La Mesa

A language for manipulating relational database tables

Matthew Jesuele, Jared Pochtar,
Charles Williamson, Michael Vitrano

Programming Languages and Translators, Fall 2011
Project Proposal

Introduction

La Mesa is a language designed to ease the use and manipulation of relational databases by allowing the programmer to view them in the object-oriented paradigm. This is accomplished by compiling La Mesa code into Java code, with the appropriate SQL API calls, that will automatically translate object-oriented data structures into tables that can be stored in the database. Similarly, the language will be able to query the database in a more natural way by allowing programmers to search for objects that meet certain criteria rather than manually querying across numerous tables to find a desired data point. This will allow programmers to forget the implementation details for storing their data in a relational database and permit them to focus on doing interesting things with the data.

Built-in Datatypes

The selection of datatypes in La Mesa is drawn directly from SQL, with some additional abstract collections to ease certain algorithmic tasks.

Primitive types in La Mesa directly correspond to a set of datatypes supported by a wide variety of SQL servers, including varchar, date, time, datetime, float/double. These can be extended by the programmer to provide additional functionality; for example, an email field could be created by adding format validation to a varchar field.

In addition to these primitive datatypes, the table datatype corresponds to an entire table, record to a single row, and attribute to a single column.

Example:

```
load “users” from /path/to/database as users
users # is a table
users[1] # is a User (extends record)
users.name # is an attribute
users{
        groups : array(2)        # creates a ‘groups’ column which takes an
                                 # array of size 2 as a value
}
```

Hypothetical Use-Case and Example Code

Our example user is a professor at a prestigious university. His department maintains a database of classes offered in the current semester; each class is a row in the database table, and the columns correspond to the instructor, the start and end times, the classroom, the course number, course name, and a description.

Our professor would like to add another class to his schedule. Classes are 75 minutes long and the professor is a caffeine fiend, so he would only like to schedule a class during a free block of time at least two hours long, giving him enough time to wait in the nigh-interminable line at the local cafe. Our professor can write a program which will output all such blocks (our professor trusts the department to have scheduled at least one class for him, and to not have scheduled any overlapping classes):

```
load "classes"
        where instructor is "Stephen Edwards"
        from /path/to/database
        as classes  # classes is still a table, but it doesn't contain the full
                    # data set

starttimes = classes.start_time.to_a  # is of type array
endtimes = classes.end_time .to_a  # is of type array

starttimes.popHead!  # Bang methods modify the structure in-place, in this case removing
                     # the head element of starttimes.
endtimes.popTail!    # And here, the tail element of endtimes.

for each gap_start, gap_end in endtimes, starttimes:
  if gap_end - gap_start is more than 2.hours
  and gap_start.hour > 9.am  # am is an integer method in the small standard
                             # library; in this case, 9.am returns a time
                             # object corresponding to 09:00
  and gap_end.hour < 9.pm:
    printn "There is a block of free time from #{gap_start} to #{gap_end}!"
```

Other Features

- Automatic filtering of user input to prevent SQL injections
- Small standard library populated with methods to ease conversion to and from SQL datatypes
- Possible inclusion of coroutines, for working efficiently with large data sets.