

# lisp-babel

Derek Feichtinger

June 4, 2016

## Contents

<b>1</b>	<b>How to use this document</b>	<b>1</b>
<b>2</b>	<b>Version information</b>	<b>2</b>
<b>3</b>	<b>using a table as input for a src block</b>	<b>2</b>
3.1	simple example . . . . .	2
3.2	passing a sub-range . . . . .	3
3.3	Investigating how tables are passed to the src block . . . . .	3
<b>4</b>	<b>calling source blocks as a function</b>	<b>4</b>
4.1	Chaining source block execution . . . . .	4
4.2	simple call syntax using CALL . . . . .	4
4.3	Naming an output table produced by a CALL . . . . .	5
<b>5</b>	<b>Inline src calls</b>	<b>6</b>
<b>6</b>	<b>Defining buffer wide variables for src blocks</b>	<b>6</b>
<b>7</b>	<b>Using a :post function for post-formatting and executing generated tables</b>	<b>6</b>

## 1 How to use this document

You should look at this document in its Org mode source form. The PDF rendering is useful to see the results of some of the export options, but the syntax of the source block is only seen in the source text.

## 2 Version information

```
(princ (concat (format "Emacs version: %s\n" (emacs-version))
               (format "org version: %s\n" (org-version))))
```

```
Emacs version: GNU Emacs 24.5.1 (x86_64-unknown-linux-gnu, GTK+ Version 3.10.8)
  of 2015-05-04 on dflt1w
org version: 8.3.2
```

## 3 using a table as input for a src block

### 3.1 simple example

We first create a table from a lisp **list of lists**. Each inner list will form a row in the resulting table. I already insert a header row with the names of three columns. A separator line can be obtained by putting the `hline` symbol into the resulting list.

```
(cons '(col1 col2 col3)
      (cons 'hline
            (loop for i from 5 to 15 collect '(',i ,(* i 5) "))))
```

col1	col2	col3
5	25	
6	30	
7	35	
8	40	
9	45	
10	50	
11	55	
12	60	
13	65	
14	70	
15	75	

We now can fill the third column by passing the table into the next source block. We force babel to treat the first row as table header by using the `:colnames yes` header argument. This also causes the result table to contain the headers (as long as the new table has the same number of columns as the original table)

Here I also demonstrate the use of the `-n` option that will export the code with line numbers.

```

1 (let (result)
2   (dolist (row tbl result)
3     (setf (nth 2 row) (* 2 (nth 1 row)))
4     (setq result (cons row result)))
5   (reverse result))

```

coll	col2	col3
5	25	50
6	30	60
7	35	70
8	40	80
9	45	90
10	50	100
11	55	110
12	60	120
13	65	130
14	70	140
15	75	150

### 3.2 passing a sub-range

It is possible to specify a sub-range for the table that is handed over through `:var`. But currently it does not work well with the `:colnames yes` option, as the following example shows.

```

1 (let (result)
2   (dolist (row tbl result)
3     (setf (nth 2 row) (* 2 (nth 1 row)))
4     (setq result (cons row result)))
5   (reverse result))

```

7	35	
8	40	80
9	45	90
10	50	100
11	55	110

### 3.3 Investigating how tables are passed to the src block

coll	col2	col3
10	str	two strings
20.5	str2	2 strings

```
(pp tbl)

((10 "str" "two strings")
 (20.5 "str2" "2 strings"))
```

Note that the `raw` value output of the source block does not yield the same. It loses the string quotes of the single entries!

```
tbl

((10 str two strings) (20 str2 2 strings))
```

## 4 calling source blocks as a function

### 4.1 Chaining source block execution

I can have another piece of code implicitly called by using its name as an input variable in another code block. So, I could directly fill the third column of our initial example table without ever having to print out that table table. We can just pass into the next function a variable `tbl` and the name of the initial code block `make-table1`.

```
(let (result)
  (dolist (row tbl result)
    (setf (nth 2 row) (* 2 (nth 1 row)))
    (setq result (cons row result)))
  (reverse result))
```

### 4.2 simple call syntax using CALL

We first define a function in a named code block called `mydouble`. The variable `x` will be passed in by defining a header argument `:var x`

```
(* 2 x)
```

Now we can call this babel function by using the code block's name `mydouble` from any place in the document. For example:

```
10
```

Another example where we pass in two variables `x` and `y`.

```
(/ x y)
```

Note that **you can/must pass additional header arguments** to the call. The ones added at the end influence the final result (e.g. putting it into a drawer), while the ones added in `[]` are evaluated in the context of the original definition (e.g whether to capture the output or return a value).

4

Another alternative calling syntax

5

### 4.3 Naming an output table produced by a CALL

If the called function produces an output table that one wants to use in subsequent function calls or in table formulas (using the **remote** keyword) one can give the **CALL** a name utilizing the syntax used for other org elements:

5	25
6	30
7	35
8	40
9	45
10	50
11	55
12	60
13	65
14	70
15	75

5	25
6	30
7	35
8	40
9	45
10	50
11	55
12	60
13	65
14	70
15	75

## 5 Inline src calls

This is the result of an inline src call in lisp:

```
15
and this is another:
15
15
```

## 6 Defining buffer wide variables for src blocks

One can use a verbatim block like this. I define a named block `myvar` and I pass it into the variable `s` of the following code block.

```
world

(concat "hello " s)

hello world
```

## 7 Using a `:post` function for post-formatting and executing generated tables

Often I produce multiple tables from a source block (e.g. printing several pandas data frames). These tables do not get aligned in the org document after the execution of the code block (even though they will get aligned upon exporting the document). Also, I may want to have table calculations using `#+TBLFM` lines executed, instead of manually having to execute them in the resulting tables.

The following function can be used in a `:post` argument for getting all tables in the output aligned and their TBLFM instructions executed, as shown further below

```
(with-temp-buffer
  (erase-buffer)
  (insert text)
  (beginning-of-buffer)
  (org-mode)
  (while
    (search-forward-regexp org-table-any-line-regexp nil t)
    (org-table-align)
```

```

(org-table-recalculate 'iterate)
(goto-char (org-table-end)))
(buffer-string))

```

```

| 5 | 22222 |
| 0 |      |
| 12 | 45 |
| ----+-----|
| 17 |      |
#+TBLFM: @>$1=vsum(@1..@-1)

```

```

| 1 | 22222 |
| 0 |      |
| 12 | 45 |

```

Example without using the `:post` function:

```

(princ
(concat
  "#+CAPTION: Test1\n"
  "|A|B|C|\n"
  "|---\n"
  "|1|20|300|\n"
  "|200|30|4|\n"
  "|---\n"
  "|||\n"
  "#+TBLFM: @>$1..@>$3=vsum(@I..@II)\n"
  "\n#+CAPTION: Test2\n"
  "|A|B|C|\n"
  "|---\n"
  "|1|20|300|\n"
  "|200|30|4|\n"
)))

```

Table 1: Test1		
A	B	C
1	20	300
200	30	4

Table 2: Test2		
A	B	C
1	20	300
200	30	4

The same example with the `:post` function:

```
(princ
(concat
  "#+CAPTION: Test1\n"
  "|A|B|C|\n"
  "|---\n"
  "|1|20|300|\n"
  "|200|30|4|\n"
  "|---\n"
  "|||\n"
  "#+TBLFM: @>$1..@>$3=vsum(@I..@II)\n"
  "\n#+CAPTION: Test2\n"
  "|A|B|C|\n"
  "|---\n"
  "|1|20|300|\n"
  "|200|30|4|\n"
))
```

Table 3: Test1		
A	B	C
1	20	300
200	30	4
201	50	304

Table 4: Test2		
A	B	C
1	20	300
200	30	4

Emacs 25.0.94.1 (Org mode 8.3.4)