CS61A Lecture 32

Friday, November 15, 2019

Announcements

- Homework 9 is due Thursday 11/21.
- Scheme Recursive Art contest entries are due Monday 12/2.

Declarative Programming

Today, we are moving to the last part of the course, which means we are leaving Scheme behind. We will learn an entirely new programming language. The idea is not just to learn new languages, but new styles. We learned about object-oriented programming with Python, functional programming with Scheme, and today we will learn about declarative programming.

Database Management

Database systems are used everywhere. They are used to maintain information over a large number of entries, for examples, a database of all students in Berkeley.

Databasement management systems (DBMS) are important, heavily used and interesting! A table is a collection of records, which are rows that have a value for each column.

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

A table has rows and columns. A column has a name and a type. And a row has a value for each column.

The standard programming language for a database is the Structured Query Language (SQL). SQL is by some measures, perhaps the most widely used programming language in the world, because many people in all fields use it, not just programmers.

What is Declarative Programming?

SQL is what we call a **declarative** programming language.

Declarative Language	Imperative Language
Examples include SQL and Prolog	Examples include Python and Scheme
A “program” is a description of the desired result	A “program” is a description of computational processes

Declarative Language	Imperative Language
The interpreter figures out how to generate that result	The interpreter carries out the execution/evaluation rules

We will **not** be building an interpreter that figures out what's the fastest way to do something. If you're interested, Berkeley offers CS186, a class that does just that.

Here is an example of SQL code:

```
create table cities as
    select 38 as latitude, 122 as longitude, "Berkeley" as name union
    select 42            , 71              , "Cambridge"        union
    select 45            , 93              , "Minneapolis"
```

This code tells SQL to create a table just like the one above. We don't need to rewrite latitude and longitude in each select statement because we already did above. Awesome!

```
select "west coast" as region, name from cities where longitude >= 115 union
select "other"               , name from cities where longitude < 115;
```

This creates a new table that sorts the locations above into locations on the West Coast, and those that aren't.

SQL Overview

The SQL language is an ANSI and ISO standard, which is the only officially recognized SQL implementation. However, many database management systems will implement custom variations that include specialized features.

However, there are some basic features that will likely work across any implementation:

- A `select` statement creates a new table, either from scratch or by projecting a table
- A `create` table statement gives a global name to a table
- Lots of other statements exist: `analyze`, `delete`, `explain`, `insert`, `replace`, `update`, etc.
- Most of the important action is in the `select` statement, as such we will primarily be focusing on that in CS61A.

SQLite

In this class, we will be using a subset of SQL called SQLite. Download version 3.8.3 or later. You can also use SQLite on our online interpreter: code.cs61a.org/sql

Selecting Value Literals

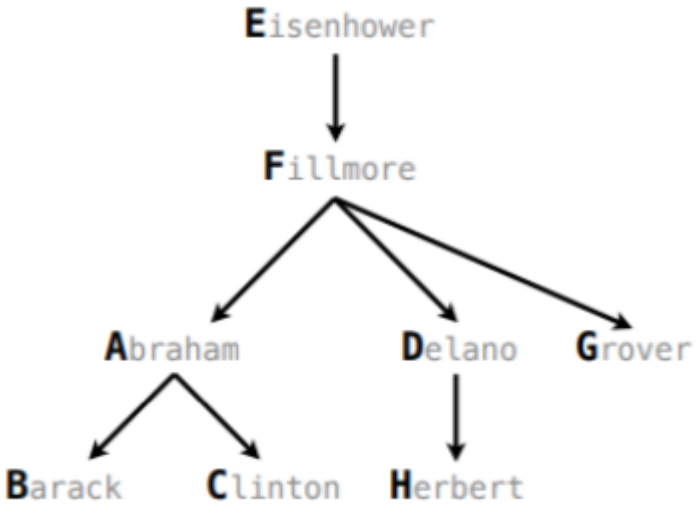
For our demos, we will be manipulating a databse of Professor DeNero's (imaginary) dogs! They're named after presidents (randomly! No political meaning behind it whatsoever).

A `select` statement always includes a comma-separated list of column descriptions. A column description is an expression, optionally followed by `as` and a column name.

```
select [expression] as [name], [expression] as [name];
```

Selecting literals creates a one-row table. The `union` of two select statements is a table containing the rows of both of their results.

```
select "delano" as parent, "herbert" as child
select "abraham" , "barack" union
select "abraham" , "clinton" union
select "fillmore" , "abraham" union
select "fillmore" , "delano" union
select "fillmore" , "grover" union
select "eisenhower" , "fillmore";
```



Naming Tables

SQL is often used as an interactive language. The result of a select statement is displayed to the user, but not stored. A `create table` statement gives that result a name, so that we can access it later on:

```
create table [name] as [select statement];
```

So to give a name to our previous table:

```
create table parents as
    select "delano" as parent, "herbert" as child union
    select "abraham" , "barack" union
    select "abraham" , "clinton" union
    select "fillmore" , "abraham" union
    select "fillmore" , "delano" union
    select "fillmore" , "grover" union
    select "eisenhower" , "fillmore";
```

And here's that table as SQL would show it:

parent	child

parent	child
abraham	barack
abraham	clinton
delano	herbert
fillmore	abraham
fillmore	delano
fillmore	grover
eisenhower	filmore

Projecting Tables

A `select` statement can specify an input table using a `from` clause.

A `subset` of the rows of the input can be selected using a `where` clause.

An ordering over the remaining rows can be declared using an `order by` clause.

Column descriptions determine how each input row is projected to a result row.

What order are these statements executed in? That's not up to you! That's up to the interpreter to figure out which is fastest.

```
select [expression] as [name], [expression] as [name]

select [columns] from [table] where [condition] order by [order];
```

So for our previous table:

```
select child from parents where parent = "abraham";
```

This statement returns a one-column table with the column "child" containing all the children whose parents were "abraham". It does not show the "parent" column.

child
barack
clinton

Or we could do this:

```
select parent from parents where parent > child
```

This statement returns a one-column table with the column "parent" containing all the parents who have children with "lesser" names (alphabetically) than them.

parent
filmore
filmore

And you can show the full table with:

```
select * from [table];
```

Arithmetic

You can even perform arithmetic in SQL. In a `select` expression, column names evaluate to row values. Arithmetic expressions can combine row values and constants.

```
create table lift as
    select 101 as chair, 2 as single, 2 as couple union
    select 102         , 0          , 3
    select 103         , 4          , 1
```

Names in SQL that you can access are names of tables, or names of columns in tables.

```
select chair, single + 2 * couple as total from lift;
```

chair	total
101	6
102	6
103	6

> **Discussion Question**
> As it turns out, every number can be represented in terms of powers of two. You will start with a table `ints` that describes how to sum powers of 2 to form various integers.
>
> ```
> create table ints as
>     select "zero" as word, 0 as one, 0 as two, 0 as four, 0 as eight union
>     select "one"         , 1       , 0       , 0        , 0 union
>     select "two"         , 0       , 2       , 0        , 0 union
>     select "three"       , 1       , 2       , 0        , 0 union
>     select "four"        , 0       , 0       , 4        , 0 union
>     select "five"        , 1       , 0       , 4        , 0 union
>     select "six"         , 0       , 2       , 4        , 0 union
>     select "seven"       , 1       , 2       , 4        , 0 union
>     select "eight"       , 0       , 0       , 0        , 8 union
>     select "nine"        , 1       , 0       , 0        , 8;
> ```

1. Write a `select` statement for a two-column table of the `word` and `value` for each integer.
2. Write a select statement for the `word` names of the powers of two.

Solutions

1. `select word, one + two + four + eight as value from ints order by value ;`
2. `select word from ints where one + two/2 + four/4 + eight/8 = 1 ;`

There were maybe easier ways to do the things we did above, especially the second question: we could've just pulled out the names of the column headings, just not with a `select` statement. We will cover more of them as we go further in the class.

SQL is built to be resillient in the sense that it expects users to forget things, so it tries to fill in info you don't specify in certain cases. This can be good, but may also lead to unexpected behavior. Be careful!