Teapot User guide

Michael Haardt (michael@moria.de)

ABSTRACT

For ages, spread sheet programs have been closely associated with financial calculations done by typical end-users. But it has shown that there is also hacker's work which can be done with them, like calculate monitor timings for various resolutions, produce convincing time statistics which justify the lack of documentation or the need for a budget increase to your employer. This first part of this user guide explains how the various functions of teapot are used, whereas the second part gives an introduction to spread sheets and explains the expression evaluator and its functions.

1. Copyright and usage conditions

I can be reached by:

Snail mail:
Michael Haardt
An Neuenhofen 40
47800 Krefeld
Germany

Email:
michael@moria.de

Disclaimer: I am in no way responsible for anything caused by using this program, so if your computer or house blows up, that's your problem.

2. Introduction to spread sheets

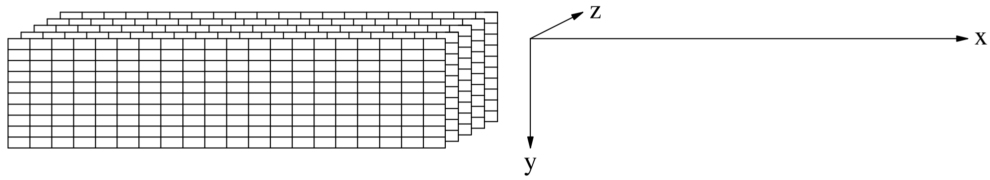
2.1 General introduction

A spread sheet consists of cells formed by rows and columns. Additionally, in many spread sheets you

---

1. Back then, around 1992, all IRC servers supported **/wallop**. If you were bored, not knowing what to do and it was just too silent and peaceful, sending a **/wallop pineapple** or **/wallop teapot** was an effective way to start a silly but entertaining operator conversation. The fact that the first versions of this program were unnamed, because I had no idea how to call it and because all suggestions sounded boring, is pure coincidence. :)

have a third dimension, which you can imagine as various sheets laying on top of each other. The third dimension allows you to hide intermediate results, show you additional results you do not want to appear in the "official" tables, keep sheets per time period (like 12 sheets for each month in a year) while allowing you to make calculations over the entire time interval and much more. The figure below shows the three dimensions:



You can think of cells as variables, which value is the value of an associated expression. The expression may be constant, like 1.23, or it may be a function of other cell values. The advantage compared to a programmable calculator is that if you change a number, you directly see all changes in other cells caused by that. Often this allows you to get a feeling how much you may change basic sizes with still getting satisfying results without having to solve the problem analytically.

Spread sheets offer many editing functions in order to modify, clear, copy and move cells or blocks of cells. Besides the usual mathematical functions, there are functions which work on blocks of cells, like calculating the sum of a block or counting all non-empty elements. Further there are functions working on character strings, because most likely you also want text besides numbers. The next section will introduce you to some of these by examples.

teapot is a traditional spread sheet and a typical UNIX program, because it does just one thing: Calculations. It does not include any graphics functions and never will, but it allows to export data in many formats, so you can use your favourite graphics software.

2.2 The first steps

Now that you should have an idea, it is probably a good time to make your first steps. This section will show you how to create and save a sheet which contains two numbers and their sum. If your system supports message catalogs, please make sure that **NLSPATH** and the **LC_*** variables are set right for your locale. Now start the program without any arguments:

teapot

You see an empty sheet with the cell cursor being at the upper left corner. Further, the status line tells you that this cell is really empty:

E @(0,0,0)=

The **E** means that you can edit the sheet. A **V** would mean that you could only view its contents. The meaning of @() will be explained soon. You are now in the command mode of teapot. Now type return to edit this cell. A complete list of command mode functions will be given later. A prompt will appear below the status line:

Cell contents: 1

Now the cell at position 0,0,0 has the integer constant 1. The status line shows you the cell contents, whereas in the sheet you see its value. Since constants are identical with their values, both are 1. Now move the cell cursor down one row and edit that cell, giving it the integer constant **41**.

Now that you have two numbers, move the cell cursor to cell 0,2,0 and give that cell the following contents:

Cell contents: @(0,0,0)+@(0,1,0)

If you were confused about the difference between contents and value of a cell, it should become more clear now: The status line shows the contents, which is the arithmetic expression to calculate the sum of two cells, whereas in the sheet you see the value of that expression: 42, which was to be expected. @(x,y,z) is a function which takes three coordinates and returns the value of the cell at the given position.

As you can see, the arithmetic expression is not too readable. If you would move cells around, it would not even work any more. For these reasons, you can use symbolic names instead of coordinates, called labels. When used in an expression, a label is like a pointer to a cell, its data type is *location*. Move to cell 0,0,0 and use **/** (slash) in command mode to get into the main menu. Depending on your screen size, you may not see all of it. In this case, move the highlighted block right (or left) to scroll through it and to see all items. Now change its label attribute: A)ttributes, L)abel:

Cell label: Paper

Then go one cell down and change its label to **Tapes**. After, move again one cell down and change the expression to:

Cell contents: @(Paper)+@(Tapes)

As you see, you can call the function @ with three integer values or with one location value. Now the expression is more understandable, at least to you. To someone else, the sheet only contained three numbers, so a little text should be added. To accomplish that, a new column needs to be inserted: B)lock, I)insert, C)olumn, W)hole column. The last menu item means that you want to insert a whole new column, not only a partial column. If you move the cursor around, you will see that everything is still fine, because you used labels. Go to cell 0,0,0 and edit it:

Cell contents: "Paper:"

This is how you enter strings. A string is a data type on its own, don't confuse this with labels. If you feel like it, leave the quotes and the colon away, and you will see the difference, because the result will not be a string, but the value of the label **Paper**, which is **&(1,0,0)**. Now change the cells below to **"Tapes:"** and **"Result:"**. This is something that is understandable to others, too.

As the last step, save your work sheet to a file: F)ile, X)DR. The native file format is XDR, so chose that. Up to now, your sheet does not have a name, so you will be prompted for one:

Save sheet to XDR file: first_step

Unless you see an error message after, your sheet is written to a file.

If you have come this far, quit (from the main menu) and you have successfully completed your first steps on using teapot.[2] Now you know cells, the difference between contents and values, you learned that labels are a good thing and you can do simple cell modifications as well as saving your work. This is enough for most applications. If the capabilities described in the next section confuse you, then it is unlikely that you need them really. Just skip that section and don't worry about it.

You may wonder what happens if you have circular dependencies, i.e. you have a cell which evaluates to its own value plus one. Well, the answer is that it depends on the order in which you create this cell. If you first give it the value 1 and after edit it to contain the expression which refers to itself plus 1, then you will find that each recalculation, like after editing other cells, will increase the value. While this may be funny, it is certainly not useful as you can not reset the cell and you have little control of its development.

What you really want is a base value and an iterative expression along with a way to control the recalculations. teapot supports this by allowing two expressions per cell. The expressions you have used so far are the ones which evaluate to the base values. Each time you edit a cell, the whole sheet will be reset, which means that all results are recalculated using the base values. After, you can clock the sheet, which is why the iterative part is also called clocked expression. A clock is an atomic operation, which means that all cell results will be recalculated in a way that the new result will only show after the entire recalculation.

An examples will demonstrate how to make use of this feature. The notation $x \rightarrow y$ means that x is the base expression and y is the clocked expression. Don't let this confuse you, as both are entered separately:

---

2. It may be a good idea to make and enjoy some tea at this point. :)

teapot does not have an → operator, but it displays the cell contents this way for increased overview. So, give the cell a base expression of **1** and a clocked expression of **@(0,0,0)+1** (using **ESC-Enter** or **Meta-Enter**) and you will see:

@(0,0,0)=1 -> @(0,0,0)+1

The sheet is currently in reset condition and the result is 1. Now clock it and you will see how the value increases.

After this introductional chapter, you should be familiar with the basic concepts in spread sheets. The next chapters explain all functions available in detail. You should read them to get an overview of the possibilities offered by teapot. Finally, we will come back to using teapot by showing some common problems and their solutions.

3. The command mode

Right after starting teapot, you are in the command mode. Many functions from the command mode are also available from menues, but using keys is faster and some things, like moving the cell cursor, are only available through keys. The command mode accepts:

Function key	ASCII key	function
Next line	^n	cursor down
Previous line	^p	cursor up
Begin	^a	cursor to column 0
End	^e	cursor to last column
	+	cursor to next layer
	–	cursor to previous layer
	<	cursor to line 0
	>	cursor to last line
	_	cursor to sheet 0
	*	cursor to last sheet
	^x <	one page left
	^x >	one page right
F10	/	main menu
F2		save menu
F3		load menu
	^x ^r	load in native format
Enter	^j, ^m	edit cell contents
	", @, *digit*	overwrite cell contents
ESC Enter	ESC ^j, ESC ^m	edit clocked cell contents
Backspace	^h	edit cell contents
	.	mark blocks (see below)
	^l	redraw screen
	^y	paste (reinsert marked) block
	^r	reset sheet
	^s	clock sheet
	ESC z	save and quit
	^x ^c	quit
Next page	^v	one page down
Previous page	ESC v	one page up

Function key	ASCII key	function
Cancel	^g, ^c	abort current action

. (dot) marks blocks: The first time it marks the beginning of a block, which is then extended by moving the cell cursor. The next time, it marks the end of the block which lets you move the cell cursor after without changing the block. The third time, it removes the block marks again.

4. The line editor

Many functions in teapot require editing a line of text, e.g. editing cell contents, typing file names and the line. Similar to the command mode, all things can be reached by control codes and most by function keys. The line editor accepts:

Function key	ASCII key	function
Previous character	^p	go to previous line
Forward character	^f	move cursor right
Begin	^a	move cursor to column 0
End	^e	move cursor to last column
Enter	^j, ^m	finish editing
	^l	redraw screen
	^t	transpose character
	^\	goto matching paren
Cancel	^g, ^c	abort editing
Backspace	^h	delete previous character
Delete	^?, ^d	delete current character
Insert		toggle insert mode

Besides the regular line editor functions as explained previously, you may use ^o to temporarily leave the editor in order to move around in the sheet if you are editing cell contents. Another ^o brings you back to the line editor. While moving around in the sheet, you can insert the value (v) or position (p) at the cursor position in the edited cell.

Aborting line editing means that you will get right back to command mode, whatever you started doing will have no effect.

5. Interactive functions

If a block of cells is marked, many interactive functions will be applied to all cells within that block. Such functions are denoted by * for this chapter.

5.1 Setting the precision

The precision only changes what is printed, teapot always uses the maximum precision for calculations. It also only affects the output, if the cell value is a floating point number. The precision of printed floating point numbers. Entering an empty precision means to set it to the default value.

5.2 Changing the cell label *

This function lets you edit the cell label of the current cell. Further it changes all occurences of it in the cell contents to the new value, unless you erased the cell label. If a block has been marked by the time you edit the cell label, all occurences of the label in contents of cells in that block will be changed.

5.3 Locking cells *

You can lock cells to protect them from accidental editing. Note that this protects you from modifying single cells. If you modify a block of cells which contains locked cells, those will be modified as well. This has been done because when using block commands, you usually know what you are doing.

5.4 Setting the column width *

The column width only affects the screen display, not the formatting of the final output with the expectation of saved formatted ASCII files. It is intended to let you make better usage of the screen for more overview. If the width is too small to display the cell value, stars will be displayed.

5.5 Copying or moving a block of cells

To copy a block of cells, mark it, then move the cell cursor to where the upper left corner of the copy should be and issue the copy command. Moving works similar, just use the move command. Of course you can mark three-dimensional blocks and copy them anywhere in the three-dimensional sheet, but doing so requires a good three-dimensional imagination to get what you want.

5.6 Filling a block of cells

To fill a block of cells, first mark a the block it should be filled with. This may be just one cell! Then move the cell cursor to where the upper left corner of the block to be filled should be and issue the fill command. You will be prompted for how often the marked block should be repeated in each dimension. For example, you may to repeat a cell 9 times below. Mark it, then move down one row. Issue the fill command and answer 1 to the number of column repetitions, 9 to rows and 1 to layers.

5.7 Clearing cells *

Clearing means to delete the cell contents and set all attributes to the default value. If you want to preserve the attributes, just edit the contents of a cell and delete them.

5.8 Insert cells *

Since work sheets can be three-dimensional, you can insert cells in all three dimensional, too. The inserted cells will be empty and their attributes have the default values. Cells will always be moved away from the front upper left corner to make room for the inserted cells. If no block is marked, you will be asked if you really only want to insert a cell of if you want to insert a whole row, line or sheet.

5.9 Delete cells *

Deleting works contrary to inserting. The deleted cells will be filled by moving neighbour cells to their positions. You will be prompted for the direction from where those cells will be taken. Deleting an entire column column-wise is done by marking the column, use the delete command and chose X direction.

5.10 Sort cells

Marked blocks of cells can be sorted after one or multiple keys, either column-wise, row-wise or depth-wise. Sorting a two dimensional block row-wise will sort lines, but if a three dimensional block is sorted row-wise, then horizontal layers will be sorted. The sort key is specified as vector which is orthogonal to the sorted elements, either in ascending or descending order. The following example illustrates the sort function. The upper left part of the screen should look like this:

0	0	1
0	1	"one"
1	2	"two"
2	3	"three"
3	4	"four"

The box shows you which block to mark. Now this block should be sorted row-wise, with the sort key being the numbers in descending order, i.e. we want the lines being numbered 4,3,2,1. Go to the block menu, then select sort. Use **R)ow**, because that is how we want to sort this block. The X position of the sort key vector is 0, because the column 0 contains the numbers. The Z position is 0, too, because those numbers are on sheet 0. Now chose **D)escending** as direction. At this point, you could add a secondary key or decide to sort the block by the keys entered so far. Use **S)ort region** to sort it. That's it, the screen should look like this now:

0	0	1
0	4	"four"
1	3	"three"
2	2	"two"
3	1	"one"

5.11 Mirror cells

Mirroring a marked block of cells can be done in three directions: Left/right, upside/down and front/back.

5.12 Saving and loading sheets

5.12.1 Name or rename a sheet

Occasionally, you may want to rename a sheet, like before making critical changes or when you load something an existing sheet to have a start for making a new one. This function allows to change the filename which is associated with a sheet.

5.12.2 Save and load in XDR format

XDR (eXternal Data Representation) is a standard invented by Sun Microsystems which defines a canonical way of storing/transporting data on external media. Its advantage is that it is widely available and that it defines a portable floating point number format. The native teapot file format uses XDR so it is portable across different machine architectures and operating systems. The advantage of this over the portable ASCII format is that due to the (usually) missing conversion calculations any floating point constants will be saved/loaded exactly without conversion errors. XDR files have no default extension.

5.12.3 Save and load in ASCII format

The ASCII file format allows easy generation/modification of saved sheets by shell scripts. Due to binary/ASCII conversion, there may be conversion errors in floating point constants. The default extension is **.asc**.

5.12.4 Save and load in CSV/GCSV format *

CSV (comma seperated value) files only contain the data, not the expressions calculating it. Many spread sheets can generate this file format and many graphics programs, like *gnuplot*(1) can read it. PlotPlus can read it if you specify **FORMAT FREE**. The field separator is a tab or comma, strings are preferably but not necessarily enclosed in double quotes and decimal numbers have a dot to mark the fractional part. The default extension is **.txt**. When loading CSV files, the sheet will not be cleared and the data will be load relative to the current cursor position. GCSV is the German variation, which uses semicolons for separating fields and a decimal comma instead of a decimal point. On load, strings without quotes and with a *0x* prefix followed by hexadecimal digits will be converted to integers.

5.12.5 Load in SC format

teapot can load simple SC sheets to convert them to teapot's native format. While loading, teapot converts all references to absolute cell positions to labels. This allows to insert and delete in such sheets without screwing the whole sheet up. teapot can not save sheets in SC format, because SC lacks many features. For now, only the most basic SC features are supported. The default extension is **.sc**.

5.12.6 Load in WK1 format

teapot can load simple WK1 sheets to convert them to teapot's native format. By default, 1-2-3 cell references are relative, so don't be surprised by a big amount of relative references in the resulting teapot sheet. For now, only the most basic 1-2-3 features are supported. The default extension is **.wk1**.

5.12.7 Save in formatted ASCII format *

The generated formatted ASCII files contain about what you see on the screen. If your sheet has more than one layer, then the various layers will be saved separated by form feeds. The default extension is **.doc**.

5.12.8 Save in tbl format *

teapot can generate *tbl*(1) table bodies in single files which are supposed to be used like this:

> **.TS**
> *options*;
> **.so** *filename*
> **.TE**

You will have to use *soelim*(1) to eliminate the **.so** requests before the tbl run. The *options*; are optional. If you use GNU roff, you will need to eliminate **.lf** requests, because this GNU roff extension confuses GNU tbl:

> **soelim** *file* **| grep -v '^\.lf'**

Alternatively, you can generate a stand-alone document, which needs no further operations to format and print. Note: If no block is marked, the whole sheet will be saved, not only the current cell. The default extension is **.tbl**.

5.12.9 Save in LaTeX format *

If you generate LaTeX 2e tables in single files, you include them in documents like this:

> **\include{***filename***}**

Alternatively, you can generate a stand-alone document, which needs no further operations to format and print. Note: If no block is marked, the whole sheet will be saved, not only the current cell. The default extension is **.tex**.

5.12.10 Save in HTML format *

You can generate html table bodies in single files which could be used in combination with server-side includes. This feature differs between the various servers, so refer to the manual for your web server for details, please.

Alternatively, you can generate a stand-alone document. Note: If no block is marked, the whole sheet will be saved, not only the current cell. The default file extension is **.html**.

5.13 Go to location

Sometimes, you directly want to go to a specific position, either to change its contents to see which cell a location expression refers to. This function lets you enter an expression, which must evaluate to a value of the type location. If so, the cursor is positioned to that location. For example, you could enter **&(10,2)** to go to cell 10,2 of the current layer or you could enter the name of a label you want to go to. Relative movements are no problem, either.

5.14 Shell

Start a sub shell. Exiting from that sub shell will bring you back into teapot.

5.15 Version information

teapot will display its version number and other entertaining things. :) If you send me email, please always write which version you are using.

6. Batch functions

Besides interactive facilities, teapot has a batch mode. Using this batch mode, shell scripts can generate output from teapot sheets. This is handy if you use *make*(1) to generate a bigger document containing tables, because you don't have to generate a tbl or LaTeX file each time you modified a sheet: make will do so. In batch mode, teapot reads batch commands from standard input. The following commands are available:

goto *location*
: Go to the specified *location*.

from *location*
: Start marking a block.

to *location*
: End marking a block.

sort-x d|a *y z* [d|a *y z* ...]
sort-y d|a *x z* [d|a *x z* ...]
sort-z d|a *x y* [d|a *x y* ...]
: Sorts the marked block as described in section *5.10*, column-wise, row-wise or depth-wise, respectivly. "d" or "a" specify the sort order to be descending or ascending. *x*, *y* and *z* specify the position of the sort key **relative** to the first cell of the marked block. Up to eight sort keys can be specified.
This example reproduces the result from *5.10*:
echo -e "from &(1,1,0)\n to &(2,4,0)\n sort-y d 0 0\n save-csv result_num.txt" \
| teapot -b doc/unsorted

save-tbl *file*
save-csv *file*
save-latex *file*
save-context *file*
save-html *file*
: Save the marked block in the specified format as *file*.

load-csv *file*
load-gcsv *file*
: Load *file* in the specified format to the last **goto** location. This is the same functionality as the interactive load described in subsection *5.12.4*.

7. Cells

In this chapter, **bold** marks literals which have to be entered exactly as shown. *Italic* mark place holders, they are not to be taken literally.

7.1 Attributes

Each cell consists of a base/reset expression, a clocked expression, the result of one of these expressions (depending on the cell being reset or clocked) and several attributes:

- A cell label, which is useful because it avoids to directly address cells by their position. A cell label must be different from function names.
- The cell adjustment, which determines if the cell value is printed left adjusted, right adjusted or centered.
- The precision for the output of floating point values. The default is 2 digits after the dot.

- If floating point numbers should be printed in scientific notation (0.123e1) or as decimal number (1.23). It only affects the output, if the cell value is a floating point number.
- If the cell is shadowed by its left neighbour. This means that the left neighbour cell additionally uses the room of the shadowed cell.
- If the cell is locked which prevents to accidentally edit or clear it. Note that block operations override this attribute, because when you deal with blocks, you usually know what you are doing.
- If special characters for e.g. roff and LaTeX should be quoted (default) or not. Not quoting them allows special effects (if you know roff or LaTeX), but is of course not portable.

7.2 Data Types

In teapot, each value has an associated data type. The following data types exist:

- empty. Empty cells have this as value.
- string. A string is a sequence of characters enclosed by double quotes: **"This is a string"**. A double quote can be part of the string, if it is quoted using a backslash: **"\""**. If you want the backslash to appear in the output instead of quoting the next character, use it to quote itself: **"\\"**. A single minus character on the screen will be output as a minus. If you want a hyphen character in the output, then use two minus characters for that.
- floating point. Floating point values are inexact, their precision and range depends on the implementation of the C type double on your system. An example is: **42.0**
- integer. Integer values are exact, their range depends on the C type long on your system. An example is: **42**
- location. Cell labels and the **&()** function have this type, but there are no location constant literals.
- error. Syntactical, semantical (type mismatch) errors cause this value, as well as division by 0 and the function **error()**. An error always has an assigned error message. Functions and operators, when applied to a value of the type error, evaluate to just that value. That way, the first error which was found deep inside a complicated expression will be showed.

8. Operators

Unlike other spread sheets, the operators in teapot check the type of the values they are applied to, which means the try to add a string to a floating point number will result in an type error. The following operators are available, listed in ascending precendence:

$x<y$ evaluates to 1 if *x* is less than *y*. If *x* or *y* are empty, they are considered to be 0 if the other is an integer number, 0.0 if it is a floating point number and the empty string if it is a string.

$x<=y$ evaluates to 1 if *x* is less than or equal to *y*. Empty values are dealt with as described above.

$x>=y$ evaluates to 1 if *x* is greater than or equal to *y*. Empty values are dealt with as described above.

$x>y$ evaluates to 1 if *x* is greater than *y*. Empty values are dealt with as described above.

$x==y$ evaluates to 1 if *x* is equal to *y*. Empty values are dealt with as described above.

$x\tilde{}=y$ evaluates to 1 if the floating point value *x* is almost to the floating point value *y*. Empty values are dealt with as described above. Almost equal means, the numbers are at most neighbours.

$x!=y$ evaluates to 1 if *x* is not equal to *y*. Empty values are dealt with as described above.

$x+y$ evaluates to the sum if *x* and *y* are numbers. If *x* and *y* are strings, the result is the concatenated string. If *x* or *y* is empty, the result is the other element. There is no dedicated logical or operation, so use **+** for that.

x−*y* evaluates to the difference if *x* and *y* are numbers. If *x* is empty, the result is **−***y*. If *y* is empty, the result is *x*.

*x****y* evaluates to the product if *x* and *y* are numbers. If *x* or *y* is empty, the result is 0. There is no dedicated logical and operation, so use * for that.

*x***/***y* evaluates to the quotient if *x* and *y* are numbers. If *x* is empty, the result is 0. If *y* is empty, the result is the error ‘‘division by 0’’.

*x***%***y* evaluates to the remainder of the division if *x* and *y* are numbers. If *x* is empty, the result is 0. If *y* is empty, the result is the error ‘‘modulo 0’’.

x^*y* evaluates to *x* to the power of *y*. If *x* or *y* are empty, they will be considered to be the integer value 0.

−*x* evaluates to **−***x* if *x* is a number. If *x* is empty, the result will be empty, too.

(*expression***)**
evaluates to the expression.

*function***(***argument***,**...**)**
evaluates to the value of the function applied to the values resulting from evaluating the argument expressions.

9. Functions

This section documents all available functions in alphabetical order. The functions are given in a C-like notation, so use **@(0,0,0)** instead of **@(integer 0, integer 0, integer 0)**. If no type is given for the result of a function, it means the result type depends on the arguments. Brackets mark optional arguments.

@([integer *x*][**,**[integer *y*][**,**[integer *z*]]]**)**
@(location *l***)**
returns the value of the cell at position *x*, *y*, *z*. If any of *x*, *y* or *z* is omitted, the coordinate of the cell is used.

location **&(**[integer *x*][**,** [integer *y*][**,** [integer *z*]]]**)**
returns a pointer to the cell at location *x*, *y*, *z*. If *z* is omitted, the *z* position of the cell is used. If *y* is missing as well, the *y* position (row) of the cell is used.

string **$(**string *env***)**
evaluates to the contents of the specified environment variable. If the variable does not exist, then an empty string will be returned.

float **abs(**float *x***)**
int **abs(**int *x***)**
evaluates to the absolute value of *x*.

float **acos(**double|int *x***)**
evaluates to the arc cosine of *x*, where *x* is given in radians.

float **arcosh(**double|int *x***)**
evaluates to the arc hyperbolic cosine of *x*, where *x* is given in radians.

float **arsinh(**double|int *x***)**
evaluates to the arc hyperbolic sine of *x*, where *x* is given in radians.

float **artanh(**double|int *x***)**
evaluates to the arc hyperbolic tangent of *x*, where *x* is given in radians.

float **asin(**double|int *x***)**
evaluates to the arc sine of *x*, where *x* is given in radians.

float **atan**(double|int *x*)
evaluates to the arc tangent of *x*, where *x* is given in radians.

clock(integer *condition*,[location[,location])
conditionally clocks the specified cell if the condition is not 0. If two locations are given, all cells in that range will be clocked. The return value of clock is empty.

float **cos**(double|int *x*)
evaluates to the cosine of *x*, where *x* is given in radians.

float **cosh**(double|int *x*)
evaluates to the hyperbolic cosine of *x*, where *x* is given in radians.

float **deg2rad**(double|int *x*)
evaluates to the degrees that are equivalent to *x* radians.

float **e**()
evaluates to the Euler constant *e*.

error **error**(string *message*)
evaluates to an error with the specified message.

eval(location)
evaluates to the value of the expression in the cell at the given *location*, but evaluated in the context of the cell using **eval()**. This function may not be used nested any deeper than 32 times.

float **float**(string *s*)
converts the given string into a floating point number.

float **frac**(float *x*)
evaluates to the fractional part of *x*.

int **int**(float *x*[,integer *neg*, integer *pos*])
converts *x* to an integer value by cutting off the fractional part. If given, the value of *neg* and *pos* determines how negative and non-negative numbers will be converted:

neg,pos		result
<−1	→	next smaller integer value
−1	→	round downward
0	→	cut fractional part off (default)
1	→	round upward
>1	→	next larger integer value

int **int**(string *s*)
converts *s* to an integer number.

string **len**(string *s*)
evaluates to the length of *s*.

float **log**(double|int *x*[,double|int *y*)
evaluates to the logarithm of *x*. If *y* is specified, the result will be the natural logarithm, otherwise it will be the logarithm to the base of *y*.

location **max**(location *l1*, location *l2*)
evaluates to the maximum in the same way **min** does for the minimum.

location **min**(location *l1*, location *l2*)
evaluates to the location of the minimum of all values in the block marked by the corners pointed to by *l1* and *l2*. Note that the empty cell is equal to 0, 0.0 and "", so if the first minimum is an empty cell, the result will be a pointer to this cell, too. If you are not interested in the location of the minimum but the value itself, use @(**min**(*l1*, *l2*)).

int **n**(location *l1***,** location *l2***)**
evaluates to the number of non-empty cells in the block marked by the corners pointed to by *l1* and *l2*.

poly(float|integer *x***,** float|integer c_n**[,** ...])
evaluates the polynome $c_n*x^n+...+c_0*x^0$.

float **rad2deg**(double|int *x*)
evaluates to the radians that are equivalent to *x* degrees.

float **rnd()**
evaluates to a pseudo-random number between 0.0 and 1.0.

float **sin**(double|int *x*)
evaluates to the sine of *x*, where *x* is given in radians.

float **sinh(**double|int *x***)**
evaluates to the hyperbolic sine of *x*, where *x* is given in radians.

string **strftime**(string *f* **[,** integer *t*])
evaluates to the time *t* formatted according to the format specified in *f*. Times in *t* are counted in seconds since epoch (1970-1-1 0:00). If *t* is empty or 0, the actual time is used. For the format specifications consult the man page of your c library, strftime (3). Example: *@(now)=int(strftime("%s"))* sets the field with label now to the actual time.

string **string**(location *l***)**
string **string**(integer *x***)**
string **string(**float *x***[,** [integer *precision*][**,** *scientific*]])
evaluates to a string containing the current value of the given cell at location *l*, or to the numeric value *x* with the given *precision*. The *scientific* flag determines if decimal (0) or scientific (unequal 0) representation is used.

int **strptime(**string *f* **,** string *datetime*)
evaluates to the seconds since epoch (1970-1-1 0:00) of the *datetime* string, parsed according to the format specified in *f*. For the format specifications consult the man page of your c library, strptime (3).

string **substr(**string *s,*integer *x,*integer *y*)
evaluates to the substring of *s* between *x* and *y*, which start at 0.

sum(location *l1***,** location *l2***)**
evaluates to the sum of all values in the block marked by the corners pointed to by *l1* and *l2*.

float **tan(**double|int *x***)**
evaluates to the tangent of *x*, where *x* is given in radians.

float **tanh(**double|int *x***)**
evaluates to the hyperbolic tangent of *x*, where *x* is given in radians.

int **x(**[location *l*]**)**
int **y(**[location *l*]**)**
int **z(**[location *l*]**)**
evaluate to the *x*, *y* and *z* position of the given location, of the currently updated cell if none is given. These functions are usually used in combination with the @ function for relative relations to other cells.

10. Problems and solutions

10.1 Why is 1.0 unequal 1.0?

If your machine uses binary floating point arithmetic, and chances are that it does, you may eventually find yourself in the following situation:

0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1

You expect to see 1.0 as result, and indeed that is what you get. Now you compare this result to the constant 1.0, but surprisingly for many users, the result is 0. Appearantly, 1.0 is unequal 1.0 for teapot.

This is not a bug in teapot, in fact it is not a bug at all. The problem is, that 0.1 (1.0/10.0) does not have an exact representation in binary floating point arithmetic, similar to how 1.0/3.0 does not have an exact representation in decimal arithmetic (or binary, for that matter). As such, a value very close to 0.1 is used, but when displaying it, it will be rounded to 0.1. The result is obvious, adding this number which is a little different from 0.1 ten times leads to a result very close to but not quite 1.0. Since it is so close, displaying it rounded to only a few digits precision shows 1.0.

To solve the comparison problem, teapot has the operator **~=**, which compares two floating point values apart from the last significant bit. Use this operator to compare the two values from above and the result will be 1, meaning they are about equal. Don't assume that a number which can be expressed with a finite number of decimal digits will be represented exactly in binary floating point arithmetic.

10.2 How do I hide intermediate results?

If you used flat, two-dimensional spread sheets before, you are probably used to hidden cells which contain intermediate results, global constants, scratch areas and the like. teapot has no way to hide cells, but you have three dimensions. Just use one or more layers for such cells and give each cell a label in order to reference and find it easily.

10.3 Why is there no conditional evaluation?

There is no special operator or function for conditional evaluation. I could add one easily, but then next someone would ask for loops and someone else for user-defined functions, variables and so on. If you need a programming language, you know where to find it.

But don't worry. :) The answer is, that conditional evaluation comes for free with teapot's orthogonal cell addressing. As an example, depending on the cell labelled **X** being negative or not, you want the result to be the string **"BAD** or **"GOOD"**. This is the solution:

eval(&((@(X)>=0)+x(BAD),y(BAD),z(BAD)))

The cell labelled **BAD** contains the string **"BAD"**, its right neighbour contains the string **"GOOD"**. If you have nested conditions, you could weight them with 1, 2, 4 and so on to address a bigger range of cells. Alternatively, you could make use of all three dimensions for nested conditions.

11. Expression grammar

digit ::= **0** | .. | **9**

hex_digit ::= **0** | .. | **9** | **a** | .. | **f**

octal_digit ::= **0** | .. | **7**

decimal_integer ::= *digit* { *digit* }

hex_integer ::= **0x** *hex_digit* { *hexdigit* }

octal_integer ::= **0** *octal_digit* { *octdigit* }

integer ::= *decimal_integer* | *hex_integer* | *octal_integer*

float ::= *digit* { *digit* } [**.**] { *digit* } [**e** | **E** [**+** | **−**] *digit* { *digit* }]

quoted_character ::= **** *any_character*

character ::= *any_character* | *quoted_character*

string ::= **"** { *character* } **"**

identifier_character ::= **_** | **@** | **&** | **.** | **$** | *alpha_character*

identifier ::= *identifier_character* { *identifier_character* | *digit* }

function ::= *identifier* **(** [*term*] { **,** [*term*] } **)**

label ::= *identifier*
parenterm ::= **(** *term* **)**
negterm ::= **–** *primary*
primary ::= *function* | *label* | *parenterm* | *negterm*
powterm ::= *primary* { **^** *primary* }
mathterm ::= *powterm* { **/** | ***** | **%** *powterm* }
factor ::= *mathterm* { **+** | **–** *mathterm* }
term ::= *factor* { **<** | **<=** | **>=** | **>** | **==** | **!=** *factor* }