

# Gimli Quick User Guide

Thomas G. Moertel

February 14, 2009

## Contents

<b>1</b>	<b>Summary of operators and parsing precedence</b>	<b>3</b>
<b>2</b>	<b>Summary of built-in functions</b>	<b>4</b>
<b>3</b>	<b>Block expressions</b>	<b>6</b>
<b>4</b>	<b>Flow-control expressions</b>	<b>7</b>
4.1	If expression . . . . .	7
4.2	Unless expression . . . . .	7
4.2.1	If-unless relationship . . . . .	8
4.2.2	Using unless for defaulting . . . . .	8
4.3	For expression . . . . .	8
<b>5</b>	<b>Working with vectors</b>	<b>9</b>
5.1	Vector construction . . . . .	9
5.2	Vector indexing . . . . .	9
5.3	Vectorized operations . . . . .	10
<b>6</b>	<b>Working with random-access lists</b>	<b>11</b>
6.1	List construction . . . . .	11
6.2	List indexing . . . . .	11
<b>7</b>	<b>Working with tables</b>	<b>12</b>
7.1	Table construction . . . . .	12
7.2	Selection . . . . .	12
7.3	Projection . . . . .	13
7.3.1	Projection to a vector . . . . .	13
7.3.2	Simple projection to a table . . . . .	13
7.3.3	Subtractive projection . . . . .	14
7.3.4	Additive projection . . . . .	14
7.3.5	Chained table projections . . . . .	14
7.3.6	Advanced column specifications . . . . .	14
7.4	Joins . . . . .	15
7.4.1	Cartesian product . . . . .	16
7.4.2	Natural joins . . . . .	16
7.4.3	Tricks, tips, and work-arounds . . . . .	17

<b>8</b>	<b>Working with functions</b>	<b>18</b>
<b>9</b>	<b>Importing and exporting data</b>	<b>19</b>
9.1	Import options . . . . .	19
9.2	Export options . . . . .	19

## I Summary of operators and parsing precedence

The following table summarizes Gimli's operators and parsing precedence. Earlier-listed operations take precedence over later-listed operations.

Description	Infix associativity	Syntax / symbol
Application	–	<i>expr</i> ( <i>expr</i> , ... )
Arithmetic power	right	<i>expr</i> ^ <i>expr</i>
Unary negation	–	– <i>expr</i>
Series construction	left	<i>expr</i> : <i>expr</i>
Selection	–	<i>table</i> [ <i>expr</i> ]
Projection to vector	left	<i>table</i> \$ <i>number-or-column</i>
Projection to table	left	<i>table</i> \$( <i>pspec</i> , ... ) where <i>pspec</i> is one of— <i>colspec</i> , ... – <i>colspec</i> , ... + <i>colspec</i> , ... where <i>colspec</i> is one of— <i>colname</i> <i>colnum</i> <i>colname=expr</i> *
Inner natural join	right	<i>table</i> ==- <i>table</i>
Left-outer natural join		<i>table</i> ==- <i>table</i>
Right-outer natural join		<i>table</i> -== <i>table</i>
Full-outer natural join		<i>table</i> === <i>table</i>
Cartesian-product join		<i>table</i> *** <i>table</i>
Infix function application	left	<i>expr</i> %( <i>function</i> %) <i>expr</i>
Multiplicative operators	left	* /
Additive operators	left	+ -
String concatenation	left	++
Comparisons	left	== != < <= > >=
Unary not	–	! <i>expr</i>
Vectorized logical operators	right	&
Scalar logical operators	right	&&
Binding	right	<i>var</i> <- <i>expr</i> <i>var</i> <<- <i>expr</i>
	left	<i>expr</i> -> <i>var</i> <i>expr</i> ->> <i>var</i>
Flow control	left	<i>expr</i> if <i>test</i> <i>expr</i> unless <i>test</i> <i>expr</i> for <i>var</i> in <i>collection</i>

## 2 Summary of built-in functions

Gimli offers the following built-in functions:

- `as.list(expr, ...)`  
Evaluates the given expressions, converts the resulting values (which can be vectors, tables, or lists) into lists, and then joins the lists into a single, all-encompassing list.
- `as.table(expr, ...)`  
Evaluates the given expressions, converts the resulting values into lists, and then joins the lists into a single, all-encompassing table.
- `in(X, C)`  
Returns a vector having one element for each element of the vector  $X$ . The value of each output element is T if the value of the corresponding element in  $X$  is contained in the vector  $C$ ; otherwise the output element is F.
- `glob(pat, ...)`  
Each of the given strings is considered to be a shell “file globbing” pattern and matched against files in the filesystem. The result is a vector of files (pathnames) that matched the given patterns or NULL if there were no matches.
- `inspect(expr, ...)`  
For each given expression, this function prints the expression, an arrow, and then the result of evaluating the expression.
- `is.na(X)`  
Returns a vector where each element is T if the corresponding element in  $X$  is NA and F otherwise.
- `length(expr, ...)`  
Evaluates each given expression as a vector and returns the sum of the vectors’ lengths.
- `match(string, regex)`  
Matches the given string against the given POSIX extended regular expression and returns one of the following values:
  - NULL, if the *string* did not match the *regex*;
  - an  $n$ -element vector, where each element contains the subset of *string* matched by the corresponding parenthesized portion of *regex*;
  - or, if there were no parenthesized portions, a 3-element vector  $[b, m, a]$  where  $b$  is the subset of *string* before the match,  $m$  is the match, and  $a$  is the subset of *string* after the match.
- `names(table)`  
Returns the names of the columns of the given *table* in the order they appear in the table.

- `print(expr, ...)`  
Prints the results of evaluating the given expressions.
- `sort(vector, ...)`  
The elements of the given vectors are sorted and returned as a single vector.
- `uniq(vector, ...)`  
The elements of the given vectors are combined into a single vector where each distinct value occurs only once (i.e., duplicates are removed).
- `var(expr)`  
Evaluates the given *expr* as a string and returns the value of the variable whose name is given by the string.

### 3 Block expressions

A block expression is a series of expressions that act as one. The expressions in the series must be separated by semicolons.

In Gimli, blocks are created using either braces or *do-end* notation, which delimit the blocks. Typically, the first form is used for single-line blocks and the second for multi-line blocks, although this convention is not mandatory.

Single-line block	Multi-line block
{ <i>expr</i> ; ... }	do
	<i>expr</i> ; ...
	end

The result of evaluating a block is the final result of evaluating the series of expressions within the block.

**Example 1.** Blocks.

Input	Result	Description
{1; 2; 3}	→ 3	three expressions as a single-line block
do 1; 2 end	→ 2	two expressions as a multi-line block

Blocks are used to insert multi-expression logic into *if*, *unless*, *for*, and *function* expressions and—at present—can only be used in these contexts.

```
if x < 3 then do
  ...
end
```

## 4 Flow-control expressions

Gimli supports the following expressions that can be used to control the flow of evaluation:

Term form	Infix form
<code>if test then expr</code>	<code>expr if test</code>
<code>if test then expr else expr</code>	–
<code>unless test then expr</code>	<code>expr unless test</code>
<code>unless test then expr else expr</code>	–
<code>for var in vector do expr; ... end</code>	<code>expr for var in vector</code>

### 4.1 If expression

The *if* expression has three forms:

```

if et then eS
if et then eS else eF
eS if et

```

All three are evaluated as follows:

1. Gimli evaluates the test expression  $e_t$  to yield the test result  $t$ .
2. If  $t$  is F, NA, or NULL, the test fails; otherwise, the test succeeds.
3. If the test succeeded, Gimli evaluates  $e_S$  and returns the result as the value of the *if* expression.
4. Otherwise, if there is an *else* clause, Gimli evaluates  $e_F$  and returns the result as the value of the *if* expression.
5. Otherwise, Gimli returns the result of the test ( $t$ ) as the value of the *if* expression.

### 4.2 Unless expression

The *unless* expression is the opposite of the *if* expression and also has three forms:

```

unless et then eF
unless et then eF else eS
eF unless et

```

All three are evaluated as follows:

1. Gimli evaluates the test expression  $e_t$  to yield the test result  $t$ .
2. If  $t$  is F, NA, or NULL, the test fails; otherwise, the test succeeds.
3. Unless the test succeeded (i.e., if the test failed), Gimli evaluates  $e_F$  and returns the result as the value of the *unless* expression.

4. Otherwise, if there is an *else* clause, Gimli evaluates  $e_S$  and returns the result as the value of the *unless* expression.
5. Otherwise, Gimli returns the result of the test ( $t$ ) as the value of the *unless* expression.

#### 4.2.1 If-unless relationship

The following equivalence summarizes the relationship between the *if* and *unless* expressions:

$$\text{if } e_t \text{ then } e_S \text{ else } e_F \equiv \text{unless } e_t \text{ then } e_F \text{ else } e_S$$

#### 4.2.2 Using unless for defaulting

The infix form of the *unless* expression makes a convenient way to specify default values:

$$\text{default unless value}$$

Unless a *value* is supplied, the *default* value will be used instead. **Note:** This trick works only for non-logical values.

### 4.3 For expression

The *for* expression is used to iterate over the elements of a collection (list or vector). It has two forms:

$$\begin{array}{ll} \text{Term form} & \text{for } x \text{ in } e_C \ e_B \\ \text{Infix form} & e_B \text{ for } x \text{ in } e_C \end{array}$$

In the term form, the body expression  $e_B$  must be a block, i.e., a series of expressions delimited by braces or *do-end* notation.

Gimli evaluates *for* expressions as follows:

1. Gimli evaluates the collection expression  $e_C$  to yield the collection  $C$  over which to iterate.
2. If  $C$  is empty, Gimli returns NULL as the value of the *for* expression.
3. Otherwise, Gimli iterates over  $C$ , binding each element in turn to the given variable  $x$  and then evaluating the body expression  $e_B$ .
4. Gimli returns the result of the last evaluation as the value of the *for* expression.

**Example 2.** For expressions.

Input	Result	Description
$x \leftarrow 1; \text{ for } y \text{ in } 1:5 \{ x \leftarrow x * y \}$	→ 120	compute 5!
$x \leftarrow 1; x \leftarrow x * y \text{ for } y \text{ in } 1:5$	→ 120	same as above

## 5 Working with vectors

Gimli's support for logical, string, and numeric values is vectorized; that is, you can create vectors containing any number of such values—as long as all the values within each vector are of the same kind. In fact, to Gimli, scalar values are really vectors of length one.

Vectors are created using bracket syntax:

```
[ elems... ]
c( elems... )
```

The first form is easier to type. The second form is provided for compatibility with R's vector-construction syntax.

### 5.1 Vector construction

A list of elements *elems* goes inside of the brackets. Each element can be a scalar value like 3 or an expression. Such element expressions will be evaluated as vectors themselves, and the resulting elements will be inserted into the vector being constructed.

**Example 3.** Vector construction.

Input		Result
[1, 2, 2+1]	→	[1, 2, 3]
c(1, 4, 5)	→	[1, 4, 5]
[1, c(2, 3)]	→	[1, 2, 3]
x <- [1, 2]; [x, 2+x]	→	[1, 2, 3, 4]

### 5.2 Vector indexing

You can select elements from a vector using indexing syntax, which can also be used on lists (see §6.2 on page 11) and, in a different way, on tables (see §7.2 on page 12):

```
vector [ expr ]
```

where *expr* can be one of the following:

<i>Expr</i>	Example	Meaning
a number	2	Select the 2nd element
a vector of numbers	[3, 1]	Select the 3rd and then the 1st elements
a negative number	-2	Select all but the 2nd element
a vector of negative numbers	-[3, 1]	Select all but the 1st and 3rd elements
a vector of logicals	[T, F]	Select elements in odd positions

Note that when indexing on a vector of logical values, the logical values will be recycled as necessary to create a one-to-one correspondence between the logicals and the elements of the vector being indexed over. The elements for which the corresponding logical value is T will be selected; the others will be omitted.

**Example 4.** Vector indexing.

Input	Result
[1,2,3,9] [2]	→ 2
[1,2,3,9] [c(2,NA,4)]	→ [2,NA,9]
[1,2,3,9] [[T,F,F,T]]	→ [1,9]
[1,2,3,9] [c(F,T)]	→ [2,9]
[1,2,3,9] [F]	→ NULL
[1,2,3,9] [T]	→ [1,2,3,9]

**5.3 Vectorized operations**

All of Gimli's standard operators, such as  $+$ ,  $!$ , and  $<$  are vectorized and can be applied directly to vectors. When Gimli applies a binary operator, such as  $+$ , to vectors  $X$  and  $Y$ , it applies the operator across the vectors *elementwise*: the first elements of both vectors are processed as a pair, and then the second elements, and so on. If the vectors are of unequal lengths, the shorter's elements will be recycled until its length matches the longer's.

**Example 5.** Operations on vectors.

Input	Result
[1,2,3,4] + 1	→ [2,3,4,5]
[1,2,3,4] * [1,0]	→ [2,0,4,0]
! [T,F,F]	→ [F,T,T]

## 6 Working with random-access lists

Gimli's lists are like vectors but can contain any kind of data, not just scalar values. You can put vectors, tables, and even lists inside of lists, and the elements of a list need not be of the same kind. Further, you can name the elements of lists, making them useful as associative arrays.

### 6.1 List construction

You can create lists via the `list` constructor:

**Example 6.** Constructing lists.

Input	Result
<code>list()</code> # empty list	<code>list()</code>
<code>list("hi")</code>	<code>[1] =&gt; "hi"</code>
<code>list(x="hi")</code>	<code>\$x =&gt; "hi"</code>
<code>list(x="hi", [T,F])</code>	<code>\$x =&gt; "hi"</code> <code>[2] =&gt; [T,F]</code>
<code>list(x=list(123))</code>	<code>\$x =&gt; [1] =&gt; 123</code>
<code>list(list(x=123))</code>	<code>[1] =&gt; \$x =&gt; 123</code>

### 6.2 List indexing

Gimli lets you select elements from lists just like from vectors but gives you additional options. Lists elements can be named, and so you can use names, in addition to numbers, for indices.

Gimli provides two ways to index lists—selection and simple projection. Selection works just like indexing on vectors: you provide a vector that describes the elements you want, and Gimli gives you a new list that contains only those elements.

**Example 7.** List indexing via selection.

Input	Result
<code>list(x=1,5,"hi") [2]</code>	<code>[1] =&gt; 5</code>
<code>list(x=1,5,"hi") [[1,3]]</code>	<code>\$x =&gt; 1</code> <code>[2] =&gt; "hi"</code>
<code>list(x=1,5,"hi") ["x"]</code>	<code>\$x =&gt; 1</code>
<code>list(x=1,5,"hi") [-1]</code>	<code>[1] =&gt; 5</code> <code>[2] =&gt; "hi"</code>
<code>list(x=1,5,"hi") [[F,T,T]]</code>	<code>[1] =&gt; 5</code> <code>[2] =&gt; "hi"</code>
<code>list(x=1,5,"hi") [F]</code>	<code>list()</code>

Simple projection, on the other hand, lets you pluck a single value from a list.

**Example 8.** List indexing via simple projection.

Input	Result
<code>list(x=1,5,"hi") \$3</code>	<code>"hi"</code>
<code>list(x=1,5,"hi") \$x</code>	<code>1</code>

## 7 Working with tables

This section describes Gimli’s table tools—selections, projections, and joins.

### 7.1 Table construction

In Gimli, tables are constructed with the `table` constructor:

```
table(colname=vector, ...)
```

where each `colname=vector` pair defines a column in the resulting table. The `colname` expression defines the name of the column, and the `vector` expression defines the column’s values. All columns of the table must have the same length.

Gimli interprets each `colname` expression as follows. First, if the expression is a name, such as `x` or `my.col`, Gimli uses the expression itself as the column’s name. Otherwise, Gimli evaluates the expression and uses the resulting string as the column’s name. If any of the column names are not unique, Gimli makes them unique by appending suffixes like `.1`, `.2`, and so on.

**Table splicing.** You can “splice” columns from existing lists and tables into a new table by listing the to-be-spliced lists and tables amongst the `colname=vector` pairs in the new table’s `table` constructor. When you do so, the vectors and columns from the spliced lists and tables will be inserted directly into the new table. As always, the lengths of all the vectors and columns must be the same.

**Example 9.** Table construction.

Input		Result
<code>table(x=[1,2])</code>	→	x 1 1 2 2
<code>x &lt;- "Z"; table(x=[1,2],(x)=[2,3])</code>	→	x Z 1 1 2 2 2 3
<code>t &lt;- table(x=[1,2]); table(t,Q=[T,F])</code>	→	x Q 1 1 T 2 2 F

### 7.2 Selection

*Selection* is the process of selecting the desired rows from a given table. The result of selection is a new table that contains only the desired rows.

In Gimli, the following syntax is used to perform selection:

```
table [ expr ]
```

For each row in *table*, Gimli evaluates the expression *expr* in the row’s context. If the expression evaluates to `F`, `NA`, or `NULL`, the row is skipped; otherwise, the row is included in the new table.

**Example 10.** Selection.

<code>t [T]</code>	select all rows from table <i>t</i>
<code>t [x&lt;3]</code>	select rows from table <i>t</i> where $x < 3$
<code>t [!is.na(y)]</code>	select rows from table <i>t</i> where <i>y</i> is not NA
<code>t [x&lt;3 &amp;&amp; !is.na(y)]</code>	select rows from table <i>t</i> where $x < 3$ and <i>y</i> is not NA

### 7.3 Projection

*Projection* is the process of manipulating the columns of a table to create a new vector or table. Simple projections can be used to take a subset of a table's columns. More complex projections can be used to merge, delete, and add columns.

#### 7.3.1 Projection to a vector

The simplest form of table projection in Gimli is projection to a vector, in which a single column of a table is extracted as a vector.

```
table $ colname
table $ colnum
```

#### 7.3.2 Simple projection to a table

Projection to a table is used to reshape the column structure of a table to result in a new table. The new table has the same number of rows as the original but its columns are determined by a *projection specification*, or *pspec* for short. For each row in the input table, Gimli evaluates the *pspec* to generate a new row, which is added to the new table.

The projection-to-table syntax is somewhat complex yet simple for common cases:

```
table $( pspec )
where pspec is one of—
colspec, ...
- colspec, ... (subtractive)
+ colspec, ... (additive)
where colspec is one of—
colname
colnum
colname=expr
*
```

A *pspec* specifies a set of column-generation rules via one or more *colspecs*. The most general *colspec* takes the form *colname=expr* and tells Gimli to generate a column named *colname* by evaluating *expr* for every row in the original table and accumulating the results as a column. All other *colspec* forms are translated into this form by Gimli before evaluation:

```
colname → colname=colname
n       → colnamen=colnamen
*       → colnamei=colnamei for all columns i
```

**Example 11.** Simple projections to tables.

```
t$(1)           take the first column of t
t$(ped.id)      take the column named "ped.id" from t
t$(ped=ped.id)  take the column named "ped.id" from t and call it "ped"
t$(*)           take all columns from t
```

You can build multi-column projections by providing a comma-separated list of *colspecs*:

**Example 12.** Multi-column projections to tables.

```
t$(1,2,4)       take the first, second, and fourth columns
t$(1,ped.id)    take the first column and the column named "ped.id"
```

### 7.3.3 Subtractive projection

Like its name implies, subtractive projection is used to remove columns from a table. It works by taking all of the columns *except* the ones you specify in your *colspecs*.

**Example 13.** Subtractive projections.

```
t$(- 1)         take all but the first column
t$(- ped.id)    take all but the column named "ped.id"
t$(- 1,2)       take all but the first and second columns
t$(- ped.id,2)  take all but the "ped.id" and second columns
```

### 7.3.4 Additive projection

Additive projection generates a new table by taking all of the columns from an input table and then adding the ones you specify in your *colspecs*. If a *colspec* defines a column that has the same name as one of the input table's columns, the *colspec*'s column will *replace* the input column. Thus additive projection can also be used to "change" columns.

**Example 14.** Additive projections.

```
t$(+ x=x+1,y=y+1)  add 1 to columns "x" and "y"
t$(+ new=x+y)      create a column "new" as the sum of columns "x" and "y"
```

### 7.3.5 Chained table projections

Any number of table projections can be chained by separating their *pspecs* by semicolons. The resulting chained projection is exactly equivalent to performing each projection in sequence:

$$table \$ ( p_1; p_2; \dots ) \equiv table \$ ( p_1 ) \$ ( p_2 ) \dots$$

### 7.3.6 Advanced column specifications

As you may recall, a *pspec* comprises one or more *colspecs*, each of which specifies a column to be generated, added, or subtracted. Typically, you specify a column by its name or number, but Gimli gives you other options—vectors, strings, or arbitrary expressions. Here are the rules Gimli uses to interpret your input as *colspecs*:

1. If the input is a number  $n$ , Gimli considers it to specify the  $n$ th column of the input table.
2. Otherwise, if the input is a variable  $x$ , Gimli considers it to specify the column named  $x$ .
3. Otherwise, if the input is of the form  $x = e$ , where  $x$  is a variable and  $e$  is an expression, Gimli considers it to specify a new column named  $x$  to be generated by evaluating  $e$  over the rows of the input table.
4. Otherwise, if the input is a star (\*), Gimli considers it to specify all of the input table's columns.
5. Otherwise, Gimli considers the input to be an arbitrary expression. Gimli evaluates the expression to yield a vector, and then interprets each element of the vector as follows:
  - (a) If the element is a number  $n$ , Gimli considers it to specify the  $n$ th column of the input table.
  - (b) Otherwise, if the input is a string value  $s$ , Gimli considers it to specify the column named  $s$ .

**Example 15.** Advanced column specifications.

<code>t\$(1+1)</code>	specifies second column
<code>t\$(1,2,3)</code>	specifies first three columns
<code>t\$(1:3)</code>	same as above
<code>t\$(1, [2,3])</code>	same as above
<code>t\$(c(1,2,3))</code>	same as above
<code>t\$(1+c(0,1,2))</code>	same as above
<code>t\$(x)</code>	specifies column named "x"
<code>x &lt;- 1; t\$((x))</code>	specifies first column
<code>x &lt;- 1; t\$(x, (x))</code>	specifies column "x" and first column
<code>x &lt;- 1; t\$(x, [x])</code>	same as above
<code>x &lt;- "y"; t\$((x))</code>	specifies column "y"
<code>x &lt;- "y"; t\$([1,x])</code>	specifies first column and column "y"

## 7.4 Joins

Gimli offers many ways to join two tables  $x$  and  $y$ . The most basic join is the Cartesian product, in which each row in  $x$  is combined with each row in  $y$ . Gimli also supports the more common, "natural" joins, in which each row in  $x$  is combined with its corresponding row in  $y$ .

Gimli's join operators are summarized below:

CARTESIAN PRODUCT	
$x *** y$	
NATURAL JOINS	
Inner	$x --- y$
	$x ==- y$
Left outer	$x ==- y$
Right outer	$x -== y$
Full outer	$x === y$

### 7.4.1 Cartesian product

The Cartesian product is straightforward and joins two tables  $x$  and  $y$  by enumerating every possible combination of rows that can be formed by drawing a row from  $x$  and combining it with a row drawn from  $y$ . More formally,

$$x***y \equiv \{R_x \# R_y \mid R_x \leftarrow \text{rows of } x, R_y \leftarrow \text{rows of } y\}.$$

Note that if  $x$  has  $M$  rows and  $y$  has  $N$  rows,  $x***y$  will have  $N \times M$  rows.

### 7.4.2 Natural joins

Natural joins assume that the tables  $x$  and  $y$  to be joined are related and that their corresponding rows can be “aligned” by finding common values across the tables in certain key columns. Gimli can usually infer the key columns by examining the tables and looking for common column names, but you can specify the join criteria by adorning the join operator with them, like so:

$x \{n\}---\{m\} y$	join using column $n$ from $x$ and column $m$ from $y$
$x \{n\}--- y$	use column $n$ from both tables
$x ==-\{n\} y$	same as above
$x \{n, o\}---\{p, q\} y$	join on two columns from each table
$x ==- y$	let Gimli figure out the key columns for you

Gimli’s natural-join operators differ in how they handle semi-aligned tables. The inner join ( $---$  or  $==-$ ) will include only those input rows that are perfectly aligned, i.e., have corresponding rows in the opposite table. The left-outer join ( $==-$ ) will include all of the left table’s rows but only perfectly aligned rows from the right table. Likewise, the right-outer join ( $-==$ ) will include all of the right table’s rows but only perfectly aligned rows from the left table. Finally, the full-outer join will include all rows from both tables, regardless of alignment.

**NA values.** When a row from one of the input tables is included in the output table but has no corresponding row in the other input table, the missing values are filled with NA values.

**Output columns.** The result of a join operation is a new table whose column structure is determined by the column structure of the tables being joined and the join type. First, *all* of the left-hand table’s columns will appear as the leftmost columns of the output table. This happens always. Next, if the join type is right-outer or full outer, the key columns of the right-hand table will be contributed to the output table; otherwise, they will be omitted. Finally, all of the non-key columns of the right-hand table will

be contributed to the output table. If any of the output columns' names would be non-unique, Gimli will make them unique by appending numeric suffixes. The following table summarizes.

Left Columns	Join Type	Right Columns		Output Columns
n x	--	n y	→	n x y
n x	==	n y	→	n x y
n x	==	n y	→	n x n.1 y
n x	===	n y	→	n x n.1 y

### 7.4.3 Tricks, tips, and work-arounds

This section includes handy tips for working with Gimli and working around some of its present limitations.

**Full-outer join with merged identifiers.** Right now, Gimli doesn't provide a full-outer join operator that automatically merges the left- and right-hand identifiers contributed by its input tables. However, the following recipe provides a manual-merge workaround:

```
gimli> en <- table(n=1:3, en=["one","two","three"]);
gimli> es <- table(n=2:4, es=["dos","tres","cuatro"]);
```

```
gimli> en === es
  n      en n.1      es
1 1  "one" NA      NA
2 2  "two" 2  "dos"
3 3 "three" 3  "tres"
4 NA      NA 4 "cuatro"
```

```
gimli> (en === es)$(+ n = n.1 unless n; -n.1)
```

```
  n      en      es
1 1  "one"      NA
2 2  "two"  "dos"
3 3 "three"  "tres"
4 4      NA "cuatro"
```

## 8 Working with functions

This section describes Gimli's rich support for functions. In Gimli, functions are first-class values: they can be created, stored in variables, and passed to other functions.

*[To be written.]*

## 9 Importing and exporting data

At present, Gimli’s facilities for importing and exporting tabular data are limited to three formats:

- **Comma separated.** Conforming to typical spreadsheet import/export rules: one row per line, each row’s values separated by a comma. Also called CSV format.
- **Tab separated.** Same as CSV format but uses tabs instead of commas to separate values. Also called TSV format.
- **Whitespace separated.** Any number of whitespace characters are used to separate values. This is the same format as Gimli’s native display format. Also called WSV format.

By default, all three forms include a header upon export and expect a header upon import. The WSV export format also includes a row count in the left-most column, which R and Gimli ignore on import.

Gimli’s import/export commands are summarized in the following table:

Import	Export
<code>var &lt;- read.csv(filename)</code>	<code>write.csv(table, filename)</code>
<code>var &lt;- read.tsv(filename)</code>	<code>write.tsv(table, filename)</code>
<code>var &lt;- read.wsv(filename)</code>	<code>write.wsv(table, filename)</code>

The result of a successful import is the table represented by the given file. The result of a successful export is the pathname to the exported file.

### 9.1 Import options

All of the `read.x` commands accept the following options, which may be combined:

Parameter	Default value	Description
<code>header</code>	T	Whether the file to be imported has a header
<code>transpose</code>	F	Whether to transpose the file’s cells upon import

**Example 16.** Import options.

```
read.csv("data.csv", h=F)  Import "data.csv", which does not have a header
read.csv("data.csv", t=T)  Import "data.csv", transposing it
```

### 9.2 Export options

All of the `write.x` commands accept the header option:

Parameter	Default value	Description
<code>header</code>	T	Whether to include a header in the output

**Example 17.** Export options.

```
read.csv(t, "data.csv", h=F)  Export table t as "data.csv", no header
```

■