Language Reference Manual
BuckCal

Date: October 27, 2014

Team

Ahmad Maruf (aim2122)

Lan Yang (ly2331)

Lingyuan He (lh2710)

mxeng Wang (mw2972)

Prachi Shukla (ps2829)

Programming Language and Translator

COMS W4115 (Fall '14)

Prof. Stephen Edwards

Table of Contents

1. Preface

This reference manual describes BuckCal, a matrix manipulation language for calculating expenses.

BuckCal has full support for mathematical matrix operations and is optimized for spreadsheet calculations. With enhanced matrix operations and data type support, programmer can make budget, record expenses, and add numbers with different units without worrying about unit conversion. BuckCal source code file have .bc suffix. Programs are translated into R scripts for excutation.

2. Lexical Component

2.1. Keyword

Keywords are reserved for language processing, and they cannot be used for identifier or other purposes.

All keywords:

if then else elif fi for in do rof disp
break continue not and or def fed return
int double bool string mat true false import

2.2. Identifier

An identifier is used to name a variable or function, it could be any combination of lower case letters (a to z), number (0-9) or underscore (_), except that the first letter must be a lower case character (a to z). An identifier cannot be an exact match with one of the reserved keywords.

Examples of valid identifiers:

```
i
matrix01
food_day2
col_name
```

Examples of invalid identifiers:

```
Var         (upper case letter)
0number     (number as the first letter)
an int      (with space)
*str        (invalid character)
```

2.3. Literal

A literal is used to express a constant value. It can be a numeric value (integer or floating point), a string (including one-character string), a numeric value with supported unit, or a matrix.

Examples:

```
12              (integer)
1.2             (floating point - double)
true            (boolean)
'hello'         (string, single character is also a string)
1.5mile         (a numeric value with unit)
{1, 2; 3, 4}    (a matrix)
```

Note that a string must be quoted by single quote; double quote is not accepted. And a literal of numeric value with unit can only be used in matrix.

2.4. Data Unit

In a matrix in BuckCal, each column can specify a single data unit, it could be any one of the following basic categories:

- (mass and weight) lb, oz, kg, g
- (distance) ft, m, km, mile
- (area) ft2, m2, km2, mile2
- (capacity and volume) ml, l, quart, gallon

The point of only including a small number of built-in units is to provide convenient access to everyday data, as it is the main usage of BuckCal. Currency, as a special case, is supposed to be without a unit in BuckCal, simply because it adds up significant overhead to support all the currencies.

Supported data units can be converted to one another, as long as the original and new units are in the same category and both are defined (compatible). Programmer can also mix up different compatible units when doing calculations, which will be automatically converted in the background if they are compatible. Otherwise, a compiler error will be raised. More details about operation with unit are in section 7.

2.5. Seperator

There are several letters served as separators:

- Semicolon - ';': separates each statement, and also each row in a matrix definition.

- Comma - ‘,’: separates arguments in a function argument list, also separates two same-row data members in a matrix definition.
- Curly bracket - ‘{}’: separates matrix content with other lexical components, i.e., left curly bracket marks the start of a matrix content, while the right square bracket marks the end.
- Parenthesis - ‘()’: used to modify operator precedence, and to wrap the argument list of a function.

Example 1:

mat ax : {1, 2; 3, 4};

This is a 2 x 2 matrix definition. Inside the brackets is the content in the matrix row, where a comma separates two number in a same row, and a semicolon separates two rows. Finally, a semicolon marks the end of this statement.

Example 2:

int i: (2 + 3) * 4;

This is an integer assignment with a pair of parenthesis to modify operator precedence. In this case, i will be 20 (addition followed by multiplication), instead of 14 when there is no parenthesis (multiplication followed by addition).

2.6. Operator

There are six (6) operators in symbol form in BuckCal:

- Plus - ‘+’
  mathematical addition for numeric values, concatenation for strings, one-on-one addition for numeric elements in matrix
- Minus - ‘-’
  Mathematical substraction for numeric values, one-on-one substraction for numeric elements in matrix
- Multiply - ‘*’
  Mathematical multiply for numeric values, one-on-one multiplication for numeric elements in matrix
- Divide - ‘/’
  Mathematical division for numeric values, one-on-one division for numeric elements in matrix
- Equal mark - ‘=’
  Used only for content comparison, output a boolean value, usually used with if statement (e.g. i = 2)
- Colon - ‘:’
  Used for variable assignment and initialization (e.g. **int** c: 3)

- Square bracket - '[]'
  Matrix subscripting. Input row and column number (which is counted from 1) separated by comma (','), and use 0 to represent all. (e.g. budget[0,1] select first column)

Specifically, below are the mathematic operators (+ - * /) and examples of their usage:

	+	-	*	/
double op double	1.0+2.1=3.1	1.0-2.1=-1.1	1.0*2.1=2.1	2.1/1.0=2.1
int op int	1+2=3	1-2=-1	1*2=2	1/2=0
string op string	's'+'u'->'su'	N/A	N/A	N/A
matrix op {int| double}	{1, 2; 3, 4} + 1 = {2, 3; 4, 5}	{1, 2; 3, 4} - 1 = {0, 1; 2, 3}	{1, 2; 3, 4} * 1.2 = {1.2, 2.4; 3.6, 4.8}	{1, 2; 3, 4} / 2.0 = {1.5, 1.0; 1.5, 2.0}
matrix op matrix	{1, 2; 3, 4} + {1, 2; 3, 4} = {2, 4; 6, 8}	{1, 2; 3, 4} - {1, 2; 3, 4} = {0, 0; 0, 0}	{1, 2; 3, 4} * {1, 2; 3, 4} = {1, 4; 9, 16}	{1, 2; 3, 4} / {1, 2; 3, 4} = {1, 1; 1, 1}

Additionally, there are three operators that are written as words, all of which are for logical operation:

- **not**: e.g. **not** A, the expression is the negation of A.
- **and**: e.g. A **and** B, the expression is true if both A and B are true.
- **or**: e.g. A **or** B, the expression is true if either A, B, or both, are true.

Details on operators and operation can be found in section 4 ('Expression').

2.7. Newlines, Whitespaces and Tabs

In BuckCal, newlines, whitespaces and tabs are only used to split lexical components, especially identifiers and literals. There is no required alignment of spacing style.

However, for the consideration of readability, in addition to separate all lexical components by whitespaces (as it is normally done), it is recommended as a good practice to only write one statement per line, and use tabs properly in a control structure or in a function to indent and align the code. Examples and more about coding style are illustrated in Section 7 ('Sample Programs').

3. Data Type

In BuckCal, these are primitive data types: number (integer and floating point number), string, and boolean. Data with same primitive data type can be composed into a matrix. Besides, number in matrix can have unit.

3.1. Primitive Data Type

- int

  The **int** data type can hold 32-bit integer values in the range of `-2,147,483,648` to `2,147,483,647`

- double

  The **double** type represents fractional numbers. Its minimum value is `2.2250738585072014e-308`, and its maximum value is `1.7976931348623157e+308.` The floating-point data type is signed.

- string

  A **string** represents a collection of characters. BuckCal does not support a built-in "character" type, therefore, a character, a literal, and a long series of characters are all referred to be a **string**. Strings are constant and immutable; their values cannot be changed after they are created. Buckcal uses the UTF-8 representation for all **string** characters, variables and literals (Refer to the definition of the U+*n* notation in the Unicode Standard.
  <http://www.unicode.org/resources/utf8.html>.

- boolean

  The boolean data type (**bool**) has only two possible values: **true** and **false**. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

3.2. Matrix

A matrix (***mat***) is a data collection type, which can store data of a single primitive data type. That means a matrix can be an int-matrix, double-matrix or string-matrix. Noticeably, there is no generic matrix or boolean-matrix. A matrix is composed of rows and columns. Rows and columns have names. By default, row names are "r1", "r2", … , and column names "c1", "c2", … . A number-matrix can have only one unit per column. To declare a matrix, it is required to specify the type of it, for example:

double mat: m;

This declares an empty matrix m that will store **double**.

3.3. Data Type Conversion

Sort of built-in functions are provided support data type conversion. The function naming convention is:

newtype : newtype_of_oldtype(oldtype X);

The conventions we support are:

int <-> **double**

[**int**|**double**] <-> **string**

[**int mat**|**double mat**] <-> **string mat**

Detail list of functions are in Appendix A. Note that int (mat) and double (mat) can be converted implicitly. So there are no int-double conversion functions.

4. Expression

4.1. Variable Declaration

All variables must be declared with its data type before used. The initial value is optional; but if there is one, it must be an expression resulting in the same type with variable.

Grammar:

datatype identifier [: initial value]

Samples are as follows:

```
int a : 1 + 2;
double d : 2.0;
string c : "This is a string";
bool b;
```

If variables are declared without initial value, their default values are as follows.

```
int a; # a = 0
double d; # d = 0.0
string s; # s = ""
bool b; # b = false
```

To create a non-empty matrix, "{}" is needed. Columns are separated with a comma, and rows are separated with a semicolon. All rows must have the same number of elements. Example:

```
mat m: {1, 2; 4, 5};
```

Matrix variable declared without initial value is empty:

```
mat m; # m = {}
```

4.2. Variable Scope

There are two levels of scope: top and function. A "top variable" is a variable with top scope, "function variable" with function scope.

A variable defined within a function has a function scope. It can only be referenced with in the function where it is defined. Top variables are defined out of function.

Scopes are isolated from each other: a variable defined in top variable cannot be referenced with in a function. If a function variable has the same name as a top variable, it is treated as a different variable and has no connect with the top one.

4.3. Variable Assignment

The assignment operator ":" stores the value of its right operand in the variable specified by its left operand. The left

operand (commonly referred to as the "left value") cannot be a literal or constant value.

The left and right operand should be of the same type. The only exceptions:

- Integer and double can be assigned to each other. Only the integer part of a double will be assigned to an int.
  Example:
  (**double** b : 1.2; **int** a;)
  a : b; # a = 1
- Double with unit can be assigned to double and integer. The unit part is ignored in assignment, so this case is equivalent to assign a double to a double/integer.

4.4. Variable Comparation

Comparison operators can be used to compare primitive variable pairs of the same type. The result of comparison is a boolean: true or false.

- "=" test if the two variables equal. "!=" returns the opposite.
- "<" test if the left operand is less than right operand. ">=" returns the opposite.
- ">" test if the left operand is greater than right operand. ">=" returns the opposite.
- String is compared by dictionary order. For boolean, true is greater than false.

4.5. Arithmetic Operation

For int and double, addition (+), subtraction(-), multiplication (*), and division(/) are supported, as well as negation(-).

If int and double are mixed, int are converted to double first.

4.6. String Operation

- Concatenation: use operator "+" to join two strings and return a new string. Example:
  string a : "b"+"c"
  we get a as "bc"
- Slicing: use function slice() to get a substring out of a string. Example:
  string a : "ABCDEFGH"
  slice(a, range(1, 4));
  we get "ABCD"
  slice(a, {1, 2, 4, 6});
  we get "ABDF"

4.7. Operator Precedence

Rules of precedence ensure when dealing with multiple operators, codes can be concise and simple while not having ambiguity. A simple example of precedence is a: b + c * d. This expression means that the result of c multiplying with d is added to b, and then the addition result is assigned to a.

In BuckCal, most operators are left-associated, some special cases would be stated later. The order of highest precedence to lowest precedence follows as the list below. Sometimes, two or more operators have the same precedence.

- Parenthesis "()"
- Function calls, matrix subscripting
- Unary negative
- Multiplication, division expressions
- Addition and subtraction expressions
- Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions
- Equal-to and not-equal-to expressions
- Logical NOT expressions
- Logical AND expressions
- Logical OR expressions
- Assignment expressions

5. Statement

Statements are implemented to cause actions and control flow within your programs.

5.1. Conditional Statement

The **if-then-else-fi** statement is used to conditionally execute part of the program, based on the truth value of a given expression. Here is the general form of an if-else-fi statement:

```
if bool-expr then
     statement1;
else
     statement2;
fi
```

If bool-expr evaluates to **true**, then only statement1 is executed. However, if bool-expr evaluates to **false**, then only statement2 is executed. The else clause is optional.
Here is an actual example:

```
if b < 0 then
```

```
    disp 0;
else
    disp b;
fi
```

The **if-then-elif-then-else-fi** statement is used to cascade the conditional execution of the program. Here is the general form of an if-elif-else-fi statement:

```
if bool-expr1 then
    statement1;
elif bool-expr2 then
    statement2;
elif bool-expr3 then
    statement3;
else
    statement4;
fi
```

Just like in if-then-else-fi, the else clause is optional here. Here is an actual example:

```
if b = 0 then
    disp "b is 0";
elif b = 1 then
    disp "b is 1";
fi
```

5.2. Loop Statement

5.2.1. Counting Loop

The counting loop statement iterates over a vector - a pre-defined collection of values. It is compact and easy to read. Here is an example:

```
int r;
for r in {1; 2; 3; 4; 5} do
    #do something
rof
```

In the above example, 'r' traverses through 1 to 5. Instructions are executed per r.

As a more useful example, the following code traverse the rows in a matrix, with the help of built-in function range() and rows():

```
# suppose mat b is defined before
```

```
int r;
for r in range(1, rows(b)) do
    #do something to b[r, 0]
rof
```

5.2.2. Conditional Loop

The conditional loop iterate until the given condition becomes false. This is suitable for cases where the condition itself can change in the loop. Here is the general form:

```
for bool-expr do
    # do something
rof
```

The loop stops when the bool-expression returns false.

5.3. Break and Continue Statement

5.3.1. Break Statement

The break statement can be used to terminate a for loop. Here is an example:

```
for r in range(1, 5) do
if r > 4 then
    break ;
else
#do something
fi
rof
```

The above example exits the for loop by executing the break statement when it finds r > 4.

5.3.2. Continue statement

The continue statement can be used in loops to terminate an iteration of the loop and begin the next iteration. Here is an example:

```
for r in range(1, 5) do
    if r < 2 then
        continue;
    else
        #do something
    fi
rof
```

The above example exits the iteration when it finds r < 2 and jumps to the next iteration.

5.4. disp Statement

The keyword disp can print any variable (including literal) and expression with a return value. The format is:

disp expr;

This will print the value of expression to stdout, in a pre-defined pretty style.

6. Function

The function is a fundamental modular in BuckCal. A function is usually designed to perform a specific task, and its name often reflects that task.

BuckCal provide some built-in functions. Users can also define their own functions.

6.1. Function Declarations

A function declaration is to specify a function's return value type, the name of function, and a list of types of arguments. Here is the general form:

[type] identifier(argument-list);

Examples of function declaration:

```
helper(int x);
mat range(int, int);
int cols(mat);
```

The declaration begins with the return value type. If the function returns nothing, then type is omitted. After that is the function name (e.g. “range”). The list of argument is between a pair of parentheses. Note that the names of formal variables are optional. In fact, the names (if any) are ignored, and only types take effect. Finally it ends with a semicolon (“;”).

Function declaration can only be in the top level.

6.2. Function Definition

A function definition begins with a keyword **def** and a declaration-similar part, which specifies the name of the function, the argument list, and its return type. The difference is argument list contains <type, identifier> pairs, where the name of formal variables are must.

Keywords **do** and **def** wrap the function body. In the function body is the declarations of local variables, and a list of expressions and statements that determine what the function does. Note that all variable declarations must appear before any expressions and statements.

Here is the general form:

def [type] identifier(argument-list) **do**

function-body

fed

A function definition cannot appear in another function's definition. That means it only exists in top-level code. Recursion is allowed - a function can call it self in definition.

6.3. Function Overloading

As is in C++, BuckCal allow functions have the same name, as long as the number or types of arguments are not identical.

For example, some library functions are overloaded:

```
int cols(int mat);
int cols(double mat);
int cols(string mat);
```

The above three functions have the same name, same number of argument, but type of arguments are different. Thus the overloading is valid.

Note that overloading by difference in return value type is not acceptable. That means user cannot define two functions almost identical except in return value type.

6.4. Function Call

BuckCal built-in functions can be called anywhere, anytime in a program. User-defined function must be defined first with a Function Definition before ready for calling. Alternatively, user can write a Function Declaration before invoking, but the function has to be implemented somewhere else.

By default, BuckCal pass arguments by value. Specially, mat (int mat, double mat, string mat) are passed by reference.

A type-call pattern for user-defined function is declared it in the source file where it is invoked, but defined somewhere else. Specially, BuckCal provide some library functions, user can just include the declarations and then use them.

6.5. Using Function in Other Files

To use functions defined in another source, a special import command can be helpful. The general form is :

import *source-file-name*

For example, to use functions defined in func.bc:

import func

Then all functions in func.bc is available after import command.

6.6. Built-in functions

See Appendix A.

7. Matrix Operation

Because of the complexity of matrix definition and operation, here is a chapter especially for matrix.

7.1. Type of matrix

Accroding to the type of its elements, a matrix should be one of the three sub-types: **int mat**, **double mat**, and **string mat**. There is no "bool mat" type, which is meaningless. mat of different sub-types are mostly similar.

7.2. Matrix definition

Matrix definition is much the same as other variable's definition. Here is an example: define an **int mat** named ax:

```
int mat ax : {1, 2, 3; 4, 5, 6; 7, 8, 9};
```

User will get a matrix ax like this:

```
1  2  3
4  5  6
7  8  9
```

Define a **string mat**:

```
string mat sx : {'ab', 'cd'; 'ef', 'gh'};
```

User will get a matrix sx like this:

```
'ab' 'cd'
'ef' 'gh'
```

A useful library function (in fact is a set of overloaded functions) can initialize or re-initialize a matrix by column:

```
double mat ax;
init_mat_col(ax, {'a', 'b', 'c'});
```

ax becomes a 0*3 matrix, whose column names are 'a' 'b' 'c'. And initialize by row and column numbers:

```
string mat sx;
init_mat_col(sx, 5, 3);
```

sx becomes a 5*3 matrix.

7.3. Matrix accessing

Operator [*row*, *col*] is for accessing a sub-matrix of a matrix. *row* and *col* can be **int** or **int mat**, as usual. Note that counting and indexing begins with 1 in BuckCal. So 0 refers to all rows/ columns.

read element(s):

```
double a : ax[1, 2];  # we get a = 2
disp ax[1, 0]; # we get the 1st row of A.
```

disp ax[**0**, range(1,2)]; # we get the first 2 columns of A.

write element(s):

ax[1, 2] : 4.0; # set (1st row, 2nd column) of A to 4.0

ax[1, 2] : {4.0}; # set (1st row, 2nd column) of A to 4.0

ax[1, 0] : range(1, (cols A));

set 1st row of A as 1, 2,...

Note that matrix can also be accessed by column/row names. In this case, *row* and *col* can be **string** or **string mat**. A string of row/column name is equivalent to an int of row/column number.

As a special case of matrix, vector can be accessed with operator [*index*], since vector is a one-dimension structure. As in mat, *index* can be an **int**/**int mat** or **string**/**string mat**.

7.4. Unit setting

By default, elements in matrix don't have units. There a two approaches to assign unit.

- **Explicitly**: The colunit function can set the unit for every column in a matrix. For example:

  colunit(mx, {'lb', 'ft'});.

  set unit for all two columns of mx. That implies cols(mx) = 2; if not, a error generated.

  colunit(mx[**0**, 2], {'ft'});

  set the unit of 2nd column of mx. Specificlly, the following case is equavalent to previous one.

  colunit(mx[1, 2], {'ft'});

  The row index is ignored.

  To unset the column unit:

  colunit(mx[0, range(1, 2)], {'', ''});

  Now the unit of 2nd column is gone! Note: there is no implicity way to remove units.

  To reset the column unit to units in the same category, just use colunit. 0 numbers are converted to new unit.

  colunit(mx[**0**, 2], {'m'});

  To reset the column unit to units in the different category: No direct way!
- **Implicitly**: The unit of a column is decided when first assigned with "double with unit". Be advised, once the unit is set, the "category" of unit is also set. So if "double with unit" is assigned to other elements of the same column, the unit have to be in the same category.

Example:

```
mat mx : {1, 2; 3, 4; 1, 5} # now mx have no units
#####################################################
mx[2, 1] : {1kg};
# the 1st column has unit 'kg', and category 'weight'
#####################################################
mx[2, 1] : {2};
# the 1st column still has unit 'kg'
#####################################################
mx[1, 1] : {1lb};
# In fact 'lb' is converted to 'kg'.
# Now mx{1, 1} = 0.45 , since 1lb = 0.45kg
#####################################################
mx[3, 1] : {1m};
# Assignment fail! 'm' belongs to 'length', not 'weight'
#####################################################
```

The column unit is a attribute of a matrix, as is name of column. And the unit can be printed when disp a matrix:

```
      c1(kg)    c2
r1    0.45      2
r2    2         4
r3    1         5
```

7.5. Other operations with unit

- `mx : rowcat(mx1, mx2);`

The units of corresponding columns in mx1 and mx2 should be of the same unit. Otherwise a runtime error is raised.

- `double a : mx[1, 1]; # a = 1`
- `mx[1, 2] : a;  # mx[1, 2] = 1`

When a double with unit assigned to a double, the unit are dropped.

- `mx : rowcat(mx1, mx2);`

The units of corresponding columns in mx1 and mx2 should be of the same unit. Otherwise a runtime error is raised.

- binary operations:

```
mx1[1, 0] + mx2[1, 0]; mx1[0, 1] - mx2[0, 2];
mx1[1, 0] * mx2[2, 0]; mx1[0, 1] - mx2[0, 1];
```

The units/names are dropped in result.

8. Sample Program

8.1. Hello-world style

```
# function definition
def hello() do
      disp 'hello';
fed

# top variable declaration
int a : 5;
int i : 0;
int x : 0;
int mat v ;
double b : 0.0;
string endstr : 'End' ;

# top level code
hello();
for i do
      if i <= 5 do
            v : colcat(v, {i});
            i : i + 1;
      else
            break;
      fi
orf
for x in v do
      if x <= 2 then
            b : b + x*x;
      elif x <= 4 then
            b : b + x;
      else
            b : b + x/2.0;
      fi
orf
disp endstr;
```

8.2. Matrix operation

```
# sum up rows! not using sum_row deliberately.
def double mat sum_up_rows(double mat mx) do
      int x;
```

```
      double mat s: range(1, cols(mx)) * 0.;
      for x in range(1, (rows mx)) do
            s: s+ mx[x, 0];
      orf
      return s;
fed

# declare top variable(mat)
double mat budget;
double mat subdget;

# initialize
init_mat_col(budget, {'Food', 'Price'});
# set unit of budget
colunit(budget[0, 1], {'lb'});
# add one row with naming
budget: addrows(budget, {1+1, 3.3}, {'John'});
# add one column and naming
budget: colcat({0}, budget);
colname(budget[0, 1], {'Paper'});
# add one row with naming
budget: addrows(budget, {250*2, 0, 5.10}, {'Tom'});
# summing
subdget: sum_up_rows(budget);
# display
disp budget;

================================================================
Note: The output of disp statement is as follows:
budget:

                Paper           Food(lb)          Price

John            0               2                 3.3

Tom             500             0                 5.10
```

Appendix A. BuckCal Library Function List

Built-in Functions

Matrix functions

int mat range(int x, int y);

Return a column vector {x; x+1; … ; y}. If y < x, return {}

int mat ranger(int x, int y);

Similar to range(), except that it returns a row vector.

int rows(int mat mx); int rows(double mat mx); int rows(string mat mx);
int cols(int mat mx); int cols(double mat mx); int cols(string mat mx);

Return the number of rows/columns in matrix mx;

int mat rowcat(int mat mx1, int mat mx2); double mat rowcat(double mat mx1, double mat mx2); string mat rowcat(string mat mx1, string mat mx2);
int mat colcat(int mat mx1, int mat mx2); double mat colcat(double mat mx1, double mat mx2); string mat colcat(string mat mx1, string mat mx2);

Return a new matrix looks like mx1 added all rows from mx2

Return a new matrix looks like mx1 added all columns from mx2

colunit(double mat mx, string mat u);

Set the units of every column of mx. *u* looks like {'kg', 'm', …}

rowame(int mat mx, string mat n); rowame(double mat mx, string mat n);
rowame(string mat mx, string mat n);
colname(mat mx, string mat n); colname(double mat mx, string mat n);
colname(string mat mx, string mat n);

Set names of rows/columns in a matrix. *n* looks like {'a', 'b', …}.

The default value are 'r1', 'r2', … / 'c1', 'c2', ...

init_mat_col(int mat mx, string mat nc); init_mat_col(double mat mx, string mat nc);
init_mat_col(string mat mx, string mat nc);

Reinitialize matrix ax as 0 rows and n columns. nc looks like {'name', 'Age', '...', ...}.

n = cols(nC), and the names of columns are specified by nc.

init_mat_size(int mat mx, int nrow, int ncol); init_mat_size(double mat mx, int nrow, int ncol);
init_mat_size(string mat mx, int nrow, int ncol);

Reinitialize matrix mx with *nrow* rows and *ncol* columns. *nrow* and *ncol* should be positive.

int mat addrows(int mat mx, int mat dr, string mat drn);
double mat addrows(double mx, double mat dr, string mat drn);
string addrows(string mx, string mat dr, string mat drn);

Add rows in dr to mx, and naming the new added rows with drn. drn looks like {'Tom', 'John', '...', ...}, and its length equals to the row number of dr.

note: the naming of this function is different "rowcat", which begins with "row". Because rowcat is a buit-in function while addrows not.

Datatype Conversion Functions

string string_of_int(string x);
string string_of_double(string x);
int int_of_string(string x);
double double_of_string(string x);
int mat mat_int_of_string(string mat x);
double mat mat_double_of_string(string mat x);
string mat mat_double_of_double(double mat x);
string mat mat_double_of_int(int mat x);

String Functions

string slice(string x, int mat idx);

Get a substring out of a string. *idx* looks like {1, 2, 3} or range(1, 3)

Math/Statistics Functions

double abs(double x);
double mat sum_row(double mat mx);
double mat sum_col(double mat mx);
Return a vector, as the sum of rows/columns in mat mx.
double mat avg_row(double mat mx);
double mat avg_col(double mat mx);
Return a vector, as the average of all rows/columns in mat mx.
double mat var_row(double mat mx);
Return a row vector v, where v[i] is the variace of numbers in mx[0, i].
double mat var_col(double mat mx);
Return a column vector v, where v[i] is the variace of numbers in mx[i, 0].