TableGen

Language Reference Manual

Andrey Falko (asf2125) Daniel Pestov (dp2297) Del Slane (djs2160)

Timothy Washington (tw2212)

October 18, 2007

1. Introduction

TableGen is an interpreted programming language designed to allow programmers to quickly generate large tables. TableGen allows experienced users to enter large data more efficiently than when using graphical spreadsheet applications, especially when data is generated via algorithmically complex functions.

The three main goals of the TableGen programming language are:

- To provide quick and easy data entry
- To allow the user to refer to data by name
- To offer simple formulation techniques for generating algorithmically complex data

2. Lexical Conventions

All information below assumes the file is being scanned in a "forward" manner: from the logical start of the file to its end-of-file character.

TableGen has four types of tokens: keywords, identifiers, constants and operators. Tokens are separated from each other by whitespaces and commas.

(a) Whitespace

Whitespace characters consist of newlines, carriage returns, tabs, and spaces; whitespace is any combination of one or more of these characters.

(b) Comments

Single-line comments begin with `//` and end with newline character.

Multi-line comments begin with `/*` and end with the first subsequent appearance of `*/`.

(c) Keywords

Keywords are the following sequences of characters:

- `foreach`
- `while`

- `func`
- `return`
- `if`
- `else`
- `elseif`
- `str`
- `num`
- `list`

(d) Identifiers

An identifier is a sequence of characters that represents the name of function or variable. It begins with any character in the English alphabet (either lowercase or uppercase) followed by any number of alphanumeric characters or an underscore (‘_’).

(e) Constants

i. Numbers

Numbers are a sequence of digits with an optional decimal point (`.`). If a number contains a decimal point, the decimal point is both preceded and followed by at least one digit.

ii. Strings

A string is a sequence of ASCII characters surrounded by double quotes (‘ ” ’). If a string is to contain any of the following characters, they must be preceded by a backslash(‘\’):

```
"    -    +    \    ,    =
```

(f) Declarations

An identifier is “declared” when it is first encountered in a file and is located on the left side of an assignment operator, and can optionally be given a user-specified type of either “number” or ”string” by prefixing the identifier with `num:` or `str:` respectively.

(g) Built-in Functions and Variables

All built-in functions and variables are treated as identifiers, and are prefixed with an underscore (‘_’).

`_row_head` and `_col_head` define the row and column headers respectively.

`_rows` and `_cols` are special lists used to define and iterate over rows and columns respectively. The elements for `_row_head` become the hash keys for `_cols`, while the element for `_col_head` become the hash keys for `_rows`. This is done to make setting and retrieval of data more intuitive. See example section to see why.

For example:

```
_cols = Monday, Tuesday, Wednesday


foreach col in _cols
    do something
```

`_row_next` and `_col_next` are pointers to the next empty row and next empty column respectively. The pointers point to a coordinate that begins a column for row. For example, `_col_next` in an empty table will point to `[0,0]`. A subsequent `_col_next` will point to `[1,0]`. If a list is put into `_row_next`, elements of the list will fill coordinates from left to right. Likewise, with `_col_next` elements will fill from top to bottom.

We have the following built in functions:

The following take numerical lists as arguments:

- `_sum()` — used to calculate row/column sums.
- `_avg()` — used to calculate the average of given values. It takes a list of elements to average.
- `_var()` — used to calculate the variance of given values
- `_tail()` — returns last element of list

- `_length()` — return length of list

3. Types

   TableGen has two types of data—numbers and strings—as well as one native data structure called a list. Every possible representation of a value is treated as a list, and constants are treated as a list of length one. Any variable declared explicitly as `num` is a list containing one number. Any other item added to such list has to be of type `num`. Any variable declared explicitly as `str` is a list containing one string. Any other item added to such list has to be of type `str`.

   (a) `num`

   This type is any real number that can be represented by the primitive data types in the Java 1.5 programming language with pre-defined operations of addition, subtraction, multiplication and division.

   (b) `str`

   This is a string of characters.

   (c) `list`

   The list type is similar to the concept of a hash—it can be assigned values to "contain" that can in turn be accessed by symbolic operations performed on the list. The elements of a list do not necessarily have to be of the same type (are *heterogeneous*), and can be accessed by the non-negative "index" integer corresponding to their position in the list.

4. Expressions and Operators

   (a) Primary Expressions

   TableGen has the following primary expressions:

   i. Identifiers

   Identifiers are names of variables which can represent numbers, strings, or lists.

ii. Constants

A constant can either be a number or a string.

iii. (expression)

Parentheses can be used to modify the precedence order of operations in expressions.

Example:

```
3 * (2 - 2)
```

iv. identifier.expression

The list element operator, the expression after the period is either an index or a key of the list preceding the dot:

```
hash.key_x
```

v. Identifier Followed By Optional List of Expressions Within Parenthesis

This a function call expression. This is an example of it:

```
aFunc (var1, var2)
```

(b) Unary Expressions

There is only one unary expression in TableGen.

i. - expression

This denotes that the value of expression after minus sign is negative. The expression has to be a number. There can be no other numeric expression immediately before the minus sign. This expression groups from right to left.

(c) Mathematical Operators

Note: mathematical operators are in order by descending precedence.

i. expression * expression

The multiplication operator, operates only on numeric types:

```
3*4
```

ii. expression / expression

The division operator, operates only on numeric types:

```
2/5.3
```

iii. expression + expression

The addition operator, both expressions have to be of the same type. If expressions are numbers, numerical addition will occur. If expressions are strings, the two string will be concatenated:

```
 3+4
"Hel" + "lo"
```

iv. expression - expression

The subtraction operator, both expressions have to be of numeric type:

```
3-1
```

(d) Coordinates

The coordinate expression has the form [expression, expression] where both expressions are numeric. Coordinates are pointers to a table cell and are used to retrieve and modify values of that cell. The first coordinate is the row, and the second coordinate is the column.

(e) Comparison Operators

The following expressions return 1 when true and 0 when false.

i. expression > expression

"greater than", both expressions must be of numeric type:

```
4>2
```

ii. expression < expression

"less than", both expressions must be of numeric type:

```
4<2
```

iii. expression >= expression

"greater than or equal to", both expressions must be of numeric type:

```
4>=2
```

iv. expression <= expression

"less than or equal to", both expressions must be of numeric type:

```
4<=2
```

v. expression == expression

"equals", expressions can either be strings or numbers and both expressions must be of the same type. If the expressions are strings, the interpreter will check - in sequence from left to right - if each ASCII character in the first string is the same as the corresponding character in the second string. If the expressions are numbers, the values will be compared for numerical equivalence:

```
"H" == "h"

4 == 4
```

vi. expression != expression

This expression is the same as the one directly above with the exception that the return values are opposite:

```
"H" != "h"

4 != 4
```

(f) Logical Operators

i. expression `&&` expression

Expressions can be of any type. This expression returns 1 if both expressions are non-zero, 0 otherwise. Lists are always non-zero, thus this logical operator will always return 1 on list comparisons.

ii. expression `||` expression

Expressions can be of any type. This expression returns 0 if both expressions are non-zero, 1 otherwise. The same principle applies to lists as in the `&&` operator.

(g) Assignment Operators

i. identifier = expression

   These expressions are grouped from right to left. The left expression is set to the value of the right expression.

ii. expression `->` coordinate

   This expression groups from left to right. The value of the left expression is placed into the coordinate expression on the right.

iii. string constant > expression

   The expression on the left can only be of type string. The expression on the right can be of any type. The expression of the left is called a key and the expression on the right is called value. This is used to assign keys to the values in a list. Note that this does not conflict with the ">" comparison operator because the comparison operator is not allowed to operate on any string expressions.

5. Statements

   (a) Normal Statement

   This is an expression followed by newline character.

   (b) Block Statement

   One or more normal statements that are indented one tab level from the surrounding sections of code

   (c) Conditional statements

   The basic conditional statement consists of the `if` keyword, followed by expression that evaluates 1 or 0, and statements that are on newlines and under at least one level of indentation from the `if` keyword. `if` block can optionally be followed by any number of `elseif` blocks that have the same form. If using `elseif`, the statement has to be followed by `else` block without any expressions attached. Abstract example:

```
if expression
```

```
    statements
elseif expression
    statements
else
    statements
```

(d) `return` statement

`return` exits out of a scope (block of code) and optionally returns a value and may only be defined inside the scope of a function definition. Abstract example:

```
return expression
```

The expression after `return` keyword is optional. If there is nothing to receive the return value, the value is discarded.

(e) `while` loop

```
while expression
    block
```

This is our basic looping mechanism.

(f) `foreach` loop

```
foreach identifier in list
    statement
```

`foreach` loop goes through the list and performs the same set of operations on every element in the list; it is convenient for iterating through all rows and columns.

(g) Function Declaration

A function declaration declares a block of code that can be executed by a function call. To define a function, start with the `func` keyword and follow it with an identifier to serve as the function's name, followed by an optional list of function arguments, each having an optional type declaration prefix. After this, place a block of code on the following line, indented one tab character. Parentheses surrounding the optional list of arguments are optional. Abstract example:

```
func identifier list
    statement
```

A more concrete example:

```
func name (arg1, arg2, arg3)
    arg1 - arg2 -> [10,4]
    arg3 -> [4,10]
```

6. Scope Rules

Blocks of code delimit “scope”—the segment of the program in which an identifier may be referenced. A variable at a lower level of tabular indentation is visible to all levels of indentation greater that its indentation level but the converse is not true; a variable at a given level of indentation may not be referenced at a level of indentation less than its own.

Functions must be declared at “global” scope, or no level of tabular indentation. A variable in a given scope may not be declared as having the same identifier as any identifier currently in scope.

7. Examples

```
num:var1 = 1
str:var2 = "one"
list:var = var1, var2

var1 -> 0,0
var2 -> 1,0
var -> 2,0
```

Output:

```
1   one   1   one
```

```
_col_head = m>"Monday", t>"Tuesday", w>"Wednesday", a>"Average", v>"Variance"
_row_head = d>"Daniel", v>"Vladimir", g>"George"


54, 70, 90 -> _row_next
66, 69, 98 -> _row_next
56, 69, 85 -> _row_next


foreach row in _rows
    _avg (row.m, row.t, row.w) -> row.a
    _var (row.m, row.t, row.w) -> row.v
```

Output:

```
                Monday      Tuesday     Wednesday   Average    Variance
Daniel          54          70          90          71.33      325.33
Vladimir        66          69          98          77.66      312.33
George          56          69          85          70         422
```

Scoping example:

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

Output:

```
5

4
```

Function Calls:

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

Equivalently:

```
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.")
```

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
```

8. Preprocessor

```perl
#!/usr/bin/perl -w
#
# This will be the main TableGen executable and preprocessor.
#
use strict;

my $prog = pop @ARGV;

SWITCH: for (@ARGV) {
        /--help/ && do { &getHelp; next };
        # More commandline options are to come. Follow
        die "Can’t process request because I do not know what $_ means. \nRun --help for help.\n";
}

sub help {
        print "TableGen expects the filename at the end of any commandline sequence: \n".
                "TableGen <options> [filename].\n";
        # Describe any options that you add here. New Print statement for each option.
        exit 0;
}


# Check input:
if (! -e $prog) {
        die "Error: The filename ($prog) of the program you wish to run does not exist.\n";
}

# Take program and shove it into memory:
open PROG, $prog or die "Could not open $prog: $!\n";

my @lines = <PROG>;

push @lines, "\n"; # Last element needs to end with a newline for
                  # the preprocessor.

# Run proprocess. Pass it as reference because we do not want
# to make copies of a potentially bit list of lines.
&preprocess (\@lines);



# Subroutines below...
sub preprocess {
        my $ref = shift;

        my $wscnt = 0;
        my $brak = 0;
        for (@{$ref}) {
                #Place square brackets around coordinates.
                if (/->\s*\d+\s*,\s*\d+\s*$/) {
                        s/(\d+\s*,\s*\d+)/\[$1\]/;
                }

                #Brackets at begining of tab increase.
                {
                        /^([\t ]+)/;
                        my @ws;
                        if (!$1) {
                                #@ws = 0;
                        } else {
                                @ws = split //, $1;
                        }

                        if (@ws > $wscnt) {
                                s/^/\{\n/;
                                $brak = $brak + 1;
                        } elsif (@ws < $wscnt) {
                                s/^/\n\}\n/;
                                $brak = $brak + 1;
                        } else {
                                # Nothing
                        }
                        $wscnt = scalar @ws;

                }

                print "$_";
        }

        if ($brak % 2) { # The number of brakets is odd.
                print "}\n";
        }
}
```

9. Grammar

```
grammar TableGen;

options {
output=AST;
k=2;
}

ASSIGNMENT_OP : '=' ;

BOOL_OPS : '<' | '<=' | '>=' ;

COLON : ':' ;

COMMA : ',' ;

DECIMAL_POINT : '.' ;

DIGIT : '0'..'9' ;

ELSE_KEYWORD : 'else' ;

ELSEIF_KEYWORD : 'elseif' ;

EQUALS  : '==' ;

FOREACH_KEYWORD : 'foreach' ;

FUNC_KEYWORD : 'func' ;

GREATER_THAN : '>' ;

IF_KEYWORD : 'if' ;

GREATER_THAN : '>' ;
```

```
IN_KEYWORD : 'in' ;

L_BRACE : '{' ;

L_BRACKET : '[' ;

L_PAREN : '(' ;

LETTER : 'a'..'z' | 'A'..'Z' ;

LINE_TERMINATOR
: ('\n' '\r') => '\n' '\r'
| '\n'
| '\r'
;

LIST_KEYWORD : 'list' ;

LOGICAL_OP
: '&&'
| '||'
;

MATH_OP
: '+'
| '-'
| '*'
| '/'
;

PUT_OP  : '->' ;

QUOTES  : '"' ;

R_BRACE : '}' ;

R_BRACKET : ']' ;

R_PAREN : ')' ;

RETURN_KEYWORD : 'return' ;

UNDERSCORE : '_' ;

VARTYPE : 'str' | 'num' ;

VALID_CHARS : '\u0020'..'\u007e' ;

WHILE_KEYWORD : 'while' ;

WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;} ;



Identifier : LETTER (LETTER | DIGIT | UNDERSCORE)* ;
```

```
String_literal : QUOTES LETTER* QUOTES ;

Number_literal
: DIGIT+ (DECIMAL_POINT DIGIT+)? ;




bool_compare : GREATER_THAN | BOOL_OPS ;

coordinate : L_BRACKET Number_literal COMMA Number_literal R_BRACKET ;

literal
: String_literal
| Number_literal
;




expression
: function_call
| literal
| location
;

function_call : Identifier L_PAREN (named_func_param | expression)* R_PAREN ;

named_func_param : Identifier COLON expression ;


location
: Identifier list_element_suffix?
| coordinate
;
list_element_suffix
: DECIMAL_POINT Identifier
;


/******************** STATEMENT TYPES *************************/
// don’t do separate declaration -- if not in scope, make it!
assignment : (VARTYPE COLON)? Identifier ASSIGNMENT_OP (expression | list_element_specification+);
list_element_specification : Identifier GREATER_THAN expression ;

put_into : expression PUT_OP location ;

if_else : IF_KEYWORD block elseif* (ELSE_KEYWORD block)? ;

elseif : ELSEIF_KEYWORD block ;

function_definition : FUNC_KEYWORD func_param* block ;
func_param : (VARTYPE COLON)? Identifier ;

for_loop : FOREACH_KEYWORD assignment bool_expression assignment block ;
```

```
bool_expression : expression bool_compare expression ;



/******************* HIGH-LEVEL **********************/
program : statement* ;

statement
: ( assignment
| put_into
| if_else
| function_definition
| for_loop
| RETURN_KEYWORD (Identifier | literal)
| block
) LINE_TERMINATOR
;

block
: L_BRACE statement* R_BRACE
;
```