# TableGen

Andrey Falko Daniel Pestov Del Slane Timothy Washington

September 25, 2007

## 0.1 Introduction

The rapid creation of algorithmically complicated data tables is currently not facilitated to a useful extent by current software such as Excel. Spreadsheet applications allow for the writing of algorithms, but it is both time consuming and cumbersome. Inputting data by tabbing through cells alone takes a long time. Data in a spreadsheet is not presentable because it is used to generate the final numbers.

TableGen will be the solution to these problems: it will be a powerful interpreted programming language with enough power to specify complex functions which can transform data. It will have an efficient syntax for assigning values to cells in a data table without any clicking, scrolling, or tabbing. Users will be able to utilize basic data structures such as arrays to easily list their data and to subsequently and conveniently compute the final results. All of this functionality will be implemented in a clean manner – unnecessary punctuation will be excluded from the language to make it less of a burden to program with.

The purpose of TableGen is to work on large input and to generate large output. It is not efficient to use TableGen to conduct a computation over $x$ variables and output one value at the end. All output in TableGen is made into cells of a table, so if you are not planning to make a table, don't use TableGen.

## 0.2 General Plan

We plan to implement a language whose syntax and semantics help people fill the cells of a table or tables with their data. Our language will allow people to:

- Fill cells, rows, or columns with minimal typing.
- Fill cells, rows, or columns by manipulating existing data.
- Output plain text, TeX/LaTeX, HTML tables.

- Allow the user to make "output modules" so that they can specify output organization other than HTML, LaTeX, etc (in language the compiler will be written in).
- Allow user to utilize statistical functions such as average, standard deviation, etc.
- Allow the user to easily add statistical functions and other functions (in our language).

We will do the following if we have time:

- Allow people to generate charts and graphs; in HTML (using css) and other formats.
- Allow the user to access and create conversion tables that can be used for convenient conversion between measurement units and systems.

## 0.3 Technical Plan

We plan to write the interpreter for the language mostly in java (backend and most of the frontend). The other parts will be in Perl (frontend). Java makes sense for a couple of reasons:

1. All of us know how to code in it.
2. Antlr is a java application, so it would be natural to utilize the same language. Perl makes sense for part of the frontend because it much easier to control the java binaries with it and to implement various command-line options with it.

The testing infrastructure will be written using a combination of Make and Perl. We plan to use Perl for the following reasons:

1. The testing infrastructure will require a lot of string comparisons, something Perl is very good at.
2. Most of us know how to code in it.

### 0.3.1 Language Design

#### Keywords and Reserved Variables or Functions

All reserved variables and functions begin with an underscore ("_").

We have a list of standard keywords:

- `for`
- `while`
- `if`
- `foreach`
- `in`

Some of the reserved variables:

- `_row_head`
- `_col_head`
- `_row_next`
- `_col_next`

Some of the reserved functions:

- `_avg(list)`: Average
- `_var(list)`: Variance

#### Types

Simple types and type inference keep users focused on the programming while cleaning up the syntax. The language still checks all types when performing operations, but only intrudes when any loss of information is possible.

The three data types are:

- Number
- String
- List

Using these three types, programmers can refer to integer and floating point numbers, strings of characters as well as lists of numbers or table data.
Users can specify types by doing:

```
num:var1 = 1

str:var2 = one

list:var = var1 var2
```

Or they can let the language decide:

```
var1 = 1

var2 = "one"

var = var1 var2
```

Anything with digits will be treated as a number. Anything in quotes will be treated as a string. Lists and arrays can be created by simply including multiple things on one line.

## Data Entry

All rows are columns are lists. These lists act as arrays and can act as hashes. The idea is that a user can refer to cells via array elements or define a key referring to a value. Users can specify a column header (`_col_head`) and row header (`_row_head`). These will serve to identify the first row and first column, and allow the user to easily cycle through them.

```
_col_head = "Monday" "Wednesday" "Tuesday" "Thursday" "Friday"

_row_head = "8:00" "12:00" "16:00" "20:00"
```

Users can assign keys to the members of the lists by using the ”>” symbol. Users can always refer to the values as arrays.

```
_col_head = "Name" "Wage" "Hours Worked" "Earned"
_row_head = "Maria" "Stanley" "Vladimir" "Glen"


wage_rate>20 hours>12  -> _row_next
wage_rate>2  hours>120 -> _row_next
wage_rate>30 hours>1   -> _row_next
wage_rate>3  hours>600 -> _row_next


func enter_pay
    foreach row in _row_head
        calculate_pay(rate:row.wage_rate, hours:row.hours) -> row.pay


func enter_pay2
    foreach row in _row_head
       calculate_pay(rate:row[0], hours:row[1]) -> row.pay


func calculate_pay num:rate num:hours
    if rate < 0
        return -1
    else
        return rate * hours
```

This program should print:

```
Name      Wage  Hours Worked  Earned
Maria     20    12            240
```

```
Stanley   2     120           240
Vladimir  30    1             30
Glen      3     600           1800
```

**Syntax and Semantics**

- Whitespace is important. The statement block will not end until indentation stops.
- Variable initialization done by using an ”=”; whitespace allowed at either end of the ”=” and between variable name and/or value.
- Variable initialization or setting done anywhere.
- A variable’s active scope is within the block in which it is declared (global scope if not in any block).

Sample Code:

```
_row_head
    ""
    a>"Monday"
    b>"Tuesday"
    c>"Wednesday"
    "Average"

_col_head
    d>"11:00-13:00"
    e>"13:00-15:00"
    f>"15:00-17:00"

for _row_head.a _row_head.c
```

```
        "My class" -> _col_head.d


for a b c

    3 4 5 -> f


_avg(_col_head.f) -> 4, 0
```

Is the same as:

```
row_header "", a="Monday", b="Tuesday", c="Wednesday"

column_header d="11:00-13:00", e="13:00-15:00", f="15:00-17:00"

for a, c put "My class" at d

for a, b, c put 3, 4, 5 at f

at f put avg_row()

"Average" -> 4, 0
```

Output:

```
                Monday     Tuesday    Wednesday   Average

11:00-13:00     My class              My class

13:00-15:00

15:00-17:00     3          4          5           4
```

**Strings**

- Standard ”\” escapes in case users desire a quote within a string.
- String variables have to be concatenated; otherwise they are just a string. e.g.

  ```
  var1 = "Hel"

  var2 = "lo"
  ```

```
var1 , var2 -> 0,0
var1var2 -> 0,1
```

Output:

```
Hello
var1var2
```

- A ”+” denotes string concatenation

**Numbers and Units**

- Interpreter will always assume any number is a floating point number, but will automatically chop off the ”.0” in 5.0. It will also round when you get something like 5.000000000000000001. If a user desires to rounding, rounding a certain decimal point, we will have special command-line options to allow the user to specify rounding.
- Automatic unit conversions and/or attachment of units to variables (if implemented):

  ```
  length = 3-in 4-cm 4-ft 203-mm
  length[0]+length[4]-in  -> 0,0
  length[3]-length[2]-mm -> 0,1
  ```

  Output:

  ```
  4.57 inches
  1259.2 millimeters
  ```

- All unit types are keywords and have to be affixed with a “-” in front. Has to be an number or number variable preceding the unit keyword.

- User will be allowed tell interpreter via command line how to output computed units (i.e. to affix the full feet or just ft).

**Subroutines and Control Structures**

- local variables can be specified in scope. e.g.

  ```
  globalVar = 2
  localVar = 3
  func compute
      localVar = 2
      localVar + globalVar -> 0,0
  localVar + globalVar -> 0,1
  ```

  Output:

  ```
  5
  4
  ```

- ”for” statements take in list of things (can be variables, strings, or numbers) separated by ”,” and does either one thing or something for each member. ”in” specifies where to do the operations. In this sample, the operations are done at `_row_head` elements. The “list of things” contains string variables. Language assumes these string variables refer to `_col_head`.

- Standard while loops.

- Subroutines are called with () at the end. Calls can be made like so:

  ```
  convert_time (3, 32, "sec")
  ```

  or so:

```
convert_time (min: 32, unit: "sec", hour: 3)
```

- Subroutines are specified like so:

```
func convert_time (num:hour, num:min, str:unit)
    if (unit == "sec")
        return (60 * min) + (60 * 60 * hour)
    else if (unit == "min")
        return 60 * hour + min
    else if (unit == "hour")
        return hour + (min / 60)
    else
        exit ("Error: You specified a unit unknown to me.")
```

- Parameters in a function call may be called in an arbitrary order since they include labels indicating what should match with the variables of a function.
- `return` is java-like.
- `exit` halts the interpreter where it is.

**Miscellaneous**

- All output will be in terms of a table.
- Rows start from the left, columns start from the top (think of x, y coordinates).
- All cells of a tables can be identified by coordinates starting with 0.
- The first row and first column can be referenced by the reserved variables `_row_head` and `_col_head` respectively.

- Reserved variables `_row_next` and `_column_next` represent the next completely empty row and column respectively.
- All values are printed into cell by using a ”dash arrow” -¿.
- Basic data structure is a list, which can be accessed randomly by numerical index.
- All variables are lists; single variables are merely one-member lists.

A simple program:

```
list1 = "A" "B" "C" "D"
list2 = "1" "2" "3" "4"


list1 -> _row_next
list2 -> _col_next
"" "" "E" "F" list1[2] -> _row_next
```

Output:

```
A B C D 1
        2
        3
        4
    E F C
```

Pascal’s Triangle as Table

```
// Factorial algorithm using a cache to compute factorials.
factCache = 1
func factorial (num:n)
    if n <= _length (factCache)
```

```
        return factCache[n]
        prod = _tail (factCache) // prod = last element of factCache
        while (_length (factCache) <= n)
            // Append prod onto end of factCache array.
            factCache = factCache (prod *= _length (factCache))
        return prod

func pascalCombination (num:n, num:k)
    return factorial (n) / (factorial (k) * factorial (n - k))

max = 10 // Set maximum # of rows to output

i = 0
j = 0
while (i <= max)
    while (j <= i)
        pascalCombination (i, j) -> (i, j)
        j++
    i++
    j = 0
```

Output:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
```

```
1   6   15  20  15  6   1

1   7   21  35  35  21  7   1

1   8   28  56  70  56  28  8   1

1   9   36  84  126 126 84  36  9   1
```