Table Generation Language Reference Manual

Mohammad Haq

November 5, 2015

Contents

1 Introduction

There are many tools for importing a data set and analyzing it. A tool like Excel might be too heavyweight and clumsy for a repetitive task that imports a small data set and does a few data manipulations. On the other hand a general purpose programming language like Perl might be too complex with need for standard library calls to do the same task. Excel does offer a tabular graphical environment. Perl, or a similar general purpose programming language, would need library calls or a slew of print statements to generate a decent formatted output. The Table Generation Language offers a balance by providing an easy to use general purpose programming language with built-in features for importing, manipulating and printing tables. It also offers an Excel-like feature for doing what-if analysis.

2 Built-In Features

2.1 Pretty Printing

When a Table data structure is printed, the built-in print function prints the data in tabular form on standard out.

2.2 CSV Input

A CSV file can be loaded in from standard input into the global Input table.

2.3 What-if Analysis

From the command line, a comma separated argument string can be provided to the executable which ends up initializing a set of global variables before program execution. This makes it easy to do what-if analysis, by being able to change inputs from the command line.

3 Lexical Conventions

3.1 Tokens

For lexical analysis, the following token classes are defined: identifiers, keywords, constants, string literals, operators and separators. Comments are ignored as well as the remaining white space.

3.2 Comments

Comments are in the C-Style. The start and end of a comment are designated by the '/*' and '*/' tokens respectively. Comments do not nest and cannot be added inside a string. An example comment is:

```
/* This is a comment */
```

3.3 Identifiers

Identifiers are case-sensitive. The identifier may include numbers, letters, and underscores. An identifier that starts with a dollar sign ($), is treated as a global variable. All other characters in an identifier return a syntax error.

3.4 Keywords

The following are keywords used in the language, and may not be used as identifiers:

Table : Used for defining Table objects.

String : String data type

Int : Int data type

Boolean : Boolean data type

Float : Float data type

Func : For defining functions

if : Used in if-then-else control structure

then : Used in if-then-else control structure

else : Used in if-then-else control structure

for : Used in for loops

foreach : Used to iterate over an array

in : Used for array membership

not : Used for negation

Input : Global Input table

Output : Global standard output

true : Boolean true value

false : Boolean false value

item : current object in iteration

self : reference to object in method

3.5 Constants

3.5.1 Int

Ints contain a sequence of digits and an optional minus sign (-) to indicate a negative number. The integer can be an number between 2,147,483,648 and 2,147,483,647.

3.5.2 String

A String literal is a sequence of characters surrounded by double quotes. For example,

```
String hello := "Hello World";
```

3.5.3 Float

A floating point number consists of a string of digits with a decimal point. The digits after the decimal point make the fractional component of the floating point number. Notice that a decimal and fractional component are required. So the number “7.” is invalid, but “7.0” is valid.

3.5.4 Boolean

The only acceptable Boolean values are true and false.

4 Data Types

All data types begin with a capital letter and have the following methods: toString() and print(). The toString() method returns a String representation of the object. The print() method prints a representation of the object to standard out.

4.1 Primitive Data Types

The primitive data types are: Int, String, Float and Boolean. More details can be found in the Constants section.

4.2 Tables

Tables have two purposes in the TBL Language. First, it can be used like a struct/record. Hereafter we call this an *Object Table*. After a Table definition is provided, a type is created, and instances of the Table can be created using the constructor which has the same name as the type. These instances have a fixed size of memory allocated to them. The value for each member can be set or accessed.

The second purpose for the Table data structure is to be *Data table*. Like a record, the definition for this type of table, has a name and type for each member. Except, a short-form function definition, without parameters or a return type, is required for each member. The return type of the function has to be the same as the type for the member, and the implicit parameter is an instance of the Object Table passed in, which can be accessed by the *item* object. The *item* object remains in scope while each of the member functions are evaluated. While each member function is evaluated, all the previous members that have been evaluated are in scope. After a data table has been defined, a Data Table instance can be created by either passing in a record table or an array of record tables.

Both Object Tables and Data Tables have built in print methods.

4.3 Arrays

An Array is a data structure that contains a fixed block of memory containing a fixed number of consecutive Object Tables.

4.3.1 Declaring Arrays

An array can be declared by indicating the data type, the name, and the number of objects it can store. If there is no initializer the array will be zeroed out. The number of elements must be greater than zero. For example,

```
Person people[10];
```

4.3.2 Initializing Arrays

An Array can be initialized by providing it a list of values separated by commas. For example,

```
Person joe := Person( "Joe", 22 );
Person jane := Person( "Jane", 21 );
Person people[2] := { joe, jane };
```

4.3.3 Accessing Array

An Array element can be accessed by indexing into the array. The array index starts at 0. For example,

```
Person jane := people[1];
```

5 Meaning of Identifiers

The identifiers in the language can be variables, function names, table names, and members of objects. Identifiers also have scope.

5.1 Variables

A variable name acts as a pointer to a block of memory for an object.

5.2 Types

5.2.1 Integer

Integers are 32-bit signed integer values

5.2.2 Strings

Strings are objects with an array of ASCII characters.

5.2.3 Floats

Floats are represented as double-precision floating point numbers.

5.2.4 Boolean

The only values are *true* and *false*.

5.2.5 Tables

New data types can be created by defining new Object Tables.

6 Conversions

There are no implicit conversions from one type to another type.

7 Expressions

7.1 Postfix Expressions

7.1.1 Array References

An array reference consists of two components. The first part has type Array and the part inside the brackets has type integral. For example:

```
name[index];        /* Get */
```

7.1.2 Function Calls

A function call consists of a name followed by list of arguments separated by commas.

For example:

```
Person jim := getJimFrom( peopleArray );
```

7.1.3 Method Calls

a method call consists of an Object Table, a period, a name, and then a comma separated list of arguments.

For example:

```
Boolean isRetired := joe.isInRetirement();
```

7.1.4 Table Member References

Consists of accessing a member from a table. The table comes first, and then a dot and then an identifier representing the field.

7.2 Multiplicative Operators

The multiplicative operators are *, /, and %. These are left-associative.

7.3 Additive Operators

The additive operators are + and -. These are left-associative.

7.4 Relational Operators

The relational operators are <, >, <=, and >=. They are left-associative.

7.5 Equality Operators

Equality Operators are == and !=. They are left-associative.

7.6 Logical AND Operator

The && operator. This is left-associative.

7.7 Logical OR Operator

The || operator. This is left-associative.

7.8 Assignment Expression

The assignment operator is := (one colon and one equal sign). This is right-associative.

7.9 if-then-else Expression

First a predicate is evaluated to a boolean type in the if clause. If the value is true, then the 'then' clause is evaluated. If the predicate returns false, then the else expression is evaluated.

8 Declarations

Declarations specify the interpretation for each identifier.

8.1 Tables

A Table declaration requires providing a type name, and defining each of the member fields, which consist of a name, a return type, and an optional short-form function. The type name must begin with an upper-case letter. All member names must begin with a lowercase letter. The field definitions are separated by commas.

Only Data table definitions can have a functions for its members. The implicit return type for the function is the type of the member field. The function also has an implicit parameter called *item*, which is the current Object Table being examined. Since this function is a short-form function, the full function definition with a return type and formal parameter list is not required. The member functions can also access previously evaluated members for the current *item* as these members remain in scope until the next *item* is evaluated.

To create a Object Table do the following:

```
Table Person{
  name -> String,
  age -> Int
};
```

To create a Data Table do the following:

```
Table Employee(Person) {
  name -> String : item.name,
  age -> Int     : item.age,
  toRetirementAge -> Int : 65 - item.age
};
```

8.2 Functions

Functions must be declared before they are called. A function definition begins with the token 'Func', followed by the return type, the name, and then the comma separated parameter list. Each parameter must provide a type. Then a block of code that is to be executed is defined. Note that the last value evaluated is returned. For example,

```
Func Int minAge( Person a, Person b ) {
  if( a.age > b.age )
    then { a.age }
    else { b.age }
}
```

8.3 Methods

Methods are functions that operate on objects. They work the same way as functions except when they are called, the object (referenced as *self* is in scope. To define a method do the following:

```
Func Boolean Person::isInRetirement() {
  if( self.age > 65 )
    then { true }
    else { false }
}
```

8.4 Arrays

An array can be declared by indicating the data type, the name, and the number of objects it can store. If there is no initializer the array will be zeroed out. The number of elements must be greater than zero.

```
Person people[10];
```

9 Statements

9.1 Expression Statements

All expressions can be converted into a statements by adding a semicolon.

9.2 for

A for loop provides the programmer a controlled iterator, which consists of an initializer, a constraint, and a post iteration. The constraint has to evaluate to a Boolean. If the constraint is satisfied (i.e. is true), the expression inside the brackets is evaluated. For example,

```
Person people[10];
Int i;

for( i := 0; i < 10; i := i + 1 ) {
    people[i] := Person( "dummy", i );
}
```

9.3 foreach

Since iterating over containers is such a common task, a built in construct exists to iterate over an array. The keyword *item* is once again used as the object being examined. For example,

```
foreach people {
  item.print();
}
```

10 Scope

Input variables that begin with the $ tag have global scope. All functions and Data Tables have global scope. Variables declared inside functions have local scope. Variables not in a function have global scope. All Types and constructors have global scope.

Special scoping rules exist for member field functions in Data Tables. The *item* object is in scope while each of the member functions are evaluated. The previously evaluated field names also remain in scope.

11 Example

Here is an example program that calculates and prints the price of cashflows. Some features shown here are CSV Input table, global input arguments, Object Table definitions, Data Table definitions, and pretty-printing of a table.

```
$cat tradeCashflows.csv
number,start,end,pay,principal,rate,yearFraction
0,1sep15,30sep15,30sep15,1000000,0.02,0.083
1,1oct15,31oct15,30oct15,1000000,0.02,0.083
2,1nov15,30nov15,31nov15,1000000,0.02,0.083
3,1dec15,31dec15,31dec15,1000000,0.02,0.083

$cat price.tgl
/* This program prints a table with the price for each cashflow */

/* $rateMultiplier is a command line argument to do what-if-analysis */
$rateMultiplier := 1.0;

/* Cashflow Object */
Table Cashflow {
  number -> Int,
  startDate -> String,
  endDate -> String,
  payDate -> String,
  principal -> Float,
  rate -> Float,
  yearFraction -> Float
};
```

```
/* Discount Factors used to price
Float discountFactors[4] := { 0.99, 0.98, 0.97, 0.96 };

/* Data Table Definition */
/* Note: Cashflow is the type of the object to be iterated */
Table Price(Cashflow) {
  number -> Int : item.number
  price -> Float :
    Float df;    /* Discount Factor */

    df := discountFactors[item.number];

    /* Price of a cashflow is rateMultiplier * principal * rate * yearFraction */
    rateMultiplier * item.principal * item.rate * item.yearFraction * df }
};

/* Build Table */
Price results := Price(Input);

/* Print Table */
results.print();

$./price -input tradeCashflows.csv -args "$rateMultiplier:=1.0"
number   price
------   -----
1        1643.4
2        1626.8
3        1610.2
4        1593.6

$./price input tradeCashflows.csv -args "$rateMultiplier:=1.5"
number   price
------   ------
1        2465.1
2        2440.2
3        2415.3
4        2390.4
```