La Mesa

Language Reference Manual

COMS 4115: Programming Languages and Translators

Professor Stephen Edwards

Michael Vitrano Matt Jesuele
Charles Williamson Jared Pochtar

1. Introduction

La Mesa is a language intended to make working with tables in a programmatic setting easier. It is designed so that a programmer can use the full functionality of tables for retrieving, storing, and manipulating data without the additional hassle of working with SQL.

Relational databases form the core of many information technology systems that need to store large amounts of data in an efficient and organized manner. While interacting with these databases with SQL is useful for efficiently accessing a particular subset of data, there are many tasks that are either cumbersome or impossible. For example, applying a function to each record in a relational table is a multistep process, requiring the programmer to write a query to retrieve the relevant data, parse it, apply the function and return the new data to the table. Further, there is no way to use SQL constructs to check input validity using regular expressions.

La Mesa attempts to simplify the interaction with relational databases by making tables first-class objects. Users are able to load information from a database, manipulate it using the familiar imperative programming paradigm and optionally commit the data back to the database.

2. Basic syntax

La Mesa is intended to be a counterpart to Java; i.e. it uses syntax derived heavily from Java and compiles into Java. However, it is distinct from Java, so that a La Mesa program and a Java program are not confused. To this end, La Mesa borrows conventions from other imperative languages, while keeping the flavor of imperative programming intact. A La Mesa program is a series of tokens; the type and nature of those tokens are described as follows.

There are five types of tokens: comments, keywords, operators, identifiers, and literals. Spaces, newlines, tabs, and any other exotic nonprinting characters constitute whitespace; they are necessary to separate tokens but are not recognized in any other way by the compiler.

2.1 Comments

The characters // introduce a comment; a newline terminates a comment. There are no multi-line comments.

2.2 Keywords

The following are keywords in La Mesa and are not to be used otherwise.

int	null	string	float	for	
if	else	while	where	table	
load	func	type	import	many	
one					

2.3 Identifiers

An identifier is anything beginning with a letter and continuing with any combination of letters, numbers, and underscores ‘_’. It can be used to describe variables, tables or functions.

2.4 Literals

Any sequence of digits is an integer literal. Any set of two sequences of digits separated by a decimal point '.' is a floating-point literal. Any sequence of characters enclosed by double quotes is a string literal.

A list delimited by parentheses is a row literal, eg ("Matt","654 W 123 St","New York").

3. Primitive Datatypes

Internally, values in La Mesa are treated as tables. In other words, while a user might type in the expression "x = 5 + 4," the La Mesa program treats the expression as "the variable x describing a table is assigned to hold the table containing the singleton value 5 added to the table containing the singleton value 4. This is done to ensure that data will not be left in an indeterminate form during program execution, it is also done to satisfy type constraint consistency. All of this will be invisible to the programmer; nevertheless certain operators acquire different powers and/or necessitate different restrictions under this policy. This is similar to the way that a database query engine such as MySQL would treat data.

Boolean Operators

== (boolean equality test), >= (boolean greater than or equal to), <= (boolean less than or equal to), != (boolean not equals), **&&** (boolean AND), || (boolean OR), **!** (boolean NOT). La Mesa uses standard boolean operators that work on strings, integers, and floats, and evaluates to either 0 (false) or something other than 0 (true). There is no special boolean data type.

Integer/Floating Point Operations

+ (addition), **-** (subtraction), ***** (multiplication), **/** (division), ****** (exponentiation). The La Mesa compiler defines all of the usual operators that work on both integer and floating point numbers. One of the operands is restricted to be a singleton; otherwise the compiler will throw an error.

String Operators

^ (string concatenation), **${**expression**}** (string interpolation). The La Mesa compiler defines two operators on strings, concatenation and interpolation. For concatenation, "string1" ^ "string2" = "string1string2". La Mesa defines interpolation as in the code fragment:

```
string suffix = "4115"
string dept_code = "COMS${suffix}"
```

In the above fragment, the variable named 'dept_code' is assigned the string "COMS4115". The value inside the curly braces must be a valid identifier, or else it will cause an error.

Table Operators

| (pipe operator) -- The pipe operator is used to iterate over table rows. La Mesa expects a valid table name to the left, and an existing or new identifier for the row to the right of the operator.

For example, the following statment iterates over the table “Classes” to find the row with id “COMS4115”:

```
classes | class {
        if class.class_id = “COMS4115” {
                //matched class_id
        }
}
```

<< (push operator) -- The push operator adds a new row to a table. The new row must have the same width and column types as the table it is being added to; otherwise a compiler error results.

>> (return operator) – takes now left hand side, adds the pushes the right hand side onto the table that will be returned. Eg:

```
func many:user explicitReturn(many:user in) {
        u|in {
                n = copy(u)
                n.name = n.name ^ “is cool”
                ret << n
        }
        return ret
}
```

becomes

```
func many:user implicitReturn(many:user in) {
        u|in {
                n = copy(u)
                n.name = n.name ^ “is cool”
                >> n
        }
}
```

:: (append operator) -- The append operator adds a new column to a table. The new column must have the same height as the table it is being added to; otherwise a compiler error results.

: (table attribute datatype resolution operator) -- Because La Mesa is strongly typed, this operator is used to specify what type of data the program is to expect. If the type resolved by the operator is not the type already in existence for the attribute, then the compiler throws an error.

. (table column access operator) -- This operator accesses a particular column within the table. For example, users.name returns the ‘name’ column of the ‘users’ table. A compiler error is produced if the user attempts to reference a column which is not in the table.

[] (table row access operator) -- This operator accesses a particular row within a table. For example, if the user wants to access the third row of a table, the user would use the syntax:

```
if (user.user_id[3] = “Matt”) {
        //user.user_id matched to “Matt”
}
```

A runtime error is produced if the user attempts to reference a row index outside the table’s range.

& (table schema copy operator) -- This operator expects a single existing table identifier, and it returns an empty table containing the exact same schema as the existing table.

3. Static typing policy and user-defined types

La Mesa is strongly- and statically-typed. Because this requires that the compiler know the schema of every table, any external databases upon which a La Mesa program depends must be provided to the compiler during compilation.

Function prototypes must specify both the type and quantity of each argument, as well as the return value. The type must be one of the primitive types, a table, or a user-defined type, as discussed below. For example, the following function takes a single integer and doubles it:

```
func one:int doubleNumber (one:int a) {
   return a + a
}
```

When the quantity is omitted, “one” is assumed, so the above is equivalent to the following:

```
func int doubleNumber (int a) {
        return a + a
}
```

In contrast, this function takes a column of integers, and returns a column of the same length with each integer doubled:

```
func many:int doubleColumn (many:int a) {
        table a = (int)
        a | cell {
                a << cell + cell
        }
}
```

User-defined types are created with the type keyword:

```
type user = {
        name: string
        age: int
        address: string
}
```

Now we can define a function which expects to recieve one:user; that is, a table with a single data row comprised of a string column named ‘name’, an integer column named ‘age’, and a string column named ‘address’:

```
func string userString(user u) {
   return “Name: ${u.name}, Age: ${u.age}, Address: ${u.address}”
}
```

4. The table

For this section, we will assume that the SQLite database file ./db.sqlite3 is provided to the compiler along with the program below, and contains a table titled “users”:

name	**address**	**city**
Joe	2320 Broadway	New York
Marcus	1234 Sunset Blvd	Los Angeles
Alex	3664 N. Halsted Blvd	Chicago

Tables in La Mesa are loaded with the load keyword:

```
load users  //now users is a table!
```

Once you have a table object, you can slice it up in various ways:

```
//we can get a single row (that is; a user)
print(users[1]) //prints “name: Joe, address: 2320 Broadway, ..."

//we can get a single column (an attribute)
print(users.name) //prints “Joe\nMarcus\n...”
```

Both users[1] and users.name are themselves tables.
users[1] is a table with one data row and a number of columns equal to the number of columns in users:

Name	**address**	**city**
Joe	2320 Broadway	New York

users.name is a table with one data column and a number of rows equal to the number of rows in users:

name
Joe
Marcus
Alex

//we can iterate over the table rows concisely with the pipe operator
users | user print(user) //prints “Joe, 2320 Broadway...\nMarcus, 1234 Sun...”

//Joe and Marcus move to London
users{name=”Joe” || name=”Marcus”}.city = “London”

4.2 Creating New Tables:
Users need to specify the type signature of the table. They do this by using the attribute datatype resolution operator **:**. The following line of code creates a new table named ‘newTable’ of type user:

table:user newTable

5. Expressions and statements

There are 2 kinds of statements: block statements and expression statements. Block statements are a series of expression statements separated by newlines and enclosed in braces. For example:

```
{
        expression-statement
        expression-statement
}
```

All other statements are expression statements. These are defined as:

```
//value is of selected stmnt
if predicate: statement (elseif pred: statement)* (else: statement)?
while (predicate) statement                    //value of the last statement
id | table statement                           //value of the last statement
table = expression-statement                   //value of the table
```

predicate is an expression of type bool.

Function Statements

Functions are the primary way of grouping code in La Mesa. Before a function can be invoked, it must be defined. A function is defined using the func keyword. After the func keyword, a signature of types is specified, specifying a table with a sequence of certain column value types to be returned by the function. All arguments are passed by reference. The identifier of the function follows, followed by a set of parentheses "()" and an optional set of typed arguments. The body of a function is enclosed in curly braces, in accordance with the block scoping convention. A code example follows.

```
type user1 {
        name_id:        int
        value:  int
}

func many:user1 doubleColumn (many:int a) {
        table b = (int, int)
        a | cell {
                b << (cell + cell, cell * cell)
        }
}
```

The above function takes in a column of integers (there are "many" ints), and creates a table with two columns, the first column containing the integer doubled, and the second column containing the integer squared. The function doubleColumn would be invoked on a table that would be receiving the finished result.

6. Scope

La Mesa is statically scoped and uses block scoping rules. Variables declared outside of any function are visible everywhere. Variable shadowing is not allowed; all names must be unique inside their scope.

Blocks are delimited by '{' and '}'. These braces are required by function definitions, if/else constructs, loop bodies, and pipe iteration, and cannot be omitted even if they enclose only a single statement. This is contrary to the behavior of many popular languages, including C and Java, and is intended to promote clarity about where blocks begin and end.

External source files can be linked in to a La Mesa program using the import statement. Similar to the C preprocessor, this is a simple text substitution, so care must be used to ensure that identifier names are unique.