OctoScript

Group Members

Conor Gourley, conor.gourley@tufts.edu | Danielle Lan, hao-wei.lan@tufts.edu

Manish Aryal, manish.aryal@tufts.edu | Sinan Unan, sinan.unan@tufts.edu

Language Description

Inspiration:

The language is created from the inspiration from one of the past projects called "Joel", SQL queries, and ORM. We wanted to create a language that would be perfect to interact with one of the structured data formats, CSV files. Hence, drawing inspirations from SQL and ORM, one of the major features of the language is the inbuilt data structure called Table.

Description:

OctoScript is a strictly typed imperative domain-specific language designed to work with data in CSV file formats. The language will have a data structure similar to Tables in SQL. The language will have better performance, compile-time type checking, lambdas and loops, and modern syntax compared to SQL. The language offers the following types:

Description	**Type**
Integer	`int`
Strings	`str`
Boolean	`bool`
Float	`float`
Null	`NULL`
Lists	`list`
Table	`table`
Lambda & named functions	`(<A>) -> <B>, where A and B are data types`

Data Representation:

The data in a CSV file is represented as a Table, where single rows or columns are represented as a List. Within a Table, columns are homogenous in type, meaning that all values of a column must have the same type. NULL values are not restricted by types within a Table and can be used to represent empty cells in a CSV. The functions and lambdas are designed to be first-class objects. The planned built-in higher-order functions for querying and modifying data are listed below in the section **Syntax**.

Algorithm:

OctoScript is meant to describe algorithms for processing and manipulating 2D structured data. In our language, we will be focusing on the CSV data. The language takes inspiration from ORM where ORM deals with SQL and database, other structured data. OctoScript deals with CSV data and gets rid of SQL entirely.

Syntax

Arithmetic Operation

```
// Comments, Variable declaration and assignment looks just like C
int i = 10;
float f = 3.5;
```

Control Flow and Boolean Logic

```
while(true & true | !false){  // Familiar boolean operators
     break;     // OctoScript only supports while loops
     continue;  // Break and continue still exist
}
```

Functions & Lambdas

```
sum(int i, int j) -> int{
     return i + j; // Function syntax differs from that in C
}
// Lambdas functions are created similarly
(int x, int y)->int{return x + y;}
```

Table and List

The built-in `Table` functions include getCol, getRow, get, update, map, getHeader, setHeader, insert, copy. Built-in list functions include get, set, insert, map, copy. Examples of these built-in functions below.

INSERT

```
table_name.insert(axis, index, list);
```

Does: Inserts a given row or column to the given index, pushes all elements after it

Parameters:

- axis : {“row”, “col”}
- Index : int; default = table.count(axis)
- L : list of ‘a

Returns: table

Precondition: Throws error if list size not equal to the size of axis. If inserted as a new row, data types have to match every column

Example Syntax: `table_name.insert(“column”, 1, [1,3,5]);`

UPDATE

```
table_name.update(row_index, col_index, val);
```

Does: Updates a single cell determined by row and column

Parameters:

- Row_Index : int
- Col_Index : int
- Val : ‘a

Returns: table

Precondition: Throws error if index of row or column out of bounds, or if data type does not match column

Example Syntax: `table_name.update(1, 5, “new_val”);`

Reading File

```
// The read function loads in a CSV
table = read(“filename.csv”, [STRING, INT], header=False,
separator=<separator>);


table = read(“filename.csv”, [<data-type>, <data-type>, ...],
header=False,  separator=<separators>);
// The separator must be a string of length 1.
// The list [<data-type>, <data-type>, ...] must have a length that
is //     equal to the number of columns in the csv file.
```

Managing headers and column name

```
table.addColHeader("age", colId);
table.getCol("age"); //col and rows can be addressed by their name
```

SQL-style functions for querying and modifying data in table

FILTER
`table_name.filter(col_index, fn);`
Does: Filters the desired column based on the predicate function
Parameters:

- col_index : int
- fn : function

Returns: table
Preconditions: Throws error if col_index is not in bounds, or if data type of column is not the same as the predicate function's parameter type

SORT
`table_name.sort(col_index, order);`
Does: Sorts the table based on the given column
Parameters:

- col_index : int
- Order : {"asc", "des"}; default = "des"

Returns: table
Preconditions: Throws error if col_index is not in bounds

JOIN
Example Syntax: `table_name.join(table2, col1, col2);`
Does: Combines rows of two tables based on a common field in the given columns
Parameters:

- table2 : table
- col1 : int
- col2 : int

Returns: table
Preconditions: Throws error if col1 and col2 are not the same type, or are not in bounds

Function chaining

```
table=read("filename.csv", [STRING,INT], header=False,
separator=",");
table.sort(1,"des").update(0,0,"newVal").getRow(0).set(1,12);
// chaining these functions allows for complex queries/updates to be
```

```
// completed in a single line.
```