Project Proposal
Programming Languages and Translators – COMS W4115

Team: Le Chang (lc2879@columbia.edu)
Qiuzi Shangguan (qs2130@columbia.edu)
Adam Dossa (aid2112@columbia.edu)
Maria Taku (mat2185@columbia.edu)

Date: September 26, 2012

TML
Table Manipulation Language

Language Description

The purpose of TML is to provide users with an efficient and simple language for building, editing, and manipulating tables/spreadsheets. For example, through only a few simple lines of code, a user can create a spreadsheet, populate the spreadsheet with data, edit the data, and even perform mathematical operations on this data (e.g., summing values in various cells, etc.) . As one could imagine, performing operations such as this in Java or C++ could take hundreds of lines of code!

At a higher level, the functionality of this language could be used to allow programmers to quickly and efficiently manage budgets, calculate yearly taxes, or otherwise keep track of various types of numerical data and the relationships between this data. It is even possible for programmers to use this feature to implement and manipulate more advanced data structures such as Stacks, Heaps, or Graphs.

Problems Solved By the Language

As mentioned above, this language allows users to create and manipulate spreadsheets through a minimal number of lines of code. According to our knowledge, functionality like this is not readily available in the common programming languages such as C/C++, Java, Python, etc. In other words, this language can be used to meet the various needs of financial, statistical and engineering software.

Syntax

1) Basic Data Types: The basic data types will be:

int: type of Integers

float: type of floating numbers
char: type of single character
string: type sequence of characters
cell: type of cells/elements in table
table: type of table/spreadsheet

2) Comments: The pound sign (#) will be used to indicate to-end-of-line comments. For example:

```
     create(4,4);                    # Comments can go here
```

3) Statement ending: Every statement (except control statements) should end with ";" . For example:

```
     delete(myTable);
```

4) Control Statements: TML will also include the basic "if/then/else" and "for" control statements. The syntax of the control statements will mimic the ones generally accepted by the programming community, below are some examples that show the grammar of the TML language:

```
for (int i = 0, i<10, i++){
        # perform some action
}

if (sumCells([*,2]>budget){
       print("your budget is too high");
} else {
        # try doing something else
}
```

5) Operators: operators are supported to perform various mathematical calculations between numbers.

Basic operator:

E + F: Addition
E - F: Subtraction
E * F: Multiplication
E/ F: Division
E > F: Greater than

E < F: Less than
E == F: Equals
E >=F: Greater than or equal to
E <= F: Less than or equal to.

6) Built-in functions (for table manipulation, etc.):

```
Table a = createTable(8,8);                          #create a new table
insertCell(tableName, [row,column], value) ;      #put element into table
deleteCell(tableName, [row,column], value);      #delete element in table
valueCell(tableName, [row,column]);              #get the value from cell
showTable(tablename);                              #print table
sumCells(tableName, [row,column]);            #sum cells in [row, column]
```

The values of the "rows" and "columns" can be either a single value, a wildcard, or a range (see the Representative Program for examples).

Representative Program

This representative program illustrates the building of a spreadsheet, populating the spreadsheet with data, editing/updating the data, and displaying the spreadsheet. Note that in general, when specifying a cell, the row/column values are put inside brackets (for example, [row,column]) to make the program easier to read.

```
# Creates a 4x4 spreadsheet, with cells initialized to NULL and returns it
Table myTable = createTable(4,4);

# Inserts the value 1000 into the cell at row 2, column 2 in myTable
insertCell(myTable,[2,2],1000);

# Inserts the value 2000 into ALL cells in column 1 in myTable
insertCell(myTable,[*,1], 2000);

# Inserts the value 3000 into ALL cells in row 1 in myTable.  Note that the previous value of
2000 in
# row 1, column 1, will be overwritten
insertCell(myTable,[1,*], 3000);
```

```
#Inserts data (1,22,3) into rows 1-3, column 3 in myTable
insertCell(myTable,[1-3,3],(1,22,33));

# Inserts the sum of column 1 into cell row 4, column 2
insertCell(myTable,[4,2],sumCell(myTable,[*,1]))

# Inserts the sum of row 1, from columns 3 to 4, into cell row 4, column 4
insertCell(myTable,[4,4],sumCell(myTable,[1,3-4]));

# Returns the value of the cell with row 4, column 4
valueCell(myTable, [4,4]);

# Prints the table to the screen
showTable(myTable);

# Compares the difference between the value in cell row 4, column 2 and the sum of the cells in
# column 1 with 3000
if ((sumCells(myTable,[*,1]) - valueCell([2,2]) - ) > 3000) {        # 9000 - 1000 > 3000 → true
        print("Your Budget is too high.");
}
```

Sample Output to Representative Program:

```
          >> 3001
          >>
             A      B       C      D    ...
    1     | 3000 |  3000  |  1   | 3000 |
    2     | 2000 |  1000  |  22  |      |
    3     | 2000 |        |  3   |      |
    4     | 2000 |  9000  |      | 3001 |
    >> Your Budget is too high.
```

Note that the column titles are output as letters (A-D) and the row title are output as numbers (1-4) to make it easier for the user to read.