# MACROS REVIEW & INTRO TO SQL Meta

## COMPUTER SCIENCE MENTORS 61A

November 25–November 29, 2024

**Recommended Timeline**

- Macros Intro: 10 minutes
- Q1. `and-odds-thanksgiving`: 8 minutes
- SQL mini-lecture: 10 minutes
- Q1 (Mentors): 10 minutes
- Q2 (Circus): 20 minutes
- Q3 (Fish): 20 minutes

**Teaching Tips**

- Before we jump into macros, it is very important to ensure that your students understand quasiquotation.
- Draw out box-and-pointer diagrams to show how the expressions in macros are being stored when the operands are unevaluated, if needed.
- If you would like a quick refresher on how to think about macros, please refer to this guide.

## 1 Macros Review

1. It's Thanksgiving, and we need to ensure the odd-indexed dishes on the table pass the "taste test"! Write a macro, `and-odds`, which takes in a list of dishes, `exprs`, and evaluates to a true value only if all of the odd-indexed elements of `exprs` evaluate to true values. If any of the odd-indexed elements evaluate to false, `and-odds` should return false.

   Can you help us determine which dishes make the cut for the Thanksgiving feast?

```
; doctests
scm> (and-odds '((= 10 10)))
#t
scm> (and-odds '((= 1 2)))
#f
```

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

```
scm> (and-odds '(#f #t #t))
#f
scm> (and-odds '((< 5 3) (= 5 5)))
#f
scm> (and-odds '((> 3 2) (< 5 0) (= 5 5)))
#t
scm> (and-odds '((< 1 5) (< 5 2) (< 3 5) (< 5 3) (< 4 5)))
#t
scm> (define a (list 1 #f 3))
a
scm> (and-odds a)
3
```

```
(define-macro (and-odds exprs)
    `(if _____________________________________________________
         _____________________________________________________
         _____________________________________________________
    )
)
```

```
(define-macro (and-odds exprs)
    `(if (> (length ,exprs) 2)
         (and (car ,exprs) (and-odds (cdr (cdr ,exprs))))
         (eval (car ,exprs))
    )
)
```

### Teaching Tips

- Quickly review boolean evaluation and boolean operations **`and`** and **`or`**.
- Begin with the intuition behind the recursive call. Have them consider an arbitrarily long list and guide them to the intuition behind the call on `(cdr (cdr exprs))`.
- If they add a base case for nil, have them consider an odd-length list.

# 2 SQL

SQL (Structured Query Language) is a declarative programming language that allows us to store, access, and manipulate data stored in databases. Each database contains tables, which are rectangular collections of data with rows and columns. This section gives a brief overview of the small subset of SQL used by CS 61A; the full language has many more features.

## 2.1 Creating Tables

### 2.1.1 SELECT

`SELECT` statements are used to create tables. The following creates a table with a single row and two columns:

```
sqlite> SELECT "Adit" AS first, "Balasubramanian" AS last;
Adit|Balasubramanian
```

`AS` is an "aliasing" operation that names the columns of the table. Note that built-in keywords such as `AS` and `SELECT` are capitalized by convention in SQL. However, SQL is case insensitive, so we could just as easily write `as` and `select`. Also, each SQL query must end with a semicolon.

### 2.1.2 UNION

`UNION` joins together two tables with the same number of columns by "stacking them on top of each other". The column names of the first table are kept.

```
sqlite> SELECT "Adit" AS first, "Balasubramanian" AS last UNION
...> SELECT "Gabe", "Classon";
Adit|Balasubramanian
Gabe|Classon
```

### 2.1.3 CREATE TABLE

To create a named table (so that we can use it again), the `CREATE TABLE` command is used:

```
CREATE TABLE scms AS
    SELECT "Adit" AS first, "Balasubramanian" AS last UNION
    SELECT "Gabe", "Classon";
```

It is nice to note that SQL syntax is supposed to mirror English grammar closely so that it is natural to understand.

The remaining examples will use the following `team` table:

```
CREATE TABLE team AS
    SELECT "Gabe" AS name, "cat" AS pet, 11 AS birth_month UNION
    SELECT "Adit",         "none",       10 UNION
    SELECT "Alyssa",       "dog",         4 UNION
    SELECT "Esther",       "dog",         6 UNION
    SELECT "Maya",         "dog",         3 UNION
    SELECT "Manas",        "none",       11;
```

## 2.2 Manipulating other tables

We can also write `SELECT` statements to create new tables from other tables. We write the columns we want after the `SELECT` command and use a `FROM` clause to designate the source table. For example, the following will create a new table containing only the `name` and `birth_month` columns of `team`:

```
sqlite> SELECT name, birth_month FROM team;
Adit|10
...
Maya|3
```

Note that the order in which rows are returned is undefined.

An asterisk `*` selects for all columns of the table:

```
sqlite> SELECT * FROM team;
Adit|none|10
...
Maya|dog|3
```

This is a convenient way to view all of the content of a table.

We may also manipulate the table columns and use `AS` to provide a (new) name to the columns of the resulting table. The following query creates a table with each teammate's name and the number of months between their birth month and June:

```
sqlite> SELECT name, ABS(birth_month - 6) AS june_dist FROM team;
Adit|4
...
Maya|3
```

### 2.2.1 `WHERE`

`WHERE` allows us to filter rows based on certain criteria. The `WHERE` clause contains a boolean expression; only rows where that expression evaluates to true will be kept.

```
sqlite> SELECT name FROM team WHERE pet = "dog";
Alyssa
Esther
Maya
```

Note that = in SQL is used for equality checking, not assignment.

### 2.2.2 `ORDER BY`

`ORDER BY` specifies a value by which to order the rows of the new table. `ORDER BY ...` may be followed by `ASC` or `DESC` to specify whether they should be ordered in ascending or descending order. `ASC` is default. For strings, ascending order is alphabetical order.

```
sqlite> SELECT name FROM team WHERE pet = "dog" ORDER BY name DESC;
Maya
Esther
Alyssa
```

## 2.3 Joins

Sometimes, you need to compare values across two tables—or across two rows of the same table. Our current tools do not allow for this because they can only consider rows one-by-one. A way of solving this problem is to create a table where the rows consist of every possible combination of rows from the two tables; this is called an **inner join**. Then, we can filter through the combined rows to reveal relationships between rows. It sounds bizarre, but it works.

An inner join is created by specifying multiple source tables in a `WHERE` clause. For example, `SELECT * FROM team AS a, team AS b;` will create a table with 36 rows and 6 columns. The table has 36 rows because each row represents one of 36 possible ways to select two rows from `team` (where order matters). The table has 6 columns because the joined tables have 3 columns each. We use `AS` to give the two source tables different names, since we are joining `team` to itself. The columns of the resulting table are named `a.name, a.pet, a.birth_month, b.name, b.pet, b.birth_month`.

For example, to determine all pairs of people with the same birth month, we can use an inner join:

```
sqlite> SELECT a.name, b.name FROM team AS a, team AS b WHERE a.name < b.name
   AND a.birth_month = b.birth_month;
Gabe|Manas
```

We include `a.name < b.name` to ensure that each pair of people is only listed once. Otherwise, we would get both `Gabe|Manas` and `Manas|Gabe`.

## 2.4 Aggregation

Aggregation uses information from multiple rows in our table to create a single row. Using an aggregation function such as `MAX`, `MIN`, `COUNT`, and `SUM` will automatically aggregate the table data into a single row. For example, the following will collapse the entire table into one row containing the name of the person with the latest birth month:

```
sqlite> SELECT name, MAX(birth_month) FROM team;
Manas|11
```

Note that there are multiple rows with the largest birth month. When this happens, SQL arbitrarily chooses one of the rows to use.

The `COUNT` aggregation function collapses the table into one row containing the number of rows in the table:

```
sqlite> SELECT COUNT(*) FROM team;
6
```

### 2.4.1 `GROUP BY`

`GROUP BY` groups together all rows with the same value for a particular column. Aggregation is performed on each group instead of on the entire table. There is then *exactly one row* in the resulting table for each group. As before, type of aggregation performed is determined by the choice of aggregation function. The following gives, for each type of pet, the information of the person with the earliest birth month who has that pet:

```
sqlite> SELECT name, pet, MIN(birth_month) FROM team GROUP BY pet;
Gabe|cat|11
Maya|dog|3
Adit|none|10
```

I like to emphasize to my students that there is always exactly one row in the resulting table for each group.

#### 2.4.2 `HAVING`

Just as `WHERE` filters out rows, `HAVING` filters out groups. For example, the following selects for all types of pets owned by more than one teammate:

```
sqlite> SELECT pet FROM team GROUP BY pet HAVING COUNT(*) > 1;
dog
none
```

### 2.5 Syntax

The clauses of a `SELECT` statement are written in the following order:

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ...;
```

However, the actual processing steps differ slightly. Notably, all row-by-row filtering (via the `WHERE` clause) occurs *before* aggregation (via the `GROUP BY` clause), ensuring that only filtered rows are aggregated.

**Teaching Tips**

- Make sure to emphasize the difference between `HAVING` and `WHERE`. Students are often confused by the similarity of these two clauses.
- `code.cs61a.org` has an interactive SQL terminal that lets you see visually the rows you extract from a given query and also the logical order in which queries are processed. This is a super useful tool.
- When I was learning about SQL, I had many confusions about edge cases. When you use `code.cs61a.org`, try asking your students if there is anything they would like you to type in the interpreter to see how it handles things.

1. CSM 61A Content Team wants to put together an end of the year party for all its mentors (and some special guests, too!) themed around everyone's favorite things! Examine the table, `mentors`, depicted below, and answer the following questions.

| Name | Food | Color | Editor | Language | Animal |
|---|---|---|---|---|---|
| Alyssa | Pork Bulgogi | Navy Blue | Vim | Java | Sea Otter |
| Vibha | Pasta | Teal | VSCode | Java | Naked Mole Rat |
| Adit | Protein Bar | Black | Vim | Python | Gorilla |
| Esther | Goldfish | Pastel Pink | VSCode | Python | Bunny |
| Aiko | Fries | Sky Blue | VSCode | Java | Cat |
| Aurelia | Dumpling | Pastel Yellow | VSCode | Python | Panda |
| Kaelyn | Popcorn | Blue | VSCode | Java | Panda |

(a) Create a new table `mentors` that contains all the information above. (You only have to write out the first two rows.)

```
CREATE TABLE mentors AS
  SELECT 'Alyssa' as name, 'Pork Bulgogi' as food, 'Navy Blue' as
     color, 'Vim' as editor, 'Java' as language, 'Sea Otter' as animal
     UNION
  SELECT 'Vibha', 'Pasta', 'Teal', 'VSCode', 'Java', 'Naked Mole Rat'
     UNION
  SELECT 'Adit', 'Protein Bar', 'Black', 'Vim', 'Python', 'Gorilla'
     UNION
  SELECT 'Esther', 'Goldfish', 'Pastel Pink', 'VSCode', 'Python',
     'Bunny' UNION
  SELECT 'Aiko', 'Fries', 'Sky Blue', 'VSCode', 'Java', 'Cat' UNION
  SELECT 'Aurelia', 'Dumpling', 'Pastel Yellow', 'VSCode', 'Python',
     'Panda' UNION
  SELECT 'Kaelyn', 'Popcorn', 'Blue', 'VSCode', 'Java', 'Panda';
```

(b) Write a query that has the same data, but alphabetizes the rows by name. (Hint: Use `order by`.)

```
Adit|Protein Bar|Black|Vim|Python|Gorilla
Aiko|Fries|Sky Blue|VSCode|Java|Cat
Alyssa|Pork Bulgogi|Navy Blue|Vim|Java|Sea Otter
Aurelia|Dumpling|Pastel Yellow|VSCode|Python|Panda
Esther|Goldfish|Pastel Pink|VSCode|Python|Bunny
Kaelyn|Popcorn|Blue|VSCode|Java|Panda
Vibha|Pasta|Teal|VSCode|Java|Naked Mole Rat
```

```
SELECT *
FROM mentors
ORDER BY name;
```

(c) Write a query that lists the food and the color of every person whose favorite language is *not* Python.

```
Pork Bulgogi|Navy Blue
Pasta|Teal
Fries|Sky Blue
Popcorn|Blue
```

```
SELECT food, color
  FROM mentors
  WHERE language != 'Python';

-- With aliasing (treating the table as a Python object) --
SELECT m.food, m.color
  FROM mentors as m
  WHERE m.language <> 'Python';
```

This is a simpler SQL problem that serves as a status check for if your students are caught up with SQL. Might be good to recap SQL keywords such as select, from, where, order by, and limit here.