
    case 45 {
        print 'var is 45'
    }
    case 46 50 55 {
        print '46 50 or 55'
    }
    range 60 100 {
        print 'between 60 and 100'
    }
    other {
        print 'var is something else'
    }
}
```

The label `case` tests the value for equality with at least one of the inputs of the case. The `range` label tests if the value is between the 2 inputs. `other` will be executed only if all the tests have failed.

## 2.5.2 Loops

There are several ways of looping in SQUIRREL :

### **for**

This command will execute a block repeatedly in accordance with the contents of a settings list. The first element of the list is a word that is assigned values from a range of numbers. The fourth element is optional. Any fourth element indicates a step, as shown in Example 40. The syntax is:

`for [word number number (number)] block`

*Example 39*

```
@> for ["i 1 100] {print :i}
1
2
3
4
...
99
100
```

*Example 40*

```
@> for ["i 50 1 -10] {print :i}
50
40
30
20
10
```

Here, the fourth element in the list specifies a step. The step is negative because the loop runs from an initial value of 50 to a lower value.

**When a loop has run its course, the first element, the "i in this instance, is destroyed.**

### **repeat**

The statement `repeat` executes a block a specified number of times. The syntax is:

`repeat number block`

The first input will be the number of times we want to execute the block.

*Example 41*

```
@> repeat 5 {print 'Hello John !'}  
Hello John !  
Hello John !  
Hello John !  
Hello John !  
Hello John !
```

### **do.while**

This command will cause repeated execution of the block as long as the Boolean expression returns the value TRUE. The syntax is:

`do.while` *block expression*

#### *Example 42*

```
@> make "j 10  
@> do.while {print :j,make "j :j-1} :j>0  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1
```

### **while**

The difference between `while` and `do.while` is that with `while`, a Boolean expression is evaluated before a block is executed. If the Boolean expression returns FALSE the first time, the block will never be executed. The syntax is:

`while` *expression block*

#### *Example 43*

```
@> make "j 10  
@> while :j>0 {print :j,make "j :j - 1}  
10  
9  
8  
7
```

```
6
5
4
3
2
1
```

### **do.until**

With `do . until`, looping continues until the Boolean expression returns TRUE. Notice that the block will always be executed at least once. The syntax is:

```
do .until block expression
```

#### *Example 44*

```
@> make "j 10
@> do.until {print :j,make "j :j-1} :j<0
10
9
8
7
6
5
4
3
2
1
0
```

### **until**

The difference between `until` and `do . until` is that with `until` a Boolean expression is evaluated before a block is executed. If the Boolean expression returns TRUE the first time, the block will never be executed. The syntax is:

```
until expression block
```

#### *Example 45*

```
@> make "j 10
@> until :j<10 {print :j,make "j :j-1}
10
```

**for.each**

The command `for .each` considers consecutive values found in a list. The syntax is:

`for .each word list block`

The word used will be the variable containing the value from the list. The block will be executed as long as there is still an element in the list.

*Example 46*

```
@> make "i 0
@> for.each "e [5 2 6 7 8 1 5] {make "i :i+:e**2,print :i}
25
29
65
114
178
179
204
```

**2.5.3 Error handling**

SQUIRREL will abort the execution of a script when an error occurs. Hopefully, it's possible to catch an unexpected error or even throw your own error using a `catch` and then a `.`

The syntax for catching an error is:

`catch word block (block)`

The syntax for throwing an error is:

`throw word thing`

*Example 47*

```
catch "divbyzero {
    if :a=0 {
        throw "divbyzero 'The value of a is zero !!'
    }
    make "j :b/:a
} {
    print 'an error has occurred'
}
```

In this example, we caught an error thrown by our self when the value of `a` was zero. The `throw` order accepts two inputs: *the name of the error* and the *return value*, which could be anything. The error's name must match the word specified in the `catch` order, otherwise the error will not be caught.

In the case of an unknown error, one that did not come from a `throw` order, you may specify the word `"error"` to catch it. An example of such an error would be a bad number of arguments.

An optional second block will be executed when an error occurs only.

*Example 48*

```
catch "error {
    print 'we generate an error bellow'
    print gseq 23
    print 'this line will never been executed if we have an error'
}
```

The order `gseq` is a primitive requiring at least two inputs. Using it alone will generate an error and we will need to catch it. The execution of the script will resume from the order after the `catch`.

Whenever an error is raised, the global variable `_error` will contains the error message.

## 2.6 Procedures

Procedures encapsulate a set of orders (a block of orders) and parameterize them.

The syntax to define a function is:

```
to word ref1 ref2 ref3 ....
...
end
```

There is no need to use the `{}` to mark a block as `to` and `end` to perform this task. For examples, to define a function which subtracts four from a value and then prints the result, we write in a script file:

```
to sub4 :a
    print :a-4
end
```

Calling it is the same as calling any primitive:

```
@> sub4 6
```

When defining a procedure, you may want to return a value<sup>4</sup>. For examples, if we want to return the result of `:a-4` and not print it, we would use the `output` primitive which returns this input as the

---

<sup>4</sup>A procedure which returns a value is called a *function*

result of the function and then stops the function. If you just wanted to stop the function, you may use the order `stop`.

When a script need to return an error code, the `output` primitive can be used to return a value (must be an integer). The script will then be ended and this value returned to the caller of the script.

We now modify our function to reflect what we want:

```
to sub4 :a
  output :a-4
end
```

Calling is similar, except that we could take the output as the input of another order:

```
@> print sub4 6
@> make "a sub4 6
```

It's possible to define default values for the inputs. To do this, we use a list instead of a variable reference. The syntax of the list is:

```
[ ref value ]
```

For example:

```
to sub :a [:b 4]
  print :a-:b
end
```

If we call it:

```
@> print sub 6
```

We will get 2, but calling it:

```
@> print sub 6 2
```

will give us 4.

Using default values is very powerful, but there's an obvious limitation. This is demonstrated in the next example:

```
to sub :a [:b 4] :c
  print :a-:b+:c
end
```

This function is syntactically correct and will not generate any errors during parsing. The execution will also work if you always specify three inputs. But if you specify only two, like:

```
@> print sub 6 2
```

An error will be reported during execution because an input is missing. Here, the value for `:c` is not known.

One of the advantages of using default values is the ability to use a previous argument. e.g.:

```
to foo :a [:b :a+4] [:c (random 40)]
  print :a+:b+:c
end
```

Calling it:

```
@> print foo 10
```

Will change the value 10 to `:a`, the value 14 to `:b` and a random number between 0 and 40 to `:c`.

This is done just before calling the function. Notice that only the global variable and the previous input are known at this time.

Once defined, a function is used like any other SQUIRREL primitive. When called, each input is assigned to a variable specified in the corresponding function.

## 2.7 Local and global variables

A function is executed by SQUIRREL in a scope, created specially for the function. It is still possible to access global variables you want to set, get or modify, but creating a local variable using the order `local` will require a step more than what is required for creating a global variable.

When a function ends, all the local variables and values created within the function will be destroyed (except the one still in use, e.g. a value returned by the function using `output` or the value set in global variables).

Creating a local variable using `local` follows the syntax:

```
local word word word word ....
```

You may use as many calls to `local` as you want and specify as many variable names (words) as you need each time. Setting the value to a local variable is the same as setting one for a global variable. For examples:

*Example 49*

```
to foo :a :b
  local "c "d
  make "c :a-5
  make "d :c//2
  output :d*2
end
```



Using the primitive `make.local` simplifies the process of creating and initializing a new local variable in a single call. The previous example becomes:

*Example 50*

```
to foo :a :b
  make.local "c :a-5
  make.local "d :c//2
  output :d*2
end
```

If you wish to modify the value of your local variable later in your function, you will use `make`.

When defining a local variable, you may *shadow* a global variable. Your function will not have access to the global variable and you will use its own local variable. For examples:

*Example 51*

```
make "c 10

to foo :a
  local "c
  make "c :a
end

foo 4
```

This will not change the value of the global variable `c`.

A function can create a global variable using the order `make` if the variable hasn't already been specified using `local`. You have to make sure when you're using variables inside your function that you define them as local variables. Otherwise the program will create global variables which will stay long after your function is over.

## 2.8 Using objects

Since *Developer Release 3*, SQUIRREL has been featuring a new type of data called an *object*. The use of an *object* is not really different from that of a simple data, e.g. a number or a string.

The Add-on *Data Structures* for examples, add to SQUIRREL's various data structures other than the `list:Vector` and `Dictionary`.

*Objects* in SQUIRREL have the peculiarity of possessing certain primitives (called *methods*) which only apply to an object.

Using a *method* is different from using a primitive :

*Example 52*

```
@> make "myarray Vector 10
@> $myarray~set 1 45
```

In this example, we are creating a `Vector` object, which is a simple array of 10 elements. We set the element 1 to the value 45. You will notice the use of `$` instead of `:` and the presence of `~`. The syntax to call a method on an object is :

*`$(variable name)~(method name) arguments of the method`*

Passing an object to a command or a function is the same as that for any standard data, using `:`.  
Example :

#### *Example 53*

```
@> make "myarray Vector 10
@> print :myarray
0 0 0 0 0 0 0 0
```

Otherwise objects are manipulated in the same way numbers are. For examples:

#### *Example 54*

```
@> make "a Vector [3 2 4 1 5 6]
@> make "b :a+1
4 3 5 2 6 7
```

#### *Example 55*

```
@> make "a Vector [3 2 4 1 5 6]
@> make "b 2.5*:a
7.5 5 10 2.5 12.5 15
```

#### *Example 56*

```
@> make "a Vector [3 2 4 1 5 6]
@> make "b Vector [1 2 3 4 5 6]
@> print :a+:b
4 4 7 5 10 12
```

## 2.9 Variable Binding

In the lifetime of a variable, the variable could have its contents changed, accessed, or the variable could be erased. The primitives `bind` and `unbind` offers the possibility to bind a variable to these events and execute a block or a function each time an event occurs.

A variable accepts only one binding for each event. If we set two binding sequentially on a variable, only the last one will be executed.

### 2.9.1 Get event

Each time the variable is read (or fetched), this event is generated. This kind of event occurs anywhere where the variable is used like in a `print` or in a math operation. For examples:

*Example 57*

```
@> make "b 4
@> bind "b "get {print 'b has been read'}
@> make "a :b+5
b has been read
```

When executed, the example will access the variable `b` and then execute the block set for the binding. We could have also used a function instead of a block. In this case we would have defined a function with one input since SQUIRREL would fill the first input of the command with the value of the variable :

*Example 58*

```
to vget :val
  print 'b read with value :' :val
end

@> make "b 4
@> bind "b "get "vget
@> make "a :b+5
b has been read with value : 4
```

Using a function instead of a block will allow us to make changes to the variable's value. The variable's value is not changed, but accessing the variable will return another value other than the real one. To do this, we just change our function to return the value we want :

*Example 59*

```
to vget :val
  print 'b read with value :' :val
  output 5
end

@> make "b 4
@> bind "b "get "vget
@> make "a :b+5
b has been read with value : 4
```

If we print the variable `a` after executing this example, we will get 10 and not 9.

### 2.9.2 erase event

When a variable is deleted (using the primitive `erase` for example), an event is sent to the variable and the binding is executed. Consider the following example :

*Example 60*

```
to erase :val
    print 'b erased'
end

@> make "b 4
@> bind "b "erase "erase
@> make "a :b+5
@> print :a
@> erase "b
b has been read with value : 4
b erased
```

When the last line is executed, the function `erase` is executed and `b erased` will be print to the console. The function could have returned a value which will be then a Boolean value. If this value is `true`, the variable will be erased, if `false` is returned, the variable will not be erased, as you can see in the next example :

*Example 61*

```
to verase :val
    print 'b erased'
    return false
end

@> make "b 4
@> bind "b "erase "verase
@> make "a :b+5
@> print :a
@> erase "b
b erased
@> print :b
b has been read with value : 4
4
```

### 2.9.3 set event

When the variable is set to a new value, the event `set` is sent to the variable. It's also possible to define a block or a function to execute when this is happening. The function must have two inputs.

*Example 62*

```

to vset :old :new
  print 'value changed to' :new 'from' :old
end

@> make "b 4
@> bind "b "set "vset
@> make "b :b+5
value changed to 9 from 4

```

If the function used for the binding returns a value, this value will be set to the variable. Otherwise the variable will have its value changed. The next example shows this:

*Example 63*

```

to vset :old :new
  print 'value changed to' :new 'from' :old
  output :old
end

@> make "b 4
@> bind "b "set "vset
@> make "b :b+5
@> print :b
value changed to 9 from 4
4

```

## 2.10 Adding comments to scripts

It's often the case that a programmer wishes to include comments to parts of their code. Comments are meant for humans only. SQUIRREL features two variants for this procedure, known as commenting. When code is parsed into instructions to the hardware for translation, comments tell the interpreter to ignore what's written. We identify them as Logo-style commenting and C-style commenting. Users familiar with these programming languages will immediately recognize the styles used.

**Logo-style commenting** Any line intended to be ignored by the interpreter must begin with a semi-colon ( ; )

**C-style commenting** Any line or succession of lines to be ignored by the interpreter must be enclosed by /\* and \*/

*Example 64*

```

make "e 2.718
; e is the base for natural logarithms
print :e

```

One line comments are easiest to handle in this way.

*Example 65*

```
/* we prefer to comment out a big chunk of code
by using the C style method of commenting.
print 'here'
print 4 5 6 7 8
*/
print 'and we execute again'
```

## 2.11 Including script

Using the keyword `#include` in a script file allow to include a file during the parsing.

If we have in a script file a function called `libfunc` and that we wish to use this function in our script, we will use `#include` to access this function:

*Example 66*

```
#include 'mylib.sqi'

libfunc 34
```

`#include` is somehow equivalent to the primitive `load` except that including a file is done during the parsing and not during the execution of the script as it's done by the primitive `load`. For example, if we wish to load a script file only in certain condition:

*Example 67*

```
if (need 'mylib.sqi') {
    #include 'mylib.sqi'
}
```

Will not work the way we want as it will **always** include the file. We must use the primitive `load`:

*Example 68*

```
if (need 'mylib.sqi') {
    load 'mylib.sqi'
}
```

Unless you are in the above situation, using `#include` instead of `load` is more efficient.

## 2.12 System global variables

When a script is run by SQUIRREL, it is possible to get the list of command-line arguments that were given using the global variable `Args`. This variable has a list for value. It is also possible to know the name and complete path of the script file executed by using the global variable `_file`. The variable `_path` will give the path.

For example, let's have the following script file:

```
if (llength :Args)>1 {
  make "dir lindex :Args 2
  make "myfile FilePanel "open "dir ["file] "single "allow ['image/*'] []
  print :dir
}
```

When run with the command: `@@> myscript.sqi '/home'` the `Args` global variable will contains `['./myscript.sqi' '/home']` and the `_file` variable will have: `'/boot/home/scripts/myscript.sqi'`.

The variable `_install` will give the path of the directory where SQUIRREL is installed on the user computer. As well, `_version` will give (as a string) the release number of SQUIRREL.

To know from where a SQUIRREL script has been launched, you can check the global variable `_from`. When the script is runned from a terminal, this variable will hold the word `"terminal`, and when the user have double-clicked on the script icon, and that the Preferred Application is SQUIRREL, the variable will have for value `"tracker`. If the script is running from the SQUIRREL Console, the value will be `"console`.

## 2.13 Using Add-Ons

Prior to the release 5.0, all the Add-Ons providing the primitives were loaded automatically by SQUIRREL. From release 5.0 on, SQUIRREL loads on start-up (except for the console version of SQUIRREL, which load them all) only the following Add-Ons:

- *Communication*
- *Control*
- *Data Processing*
- *Workspace*

To get access to the other Add-Ons, you need to specify in your script which ones to load, using the primitive `use`. For example, if we need to use the `timing` primitive from the *Time* Add-on, we will add this to our script:

```
use 'Time'
```

```
to test :nb
  for ["i 1 :nb] {
    ; do something
  }
end

print 'time : ' (timing "test 1000) 'microseconds'
```

use accepts as many inputs as needed to load all the Add-Ons you need. The name of the Add-Ons need to be given with correct spelling and it's case sensitive.

The benefit is a real gain in starting time (specialy if you don't need the *GUI* Add-On) for your script and also lower use of the memory.

If you happend to load twice (or more) the same Add-On, SQUIRREL will simply replace all the primitives defined in that Add-On by what it will suppose is a new version of them.



## Chapter 3

# Standard Objects

This chapter describes the various *Objects* than SQUIRREL handle.

### 3.1 Application Messaging

Using the object `Message`, it is possible with SQUIRREL to send or receive messages from/to another application or to receive a drag and drop notification on a widget.

The primitive `Message` is used to create a message object. The primitive accept as only input a number (integer or hexadecimal) which is a value that captures what the message is about.

A message could store pretty much any basics SQUIRREL object like : word, number or list. When a list is given, the list elements will be added with the same data name. They must be of the exact same object type, else the method `add` will fail.

A SQUIRREL script accept to receive message by using a hook function that the script must define : `_MessageReceived`. This function has only one input which is the received message:

*Example 1*

```
to _MessageReceived :msg

    switch ($msg~what) {
        case 0x80 {
            print 'Received :' $msg~find "test
            $msg~reply Message 0xFF
        }
    }

end
```

"int8	an int8 or uint8
"int16	an int16 or uint16
"int32	an int32 or uint32
"int64	an int64 or uint64
"float	a float
"double	a double
"string	a string
"point	a BPoint
"rect	a BRect

Table 3.1: Message datatype

When adding data in a message, it is possible to specify the exact datatype. The following table list them all:

A message object has several methods that allow to get/set the data stored into it, as well as sending it to an application:

#### **Message~add**

```
~add (word | string) thing (word)
```

Put data in the message. The first input is the name of the data to add, the second is the value to store. If a third input is given it must be a word that describe the exact datatype to use. The method will output `true` if no error occurs, `false` else.

```
@> $msg~add "length 56.78
```

#### **Message~delivered**

```
~delivered
```

Output `true` if the message has been delivered and so that it is possible to send a reply to that message.

```
@> print $msg~delivered
true
```

#### **Message~empty**

```
~empty
```

Remove all data from a message. Output `true` if no error is encountered.

```
@> $msg~add "test 'hello there'
@> $msg~empty
@> print $msg~has "test
false
```

**Message~find**

```
~find word | string
```

Output a new object created from the data(s) read in the message for the name given as input of the method. An error is raised if the name is not found.

```
@> $msg~add "test 'hello there'
@> print $msg~find "test
hello there
```

**Message~has**

```
~has (word | string) (word)
```

Output true if a data is found for the name given as first input. If a second input is given, it should match the datatype of the data found.

```
@> $msg~add "test 'hello there'
@> print $msg~has "test
true
@> print $msg~has "test "int32
false
```

**Message~is.empty**

```
~is.empty
```

Output true if a the message is empty of any data.

```
@> $msg~add "test 'hello there'
@> print $msg~is.empty
false
```

**Message~is.remote**

```
~is.remote
```

Output true if a the message has been sended by another application.

```
@> print $msg~is.remote
true
```

**Message~is.reply**`~is.reply`

Output true if a the message is a reply to another message.

```
@> $msg~add "test 'hello there'
@> print $msg~is.reply
false
```

**Message~is.waiting**`~is.waiting`

Output true if a the sender of the message is waiting for a reply.

```
@> print $msg~is.waiting
false
```

**Message~names**`~names`

Output a list of all the data names in the message.

```
@> $msg~add "num 23.56
@> $msg~add "str 'what can I do ?'
@> $msg~add "foo [34 65 23 90]
@> show $msg~names
['num' 'str' 'foo']
```

**Message~previous**`~previous`

Output the message to which the current message is a reply. false is outputted if the message is not a reply.

```
@> make "org $rep~previous
```

**Message~rem**`~rem word | string`

Remove a data from the message. Output true if it succes.

```
@> $msg~rem "test
```

**Message~replace**

```
~replace (word | string) thing (word)
```

Replace data in the message. The first input is the name of the data to be replaced, the second if the value to store. If a third input is given it must be a word that describes the exact datatype to use. The method will output `true` if no error occurs, `false` else. The datatype **MUST** be the same as the data to replace.

```
@> $msg~add "length 56.78  
@> $msg~replace "length 34
```

**Message~reply**

```
~reply message (word)
```

Send a message as the reply to the message given as first input. The message is **NOT** destroyed by this method. A message could then be sent several times. If a second input is given, it must be the word `"reply`. In this case, the method will block until it receives a reply from the target of the message. The method outputs `true` if the reply has been sent.

```
make "r Message 0x78  
$r~add "data 45.8  
$msg~reply :r
```

**Message~send**

```
~send string (word)
```

Send the message to an application whose signature is given as first input. The message is **NOT** destroyed by this method. A message could then be sent several times. If a second input is given, it must be the word `"reply`. In this case, the method will block until it receives a reply from the target of the message. The method outputs `true` if the message has been sent.

```
@> $msg~send 'application/x-vnd.Foo-Inc.MyApp'
```

**Message~timeout**

```
~timeout word (number)
```

Set the timeout of the message when sending it, if the first input is `"send` or output the current timeout if no second input is given. If the first input is `"reply` the timeout will be `ON` to wait for a reply. A value of `-1` sets an infinite timeout.

```
@> $msg~timeout "reply 50000
```

**Message~what**

~what

Output the number that defines what the message is all about.

```
@> print $msg~what
128
```

## 3.2 Image object

SQUIRREL 5.3 introduces a new kind of object that allow to handle images. This object can load and manipulate any kind of images for which there is an installed translator.

The primitive `Image` is used to create an *Image* object. It accept either a filename or a given width and height (in a list) in pixels. When no filename is given, the image will be created empty.

The primitive `trans.mime` outputs a list of all MIME type for which there is a Translator, whiles the primitive `trans.name` outputs a list of the image format names.

An *Image* object had severals methods described bellow:

**Image~height**

~height

Output the height in pixel of the image.

```
@> $img~load 'ariane5.jpg'
@> print $img~height
600
```

**Image~length**

~length

Output the memory used by the Image (in bytes).

```
@> $img~length
1228800
```

**Image~load**

`~load string`

Load an file into the image and discard any image already loaded. The method will select the correct Translator according to the file's MIME type.

```
@> $img~load 'myimage.gif'
```

**Image~mime**

`~mime word (string)`

Set or get the MIME type of the image. If the MIME type is set it will be used when the image is save.

```
@> $img~load 'myimage.jpg'
@> print $img~mime "get
image/jpeg
```

**Image~path**

`~path word (string)`

Set or get the path of the image.

```
@> $img~load 'myimage.jpg'
@> $img~path "set '/this/place/this_name.gif'
@> $img~save
```

**Image~save**

`~save (string word)`

Save the image into a file. If no input is given, the method use the path and MIME type of the image object. Otherwise, the input must be the path of the file to create and the name of the format to store the image in.

```
@> $img~load 'myimage.jpg'
@> $img~save 'myimage.gif' "GIF
```

**Image~valid?**

`~valid?`

Output true if the image is a correct image. When an image object is created from a file, the image is only valid if the loading has been a success.

```
@> $img~load 'myimage.cpp'
@> print $img~valid?
false
```

**Image~width**

~width

Output the width in pixel of the image.

```
@> $img~load 'ariane5.jpg'
@> print $img~width
800
```



## Chapter 4

# Primitives

SQUIRREL is exploiting the system of Add-Ons from BeOS . All the primitives are actually in several Add-Ons. A future release will include SQUIRREL 's API, which will allow third parties to write Add-Ons very easily.

This chapter describes all the primitives defined in each standard Add-on delivered with this release.

### 4.1 Add-ons management

This set of primitives allows you to seek details on the Add-on that is loaded. In the next release it will be possible to load and unload Add-ons using primitives.

#### **addon.list**

```
addon.list
```

Output a list of all the Add-on loaded.

```
@> show addon.list
['AddOn' 'Communication' 'Control' 'Data Handling' 'Inspector'
'List Handling' 'Mathematics' 'String' 'Handling' 'Time' 'Workspace']
```

#### **addon.info**

```
addon.info string
```

Print information about an Add-on.

```
@> print addon.info 'Inspector'
Purpose      : Allow access to information on the interpreter
Author       : [e-]
Version      : 0.5
Last Update  : 08/09/99
```

**addon.func**

`addon.func` *string*

Output a list of all the primitive defined in the Add-on given as input.

```
@> print addon.func 'AddOn'  
addon.func addon.info addon.list
```

## 4.2 Communication

Output and input within your SQUIRREL scripts are made possible by the primitive implemented in this Add-on.

This Add-On features two kinds of Message Boxes (which BeOS calls Alert). One is called *Question* and the other is *Info*. The only difference between those two is their use. A *Question* message box will block all activity and access to the application's window since it's a modal window. However, the *Info* message box will not block the application's window. An *Info* window will also not return the button used by the user. Its purpose is to inform the user rather than ask him a question.

Both of the message boxes display an icon on the left of the message which could be specified when creating the message box by including one of the following words :

"none	no icon
"info	
"idea	
"warning	
"stop	
"bug	

Table 4.1: Type of message box

Both primitives will accept a 32x32 *Image* object instead of the previous words.

### Info

`Info word list word—string (thing)+`

The primitive `Info` creates an *Info* message box. The first input must be one of the words representing a certain type of icon, or an *Image* object. The second input is a list of strings or words, which will be the label of the buttons. The third input is the title of the message box, and it can be a string or a word. The primitive returns immediately.

```
@> Info "idea ["ok] '' 'An easier way to do that is to ....'
```

## load

`load string1 string2 ...`

Load all the files (if they are valid SQUIRREL script file), whose filenames (and paths) were given in inputs

```
@> load '../Scripts/constants.sqi' '../Script/math.sqi'
```

## parse.anything

`parse.anything string`

Parse the *string* in input as anything and output a new object. The string must respect the correct syntax.

```
@> make "str '['this 'is 'a 'list']'
@> make "val parse.anything :str
```

## parse.block

`parse.block string`

Parse the *string* in input as a block and output a new block object.

```
@> make "str 'print \'hello there\''
@> make "val parse.block :str
```

## parse.float

`parse.float string`

Parse the *string* in input as a float and output a new float object.

```
@> make "str '4.2345'
@> make "val parse.float :str
```

## parse.integer

`parse.integer string`

Parse the *string* in input as an integer and output a new integer object.

```
@> make "str '748'
@> make "val parse.integer :str
```

**parse.list**

`parse.list string`

Parse the *string* in input as a list and output a new list object.

```
@> make "str '2 6 8 3 4 6'
@> make "val parse.list :str
```

**parse.number**

`parse.number string`

Parse the *string* in input as a number and output a new number object.

```
@> make "str '748'
@> make "val parse.number :str
```

**parse.string**

`parse.string string`

Parse the *string* in input as a string and output a new string object.

```
@> make "str 'this is a string'
@> make "s parse.string :str
```

**parse.word**

`parse.word string`

Parse the *string* in input as a word and output a new word object.

```
@> make "str 'this'
@> make "w parse.word :str
```

**Precision**

`Precision (number)`

Set or get the floating point number precision (number of digits displayed after the decimal)

```
@> print Precision
3
@> Precision 4
```

**print**

```
print thing1 thing2 thing3 thing4 ...
```

Print all the inputs to the standard output. A floating point number is displayed with a precision which is set in the Preferences. Lists are printed without the [ ], and each input is printed separated by a space. A carriage return is added at the end.

```
@> print 4.56789 'hello' [2 3 4 5]
4.567 hello 2 3 4 5
```

**Question**

```
Question word list (thing)+
```

The primitive *Question* creates a *Question* message box. The first input must be one of the words representing a certain type of icon or an *Image* object. The second input is a list of strings or words, which are the buttons to display in the message box. The third input is the title of the message box. All the other inputs will be concatenated in the message displayed. The primitive will output 1 if the button used by the user is the first in the list, and 2 if it's the second and so forth.

```
@> Question "stop ["Yes "No] 'Erasing all files' 'Are your sure ?'
```

**read.anything**

```
read.anything
```

Read anything from the standard input and output a new object with the value. Correct syntax must be applied.

```
@> type 'Enter something : ', make "thing read.anything
Enter something : [1 2 3 4 5]
@> print is.list :thing
true
```

**read.block**

```
read.block
```

Read a block from the standard input and output a new object with the value. Only one line block is valid.

```
@> type 'Enter your orders: ', make "block read.block
Enter your orders: print "a, make "b 456, add :a :b
@> run :block
4
460
```

**read.float**

```
read.float
```

Read a float from the standard input and output a new object with the value.

```
@> type 'Enter the value of v: ', make "v read.float
Enter the value of v: 4.67
@> print :v
4.67
```

**read.integer**

```
read.integer
```

Read an integer from the standard input and output a new object with the value.

```
@> type 'How old are you ? ', make "old read.integer
How old are you ? 25
@> print :old
25
```

**read.list**

```
read.list
```

Read a list from the standard input and output a new object. The space between inputs will mark each element of the list.

```
@> type 'Enter a list: ', make "mylist read.list
Enter a list: 3 6 2 3.4 'this is a string' "myword
@> show :mylist
[3 6 2 3.4 'this is a string' "myword]
```

**read.number**

```
read.number
```

Read a number (integer or float) from the standard input and output a new object with the value.

```
@> type 'Enter a number: ', make "num read.number
Enter a number: 345
@> print :num
345
```

### **read.string**

`read.string`

Read a string from the standard input and output a new object with the value.

```
@> type 'Enter your name: ', make "name read.string
Enter your name: Joe
@>
```

### **read.word**

`read.word`

Read a word from the standard input and output a new object with the value.

```
@> type 'Enter a function name: ', make "func read.word
Enter a function name: thing
@>
```

### **show**

`show thing1 thing2 thing3 thing4 ...`

Print all the inputs to the standard output. Floating point numbers are displayed with a precision which is set in the Preferences. Lists are printed with the `[ ]`, and each input is printed separated by a space. A carriage return is added at the end.

```
@> show 4.56789 'hello' [2 3 4 5]
4.567 'hello' [2 3 4 5]
```

### **type**

`type thing1 thing2 thing3 thing4 ...`

Print all the inputs to the standard output. Floating point numbers are displayed with a precision which is set in the Preferences. Lists are printed without the `[ ]`, but no space is added between inputs. A carriage return is added at the end.

```
@> type 4.56789 'hello' [2 3 4 5]
4.567 hello 2345
```



## 4.3 Control

Most control structures are implemented deep inside the interpreter, but some of the simplest are in this Add-on.

### call

```
call word thing1 thing2 thing3 ...
```

The command `call` executes commands (primitive or procedure) given as its first input. All the other inputs will be treated as regular inputs to this command.

```
@> call "print 'This is a test:' sum 4 5  
This is a test: 9
```

### break

```
break
```

This order only makes sense within a loop. It will stop the execution of the loop and return without executing the remaining orders in the loop block.

```
for ["i 1 100] {  
    if (myfunc :i (random 45))<34 {break}  
    dosomething_with :i  
}
```

### continue

```
continue
```

This order only makes sense within a loop. It will advance in the execution of the loop and not execute the remaining orders in the loop block.

```
for ["i 1 100] {  
    if :i%10 {continue}  
    print :i  
}
```

Will produce:

```
10
20
30
40
50
60
70
80
90
100
```

### **stop**

```
stop
```

Stop the execution of a function. Execute the next order after calling the function.

```
to foo :a
  if :a=0 {stop}
  print :a
end
```

### **test**

```
test expression
```

Evaluate the expression in input and remember the result. The result will be used by an `iftrue` and `iffalse`.

```
@> test :a < 10
```

### **wait**

```
wait number
```

Suspend the execution for a *number* seconds. The input must be an integer.

```
@> wait 2
```

## 4.4 Data processing

Several primitives are implemented in this Add-on to manipulate data.

### **butfirst**

`butfirst` *thing*

Output all, but the first element of the input, if the input is a list. Otherwise if the input is a string or a word, output all, but the first of its characters.

```
@> print butfirst [3 2 4 5]
2 4 5
@> print butfirst 'hello'
ello
```

### **butlast**

`butlast` *thing*

Output all, but the last element of the input, if the input is a list. Otherwise if the input is a string or a word, output all, but the last of its characters.

```
@> print butlast [3 2 4 5]
3 2 4
@> print butlast 'hello'
hell
```

### **clone**

`clone` *thing*

Output a new object to be the exact copy of the input.

```
@> make "a [1 2 3 4 5 6 7 8 9 0]
@> make "b clone :a
```

### **deepclone**

`deepclone` *thing*

Output a new object to be the exact copy of the input. This order is different from `clone` when used on a list as it clone as well all the elements of the list.

```
@> make "a [1 2 3 4 5 6 7 8 9 0]
@> make "b deepclone :a
```

**first**

*first thing*

If the input is a list, the primitive outputs its first element. Otherwise outputs the first character of the string or word if the input states it.

```
@> print first [3 2 4 5]
3
@> print first 'hello'
h
```

**is.block**

*is.block thing*

Output *true* if the input is a block.

```
@> print is.block 'test'
false
@> print is.block {print 'hello'}
true
```

**is.bool**

*is.bool thing*

Output *true* if the input is a Boolean value.

```
@> print is.bool 4
false
@> print is.bool true
true
```

**is.float**

*is.float thing*

Output *true* if the input is a floating point number.

```
@> print is.float 4.56
true
@> print is.float 23
false
```

**is.integer**

`is.integer thing`

Output *true* if the input is an integer number.

```
@> print is.integer 4.56
false
@> print is.integer 23
true
```

**is.list**

`is.list thing`

Output *true* if the input is a list.

```
@> print is.list [1 2 3 4]
true
@> print is.list 3
false
```

**is.number**

`is.number thing`

Output *true* if the input is a number (float or integer).

```
@> print is.number 4.56
true
@> print is.number 23
true
```

**is.object**

`is.object thing`

Output *true* if the input is an object (Array, File ...).

```
@> print is.object 4.56
false
@> make "a Array 10
@> print is.Object :a
true
```

**is.string**

`is.string` *thing*

Output *true* if the input is a string.

```
@> print is.string 'hell there'
true
@> print is.string "test"
false
```

**is.word**

`is.word` *thing*

Output *true* if the input is a word.

```
@> print is.word [1 2 3 4]
false
@> print is.word "test"
true
```

**item**

`item` *number thing*

Output the *number* at the position given from the second input if the second input is a list. Otherwise if it's a string or a word, output the corresponding character to the position given. The first element is at position 1.

```
@> print item 1 [3 2 4 5]
3
@> print item 5 'hello'
o
```

**last**

`last` *thing*

Output the last element of the input if the input is a list. Otherwise if it's a string or a word, output its last character.

```
@> print last [3 2 4 5]
5
@> print last 'hello'
o
```

**word**

`word word1 word2 word3 ...`

Output the concatenation of the input as a word.

```
@> print word "a "b "c "d "e "f "j  
abcbefbj
```

## 4.5 Data structures

Using a list in SQUIRREL is very simple and powerful. However, it is possible that a list may not meet the requirements for more efficiency or simply that a list can not adapt to a particular task. This Add-on gives SQUIRREL two data structures : Vector and Dictionary. Each of these data structures are an object that has methods.

### Dictionary

Dictionary

Create a new *Dictionary*. An object of this type has most of the functionalities of an array, but the indexes are not limited to integers. An index could be a number, word, or string. The dictionary's index is referred to as the *key*.

The primitive `is.dictionary` will return `true` only if the input is a Dictionary.

Several *methods* are available for a dictionary, for example, to get and set the value of the element or to perform other operations :

#### Dictionary~av

~av

Output the average value of all the numerical values in the dictionary. 0 will be the output if no numerical value is found.

```
@> make "v Dictionary
@> $v~set "jim 25
@> $v~set "tom 34
@> $v~set "eric 20
@> $v~set "jon 31
@> print $v~av
27.5
```

#### Dictionary~empty

~empty

Erase all the elements of the dictionary.

```
@> make "v Dictionary
@> $v~set "jim 25
@> $v~set "tom 34
@> $v~set "eric 20
@> $v~set "jon 31
@> $v~empty
```



**Dictionary~erase**

`~erase key`

Erase the element with the key *key* in the dictionary.

```
@> make "v Dictionary
@> $v~set "jim 25
@> $v~set "tom 34
@> $v~set "eric 20
@> $v~set "jon 31
@> $v~erase "tom
```

**Dictionary~exists**

`~exists key`

Output true if the key *key* exists in the dictionary, false else.

```
@> make "v Dictionary
@> $v~set "jim 25
@> $v~set "tom 34
@> $v~set "eric 20
@> $v~set "jon 31
@> print $v~exists "kim
false
```

**Dictionary~find**

`~find thing`

Output the key of the first occurrence of the value *thing* in the dictionary. If the value is not found, false will be the output.

```
@> make "v Dictionary
@> $v~set "jim 25
@> $v~set "tom 34
@> $v~set "eric 20
@> $v~set "jon 31
@> print $v~find 34
tom
@> print $v~find "Dave
false
```

**Dictionary~find.all**

```
~find.all thing
```

Output a list of all the keys with the value *thing* in the dictionary. If the value is not found the list will be empty.

```
@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 20
@> $v~set "jon 31
@> print $v~find.all 20
[eric jim]
@> print $v~find.all 40
[]
```

**Dictionary~find.if**

```
~find.if word (thing ...)
```

Output the key of the first element in the dictionary, which when given as the first argument to the function, *word* returns the value `true`. The other inputs *thing*, will be given to the function *word*. If no element matches, `false` will be the output.

```
to lessthan :element :bound
  output :element<:bound
end

@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> print $v~find.if "lessthan 30
eric
@> print $v~find.if "lessthan 20
false
```

**Dictionary~find.if.all**

```
~find.if.all word (thing ...)
```

Output a list of the key of the elements in the dictionary, which when given as the first argument to the function, *word* returns the value `true`. The other inputs *thing*, will be given to the function *word*. If no element matches, the list will be empty.

```

to lessthan :element :bound
  output :element<:bound
end

@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> print $v~find.if.all "lessthan 30
[eric jim]
@> print $v~find.if.all "lessthan 20
[]

```

### Dictionary~find.if.last

`~find.if.last word (thing ...)`

Output the key of the last element in the dictionary, which when given as the first argument to the function, *word* returns the value true. The other inputs *thing*, will be given to the function *word*. If no element matches, false will be the output.

```

to lessthan :element :bound
  output :element<:bound
end

@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> print $v~find.if.last "lessthan 30
jim
@> print $v~find .if.last "lessthan 20
false

```

### Dictionary~get

`~get key`

Output the value for the key *key* in the dictionary. An error will be thrown if the key is not found in the dictionary.

```

@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25

```

```
@> $v~set "jon 31
@> print $v~get "tom
34
```

### Dictionary~iterate

```
~iterate word (thing ...)
```

Execute the function *word* for each element of the dictionary. The function called will receive as its input, the element and the other inputs of the method *thing* .... The method will return a Dictionary containing what the function *word* returned for each element.

```
to add :a :b
    output :a+:b
end

@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> make "b $v~iterate "add 1
```

### Dictionary~iterate.i

```
~iterate.i word (thing ...)
```

Execute the function *word* for each element of the dictionary. The function called will receive as its input, the position and value of the element in the dictionary. The other inputs to the method *thing* ... will be added. The method will return a Dictionary containing what the function *word* returned for each element.

```
to foo :key :value :coef
    output :value + :coef
end

@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> make "b $v~iterate.i "foo 3
```

### Dictionary~max

```
~max
```

Output the maximum value of all the numerical values in the dictionary. 0 will be the output if no numerical value is found.

```
@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> print $v~max
34
```

### Dictionary~min

`~min`

Output the minimum value of all the numerical values in the dictionary. 0 will be the output if no numerical value is found.

```
@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
@> print $v~min
20
```

### Dictionary~set

`~set key thing`

Set the value for the key *key* in the dictionary with the value *thing*.

```
@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
@> $v~set "eric 25
@> $v~set "jon 31
```

### Dictionary~size

`~size`

Output the number of elements in the dictionary.

```
@> make "v Dictionary
@> $v~set "jim 20
@> $v~set "tom 34
```

```
@> $v~set "eric 25
@> $v~set "jon 31
@> print $v~size
4
```

## Vector

*Vector (size (initial\_value))*

Create a new *Vector*. If the *size* is specified, the vector will be of this size. If an *initial\_value* is given, the vector element will be set to this value. A vector in Squirrel doesn't have a fixed capacity, even when the size is specified.

*Vector list* If the first input is a list, the content of the list will be used as the content of the array, and the its size will be set to the size of the list.

The index in a vector starts from 1. The primitive `is.vector` will return `true` only if the input is a *Vector*.

Several *methods* are available on an vector:

### Vector~append

*~append thing*

Add the input *thing* to the end of the vector. The size of the vector is increased.

```
@> make "v Vector [1 2 3 4]
@> $v~append 5
@> print :v
1 2 3 4 5
```

### Vector~av

*~av*

Output the average value of all the numerical values in the array. 0 will be the output if no numerical value is found.

```
@> make "v Vector [1 2 3 4]
@> print $v~av
2.5
```

### Vector~erase

*~erase index*

Erase the element at the position *index* in the vector. The size of the vector is reduced by one, and the elements following the one just erased are shifted to the left.

```
@> make "v Vector [6.4 2.4 2.1 8.34]
@> $v~erase 3
@> print :v
6.4 2.4 8.34
```

### Vector~find

*~find thing*

Output the position of the first occurrence of the value *thing* in the vector. If the value is not found, 0 will be the output.

```
@> make "v Vector [6.4 2.4 2.1 8.34]
@> print $v~find 4.56
0
@> print $v~find 2.4
2
```

### Vector~find.all

*~find.all thing*

Output a list of all the positions of the occurrence of the value *thing* in the vector. If the value is not found the list will be empty.

```
@> make "v Vector [6.4 1 2.4 2.1 1 8.34 1]
@> print $v~find.all 1
[2 5 7]
@> print $v~find.all 4
[]
```

### Vector~find.if

*~find.if word (thing ...)*

Output the position of the first element in the vector, which when given as the first argument as the function, *word* returns the value true. The other inputs *thing*, will be given to the function *word*. If no element matches 0 will be the output.

```
to lessthan :element :bound
  output :element<:bound
end
@> make "v Vector [6.4 2.4 2.1 8.34]
@> print $v~find.if "lessthan 3
```

```
2
@> print $v~find.if "lessthan 2"
0
```

### Vector~find.if.all

`~find.if.all word (thing ...)`

Output a list of the positions of the element in the vector, which when given as the first argument as the function, *word* returns the value `true`. The other inputs *thing*, will be given to the function *word*. If no element matches the list will be empty.

```
to lessthan :element :bound
  output :element<:bound
end
@> make "v Vector [6.4 2.4 2.1 8.34]
@> print $v~find.if.all "lessthan 3"
[2 3]
@> print $v~find.if.all "lessthan 2"
[]
```

### Vector~find.if.last

`~find.if.last word (thing ...)`

Output the position of the last element in the vector, which when given as the first argument to the function, *word* returns the value `true`. The other inputs *thing*, will be given to the function *word*. If no element matches 0 will be the output.

```
to lessthan :element :bound
  output :element<:bound
end
@> make "v Vector [6.4 2.4 2.1 8.34]
@> print $v~find.if.last "lessthan 3"
3
@> print $v~find .if.last "lessthan 2"
0
```

### Vector~get

`~get index`

Output the value at the position *index* in the vector. *index* must be an integer between 1 and the size of the vector, otherwise an error will be thrown.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> print $v~get 3
2.1
```



**Vector~iterate**

```
~iterate word (thing ...)
```

Execute the function *word* for each element of the array. The function called will receive as its input, the element and the other inputs of the method *thing ...*. The method will output a Vector containing what the function *word* returned for each element.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> make "b $v~iterate "rsin
@> print :b
0.1165 0.6755 0.8632 0.6755 0.8842
```

**Vector~iterate.i**

```
~iterate.i word (thing ...)
```

Execute the function *word* for each element of the array. The function called will receive as its input, the position and value of the element in the vector. The other inputs to the method *thing ...* will be added. The method will output a Vector containing what the function *word* returned for each element.

```
to foo :position :value :coef
    output :position * :value + :coef
end
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> make "b $v~iterate.i "foo 3
@> print :b
9.4 7.8 9.3 12.6 44.7
```

**Vector~max**

```
~max
```

Output the maximum value of all the numerical values in the array. 0 will be the output if no numerical value is found.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> print $v~max
8.34
```

**Vector~min**

```
~min
```

Output the minimum value of all the numerical values in the array. 0 will be the output if no numerical value is found.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> print $v~min
2.1
```

### Vector~set

*~set index thing*

Set the value at the position *index* in the vector with the value *thing*. *index* must be an integer between 1 and the size of the vector, otherwise an error will be thrown.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> $v~set 3 4.3
@> print :v
6.4 2.4 4.3 2.4 8.34
```

### Vector~size

*~size*

Output the number of elements in the vector.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> print $v~size
5
```

### Vector~sort

*~sort (word)*

Sort the vector in ascending order if "asc is specified (or if no input is given). Sort the vector in descending order if "des is given.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> $v~sort
@> print :v
2.1 2.4 2.4 6.4 8.34
```

### Vector~sort.new

*~sort.new (word)*

Output a new sorted clone of the vector in ascending order if "asc is specified (or if no input is given). Output a new sorted clone of the vector in descending order if "des is given.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> make "w $v~sort.new "des
@> print :w
8.34 6.4 2.4 2.4 2.1
```

**Vector~reverse**

`~reverse (word)`

Reverse the element of the vector.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> $v~reverse
@> print :v
8.34 2.4 2.1 2.4 6.4
```

**Vector~reverse.new**

`~reverse.new (word)`

Output a reversed clone of the vector.

```
@> make "v Vector [6.4 2.4 2.1 2.4 8.34]
@> make "w $v~reverse.new
@> print :w
8.34 2.4 2.1 2.4 6.4
```

## 4.6 Exec

This Add-on allows us to execute external programs and to collect their output.

### **exec.bg**

`exec.bg word | [word things...] string (thing)*`

Execute the program given as the second input. All the inputs to the primitive after the second, will be given as arguments to the program. The first input is the name of a function to execute each time the program outputs a line. The output will be given as the first input of the command. If a list is used instead of a word, it will contain as its first element, the name and inputs of the function to call. The primitive outputs the id of the thread.

```
to process :line
    print :line
end
```

```
@> exec.bg "process '/bin/ls'
```

### **exec.wait**

`exec.wait word string (thing)*`

Execute a program given as the second input. All the inputs to the primitive after the second, will be given as arguments to the program. The first input is a variable name. This variable will contain a list of all the outputs from the program. The primitive returns when the program has finished and output the error code returned by the programmed executed.

```
@> exec.wait "term '/bin/sh' '-c' 'exec echo $TERM'
@> print :term
dumb
```

### **launch**

`launch string`

Launch the preferred application of the file given as input (complete path).

```
@> launch 'mydoc.html'
```

## 4.7 File Input/Output

Using SQUIRREL to read and write to a text file is possible with the use of the following primitives:

### **fclose**

`fclose file`

Close an opened text file.

```
@> fclose :myfile
```

### **feof**

`feof file`

Output true if there are no more lines to read from the text file. The End of File has been reached.

```
@> if not (feof :myfile) {make "line fgets :myfile}
```

### **fgets**

`fgets file`

Read a line from the text file *file* and output it into a string. Throw an error if the end of the file has been reached.

```
@> make "line fgets :myfile
```

### **fopen**

`fopen mode path`

Open a text file *path* in a certain *mode* ("in" "out" "append) and output a file object.

```
@> make "myfile fopen "in 'test.txt'
```

### **fputs**

`fputs file thing1 thing2 thing3 ...`

Write a line in the text file *file* composed of the concatenation of the inputs *thing1 thing2 ...*

```
@> fputs :myfile 12 ' ' 56.56
```

**is.file**

`is.file thing`

Output true if the input *thing* is a file and false otherwise.

```
@> print is.file 4  
false
```

## 4.8 Image processing

This Add-on contains several image processing primitives.

### **img.crop**

`img.crop image width height`

Crop the first input to the size given as second and third input. The primitive output a new image object.

```
@> make "img2 img.crop :img 100 100
```

### **img.scale**

`img.scale image word [number number — list]`

Scale the image given as first input. The second input is either "factor or "size. The next inputs shall be a list of two numbers, or two numbers. When the second input is "factor the two numbers give a factor by which the image shall be scaled (the value shall be between 0 and 1). When the second input is size, the two numbers are width and height of the destination image. The primitive output a new image object.

```
@> make "img2 img.scale :img "factor 0.4 0.4
```

## 4.9 Inspector

Used mostly for debugging, the Add-on *Inspector* provides various information on SQUIRREL . For examples, the heap content, or the current memory usage.

### deth

deth

Print a dump of all the objects in the *eternal heap*

```
@> deth
BOL[1][1][0] - true
BOL[2][2][0] - false
DBL[4][0][0] - 23.6
INT[7][0][0] - 12
STR[11][0][0] - hello there
KWO[16][1][0] - abc
INT[18][1][0] - 2345
LST[20][1][0] - 1 2 3 4 5
INT[21][1][0] - 1
INT[22][1][0] - 2
INT[23][1][0] - 3
INT[24][1][0] - 4
INT[25][1][0] - 5
```

### dgloh

dgloh

Print a dump of all the objects in the *global heap*

```
@> dgloh
BOL[1][1][0] - true
BOL[2][2][0] - false
DBL[4][1][0] - 23.6
INT[7][1][0] - 12
STR[11][1][0] - hello there
```

### dloh

dloh

Print a dump of all the objects in the *local heap* (useful when used within a function)



```
@> dloh
BOL[1][1][0] - true
BOL[2][2][0] - false
DBL[4][1][0] - 23.6
INT[7][1][0] - 12
STR[11][1][0] - hello there
```

### **dump**

`dump thing1 thing2 thing3 ...`

Print internal information about the inputs in the format:

```
type[id][ref][status] - value
```

```
@> make "a 23.56
@> dump :a
DBL[4][1][0] - 23.6
```

### **dtreeh**

`dtreeh`

Print a dump of all the objects in the *tree heap*

```
@> dtreeh
BLK[46][1][0] -
{block}[
{builtin}[]

BUI[47][1][0] - {builtin}[]
```

### **mem.usage?**

`mem.usage?`

Output the current memory usage in bytes.

```
@> print mem.usage?
4284416
```

### **pglov**

`pglov`

Print all the global variables and their values.

```
@> pglov
a = abc
b = 2345
c = 1 2 3 4 5
d = false
```

### **plov**

```
plov
```

Print all the local variables and their values.

```
@> plov
i = 23
j = hello there
k = false
```

### **sgloh**

```
sgloh
```

Print information about the *global heap* e.g. number of objects present and total current number of objects created.

```
@> sgloh
Size = 0 Last ID = 0
```

### **sloh**

```
sloh
```

Print information about the *local heap* e.g. number of objects present and total current number of objects created.

```
@> sloh
Size = 0 Last ID = 0
```

## 4.10 List processing

This Add-on contains several primitives which operate on Lists.

### **fput**

`fput thing list`

Output a list formed by the concatenation of the first input and the second input.

```
@> show fput 34.56 ['foo' 345]
[34.56 'foo' 345]
```

### **gseq**

`gseq from to (count)`

Output a list of the numbers from *from* to *to*. If *count* is specified, numbers will be equally spaced and the size of the list will be *count*.

```
@> print gseq 5 10
5 6 7 8 9 10
@> print gseq 1 10 20
1 1.47 1.95 2.42 2.89 3.37 3.84 4.32 4.79 5.26
5.74 6.21 6.68 7.16 7.63 8.11 8.58 9.05 9.53 10
```

### **lappend**

`lappend list thing1 thing2 thing3 ...`

Append the inputs to the list specified as the first input. This primitive mutates the list.

```
@> make "a [1 2 3 4]
@> lappend :a 5 6 7 8 [9]
@> show :a
[1 2 3 4 5 6 7 8 [9]]
```

### **lempty**

`lempty list`

Output *true* if the list is empty, and *false* otherwise.

```
@> print lempy [1 2 3 4]
false
```

**lfind**

`lfind list thing`

Output the position in the list given as first input, of the first occurrence of the second input. Output 0 if not found.

```
@> make "l [23 56 "hello 'this is it !']
@> print lfind :l "hello
3
```

**lindex**

`lindex list index (thing)`

Output the value at the position `index` in the list, or change the value at this position if a third input is specified. This primitive mutates the list.

```
@> print lindex [1 2 3 4 5] 3
3
@> make "a [1 2 3 4 5]
@> make "b :a
@> lindex :a 5 99
@> show :b
[1 2 3 4 99]
```

**list**

`list thing1 thing2 thing3 ...`

Output a list whose members are in the inputs.

```
@> make "mylist list 2 5.6 'hello' "foo
@> show :mylist
[2 5.6 'hello' "foo]
```

**ljoin**

`ljoin list (thing)`

Output the concatenation of all the elements of the list `list` separated by `thing` if specified in a string.

```
@> print ljoin [1 2 3 4 5] ';'
1;2;3;4;5
```

**llength**

`llength list`

Output the length of the list.

```
@> print llength [2 5 1 76 8]
5
```

**lput**

`lput thing list`

Output a list formed by the concatenation of the second input and the first input.

```
@> show lput 'end' ['foo' 345]
['foo' 345 'end']
```

**lremove**

`lremove list thing`

Remove the first occurrence of the second input from the first input.

```
@> lremove :list 4
```

**lscan**

`lscan list "var1" "var2" "var3" ...`

Store the element of the list starting from the first one into each given variable name, and output the rest of the list. If there are less elements than variables in the list, the variable values will be set to an empty list.

```
@> make "rest" lscan [1 2 3 4 5 6] "a" "b" "c"
@> show :rest
[4 5 6]
@> show :a :b :c
1 2 3
```

**lsub**

`lsub list from (to)`

Output a sublist of the list given as first input, from the index *from* to the index *to* if given, else to the end of the list.

```
@> show lsub ["a "b "c "d "e "f] 3 5
["c "d "e]
```

### **reverse**

`reverse` *list*

Output a list in the reverse order from that given as input.

```
@> show reverse [1 2 3 4 5]
[5 4 3 2 1]
```

### **sentence**

`sentence` *thing1 thing2 thing3 ...*

Output a list whose members are in the inputs. Lists in the inputs are flattened.

```
@> make "mylist sentence 2 5.6 [1 2 3 4]
@> show :mylist
[2 5.6 1 2 3 4]
```

## 4.11 Mail

This Add-On give to SQUIRREL script to build and send emails through a new *Mail* object.

A Mail object is not automatically destroyed when sended, it's to the developer to do it, if we want.

### 4.11.1 Primitives

The following primitives are available:

#### **is.mail**

*is.mail thing*

Output true if the input is a Mail object

```
@> make "msg Mail
@> print is.mail 5
false
@> print is.mail :msg
true
```

#### **Mail**

*Mail*

Output a new mail object.

```
@> make "msg Mail
@>
```

#### **Send**

*Send (mail (word))*

Send an email if the email is given as first input. If no input given, all the pending emails in the *Out folder* will be send. If a second input is given, it must be the word "now or "later. "now will send the message now.

```
@> Send :msg "later
```

### 4.11.2 Mail Object

To fill a mail, we use the severals methods available :

**Mail~attachment***~attachment word (string)*

The method acts on the files attached to the email. If the first input is "set the second input is added to the list of attachments. If the first input is "chg, the second input will replace the current list. If the first input is "del (no second input needed), the list will be deleted. If it's "get, the method will output the current attached files.

```
@> $msg~attachment "set 'myfile.zip'
```

**Mail~bcc***~bcc word (string)*

The method acts on the blind carbon-copy recipients of the mail. If the first input is "set the second input is added to the list of recipient. If the first input is "chg, the second input will replace the current list of recipients. If the first input is "del (no second input needed), all the recipients will be removed from the list. If it's "get, the method will output the current list of recipients.

```
@> $msg~bcc "set 'joe@cool.net'
```

**Mail~cc***~cc word (string)*

The method acts on the carbon-copy recipients of the mail. If the first input is "set the second input is added to the list of recipient. If the first input is "chg, the second input will replace the current list of recipients. If the first input is "del (no second input needed), all the recipients will be removed from the list. If it's "get, the method will output the current list of recipients.

```
@> $msg~cc "set 'bill@cool.net'
```

**Mail~content***~content word (string)*

The method acts on the content of the mail. If the first input is "set the second input is added a new line in the content. If the first input is "chg, the second input will replace the current content. If the first input is "del (no second input needed), the content will be deleted. If it's "get, the method will output the current content.

```
@> $msg~content "set 'Hello Bob,\n'
@> $msg~content "set 'Can you tell me where you are ?\n'
@> $msg~content "set 'Thanks\n'
```



**Mail~from**

*~from word (string)*

Set or get the sender email of the message. By default, the sender email is set at sending time from the BeOS settings.

```
@> $msg~from "set 'jlv@bemail.org'
```

**Mail~to**

*~to word (string)*

The method acts on the recipient of the mail. If the first input is "set the second input is added to the list of recipient. If the first input is "chg, the second input will replace the current list of recipients. If the first input is "del (no second input needed), all the recipients will be removed from the list. If it's "get, the method will output the current list of recipients.

```
@> $msg~to "set 'bob@cool.net'
```

**Mail~reply**

*~reply word (string)*

Set or get the reply field of the message. By default, the reply is set at sending time from the BeOS settings.

```
@> $msg~reply "set 'me@me.org'
```

**Mail~subject**

*~subject word (string)*

Set or get the subject of the email.

```
@> $msg~subject "set 'Business Plan Rev0.8'
@> print $msg~subject "get
Business Plan Rev0.8
```

## 4.12 Mathematics

To perform mathematical operations such as `max`, and `rsin`, you may use this Add-on. It provides random number generations and contains a large amount of Mathematical primitives.

### **abs**

`abs` *number*

Output the absolute value of the input

```
@> print abs -4.78
4.78
```

### **av**

`av` *thing1 thing2 thing3 thing4 ...*

Output the average value of the input. The input may be a *number* or a *list*. Any input of other types will be ignored.

```
@> print av 3 6 2 9.6 [3 2 4 0.5]
3.762
```

### **ceil**

`ceil` *number*

Output the smallest integer that is not less than the input.

```
@> print ceil 4.576
5
```

### **deg2rad**

`deg2rad` *number*

Output the equivalent angle in radians for the degree given (converting degrees to radians)

```
@> print deg2rad 120
2.094
```

**difference**

`difference number1 number2`

Output the difference between *number1* and *number2*.

```
@> print difference 4 5  
1
```

**dacos**

`dacos number`

Output the arccosine of the input given in degrees

```
@> print dacos 45  
0.9033
```

**dasin**

`dasin number`

Output the arcsine of the input given in degrees

```
@> print dasin 180  
nan
```

**datan**

`datan number`

Output the arctangent of the input given in degrees

```
@> print datan 45  
0.6658
```

**dcos**

`dcos number`

Output the cosine of the input given in degrees

```
@> print dcos 180  
-1
```

**dcosh**

*rcosh number*

Output the hyperbolic cosine of the input given in degrees

```
@> print dcosh 180  
11.59
```

**dsin**

*dsin number*

Output the sine of the input given in degrees

```
@> print dsin 60  
0.866
```

**dsinh**

*dsinh number*

Output the hyperbolic sine of the input given in degrees

```
@> print dsinh 180  
11.55
```

**dtan**

*dtan number*

Output the tangent of the input given in degrees

```
@> print dtan 140  
-0.8391
```

**dtanh**

*dtanh number*

Output the hyperbolic-tangent of the input given in degrees

```
@> print dtanh 78  
0.8767
```

**erf**

*erf number*

Output the error function of the input.

```
@> print erf 4.5  
1
```

**erfc**

*erfc number*

Output the complementary error function of the input.

```
@> print erfc 4.5  
1.96616e-10
```

**exp**

*exp number*

Output the exponential value of the input.

```
@> print exp 4.7  
109.95
```

**floor**

*floor number*

Output the largest integer that is not greater than the input.

```
@> print floor 4.576  
4
```

**gamma**

*gamma number*

Output the log gamma function of the input.

```
@> print gamma 4.5  
2.45374
```

**hypot**

`hypot number1 number2`

Output the Euclidean distance function of the input

```
@> print hypot 2.5 6.8
7.245
```

**incr**

`incr word (number)`

Increment the value stored in a variable by 1 or by the number specified as the second input. The first input must be a known variable name.

```
@> make "a 1
@> incr "a -1
@> print :a
0
```

**int**

`int number`

Output a cast of the input into an integer.

```
@> print int 4.576
4
```

**log10**

`log10 number`

Output the logarithm (base 10) of the input.

```
@> print log10 5000
3.699
```

**ln**

`ln number`

Output the natural logarithm of the input.

```
@> print ln 5000
8.517
```

**max**

`max thing1 thing2 thing3 thing4 ...`

Output the maximum value of the inputs. The inputs may be *numbers* or *lists*. Any input of other types will be ignored.

```
@> print max 3 6 2 9.6 [3 2 4 0.5]
9.6
```

**min**

`min thing1 thing2 thing3 thing4 ...`

Output the minimum value of the inputs. The inputs may be *numbers* or *lists*. Any input of other types will be ignored.

```
@> print min 3 6 2 9.6 [3 2 4 0.5]
0.5
```

**minus**

`minus number`

Output the negative value of the input.

```
@> print minus -5
5
@> print minus 45.78
-45.78
```

**product**

`product thing1 thing2 thing3 thing4 ...`

Output the product of all the inputs. The inputs may be *numbers* or *lists*. Any input of other types will be ignored.

```
@> print product 3 6 2 9.6 [3 2 4 0.5]
4147.2
```

**quotient**

`quotient number1 number2`

Output the quotient of *number1* over *number2*.

```
@> print quotient 5 7
0.714
```

**modulo**

`modulo number1 number2`

Output the remainder from performing the integer quotient of *number1* over *number2*. Both numbers must be integers.

```
@> print modulo 10 2
0
```

**power**

`power number1 number2`

Output the power of *number1* by *number2*

```
@> print power 3.4 5
454.354
```

**racos**

`racos number`

Output the arccosine of the input given in radians

```
@> print racos 0.4
1.159
```

**rad2deg**

`rad2deg number`

Output the corresponding angle in degrees, given the angle in radians (converting from radians into degrees)

```
@> print rad2deg 2.094
120
```

**random**

`random (min) max`

Output a random number between the given min and max. If only one input is given, it will be interpreted as the max and the output will be a number between 0 and max.

```
@> print random 40
15
```



**rcos**

*rcos number*

Output the cosine of the input given in radians

```
@> print rcos 60  
-0.952
```

**rcosh**

*rcosh number*

Output the hyperbolic cosine of the input given in radians

```
@> print rcosh 0.56  
1.160
```

**rasin**

*rasin number*

Output the arcsine of the input given in radians

```
@> print rasin 0.4  
0.411
```

**rsin**

*rsin number*

Output the sine of the input given in radians

```
@> print rsin 60  
-0.304
```

**rsinh**

*rsinh number*

Output the hyperbolic sine of the input given in radians

```
@> print rsinh 0.56  
0.589
```

**ratan**

*ratan number*

Output the arctangent of the input given in radians

```
@> print ratan 0.6  
0.540
```

**rtan**

*rtan number*

Output the tangent of the input given in radians

```
@> print rtan 60  
0.32
```

**rtanh**

*rtanh number*

Output the hyperbolic tangent of the input given in radians

```
@> print rtanh 0.7  
0.604
```

**sqrt**

*sqrt number*

Output the square root of the input.

```
@> print sqrt 25  
5
```

**sum**

*sum thing1 thing2 thing3 thing4 ...*

Output the sum of the all the inputs. The inputs may be *numbers* or *lists*. Any input of other types will be ignored.

```
@> print sum 3 6 2 9.6 [3 2 4 0.5]  
30.01
```

## 4.13 String processing

This Add-on offers some string processing primitives for strings as well as words.

### **safter**

`safter string1 string2`

Output the part of the first input which is after the last occurrence of the second input in the first.

```
@> print safter 'image/jpeg' 'image/'  
jpeg
```

### **sbefore**

`sbefore string1 string2`

Output the part of the first input which is before the first occurrence of the second input in the first.

```
@> print sbefore 'image/jpeg' '/'  
image
```

### **scmp**

`scmp string1 string2`

Compare two strings (return 0 if they are equal, a positive value if  $s1 > s2$ , and a negative value otherwise).

```
@> print scmp 'this is my string' 'and another'  
1
```

### **serase**

`serase string number (number)`

Output the string given as first input where a part of it has been erased from the position given as second input to the end of the string if no third input is given, or for the given number of characters.

```
@> print serase 'the weather is great' 12  
the weather
```

**sfind**

`sfind string string`

Output the position of the first occurrence of the second input in the first input. `false` is outputted if the string has not been found.

```
@> print sfind 'hello there !!' 'there'
7
```

**sfind.all**

`sfind string string`

Output a list of the position of all the occurrence of the second input in the first input. The list will be empty if no occurrence has been found.

```
@> print sfind.all 'I disagree. I think it\'s good' 'I'
[1 13]
```

**sfind.last**

`sfind.last string string`

Output the position of the last occurrence of the second input in the first input. `false` is outputted if the string has not been found.

```
@> print sfind.last 'hello there !!' '!'
14
```

**sfirst.not.of**

`sfirst.not.of string string`

Output the position of the first character from the first string that is not present in the second string. `false` is outputted if they are all found.

```
@> print sfirst.not.of 'hello there !!' '!?,..'
1
```

**sfirst.of**

`sfirst.of string string`

Output the position of the first occurrence of any characters of the second input in the first input. `false` is outputted if none found.

```
@> print sfirst.of 'hello there !!' '!?,..'
13
```

**sinsert**

`sinsert string number (thing)+`

Output the string given as first input where the third (and the others) inputs has been inserted at the position given as second input.

```
@> print sinsert 'the weather is great' 16 'not '  
the weather is not great
```

**slast.not.of**

`slast.not.of string string`

Output the position of the last occurrence of any characters not of the second input in the first input. false is outputted if none found.

```
@> print slast.not.of 'hello there !!' '!?,. '  
12
```

**slast.of**

`slast.of string string`

Output the position of the last occurrence of any characters of the second input in the first input. false is outputted if none found.

```
@> print slast.of 'hello there !!' '!?,. '  
14
```

**sleft**

`sleft string number`

Output the first n characters of the string.

```
@> print sleft 'The weather is great' 5  
The w
```

**slength**

`slength string`

Output the size of the string given as input

```
@> print slength 'this is a test'  
14
```

**smatch**

*smatch string string*

Output true if the pattern given as second input match the first input. false else. Use a \* in the pattern string to match any kind of characters.

```
@> print smatch 'joe@foo.zorg' '*zorg'
true
```

**split**

*split string separator*

Split the string *string* into list elements using the string *separator*. Each element will be *parsed*.

```
@> show split '34;45;23;56;78' ';'
[34 45 23 56 78]
```

**split.as.string**

*split.as.string string separator*

Split the string *string* into list elements using the string *separator*. Each element will be a string.

```
@> show split 'foo;34;45;23;56;78' ';'
['foo' 34 45 23 56 78]
```

**sreplace**

*sreplace string number thing*

Output the string given as first input where the third input has replaced the part of the string at the position given by the second input.

```
@> print sreplace 'The weather is fine' 16 'bad '
The weather is bad
```

**sright**

*sright string number*

Output the last n characters of the string.

```
@> print sright 'The weather is great' 5
great
```

**stolower**

`stolower string`

Output a string which is a lower case version of the input.

```
@> print stolower 'This IS A STring'
this is a string
```

**stoupper**

`stoupper string`

Output a string which is an upper case version of the input.

```
@> print stoupper 'this is a test'
THIS IS A TEST
```

**strim**

`strim string character`

Output the first input trimmed (both left and right) off the second input.

```
@> show strim '    hello there !    ' ' '
'hello there !'
```

**strim.l**

`strim.l string character`

Output the first input trimmed left off the second input.

```
@> show strim.l '    hello there !    ' ' '
'hello there !    '
```

**strim.r**

`strim.r string character`

Output the first input trimmed right off the second input.

```
@> show strim.r '    hello there !    ' ' '
'    hello there !'
```

**string**

`string thing1 thing2 thing3`

Output a string which is the concatenation of the inputs

```
@> print string 'this is a test' 56 3.6 "hello  
this is a test563.6hello
```

**substr**

`substr string number (number)`

Output a part of the string given as first input. The second input is the index of the first character of the string to extract. The second input is the number of characters to extract. If this input is not given, the primitive will extract all the characters up to the end of the string.

```
@> print substr 'this is a test' 6 2  
is
```



## 4.14 Storage

This Add-on contains various primitives which offer access to the BeOS file-system. The BeOS file-system features file attribute handling, mime type and directory browsing.

### **attr.del**

`attr.del file_path attribute_name`

Delete the attribute specified by the string *attribute\_name* in the file *file\_path*.

```
@> attr.del '/boot/home/myfile.txt' 'myattribute'
```

### **attr.exists**

`attr.exists file_path attribute_name`

Output true if the attribute *attribute\_name* exists in the file *file\_path*, and output false otherwise.

```
@> print attr.exists '/boot/home/myfile.txt' 'myattribute'
true
```

### **attr.get**

`attr.get file_path attribute_name`

Output the value of an attribute specified by the string *attribute\_name* in the file *file\_path*.

```
@> print attr.get '/boot/home/myfile.txt' 'myattribute'
12.45
```

### **attr.list**

`attr.list file_path`

Output a list of all the attributes in the file *file\_path*.

```
@> show attr.list '/boot/home/myfile.txt'
["wrap "alignment "styles "myattribute "BEOS:TYPE]
```

### **attr.set**

`attr.set file_path attribute_name thing`

Set the value of the attribute *attribute\_name* in the file *file\_path* to *thing*. Only strings and numbers are accepted as valid values to be stored in an attribute.

```
@> attr.set '/boot/home/myfile.txt' 'myattribute' 56.89
```

**dir.current**

`dir.current (string)`

If the primitive has an input, the current directory will be changed to this input. Otherwise the primitive will output the path of the current directory.

```
@> print dir.current
/boot/home/e-/New Squirrel/Tests
```

**dir.exists**

`dir.exists string`

Test if the directory given as the first input exists. The primitive outputs `true` if the directory exists, and `false` otherwise.

```
@> print dir.exists /boot/home/foo
false
```

**dir.list**

`dir.list string — ((string) word string)`

Output a list of all the entries in a directory. If the first input is not a string, the current directory is used, else the first input is the path of the directory to list the contents. If more than one input is given, the primitive will only output the directory entries which meet specific criteria. The first criteria is a word that specifies the type of entry to select: "file" "directory" "link" or "any". The next input must be a string that allows us to select entries by name.

```
@> print dir.list "any" "t*.sqi"
thread.sqi test_hello.sqi test.sqi
```

**dir.contains**

`dir.contains (string) word string`

Output `true` if the current directory (or the directory specified as the first input) contains an entry that matches the type. The name of the entry is given as the last input.

```
@> print dir.contains '/boot/apps' "directory" 'Squirrel'
true
```

**entry.delete**

`entry.delete string`

Output true if the entry given as input has been deleted or false if not.

```
@> print entry.delete 'myfile.txt'
true
```

**entry.exists**

`entry.exists string`

Output true if the entry given as input exists. false else.

```
@> print entry.exists '/boot/apps'
true
@> print entry.exists '/biit/apps'
false
```

**entry.icon**

`entry.get file_name word word (image)`

Set or get the icon for a given file as an *Image* object. The first input is the file path, the second input is the word "get or "set. The third input indicates whether we want the "small icon or the "large one. If we set the icon, an *Image* as fourth input is required.

```
@> make "icn entry.icon 'myfile.txt' "get "small
```

**entry.isdir**

`entry.isdir string`

Output true if the input is a directory, and output false otherwise.

```
@> print entry.isdir '/boot/apps'
true
@> print entry.isdir 'foo.txt'
false
```

**entry.isfile**

`entry.isfile` *string*

Output true if the input is a standard file, and output false otherwise.

```
@> print entry.isfile '/boot/apps'
false
@> print entry.isfile 'foo.txt'
true
```

**entry.islink**

`entry.islink` *string*

Output true if the input is a symbolic link to a file, and output false otherwise.

```
@> print entry.islink '/boot/apps'
false
@> print entry.islink 'foo2.txt'
true
```

**entry.match**

`entry.match` *string string*

Output true if the second input matches the regular expression given as the first input.

```
@> print entry.match 'foo*45*h.dat' 'hello45rgh.dat'
false
```

**entry.move**

`entry.move` *string string*

Move the entry as first input to the path given as second input. The primitive output false if it fails.

```
@> entry.move 'myfile.txt' '../../'
```

**entry.rename**

`entry.rename` *string string*

Rename the entry as first input to the name given as second input. The primitive output false if it fails.

```
@> entry.rename 'myfile.txt' 'data.txt'
```

**entry.reveal**

`entry.reveal` *string*

Output the path of the original file if the input is a *link*.

```
@> print entry.reveal 'foo2.txt'  
foo.txt
```

## entry.stats

`entry.stats entry.stats string word word (thing)`

Set or Get *statistical* information of the entry given as first input. The second input is the word "get or "set that indicates if we want to set or to get an information. The third input indicates the information to access :

Name	information
"access	Time at which the entry was last accessed (open)
"creation	Time at which the entry was created
"group	Group ID of the entry owner
"owner	User ID of the entry owner
"rights	Access rights of the entry
"size	Size in bits of the entry
"update	Time at which the entry was last updated

Table 4.2: Entry's stats

Size of an entry can NOT be set. Reading the entry rights return a list that contains strings. Each strings describes the rights for the user, group and other : [ 'urx' 'grwx' 'owx' ]. The first character of each string indicates if it is the rights for the user, group, other or for all. The rest of the string describes the right read write execute. To add or remove right, a + or a - can be added to the right string as the second character: 'g-r' will make the entry unreadable for the group. When setting the rights, the fourth input can be a list of strings or a string.

```
@> print ctime entry.stats :_file "get "access
Sun Nov 19 10:30:20 2000
@> entry.stats 'myfile.txt' "set "rights ['o-rwx' 'g-wx']
```

## FilePanel

`FilePanel mode dir flavor selection filter mimes names`

Provide an "Open File" or "Save File" window and provides the user to select one or more file(s). Output the user's selection.

The primitives inputs are :

1. **mode** is a word "open or "save that indicate if we want to open a file or to save it.
2. **dir** is a variable name when we want to know the last directory visited by the user. Else, it's a string that give the directory from where the user must start.
3. **flavor** is a list (must contain at least one element) that indicates what flavor of node the user can select from : "file "directory "link (It is possible to combine them , eg. ["file "link]).
4. **selection** describes if the user can selects more than a file or only one. It must be the word : "single or "multiple. When the user select more than one file, the output of the primitive is a list.

5. **filter** is the word "allow or "disallow. It indicates if the selection given as inputs 6 and 7 must be shown or not shown.
6. **mimes** is the list of mime types allowed or diallowed, eg. [ 'image/\*' 'video/\*' ].
7. **names** is the list of filenames allowed or diallowed, eg. [ '\*.zip' '\*.tgz' ].

If the user cancels the panel, the primitive output the boolean value false.

```
@> make "dir '/boot/home'
@> make "myfile FilePanel "open "dir ["file] "single "allow ['image/*'] []
@> print :dir
/boot/home/Images
```

### **mime.delete**

`mime.delete mime_type`

Delete a mime-type from the BeOS mime database.

```
@> mime.delete 'text/jlv-text'
```

### **mime.desc**

`mime.desc word mime_type word (string)`

Set or get the description of a mime type. The first input is the word "short or "long that indicate if we are working on the long or short description. The second input is the mime-type. The third input is the word "get or set that indicates if we want to set or get the decription. If we want to set it, we need to give as 4th input a string that will be the description.

```
@> mime.desc "short 'text/jlv-text' "set 'JLV\'s special format'
```

### **mime.exists**

`mime.exists mime_type`

Output true if the mime-type given as input exists in the BeOS mime database.

```
@> print mime.exists 'text/plain'
true
```

### **mime.get**

`mime.get file_path`

Output the MIME type of the file *file\_path* as a string.

```
@> print mime.get /boot/home/myfile.txt'
text/plain
```

### **mime.icon**

`mime.get mime_type word word (image)`

Set or get the icon for the MIME type as an *Image* object. The first input is the MIME type, the second input is the word "get" or "set". The third input indicates whether we want the "small" icon or the "large" one. If we set the icon, an *Image* as fourth input is required.

```
@> make "icn mime.icon 'image/jpeg' "get "large
```

### **mime.install**

`mime.install mime_type`

Install a mime type in the BeOS Mime database.

```
@> mime.install 'text/jlv-text'
```

### **mime.set**

`mime.set file_path mime_type`

Set the MIME type of the file *file\_path* to the string *mime\_type*. The MIME type must be valid, otherwise an error will be issued.

```
@> mime.set '/boot/home/myfile.txt' 'text/mydata'
```



## 4.15 Threading

SQUIRREL supports *multi-threading* with this Add-on. A thread could be either a function or a block and will run concurrently and asynchronously to any other SQUIRREL thread. A thread has an ID which is unique. Using this ID as its reference, it's possible to kill, suspend, or wait for the end of thread. Each thread has a priority. Higher priority threads will run more often and will terminate faster than a lower priority thread. The following table contains all the priorities supported by SQUIRREL :

Name
"low
"normal
"display
"urgent_display
"realtime_display
"urgent
"realtime

Table 4.3: Thread priority (from low to high)

When the thread is a function, the function can output a value. In case another thread is waiting, the output of the function will be returned to the `Thread.waitFor` primitive.

It may be necessary in a threading script to use a locker (or several lockers) to insure that a critical section of the script will be protected from concurrent threads. For example, the modification of the value of a global variable shared between several threads.

### Locker

Locker

Output a new locker object. This object has a two member functions that locks it `~lock` or unlocks it `~unlock`.

```
make "mylock Locker
make "sum 0

to thread_1
    mylock~lock
    incr "sum 10
    mylock~unlock
end

make "th1 Thread "normal "thread_1
make "th2 Thread "normal "thread_1
Thread.hop :th1 :th2
```

## Priority

Priority (*word*)

Output or set the priority of the calling thread. If an input is given, it should be a valid word.

```
@> Priority "low"
```

## snooze

snooze *integer*

Pause the calling thread for a given number of microseconds.

```
@> snooze 10000
```

## Thread

Thread *word block* | (*word (thing)\**)

Create a new thread. The first input is the priority of the thread. The second input could either be a block or a word. When the name of a function is given, the restart inputs of the primitives will be fed to the function when it is executed. The primitive outputs the *thread id* of the thread. The thread does not start at the end of this primitive.

```
to thread_1 :iter
  make.local "s 0
  for ["i 1 :iter] {
    make "s :i
  }
  output :s
end

@> make "th1 Thread "low "thread_1 20
```

## ThreadID

ThreadID

Output the *thread id* of the calling thread.

```
@> print ThreadID
367
```

### Thread.hop

Thread.hop (*number*)+

Start or resume one or more thread(s). The input must be the *thread ids* of the threads to run.

```
@> Thread.hop :th1
```

### Thread.hoping

Thread.hoping

Output a list of the *thread ids* of all the running threads.

```
@> show Thread.hoping  
[ 453 ]
```

### Thread.kill

Thread.kill (*number*)+

Kill one or more thread(s). The input must be the *thread ids* of the threads to kill.

```
@> Thread.kill :th1
```

### Thread.priority

Thread.priority *number* (*word*)

Output or set the priority of a thread given as the first input. If a second input is given, it should be a valid word.

```
@> print Thread.priority :th1  
normal
```

### Thread.suspend

Thread.suspend (*number*)+

Suspend one or more thread(s). The input must be the *thread ids* of the threads to suspend.

```
@> Thread.suspend :th1
```

## Thread.waitFor

Thread.waitFor *number* | (*word* (*number*)*+*)

Wait for the end of one or more thread(s). When the primitive should wait for several threads, the first input must be the word: "all or "first. When "all is used, the primitive will wait for the end of all the threads before producing a list of the thread's output values. When "first is used, the primitive will return when the first thread ends and then outputs the thread's output values.

```
@> print Thread.waitFor :th1
```

## Wait

Wait *word*

The calling thread waits for the variable given as input to be updated by another thread. The primitive output true if the thread as wait, false else.

```
@> Wait "mtvar
```

## 4.16 Time

This Add-on contains primitives to access time.

### **clock**

`clock`

Output an approximation for the current processor time used by the program.

```
@> print clock
1
```

### **ctime**

`ctime long`

Output the time in a string format from a time value given as the second input.

```
@> print ctime time
Thu Aug 19 14:30:06 1999
```

### **c2sec**

`c2sec long`

Output the time in seconds from a clock value.

```
@> make "t0 clock
@> do_something_rather_long
@> print 'Time elapsed: ' c2sec :t0-(clock)
0.34
```

### **time**

`time`

Output the time elapsed in seconds since Epoch (00:00:00 UTC, January 1, 1970).

```
@> print time
935108764
```

**timing**

```
timing block | (word things ...)
```

Output the elapsed time in microseconds of the execution of the block or function.

```
@> print timing "foo 45 "hello  
4234
```

## 4.17 Workspace

This Add-on contains most of the workspace management primitives.

### **bind**

`bind word word block | word things ....`

Bind the variable given as the first input to the event given as the second input. The first and second input must be one of the valid words : "get "set "erase. The third input could either be a block or a word defining a function (in this case, we could also add inputs to this function). Each time an event occurs on the variable, the block or the function will be executed.

```
@> bind "myvar" "get" {
      print 'Variable readed'
}
```

### **env.exists**

`env.exists word`

Check if an environment variable exists. Output true or false .

```
@> print env.exists SHELL
true
```

### **env.get**

`env.get word`

Output the value of an environment variable. If the variable doesn't exist, an exception will be raised.

```
@> print env.get "SHELL"
/bin/sh
```

### **env.list**

`env.list`

Output a list of all the environment variables.

```
@> print env.list
PWD BETOOLS BE_C_COMPILER HOSTNAME BE_DEFAULT_CPLUS_FLAGS
BEINCLUDES BELIBRARIES SAFEMODE TTY BE_LINKER ADDON_PATH
USER MACHTYPE BUILDHOME OLDPWD LIBRARY_PATH BE_HOST_CPU
BE_DEFAULT_C_FLAGS SHLVL GROUP SHELL BE_CPLUS_COMPILER
HOSTTYPE OSTYPE HOME TERM PATH _
```

**env.set**

`env.set word thing`

Set the value of an environment variable. The first input is the variable name, the second input is the value. **The variable is not exported.**

```
@> env.set "FOO 34
```

**erase**

`erase word1 word2 word3 ...`

Erase all the inputs from the workspace if they are variable or function names.

```
@> erase "myvar "myfunc
```

**gc**

`gc`

Performs a "garbage collection" and then outputs the number of objects destroyed.

```
@> print gc
2
```

**help**

`help word`

Prints an online help on the *word* given if it's a primitive name.

```
@> help "is.var
AddOn   :      Workspace
Purpose :      Return true if the argument is the name of a variable
Usage   :      is.var "name
```

**is.pred**

`is.pred word`

Output *true* if *word* is the name of a function or primitive

```
@> print is.pred "list
true
@> print is.pred "myfunc
true
```



**is.proc**

`is.proc word`

Output *true* if *word* is the name of a function

```
@> print is.proc "list  
false
```

**is.prim**

`is.prim word`

Output *true* if *word* is the name of a primitive

```
@> print is.prim "list  
true
```

**name**

`name thing word`

Makes the value *thing* a synonym for the variable *word*.

```
@> name 3.1416 "pi  
@> print :pi  
3.141
```

**thing**

`thing word`

Output the value of the variable *word*.

```
@> make "a 'hello'  
@> make "b thing "a
```

**unbind**

`unbind word word`

Unbind the variable given as the first input for the event given as the second input. The first and second input must be one of the valid words: "get "set "erase.

```
@> unbind "myvar "get
```

# Chapter 5

## Release notes

### 5.1 Release 5.3

#### 5.1.1 Changes

- Primitives `Question` and `Info` can accept an *Image* as first input. Both primitives request now an extra input which is the title of the message box.
- Strings can be spread over several line. Tabulation within a string is no longer taken into account.
- Use `\\` followed by a carriage return to spread an instruction over two lines.
- Renamed primitives `mime.exist` `dir.exist` `attr.exist` in `mime.exists` `dir.exists` `attr.exists`.
- Renamed method `exist` of the *Dictionary* object in `exists`.

#### 5.1.2 Additions

- Added *Image* object.
- New Add-On *Image Processing* with two primitives : `img.crop` and `img.scale`.
- Added primitive `mime.icon`, `entry.icon` and `entry.exists` in the *Storage* Add-On.
- Added primitive `lremove` in the *List Processing* Add-On.

#### 5.1.3 Bugs fixed

- Severals issues in the way SQUIRREL was terminating have been solved.
- Fixed a bug in the `FilePanel` primitive (file selected but see as panel canceled).
- Fixed possible crash of SQUIRREL (in `SQLib.a`).

## 5.2 Release 5.2b

### 5.2.1 Changes

- Speed improvement of function's execution time. Up to 3 times faster than for the previous release.

### 5.2.2 Additions

- Added global variable `_error` that contains the error message when an error is thrown.

### 5.2.3 Bugs fixed

None.

## 5.3 Release 5.2

### 5.3.1 Changes

- Primitive `difference` return always a positive number.
- Using `output` outside in function return the value as the error code of the script.
- Removed SQUIRREL banners displayed when running SQUIRREL from a Terminal.

### 5.3.2 Additions

- Global variable `_from` that indicates from where a script has been run (terminal, tracker or SQUIRREL console).
- Primitives `Info` and `Question` that display a messagebox. Moved from the *GUI Add-On* in the *Communication Add-On*.
- Primitives `env.exists` `env.list` `env.get` `env.set` to access Environment variables.
- Primitive `entry.stats` to the *Storage Add-On* that give access to the *Statistical* informations on an entry.

### 5.3.3 Bugs fixed

None.

## 5.4 Release 5.1b and 5.1c

### 5.4.1 Notes

Those two upgrades have been applied to SQUIRREL 5.1 to mostly fix problems.

### 5.4.2 Changes

None.

### 5.4.3 Additions

- Added primitive `lfind` that output the position of something within a list.

### 5.4.4 Bugs fixed

- Crash caused by `FilePanel` when using a filter (*Storage Add-On*)
- Crash when killing a non existent thread (*Thread Add-On*)
- Impossibility to use list in expression such like `:a <> [ 3 4 5 ]`

## 5.5 Release 5.1

### 5.5.1 Notes

Starting from this release, SQUIRREL come in two release package, one for the developers and one for the user.

This version had quite some additions, thanks to all that gave me feedback at BeGeistert 5 in Düsseldorf (7-8 October 2000).

### 5.5.2 Changes

- Name of the executables have change, loosing it `.dr`.
- The primitive `with` have been renamed `use` to be more clear.
- Lists can contains variable, function/primitive calls and block.

### 5.5.3 Additions

- Added primitives `mime.desc`, `mime.install`, `mime.delete` and `mime.exist` to the *Storage Add-On* to give access to the BeOS Mime-database.
- Added `cc` and `bcc` methods to the *Mail* object to set/get the *carbon copy* and *blind carbon copy* fields.
- Added primitives `strim`, `strim.r`, `strim.l` and `smatch` to the *String Processing Add-On*.
- Added method `exist` to the *Dictionary* object of the *Data Structures Add-On*.
- Added global system variables: `_path`, `_install` and `_version`.

### 5.5.4 Bugs fixed

- Crash when storing the output of a function in a list.
- Crash when not using the output of the last function/primitive called in a block
- Fixed some memory problems

## 5.6 Developer Release 5.0

### 5.6.1 Notes

Although this release is numbered 5.0 it's mostly a maintenance release.

### 5.6.2 Changes

- Add-Ons are no longer all loaded when SQUIRREL start (except for the console version).
- `true` and `false` are no longer primitives but literal value.
- Definition of a list can be spread over several line

```
make "lst [  
    34.78  
    23.67  
    12.90  
]
```

### 5.6.3 Additions

- `with` primitive that allow to load one or more Add-On(s).
- `timing` primitive to the *Time* Add-On that output the elapsed time in *microseconds*.

### 5.6.4 Bugs fixed

- `substr` was crashing.

## 5.7 Developer Release 4.9

### 5.7.1 Notes

This release is the first version to be compiled for BeOS 5.0. Use with older version of BeOS is not supported.

Check the *GUI* Add-On documentation for more details about this release.

### 5.7.2 Changes

none

### 5.7.3 Additions

- *Mail* Add-On, that give the ability to send email from SQUIRREL
- Primitive `FilePanel` in the *Storage* Add-On, that allow the user to select one or more files from the disk(s).
- Global variable `_file` that give the full path of the current running script file.
- Primitive `lsub` in the *List Processing* Add-On, that return a part of a list.
- Methods `empty` to the Dictionary object (*Data Structure* Add-On), that empty the dictionary.

### 5.7.4 Bugs fixed

- Use of a string/word in a boolean expression
- *Segmentation Fault* when setting a file's attribute using the primitive `attr.set`, with a word value.
- SQUIRREL quitting before executing anything

## 5.8 Developer Release 4.8

### 5.8.1 Notes

This release enable the ability to send or receive message from another application. Sending a message to a SQUIRREL script could be a problem when several script are running as the signature of the scripts is always the signature of SQUIRREL . This problem will be fixed in the coming releases.

### 5.8.2 Changes

- The `catch` structure support now a second block to execute when an error is caught.

### 5.8.3 Additions

- Control Structure `switch`
- Hexadecimal number like `0x34` now supported
- Message object and Application messaging

### 5.8.4 Bugs fixed

- A little bug in the display of float number fixed (another :)

## 5.9 Developer Release 4.7

### 5.9.1 Notes

This release introduce *Skippy* which replace the old *2D Drawing Board* Add-on. This new add-on had it own documentation file.

### 5.9.2 Changes

- More informations are provided when a parsing error occurs

### 5.9.3 Additions

- The directive `#include` allow to include a script file during the parsing.
- Several new primitives to the Add-on *String Processing* have been added: `substr`, `insert`, `sreplace`, `serase`, `sleft`, `sright`, `sfind`, `sfind.all`, `sfind.last`, `sfirst.of`, `slast.of`, `sfirst.not.of`, `slast.not.of`.
- Three new primitives have been added to the add-on *Storage*: `entry.delete`, `entry.rename`, `entry.move`.
- New primitive `Wait` to the *Threading* Add-on that wait for a variable to be updated by another thread.

### 5.9.4 Bugs fixed

- Output of temporary object as input of a primitive/method/function was crashing in certain configuration.

## 5.10 Developer Release 4.5

### 5.10.1 Notes

One of the main objectives of this release was to introduce the new *Threading Add-on*. The *Threading Add-on* offers the possibility of performing multi-threading in SQUIRREL .

### 5.10.2 Changes

- There are now two executables in a SQUIRREL distribution

### 5.10.3 Additions

- New Add-on *Exec* allows one to execute external programs and to collect the output
- Several primitives have been added to the add-on for browsing directories e.g. *Storage*.
- *Threading* Add-on for multi-threading

### 5.10.4 Bugs fixed

- output in a loop within a function doesn't stop the function.

## 5.11 Developer Release 4

### 5.11.1 Notes

The GUI Add-On on the previous versions has been replaced by a newer version. This Add-On is no longer described in this manual but is described in a another manual.

### 5.11.2 Changes

- New version of the GUI Add-On has been totally recoded. This version makes obsolete all the old GUI commands and features. Check the GUI Add-On manual for more details.
- The command `gseq` has been changed to create a list in reverse order when the first input is greater than the second.

### 5.11.3 Additions

- New *Variable Binding* commands : `bind` and `unbind`
- New command `Precision` which changes the number of digits shown after the decimal point of a float number.

### 5.11.4 Bugs fixed

- A bug in `for .each` has been fixed
- One flaw in the "garbage collection" has been corrected
- Using a member call in a boolean expression
- Float display problems fixed
- Commands : `is.float` `is.object` and `is.number` weren't correct.

## 5.12 Developer Release 3

### 5.12.1 Notes

#### About this release

This release introduces the way SQUIRREL handles and uses objects. Three new Add-ons are also added, two of which directly exploits the new object ability.



### Data Structure Add-on

This Add-on offers two new data structures to be used in addition to the usual `list`: `Vector` and `Dictionary`.

A `Vector` is a dynamic array which offers better performance and more capabilities than a `list`. On the other hand, a `Dictionary` should be seen as a useful way to store information like in a `C` structure. However, it offers good performance, though not as good as a `Vector`.

### File Input/Output Add-on

Reading and writing from and to a text file is possible using the primitive defined in this Add-on.

### Storage Add-on

This new Add-on contains a first set of *File System* related primitives. For this release only the *file attributes* handling primitives are working. By using them, one is able to save numbers, strings, words, lists, `Vectors` and `Dictionaries` in the attributes of the file. There are also two primitives of this Add-on which manipulate MIME type.

## 5.12.2 Changes

- An error in the user manual has been fixed for the `foreach` loop which is `for . each`
- Some optimizations of the interpreter has been performed.

## 5.12.3 Additions

- New Object handling ability
- New "File I/O" Add-on
- New primitives `split` and `split.as.string` which splits a string in list elements added to the "String processing" Add-on
- New primitive `ljoin` added to "List processing". This primitive creates a string from the concatenation of all the elements of a list
- New "Data Structures" Add-on with the object `:Dictionary` and `Vector`
- New primitive `make.local` added to "Workspace" which simplifies the creation and initialization of a local variable
- New primitive `is.object` added to "Data processing". This primitive returns `true` if the input is an object (not a number, string, word or list)
- New "Storage" Add-on which offers file attributes and mime type manipulations

## 5.12.4 Bugs fixed

- Using `make` within a function when creating a variable (first use)
- Segmentation Fault when using the operator `+ - ...` with a list
- Problem with the use of an object within a function

## 5.13 Developer Release 2

### 5.13.1 Notes

#### About this release

Except for the *GUI Add-on* which is introduced in this release, there is no particular evolution of SQUIRREL between DR2 and DR1. A few bugs have been fixed and some minor changes have been applied. You may find more detail in this chapter about all that.

#### The GUI Add-on

The GUI Add-on shipped with this release is not yet complete. It's a first version, almost an *Alpha* version, which is subject to a lot of additions and changes in the coming releases.

Several features are missing in this release:

- No font handling
- No automatic placement tool (like the *packer* of Tcl/Tk)
- Lots Views objects are missing
- Drawing in a view is not implemented
- Lack of a good Tutorial

But this version works well with the GUI objects already implemented.

### 5.13.2 Changes

- The operator power  $\wedge$  has been changed to  $**$ , so for example:

```
@> print 5\^2
25
```

is now:

```
@> print 5**2
25
```

- The *Arithmetic* Add-on has been renamed *Mathematics*.
- The primitive `avg` which computes the average value of the input has been renamed `av`

### 5.13.3 Additions

- SQUIRREL could be set to be the *preferred application* of a SQUIRREL script file, and the file will be loaded and executed.
- New Add-on *GUI*
- The mathematical expression operators has been completed with `&` and `|`, which performs bit wise operations.

- New primitive `call` which calls a command specified by a word, in the *Control* Add-on
- New primitives in the *Mathematics* Add-on supporting angles in degree: `deg2rad` `rad2deg` `dsin` `dcos` `dtan` `dasin` `dacos` `datan` `dsinh` `dcosh` `dtanh`
- New primitive `string` which creates a string from the concatenation of the inputs, to the *String Processing* Add-on

#### 5.13.4 Bugs fixed

- Non quitting SQUIRREL when in `noconsole` mode
- Exception handling of unknown variables
- Wrong Precedence's of the math operators
- Segmentation fault when quitting SQUIRREL (both `noconsole` and `console` mode)
- Function calling

### 5.14 Developer Release 1

Please consider this release as a first iteration of SQUIRREL and will therefore, be far from being perfect. Although the interpreter is working well, several things are missing and will be added in the coming releases. In addition, the various Add-ons will be completed and some new ones will be added.

The availability of the SQUIRREL Add-ons API in the near future will allow third parties to write Add-ons, and will therefore, increase the versatility of SQUIRREL .

- Although recursive algorithms are working fine, a non-recursive approach is somewhat better when performance is an issue
- The *Automatic Garbage collection* is a prototype. The performance optimization is not yet perfect but is fully working.
- If you don't wish to use the *Automatic Garbage collection*, we recommend that you use the `gc` order to perform *Garbage collection* when performing heavy tasks. As well, using the `erase` order to destroy unused variables is recommended. But you're encouraged to use the *Automatic Garbage collection*.

# Index

- `*`, 12
  - `+`, 12
  - `-`, 12
  - `/`, 12
- `#include`, 33
- `**`, 12
- `<`, 14
- `<=`, 14
- `<>`, 14
- `=`, 14
- `>`, 14
- `>=`, 14
- `Args`, 34
- `Dictionary`, 59
  - `~av`, 59
  - `~empty`, 59
  - `~erase`, 60
  - `~exists`, 60
  - `~find`, 60
  - `~find.all`, 61
  - `~find.if`, 61
  - `~find.if.all`, 61
  - `~find.if.last`, 62
  - `~get`, 62
  - `~iterate`, 63
  - `~iterate.i`, 63
  - `~max`, 63
  - `~min`, 64
  - `~set`, 64
  - `~size`, 64
- `FilePanel`, 105
- `Image`
  - `~height`, 41
  - `~length`, 41
  - `~load`, 42
  - `~mime`, 42
  - `~path`, 42
  - `~save`, 42
  - `~valid?`, 42
  - `~width`, 43
- `Info`, 46
- `Locker`, 108
- `Mail`, 82
  - `~attachment`, 83
  - `~bcc`, 83
  - `~cc`, 83
  - `~content`, 83
  - `~from`, 84
  - `~reply`, 84
  - `~subject`, 84
  - `~to`, 84
- `Message`
  - `~add`, 37
  - `~delivered`, 37
  - `~empty`, 37
  - `~find`, 38
  - `~has`, 38
  - `~is.empty`, 38
  - `~is.remote`, 38
  - `~is.reply`, 39
  - `~is.waiting`, 39
  - `~names`, 39
  - `~previous`, 39
  - `~rem`, 39
  - `~replace`, 40
  - `~reply`, 40
  - `~send`, 40
  - `~timeout`, 40
  - `~what`, 41
- `Precision`, 48
- `Priority`, 109
- `Question`, 49
- `Send`, 82
- `Thread`, 109
  - `Thread.hop`, 110
  - `Thread.hoping`, 110
  - `Thread.kill`, 110
  - `Thread.priority`, 110

Thread.suspend, 110  
Thread.waitfor, 111  
ThreadID, 109  
Vector, 65  
    ~append, 65  
    ~av, 65  
    ~erase, 65  
    ~find, 66  
    ~find.all, 66  
    ~find.if, 66  
    ~find.if.all, 67  
    ~find.if.last, 67  
    ~get, 67  
    ~iterate, 68  
    ~iterate.i, 68  
    ~max, 68  
    ~min, 68  
    ~reverse, 70  
    ~reverse.new, 70  
    ~set, 69  
    ~size, 69  
    ~sort, 69  
    ~sort.new, 69  
Wait, 111  
\\', 15  
\\, 15  
\\b, 15  
\\f, 15  
\\n, 15  
\\r, 15  
\\t, 15  
\\v, 15  
\_error, 25  
\_file, 34  
\_from, 34  
\_install, 34  
\_path, 34  
\_version, 34  
abs, 85  
addon.func, 45  
addon.info, 44  
addon.list, 44  
attr.del, 100  
attr.exists, 100  
attr.get, 100  
attr.list, 100  
attr.set, 100  
av, 85  
bind, 29, 114  
break, 52  
butfirst, 54  
butlast, 54  
c2sec, 112  
call, 52  
catch, 24  
ceil, 85  
clock, 112  
clone, 54  
continue, 52  
ctime, 112  
dacos, 86  
dasin, 86  
datan, 86  
dcos, 86  
dcosh, 87  
deepclone, 54  
deg2rad, 85  
deth, 75  
dgloh, 75  
difference, 86  
dir.contains, 101  
dir.current, 101  
dir.exists, 101  
dir.list, 101  
dloh, 75  
do.until, 23  
do.while, 22  
dsin, 87  
dsinh, 87  
dtan, 87  
dtanh, 87  
dtreeh, 76  
dump, 76  
end, 25  
entry.delete, 102  
entry.exists, 102  
entry.icon, 102  
entry.isdir, 102  
entry.isfile, 103  
entry.islink, 103  
entry.match, 103  
entry.move, 103  
entry.rename, 103  
entry.reveal, 104  
entry.stats, 105  
env.exists, 114

env.get, 114  
env.list, 114  
env.set, 115  
erase, 115  
erf, 88  
erfc, 88  
exec.bg, 71  
exec.wait, 71  
exp, 88  
fclose, 72  
feof, 72  
fgets, 72  
first, 55  
floor, 88  
fopen, 72  
for, 21  
for.each, 24  
fput, 78  
fputs, 72  
gamma, 88  
gc, 115  
gseq, 78  
help, 115  
hypot, 89  
if, 18  
ifelse, 19  
iffalse, 19  
iftrue, 19  
img.crop, 74  
img.scale, 74  
incr, 89  
int, 89  
is.block, 55  
is.bool, 55  
is.dictionary, 59  
is.file, 73  
is.float, 55  
is.integer, 56  
is.list, 56  
is.mail, 82  
is.number, 56  
is.object, 56  
is.pred, 115  
is.prim, 116  
is.proc, 116  
is.string, 57  
is.vector, 65  
is.word, 57  
item, 57  
lappend, 78  
last, 57  
launch, 71  
lempty, 78  
lfind, 79  
lindex, 79  
list, 79  
ljoin, 79  
llength, 80  
ln, 89  
load, 47  
local, 27  
log10, 89  
lput, 80  
lremove, 80  
lscan, 80  
lsub, 80  
make, 28  
make.local, 28  
max, 90  
mem.usage?, 76  
mime.delete, 106  
mime.desc, 106  
mime.exists, 106  
mime.get, 106  
mime.icon, 107  
mime.install, 107  
mime.set, 107  
min, 90  
minus, 90  
modulo, 91  
name, 116  
not, 14  
output, 25  
parse.anything, 47  
parse.block, 47  
parse.float, 47  
parse.integer, 47  
parse.list, 48  
parse.number, 48  
parse.string, 48  
parse.word, 48  
pglov, 76  
plov, 77  
power, 91  
print, 49  
product, 90

quotient, 90  
racos, 91  
rad2deg, 91  
random, 91  
rasin, 92  
ratan, 93  
rcos, 92  
rcosh, 92  
read.anything, 49  
read.block, 49  
read.float, 50  
read.integer, 50  
read.list, 50  
read.number, 50  
read.string, 51  
read.word, 51  
repeat, 21  
reverse, 81  
rsin, 92  
rsinh, 92  
rtan, 93  
rtanh, 93  
safter, 94  
sbefore, 94  
scmp, 94  
sentence, 81  
serase, 94  
sfind, 95  
sfind.all, 95  
sfind.last, 95  
sfirst.not.of, 95  
sfirst.of, 95  
sgloh, 77  
show, 51  
sininsert, 96  
slast.not.of, 96  
slast.of, 96  
sleft, 96  
slength, 96  
sloh, 77  
smatch, 97  
snooze, 109  
split, 97  
split.as.string, 97  
sqrt, 93  
sreplace, 97  
sright, 97  
stolower, 98  
stop, 53  
stoupper, 98  
strim, 98  
strim.l, 98  
strim.r, 98  
string, 99  
substr, 99  
sum, 93  
switch, 20  
test, 19, 53  
thing, 116  
throw, 24  
time, 112  
timing, 113  
to, 25  
trans.mime, 41  
trans.name, 41  
type, 51  
unbind, 29, 116  
until, 23  
use, 34  
wait, 53  
while, 22  
word, 58