Tablicious for Octave

version 0.1.0-SNAPSHOT, April 2019

Andrew Janke

Short Contents

Table of Contents

1 Introduction

This is the manual for the Tablicious package version 0.1.0-SNAPSHOT for GNU Octave.

Tablicious provides Matlab-compatible tabular data support for GNU Octave. This includes a `table` class with support for filtering and join operations, Missing Data support, and `string` and `categorical` data types.

This document is a work in progress. You are invited to help improve it and submit patches.

Tablicious was written by Andrew Janke `<floss@apjanke.net>`. Support can be found on the Tablicious project GitHub page (`https://github.com/apjanke/octave-tablicious`).

2 Getting Started

The easiest way to obtain Tablicious is by using Octave's `pkg` package manager. To install the development prerelease of Tablicious, run this in Octave:

```
     pkg install https://github.com/apjanke/octave-tablicious/releases/download/v0.1.0-SNAP
```

(Check the releases page at `https://github.com/apjanke/octave-tablicious/releases` to find out what the actual latest release number is.)

For development, you can obtain the source code for Tablicious from the project repo on GitHub at `https://github.com/apjanke/octave-tablicious`. Make a local clone of the repo. Then add the `inst` directory in the repo to your Octave path.

3 Table Representation

Tablicious provides the `table` class for representing tabular data.

3.1 `table` Class

A `table` is an array object that represents a tabular data structure. It holds multiple named “variables”, each of which is a column vector, or a 2-D matrix whose rows are read as records.

4 Missing Functionality

Tablicious is based on Matlab's table API and supports most of its major functionality. But not all of it is implemented yet. The missing parts are currently:

- `timetable`
- Moving window methods in `fillmissing`
- `summary()` for `table` and `categorical`
- Assignment to table variables using `.`-indexing
- File I/O like `readtable()` and `writetable()`

It is the author's hope that all these will be implemented some day.

5 Function Reference

5.1 Functions by Category

5.1.1 Tables

5.1.2 Data Types

5.1.3 Missing Data

5.1.4 Miscellaneous

5.2 Functions Alphabetically

5.2.1 array2table

```
out = array2table (c)                                          [Function]
out = array2table (..., 'VariableNames', VariableNames)        [Function]
out = array2table (..., 'RowNames', RowNames)                  [Function]
```

Convert an array to a table.

Converts an array to a table, with columns in the array becoming variables in the output table. This is typically used on numeric arrays, but it can be applied to any type of array.

You may not want to use this on cell arrays, though, because you will end up with a table that has all its variables of type cell. If you use `cell2table` instead, columns of the cell array which can be condensed into primitive arrays will be. With `array2table`, they won't be.

See also: Section 5.2.3 [cell2table], page 6, Section 5.2.19 [table], page 8, Section 5.2.18 [struct2table], page 8,

5.2.2 categorical

`categorical` [Class]

Categorical variable array.

A `categorical` array represents an array of values of a categorical variable. Each `categorical` array stores the element values along with a list of the categories, and indicators of whether the categories are ordinal (that is, they have a meaningful mathematical ordering), and whether the set of categories is protected (preventing new categories from being added to the array).

In addition to the categories defined in the array, a categorical array may have elements of `"undefined"` value. This is not considered a category; rather, it is the absence of any known value. It is analagous to a `NaN` value.

This class is not fully implemented yet. Missing stuff: - gt, ge, lt, le - Ordinal support in general - countcats - summary

`uint16 code` [Instance Variable of `categorical`]

The numeric codes of the array element values. These are indexes into the `cats` category list.

This is a planar property.

`logical tfMissing` [Instance Variable of `categorical`]

A logical mask indicating whether each element of the array is missing (that is, undefined).

This is a planar property.

`cellstr cats` [Instance Variable of `categorical`]

The names of the categories in this array. This is the list into which the `code` values are indexes.

`scalar_logical isOrdinal` [Instance Variable of `categorical`]

A scalar logical indicating whether the categories in this array have an ordinal relationship.

5.2.3 cell2table

out = `cell2table (`*c*`)` [Function]
out = `cell2table (..., 'VariableNames', `*VariableNames*`)` [Function]

out = `cell2table (..., 'RowNames',` *RowNames*`)` [Function]

Convert a cell array to a table.

Converts a 2-dimensional cell matrix into a table. Each column in the input *c* becomes a variable in *out*. For columns that contain all scalar values of `cat`-compatible types, they are "popped out" of their cells and condensed into a homogeneous array of the contained type.

See also: Section 5.2.1 [array2table], page 5, Section 5.2.19 [table], page 8, Section 5.2.18 [struct2table], page 8,

5.2.4 colvecfun

Not documented

5.2.5 contains

Not documented

5.2.6 discretize

Not documented

5.2.7 dispstrs

Not documented

5.2.8 endsWith

Not documented

5.2.9 fillmissing

Not documented

5.2.10 ismissing

Not documented

5.2.11 isnannish

Not documented

5.2.12 missing

Not documented

5.2.13 pp

Not documented

5.2.14 rmmissing

Not documented

5.2.15 standardizeMissing

Not documented

5.2.16 startsWith

Not documented

5.2.17 string

Not documented

5.2.18 struct2table

Not documented

5.2.19 table

`table` [Class]

Tabular data array containing multiple columnar variables.

A `table` is a tabular data structure that collects multiple parallel named variables. Each variable is treated like a column. (Possibly a multi-columned column, if that makes sense.) The types of variables may be heterogeneous.

A table object is like an SQL table or resultset, or a relation, or a DataFrame in R or Pandas.

A table is an array in itself: its size is *nrows*-by-*nvariables*, and you can index along the rows and variables by indexing into the table along dimensions 1 and 2.

`cellstr VariableNames` [Instance Variable of `table`]

The names of the variables in the table, as a cellstr row vector.

`cell VariableValues` [Instance Variable of `table`]

A cell vector containing the values for each of the variables. `VariableValues(i)` corresponds to `VariableNames(i)`.

`cellstr RowNames` [Instance Variable of `table`]

An optional list of row names that identify each row in the table. This is a cellstr column vector, if present.

5.2.19.1 table.table

`obj` `= table ()` [Constructor]

Constructs a new empty (0 rows by 0 variables) table.

`obj` `= table (`*`var1`*`,` *`var2`*`, ...,` *`varN`*`)` [Constructor]

Constructs a new table from the given variables. The variables passed as inputs to this constructor become the variables of the table. Their names are automatically detected from the input variable names that you used.

`obj` `= table ('Size',` *`sz`*`, 'VariableTypes',` *`varTypes`*`)` [Constructor]

Constructs a new table of the given size, and with the given variable types. The variables will contain the default value for elements of that type.

`obj` `= table (..., 'VariableNames',` *`varNames`*`)` [Constructor]
`obj` `= table (..., 'RowNames',` *`rowNames`*`)` [Constructor]

Specifies the variable names or row names to use in the constructed table. Overrides the implicit names garnered from the input variable names.

5.2.19.2 table.summary

`summary (`*`obj`*`)` [Method]
`s`` = summary (`*`obj`*`)` [Method]

Summary of table's data.

Displays or returns a summary of data in the input table. This will contain some statistical information on each of its variables.

This method is not implemented yet.

5.2.19.3 table.prettyprint

`prettyprint (`*`obj`*`)` [Method]

Display table's values in tabular format. This prints the contents of the table in human-readable, tabular form.

Variables which contain objects are displayed using the strings returned by their `dispstrs` method, if they define one.

5.2.19.4 table.table2cell

`c`` = table2cell (`*`obj`*`)` [Method]

Converts table to a cell array. Each variable in *obj* becomes one or more columns in the output, depending on how many columns that variable has.

Returns a cell array with the same number of rows as *obj*, and with as many or more columns as *obj* has variables.

5.2.19.5 table.table2struct

`s`` = table2struct (`*`obj`*`)` [Method]
`s`` = table2struct (..., 'ToScalar', `*`trueOrFalse`*`)` [Method]

Converts *obj* to a scalar structure or structure array.

Row names are not included in the output struct. To include them, you must add them manually: s = table2struct (tbl, 'ToScalar', true); s.RowNames = tbl.Properties.RowNames;

Returns a scalar struct or struct array, depending on the value of the `ToScalar` option.

5.2.19.6 table.table2array

`s`` = table2struct (`*`obj`*`)` [Method]

Converts *obj* to a homogeneous array.

5.2.19.7 table.varnames

`out`` = varnames (`*`obj`*`)` [Method]

Get variable names for a table.

Returns cellstr.

5.2.19.8 table.istable

`tf`` = istable (`*`obj`*`)` [Method]

True if input is a table.

5.2.19.9 table.size

sz = `size` (*obj*) [Method]

Gets the size of a table.

For tables, the size is [number-of-rows x number-of-variables]. This is the same as `[height(obj), width(obj)]`.

5.2.19.10 table.length

out = `length` (*obj*) [Method]

Length along longest dimension

Use of `length` is not recommended. Use `numel` or `size` instead.

5.2.19.11 table.ndims

out = `ndims` (*obj*) [Method]

Number of dimensions

For tables, `ndims(obj)` is always 2.

5.2.19.12 table.squeeze

obj = `squeeze` (*obj*) [Method]

Remove singleton dimensions.

For tables, this is always a no-op that returns the input unmodified, because tables always have exactly 2 dimensions.

5.2.19.13 table.sizeof

out = `sizeof` (*obj*) [Method]

Approximate size of array in bytes. For tables, this returns the sume of `sizeof` for all of its variables' arrays, plus the size of the VariableNames and any other metadata stored in *obj*.

This is currently unimplemented.

5.2.19.14 table.height

out = `height` (*obj*) [Method]

Number of rows in table.

5.2.19.15 table.rows

out = `rows` (*obj*) [Method]

Number of rows in table.

5.2.19.16 table.width

out = `width` (*obj*) [Method]

Number of variables in table.

Note that this is not the sum of the number of columns in each variable. It is just the number of variables.

5.2.19.17 table.columns

out = `columns` (*obj*) [Method]

Number of variables in table.

Note that this is not the sum of the number of columns in each variable. It is just the number of variables.

5.2.19.18 table.numel

out = `numel` (*obj*) [Method]

Total number of elements in table.

This is the total number of elements in this table. This is calculated as the sum of numel for each variable.

NOTE: Those semantics may be wrong. This may actually need to be defined as `height(obj) * width(obj)`. The behavior of `numel` may change in the future.

5.2.19.19 table.isempty

out = `isempty` (*obj*) [Method]

Test whether array is empty.

For tables, `isempty` is true if the number of rows is 0 or the number of variables is 0.

5.2.19.20 table.ismatrix

out = `ismatrix` (*obj*) [Method]

Test whether array is a matrix.

For tables, `ismatrix` is always true, by definition.

5.2.19.21 table.isrow

out = `isrow` (*obj*) [Method]

Test whether array is a row vector.

5.2.19.22 table.iscol

out = `iscol` (*obj*) [Method]

Test whether array is a column vector.

For tables, `iscol` is true if the input has a single variable. The number of columns within that variable does not matter.

5.2.19.23 table.isvector

out = `isvector` (*obj*) [Method]

Test whether array is a vector.

5.2.19.24 table.isscalar

out = `isscalar` (*obj*) [Method]

Test whether array is scalar.

5.2.19.25 table.hasrownames

out = `hasrownames` (*obj*) [Method]

True if this table has row names defined.

5.2.19.26 table.vertcat

out = `vertcat` (*varargin*) [Method]

Vertical concatenation.

Combines tables by vertically concatenating them.

Inputs that are not tables are automatically converted to tables by calling table() on them.

The inputs must have the same number and names of variables, and their variable value types and sizes must be cat-compatible.

5.2.19.27 table.horzcat

out = `horzcat` (*varargin*) [Method]

Horizontal concatenation.

Combines tables by horizontally concatenating them. Inputs that are not tables are automatically converted to tables by calling table() on them. Inputs must have all distinct variable names.

Output has the same RowNames as `varargin{1}`. The variable names and values are the result of the concatenation of the variable names and values lists from the inputs.

5.2.19.28 table.repmat

out = `repmat` (*obj*, *sz*) [Method]

Replicate matrix.

Repmats a table by repmatting each of its variables vertically.

For tables, repmatting is only supported along dimension 1. That is, the values of sz(2:end) must all be exactly 1.

Returns a new table with the same variable names and types as tbl, but with a possibly different row count.

5.2.19.29 table.setVariableNames

out = `setVariableNames` (*obj*, *names*) [Method]

Set variable names.

Sets the `VariableNames` for this table to a new list of names.

names is a cellstr vector. It must have the same number of elements as the number of variables in *obj*.

5.2.19.30 table.setRowNames

out = `setRowNames` (*obj*, *names*) [Method]

Set row names.

Sets the row names on *obj* to *names*.

names is a cellstr column vector, with the same number of rows as *obj* has.

5.2.19.31 table.resolveVarRef

[*ixVar*, *varNames*] = `resolveVarREf` (*obj*, *varRef*) [Method]

Resolve a variable reference against this table.

A *varRef* is a numeric or char/cellstr indicator of which variables within *obj* are being referenced.

Returns: *ixVar* - the indexes of the variables in *obj* *varNames* - a cellstr of the names of the variables in *obj*

Raises an error if any of the specified variables could not be resolved.

5.2.19.32 table.subsetRows

out = `subsetRows` (*obj*, *ixRows*) [Method]

Subset table by rows.

Subsets this table by rows.

ixRows may be a numeric or logical index into the rows of *obj*.

5.2.19.33 table.subsetvars

out = `subsetvars` (*obj*, *ixVars*) [Method]

Subset table by variables.

Subsets table *obj* by subsetting it along its variables.

ixVars may be: - a numeric index vector - a logical index vector - `":"` - a cellstr vector of variable names

The resulting table will have its variables reordered to match ixVars.

5.2.19.34 table.removevars

out = `removevars` (*obj*, *vars*) [Method]

Remove variables from table.

Deletes the variables specified by *vars* from *obj*.

vars may be a char, cellstr, numeric index vector, or logical index vector.

5.2.19.35 table.movevars

out = `movevars` (*obj*, *vars*, *relLocation*, *location*) [Method]

Move around variables in a table.

vars is a list of variables to move, specified by name or index.

relLocation is `'Before'` or `'After'`.

location indicates a single variable to use as the target location, specified by name or index. If it is specified by index, it is the index into the list of *unmoved* variables from *obj*, not the original full list of variables in *obj*.

Returns a table with the same variables as *obj*, but in a different order.

5.2.19.36 table.setvar

out = `setvar` (*obj*, *varRef*, *value*) [Method]

Set value for a variable in table.

This sets (replaces) the value for a variable that already exists in *obj*. It cannot be used to add a new variable.

5.2.19.37 table.convertvars

out = `convertvars` (*obj*, *vars*, *dataType*) [Method]

Convert variables to specified data type.

Converts the variables in *obj* specified by *vars* to the specified data type.

vars is a cellstr or numeric vector specifying which variables to convert.

dataType specifies the data type to convert those variables to. It is either a char holding the name of the data type, or a function handle which will perform the conversion. If it is the name of the data type, there must either be a one-arg constructor of that type which accepts the specified variables' current types as input, or a conversion method of that name defined on the specified variables' current type.

Returns a table with the same variable names as *obj*, but with converted types.

5.2.19.38 table.head

out = `head` (*obj*) [Method]
out = `head` (*obj*, *k*) [Method]

Get first K rows of table.

Returns the first *k* rows of *obj*, as a table.

k defaults to 8.

If there are less than *k* rows in *obj*, returns all rows.

5.2.19.39 table.tail

out = `tail` (*obj*) [Method]
out = `tail` (*obj*, *k*) [Method]

Get last K rows of table.

Returns the last *k* rows of *obj*, as a table.

k defaults to 8.

If there are less than *k* rows in *obj*, returns all rows.

5.2.19.40 table.join

`[C, ib] = join (A, B)` [Method]
`[C, ib] = join (A, B, ...)` [Method]

Combine two tables by rows using key variables, in a restricted form.

This is not a "real" relational join operation. It has the restrictions that: 1) The key values in B must be unique. 2) Every key value in A must map to a key value in B. These are restrictions inherited from the Matlab definition of table.join.

You probably don't want to use this method. You probably want to use innerjoin or outerjoin instead.

See also: Section 5.2.19.41 [table.innerjoin], page 15, Section 5.2.19.42 [table.outerjoin], page 15,

5.2.19.41 table.innerjoin

`[out, ixa, ixb] = innerjoin (A, B)` [Method]
`[...] = innerjoin (A, B, ...)` [Method]

Combine two tables by rows using key variables.

Computes the relational inner join between two tables. “Inner” means that only rows which had matching rows in the other input are kept in the output.

TODO: Document options.

Returns: *out* - A table that is the result of joining A and B *ix* - Indexes into A for each row in out *ixb* - Indexes into B for each row in out

5.2.19.42 table.outerjoin

`[out, ixa, ixb] = outerjoin (A, B)` [Method]
`[...] = outerjoin (A, B, ...)` [Method]

Combine two tables by rows using key variables, retaining unmatched rows.

Computes the relational outer join of tables A and B. This is like a regular join, but also includes rows in each input which did not have matching rows in the other input; the columns from the missing side are filled in with placeholder values.

TODO: Document options.

Returns: *out* - A table that is the result of the outer join of A and B *ixa* - indexes into A for each row in out *ixb* - indexes into B for each row in out

5.2.19.43 table.outerfillvals

`out = outerfillvals (obj)` [Method]

Get fill values for outer join.

Returns a table with the same variables as this, but containing only a single row whose variable values are the values to use as fill values when doing an outer join.

5.2.19.44 table.semijoin

`[outA, ixA, outB, ixB] = semijoin (A, B)` [Method]

Natural semijoin.

Computes the natural semijoin of tables A and B. The semi-join of tables A and B is the set of all rows in A which have matching rows in B, based on comparing the values of variables with the same names.

This method also computes the semijoin of B and A, for convenience.

Returns: *outA* - all the rows in A with matching row(s) in B *ixA* - the row indexes into A which produced *outA* *outB* - all the rows in B with matching row(s) in A *ixB* - the row indexes into B which produced *outB*

5.2.19.45 table.antijoin

`[outA, ixA, outB, ixB] = antijoin (A, B)` [Method]

Natural antijoin (AKA "semidifference").

Computes the anti-join of A and B. The anti-join is defined as all the rows from one input which do not have matching rows in the other input.

Returns: *outA* - all the rows in A with no matching row in B *ixA* - the row indexes into A which produced *outA* *outB* - all the rows in B with no matching row in A *ixB* - the row indexes into B which produced *outB*

5.2.19.46 table.cartesian

`[out, ixs] = cartesian (A, B)` [Method]

Cartesian product of two tables.

Computes the Cartesian product of two tables. The Cartesian product is each row in A combined with each row in B.

Due to the definition and structural constraints of table, the two inputs must have no variable names in common. It is an error if they do.

The Cartesian product is seldom used in practice. If you find yourself calling this method, you should step back and re-evaluate what you are doing, asking yourself if that is really what you want to happen. If nothing else, writing a function that calls cartesian() is usually much less efficient than alternate ways of arriving at the same result.

This implementation does not remove duplicate values. TODO: Determine whether this duplicate-removing behavior is correct.

The ordering of the rows in the output is not specified, and may be implementation-dependent. TODO: Determine if we can lock this behavior down to a fixed, defined ordering, without killing performance.

5.2.19.47 table.groupby

`[out] = groupby (obj, groupvars, aggcalcs)` [Method]

Find groups in table data and apply functions to variables within groups.

This works like an SQL `"SELECT ... GROUP BY ..."` statement.

groupvars (cellstr, numeric) is a list of the grouping variables, identified by name or index.

aggcalcs is a specification of the aggregate calculations to perform on them, in the form `{`*out_var*`, `*fcn*`, `*in_vars*`; ...}`, where: *out_var* (char) is the name of the output variable *fcn* (function handle) is the function to apply to produce it *in_vars* (cellstr) is a list of the input variables to pass to fcn

Returns a table.

5.2.19.48 table.grpstats

`[`*out*`] = grpstats (`*obj*, *groupvar*`)` [Method]
`[`*out*`] = grpstats (..., 'DataVars', `*DataVars*`)` [Method]

Statistics by group.

See also: Section 5.2.19.47 [table.groupby], page 16.

5.2.19.49 table.union

`[`*C*`, `*ia*`, `*ib*`] = union (`*A*, *B*`)` [Method]

Set union.

Computes the union of two tables. The union is defined to be the unique row values which are present in either of the two input tables.

Returns: *C* - A table containing all the unique row values present in A or B. *ia* - Row indexes into A of the rows from A included in C. *ib* - Row indexes into B of the rows from B included in C.

5.2.19.50 table.intersect

`[`*C*`, `*ia*`, `*ib*`] = intersect (`*A*, *B*`)` [Method]

Set intersection.

Computes the intersection of two tables. The intersection is defined to be the unique row values which are present in both of the two input tables.

Returns: *C* - A table containing all the unique row values present in both A and B. *ia* - Row indexes into A of the rows from A included in C. *ib* - Row indexes into B of the rows from B included in C.

5.2.19.51 table.setxor

`[`*C*`, `*ia*`, `*ib*`] = setxor (`*A*, *B*`)` [Method]

Set exclusive OR.

Computes the setwise exclusive OR of two tables. The set XOR is defined to be the unique row values which are present in one or the other of the two input tables, but not in both.

Returns: *C* - A table containing all the unique row values in the set XOR of A and B. *ia* - Row indexes into A of the rows from A included in C. *ib* - Row indexes into B of the rows from B included in C.

5.2.19.52 table.setdiff

`[C, ia] = setdiff (A, B)` [Method]

Set difference.

Computes the set difference of two tables. The set difference is defined to be the unique row values which are present in table A that are not in table B.

Returns: *C* - A table containing the unique row values in A that were not in B. *ia* - Row indexes into A of the rows from A included in C.

5.2.19.53 table.ismember

`[tf, loc] = ismember (A, B)` [Method]

Set membership.

Finds rows in A that are members of B.

Returns: *tf* - A logical vector indicating whether each A(i,:) was present in B. *loc* - Indexes into B of rows that were found.

5.2.19.54 table.ismissing

`out = ismissing (obj)` [Method]
`out = ismissing (obj, indicator)` [Method]

Find missing values.

Finds missing values in *obj*'s variables.

If indicator is not supplied, uses the standard missing values for each variable's data type. If indicator is supplied, the same indicator list is applied across all variables.

All variables in this must be vectors. (This is due to the requirement that `size(out) == size(obj)`.)

Returns a logical array the same size as *obj*.

5.2.19.55 table.rmmissing

`[out, tf] = rmmissing (obj)` [Method]
`[out, tf] = rmmissing (obj, indicator)` [Method]
`[out, tf] = rmmissing (..., 'DataVariables', vars)` [Method]
`[out, tf] = rmmissing (..., 'MinNumMissing', minNumMissing)` [Method]

Remove rows with missing values.

Removes the rows from *obj* that have missing values.

If the 'DataVariables' option is given, only the data in the specified variables is considered.

Returns: *out* - A table the same as *obj*, but with rows with missing values removed. *tf* - A logical index vector indicating which rows were removed.

5.2.19.56 table.standardizeMissing

`out = standardizeMissing (obj, indicator)` [Method]
`out = standardizeMissing (..., 'DataVariables', vars)` [Method]

Insert standard missing values.

Standardizes missing values in variable data.

If the *DataVariables* option is supplied, only the indicated variables are standardized.

indicator is passed along to `standardizeMissing` when it is called on each of the data variables in turn. The same indicator is used for all variables. You can mix and match indicator types by just passing in mixed indicator types in a cell array; indicators that don't match the type of the column they are operating on are just ignored.

Returns a table with same variable names and types as *obj*, but with variable values standardized.

5.2.19.57 table.varfun

out = `varfun` (*fcn*, *obj*) [Method]
out = `varfun` (..., `'OutputFormat'`, *outputFormat*) [Method]
out = `varfun` (..., `'InputVariables'`, *vars*) [Method]
out = `varfun` (..., `'ErrorHandler'`, *errorFcn*) [Method]

Apply function to table variables.

Applies the given function *fcn* to each variable in *obj*, collecting the output in a table, cell array, or array of another type.

5.2.19.58 table.rowfun

out = `varfun` (*fcn*, *obj*) [Method]
out = `varfun` (..., `'OptionName'`, *OptionValue*, ...) [Method]

This method is currently unimplemented. Sorry.

5.2.19.59 table.findgroups

[*G*, *TID*] = `findgroups` (*obj*) [Method]

Find groups within a table's row values.

Finds groups within a table's row values and get group numbers. A group is a set of rows that have the same values in all their variable elements.

Returns: *G* - A double column vector of group numbers created from *obj*. *TID* - A table containing the row values corresponding to the group numbers.

5.2.19.60 table.evalWithVars

out = `evalWithVars` (*obj*, *expr*) [Method]

Evaluate an expression against table's variables.

Evaluates the M-code expression *expr* in a workspace where all of *obj*'s variables have been assigned to workspace variables.

expr is a charvec containing an Octave expression.

As an implementation detail, the workspace will also contain some variables that are prefixed and suffixed with `"__"`. So try to avoid those in your table variable names.

Returns the result of the evaluation.

Examples:

```
[s,p,sp] = table_examples.SpDb
tmp = join (sp, p);
shipment_weight = evalWithVars (tmp, "Qty .* Weight")
```

5.2.19.61 table.restrict

out = `restrict` (*obj*, *expr*) [Method]
out = `restrict` (*obj*, *ix*) [Method]

Subset rows using variable expression or index.

Subsets a table row-wise, using either an index vector or an expression involving *obj*'s variables.

If the argument is a numeric or logical vector, it is interpreted as an index into the rows of this. (Just as with ‘subsetRows (this, index)‘.)

If the argument is a char, then it is evaulated as an M-code expression, with all of this' variables available as workspace variables, as with `evalWithVars`. The output of expr must be a numeric or logical index vector (This form is a shorthand for `out = subsetRows (this, evalWithVars (this, expr))`.)

TODO: Decide whether to name this to "where" to be more like SQL instead of relational algebra.

Examples:

```
[s,p,sp] = table_examples.SpDb;
prettyprint (restrict (p, 'Weight >= 14 & strcmp(Color, "Red")'))
```

5.2.20 tableOuterFillValue

Not documented

6 Copying

6.1 Package Copyright

Tablicious for Octave is covered by the GNU GPLv3.

All the code in the package is GNU GPLv3.

The Fisher Iris dataset is Public Domain.

6.2 Manual Copyright