BuckCal

A Matrix Manipulation Language for Calculating Expenses

Ahmad Maruf (aim2122): Manager
Meng Wang (mw2972): Language Guru
Prachi Shukla (ps2829): Systems Architect
Lan Yang (ly2331): Verification & Validation
Lingyuan He (lh2710): Systems Integrator

Programming Language & Translators (COMS W4115)
Instructor: Professor Stephen Edwards

Fall 2014

September 24, 2014
**Proposal:**
BuckCal is a matrix manipulation language. It has full support for mathematical matrix operations and is optimized for spreadsheet calculations. With enhanced matrix operations and data type support, we can make budget, record expenses, and add numbers with different units without worrying about unit conversion. Programs written with BuckCal include employee payroll calculator, statistical computation, and estimation of budget.

**Motivation:**
Calculating expenses is amongst the most common daily tasks. For example, keeping track of our daily/weekly expenses or how much money we owe to others. BuckCal is designed to record all the data in a matrix, calculate daily expenses on various commodities, and estimate savings. It allocates the bills to members of matrix, so that every member gets a clear thought of how much he/she owes to others.

Someone may suggest numeric computing softwares such as MATLAB and visual spreadsheet processing application like Microsoft Excel. However, MATLAB is not convenient enough for simple daily use and Microsoft Excel is not designed for straightforward programming. To solve these problems, BuckCal combines the advantage of both MATLAB and MS Excel by offering powerful computation and ease of use. Moreover, what differentiates BuckCal from those above is that it deals with different units of currency and goods. In short, our project aims to introduce a simple, comprehensible and easily programmable language to analyze the money we spend.

**Language Features:**
1. Built-in types, with a few examples:
   - integer (123, -5)
   - double (1.23)
   - string (‘This’, ‘a’)
   - boolean (true/false)
   - matrix (composed of basic data types)
   - vector (Matrix with only one column or row)
2. Operators:
   - + (Addition)
   - - (Subtraction)
   - > (greater than)
   - < (smaller than)
   - = (equal)
   - <> (unequal)
   - : (assignment)
   - . (indicating the reverse of function name and argument)
3. Units (‘kg’, ‘lb’, ‘l’, ‘qt’) are well-defined, combined, and auto converted in matrix for computing (e.g. adds 1kg with 1 lb equals to 1.4 kg)
4. Console/File Output (print matrix to console / a file)
5. Proper indentation (Consistency required)
6. Every statement ends in a semicolon (;)
7. No OOP support: no class, namespace
8. Function call by value
9. Naming conventions of variables: only letters, case-sensitive.
10. Strong static types: static type-checking
11. Matrix Operations:
   - Sum up rows/columns
   - Add/drop a row/column
   - Rename rows and columns
   - Access an element in a matrix by name or index

**Keywords:**

int (integer)
double
string
bool (boolean)
mat (instantiate a matrix)
def .... fed (function declaration)
for ... in ... orf (iteration)
if ... then ... else ... fi (branch)
disp (print to console)
export (print to file)

**Programming Features:**

1. Comments
   
   # This is a single-line comment. No multi-line comment support
   This is not comment but # From here to end-of-line is comment

2. Declaration
   
   int uninit; # declare an integer without initialization
   int i: 2; # declare an Integer variable a and initialize as 2
   double d: 2.50; # declare an floating-point number 2.50
   string s: “this string”; # declare a string s and initialize it
   bool a: true; # declare a boolean
   mat m: [1, 2; 4, 5]; # declare a matrix: “;” starts a new row, “,” splits elements in same row

3. Basic Expressions
   
   int a: 1 + 2; # yields 3
   double b: 2.0 - 3; # yields -1.0
   if b < 0 then
      disp a;

else
    disp b; # end of if
fi

for i : in [1, 2, 3, 4] # iteration with a for-loop
    disp i;
endof # end of for

int foo_bar: int x  # define a function “foo_bar”, with one int argument x, return a int
def x + 5  # foo_bar x will returns x+5
    # ‘def’ also indicates the end of function
int c: foo_bar 3  # call function foo_bar, yield 8
int c: 3.foo_bar  # equivalent to previous line

Library Support:
1. addrow, addcol:
   add a row/column to an existing matrix

2. sumrow, sumcol:
   sum up all the rows/columns and put the result as a new row/column in matrix

3. rowname, colname:
   Access the name of all rows/cols in a matrix

Example Code:  (black: source code; blue: output; green: comments)

# Matrix must be explicitly declared
# Create an empty matrix "budget"
mat budget ;
# John bought 1lb food with $3.30
# Add one row "John" to budget,
# column "Food" = 2lb, column "Price" = 3.3
addrow budget 'John' ['Food': 2lb, 'Price': 3.3] ;
    # Similarly, one more
addrow budget 'Tom' ['Paper': 500, 'Price': 5.10] ;
disp budget ;  # export budget budget.txt
budget:
    Food  Price  Paper
John   2lb    3.3   0
Tom    0      5.10  500
# John bought 1kg of food for 4$
addrow budget John ['Food': 1kg, 'Price': 4] ;
    #display budget matrix
disp budget ;
budget:

<table>
<thead>
<tr>
<th></th>
<th>Food</th>
<th>Price</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2lb</td>
<td>3.3</td>
<td>0</td>
</tr>
<tr>
<td>Tom</td>
<td>0</td>
<td>5.10</td>
<td>500</td>
</tr>
<tr>
<td>John</td>
<td>1kg</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

# sum up!
#sum_row : sums up the columns and add a new row with all the results
mat sbudget : sumrow budget ;
disp sbudget ;
sbudget:

<table>
<thead>
<tr>
<th></th>
<th>Food</th>
<th>Paper</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2lb</td>
<td>0</td>
<td>3.3</td>
</tr>
<tr>
<td>Tom</td>
<td>0</td>
<td>500</td>
<td>5.10</td>
</tr>
<tr>
<td>John</td>
<td>1kg</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>sum</td>
<td>1.9kg</td>
<td>500</td>
<td>12.40</td>
</tr>
</tbody>
</table>

# note: "a.func" is equivalent to "func a"
# No "class" or "name-space" support
# define function 'split_bill' that takes an argument as matrix
mat split_bill : mat b
mat result ; # to store result
for r: in b.rows # scan every row
    idx: locate r.name result.cols.names # whether name known
    if idx = null then # new name ?
        addcol result r.name ['Debit': r('Price')] # add a column
    else # known name
        accum result(idx) r('Price'); # accumulate the price
    fi # end if
rof # end of for-loop
Here we got:
bill :

<table>
<thead>
<tr>
<th></th>
<th>Debit</th>
<th>Tom</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>7.30</td>
<td>5.10</td>
</tr>
</tbody>
</table>
# sum up price of every one
mat result1: sumcol result ;
Here we got:

<table>
<thead>
<tr>
<th></th>
<th>Debit</th>
<th>Tom</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>7.30</td>
<td>5.10</td>
<td>12.40</td>
</tr>
</tbody>
</table>
# add a row, filled with the same value
addrow result ‘SplitExp’ [result1(Debit, sum) / result.width ] ;
# Here we got
result
<table>
<thead>
<tr>
<th></th>
<th>John</th>
<th>Tom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debit</td>
<td>7.30</td>
<td>5.10</td>
</tr>
<tr>
<td>SplitExp</td>
<td>6.20</td>
<td>6.20</td>
</tr>
</tbody>
</table>

\[ \text{mat result: diffrow result ;} \]

<table>
<thead>
<tr>
<th></th>
<th>John</th>
<th>Tom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debit</td>
<td>7.30</td>
<td>5.10</td>
</tr>
<tr>
<td>SplitExp</td>
<td>6.20</td>
<td>6.20</td>
</tr>
<tr>
<td>diff</td>
<td>1.10</td>
<td>-1.10</td>
</tr>
</tbody>
</table>

\[ \text{result(-1, :) rowname Budget ;} \]

\[ \text{def result ; } \] # return result and end the function

\[ \text{# call function} \]

\[ \text{# the results are demonstrated above} \]

\[ \text{mat bill : split_bill budget ;} \]

\[ \text{disp bill;} \]

\[ \text{bill} \]

<table>
<thead>
<tr>
<th></th>
<th>John</th>
<th>Tom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debit</td>
<td>7.30</td>
<td>5.10</td>
</tr>
<tr>
<td>SplitExp</td>
<td>6.20</td>
<td>6.20</td>
</tr>
<tr>
<td>Budget</td>
<td>1.10</td>
<td>-1.10</td>
</tr>
</tbody>
</table>