Table Generation Language

Mohammad Haq

October 1, 2015

Abstract

The Table Generation Language offers general purpose programming using Records and Functions with built in support for generating tables.

1 Motivation

Printing results to an output stream is one of the fundamental tasks of any program. A domain specific language that allows easy generation of formatted output can be a boon for organization and readability of data. Tables are a natural way for grouping and organizing data. Look at relational databases. They show that that keeping data in tables is a natural way to organize data. The table generation language has built in language constructs that make it easy to take data and arrays of data and print them out in table form. It is an imperative programming language with declarative features for the table generation.

2 Language Types

2.1 Primitive Types

The supported primitive types are integers and strings, which are boxed types.

2.2 Records

These are basically immutable structs. Records can be constructed using a Record constructor, which defines the fields that the record type has. The allowable fields are other Records and primitives.

2.3 Arrays

Arrays can only hold data of a single type.

2.4 Functions

Functions are first class objects of type Function. Every function evaluates to a value and returns it.

2.5 Tables

The program initiates with a single global Table. The Table is a special built-in type that acts as an associative array, and contains all the meta-data about the table (labels, columns, column-functions, formatting, etc). It also acts as a convenient associative array for persisting global data.

3 Language Features

3.1 Basic Control Structures

The three control structures offered are 'For' loops, 'If-Then-Else', and 'For-In'. 'For–In' is used for iterating over arrays.

3.2 String Representation

Every type has a string representation. The final result of a program is a String. It can be an empty string or an entire document. Strings are conveniently and efficiently concatenated under the hood.

3.3 Support for Tables

The basic output of a program is a 1x1 table, the Default Table, without any formatting. Using built in language constructs one can easily build more complicated tables, with different types of formatting, for output. More details about the Table generation constructs can be found in the syntax section of this document.

3.4 Global Associative Array

The DefaultTable acts as a global associative array for persisting objects. Each defined table also has its own global fields for the life of the table.

3.5 Column-function Coupling

Each column of the table is associated with a function. The DefaultTable has a single-function (the 'main' function) for the single column that it has. Newly defined tables will define a function for each column.

3.6 Table-Array Coupling

A special language construct exists that allows an array to be associated with a table such that the set of column-functions are applied to each element of the array.

3.7 Table-Generation

When the program is executed, starting with the DefaultTable, each column-function will be executed for the table. Of there is an array associated with the table, then the set of column-functions will be called for each item in the array. The strings will be concatenated, with the proper formatting, creating the string representation for the program, which contain string representations of tables.

3.8 Reference Counted

All objects are reference counted, including Integers and Strings.

3.9 Target Language

The output language for the compiler is C. C is chosen for its superior handling of strings and portability (compared to assembly language).

4 Syntax

4.1 Primitive Types

```
Integer name;      /* 32 bit int */
String name;       /* under the hood, c-style character array with size */
```

```
"I am a String"
```

Double quoted items are strings. The plus operator is a built in function for concatenating strings.

4.2 Records

```
Type Name( Type field1, Type field2, etc ).
```

This is the constructor for the record. All types start with a capital letter and all variables start with lower case letters.

4.3 Arrays

```
Array name(size,Type)

name[index] = object;  /* Set */
name[index];           /* Get */
```

4.4 Functions

```
Func Type name( Type arg1, Type arg2, etc )
{ expression }
```

4.5 Tables

```
Table<name>
```

This defines a new table.

```
Table<name><arrayName><arrayItem>
```

This defines a table with an array association.

```
<* Column Name *>{ column-function }
```

This defines a column with its associated column-function within a table

```
EndTable
```

This ends the table declaration

4.6 Comments

```
/* C Style comments */
```

4.7 Control Structures

4.7.1 if-then-else

```
if( expr ) then { expre } else {expr};
```

4.7.2 for loops

```
for( expr ; expr ; expr )
{
expr
}
```

4.7.3 for-in

```
for arrayItem in array
{
expr
}
```

5 Example Programs

5.1 Hello World

```
‘‘Hello World’’

------
OUTPUT
------
Hello World
```

5.2 Create and Print a Record

```
/* Create a new Record Constructor */

Type Person( String name, Integer age );

/* Create a new Person */

Person joe = Person( "Joe", 22 );

/* Create a 1x2 Table */

Table<"Person">
<* "Name" *>{ joe.name };
<* "Age" *>{ joe.age };
EndTable

------
OUTPUT
------

Person
Name   Age
Joe    22
```

5.3 Print an Array of Records

```
-------
PROGRAM
-------

/* Two new people */
Person mike = Person( ‘‘Mike’’, 25 );
Person jane = Person( ‘‘Jane’’, 22 );

/* New Array */
Array people(2,Person);

people[0] = mike;
people[1] = jane;

/* New Table */

Table<’’People’’><people><person>
<* ‘‘Name’’ *>{ person.name }
<* ‘‘Old Age’’ *> { person.age * 2 + 10 } /* Note Age expression */
EndTable

------
OUTPUT
------

People
Name    Old Age
Mike    60
Jane    54
```

5.4 GCD

```
-------
PROGRAM
-------

/* GCD Recursive Function */

Func Integer gcd( Integer a, Integer b )
{
  if( a == 0 )
  then { b }
  else { gcd(b % a, a) }
}

gcd( 10, 12 );

------
OUTPUT
------

2
```

6 Conclusion

This is a simple programming language that can generate pretty-printed data. Since Tables are built-in types, it would be pretty easy to add attributes, formatting, and other special functionality as extensions (outside the scope of this project). The example programs show both the generation and the printing of the data. The true power of the formatting becomes evident when the data (in Record form) is already available either through built-in functionality or libraries.