SQL AND TAIL RECURSION

COMPUTER SCIENCE MENTORS 61A

April 11 to April 15, 2016

1 Tail Recursion

1. What is a tail context/tail call? What is a tail recursive function?

Solution: A tail call is a call expression in a tail context. A tail context is usually the final action of a procedure/function.

A tail recursive function is where all the recursive calls of the function are in tail contexts.

An ordinary recursive function is like building up a long chain of domino pieces, then knocking down the last one. A tail recursive function is like putting a domino piece up, knocking it down, putting a domino piece up again, knocking it down again, and so on. This metaphor helps explain why tail calls can be done in constant space, whereas ordinary recursive calls need space linear to the number of frames (in the metaphor, domino pieces are equivalent to frames).

2. Why are tail calls useful for recursive functions?

Solution: When a function is tail recursive, it can effectively discard all the past recursive frames and only keep the current frame in memory. This means we can use a constant amount of memory with recursion, and that we can deal with an unbounded number of tail calls with our Scheme interpreter.

Answer the following questions with respect to the following function:

```
(define (sum-list lst)
        (if (null? lst)
                0
                (+ (car lst) (sum-list (cdr lst)))
        )
)
```

3. Why is sum-list not a tail call? Optional: draw out the environment diagram of this sum-list with list: (1 2 3). When do you add 2 and 3?

Solution: Sum list is not the last call we make, its actually the other addition which we do after we evaluate sum-list. Sum list is not the last expression we evaluate.

4. Rewrite sum-list in a tail recursive context.

Solution:

```
(define (sum-list-tail lst)
        (define (sum-list-helper lst sofar)
                (if (null? lst)
                        sofar
                        (sum-list-helper
                                (cdr lst)
                                (+ sofar (car lst))
                        )
                )
        )
        (sum-list-helper lst 0)
)
```

2 SQL

Name	Food	Color	Editor	Language
Tiffany	Thai	Purple	Notepad++	Java
Diana	Pie	Green	Sublime	Java
Allan	Sushi	Orange	Emacs	Ruby
Alfonso	Tacos	Blue	Vim	Python
Kelly	Ramen	Green	Vim	Python

5. Create a new table **mentors** that contains all the information above.

Solution:
```
create table mentors as
    select 'Tiffany' as name, 'Thai' as food, 'Purple' as
       color, 'Notepad++'
 as editor, 'Java' as language union
    select 'Diana', 'Pie', 'Green', 'Sublime', 'Java' union
    select 'Allan', 'Sushi', 'Orange', 'Emacs', 'Ruby'
       union
    select 'Alfonso', 'Tacos', 'Blue', 'Vim', 'Python'
       union
    select 'Kelly', 'Ramen', 'Green', 'Vim', 'Python';
```

6. Write a query that lists all the mentors along with their favorite food if their favorite color is green.
   Output:
```
Diana|Pie
Kelly|Ramen
```

Solution:
```
SELECT m.name, m.food
  FROM mentors as m
  WHERE m.color = 'Green';

WITHOUT ALIASING:

SELECT name, food
  FROM mentors
  WHERE color = 'Green';
```

7. Write a query that lists all the mentors along with their favorite food if their favorite color is green.
   Output:
```
Sushi|Orange
Pie|Green
Thai|Purple
```

8. Write a query that lists all the pairs of mentors who like the same language. (How can we make sure to remove duplicates?)
   Output:
   ```
   Kelly|Alfonso
   Tiffany|Diana
   ```

   Solution:
   ```
   SELECT m1.name, m2.name
       FROM mentors as m1, mentors as m2
       WHERE m1.language = m2.language and m1.name > m2.name;
   ```

9. Write a query that has the same data, but alphabetizes the rows by name. (Hint: Use order by.)
   Output:
   ```
   Alfonso|Tacos|Blue|Vim|Python
   Allan|Sushi|Orange|Emacs|Ruby
   Diana|Pie|Green|Sublime|Java
   Kelly|Ramen|Green|Vim|Python
   Tiffany|Thai|Purple|Notepad++|Java
   ```

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

3 Fish Population

The 61A mentors want to start a fish hatchery, and they need your help to analyze the data they've collected for the fish populations! Also, running a hatchery is expensive – they'd like to make some money on the side by selling some seafood (only older fish of course) to make delicious sushi.

The following table contains a subset of the data that has been collected. The SQL column names are listed in brackets. Note: we must be able to extend your queries to larger tables! (i.e, don't hard code your answers)

fish*

Species [species]	Population [pop]	Breeding Rate [rate]	$/piece [price]	# of pieces [pieces]
Salmon	500	3.3	4	30
Eel	100	1.3	4	15
Yellowtail	700	2.0	3	30
Tuna	600	1.1	3	20

*(This was made with fake data, do not actually sell fish at these rates)

10. **Aggregation**

(a) Profit is good, but more profit is better. Write a query to select the species that yields the most number of pieces for each price. Your output should include the species, price, and pieces.

Solution:
```
SELECT species, price, MAX(pieces) FROM fish GROUP BY
   price;
```

(b) Write a query to find the three most populated fish species.

Solution:
```
SELECT species FROM fish ORDER BY -pop LIMIT 3;
```

(c) Write a query to find the total number of fish in the "ocean." Additionally, include the number of species we summed. Your output should have the number of species and the total population.

Solution:
```
SELECT COUNT(species), SUM(pop) FROM fish;
```

(d) Business is good, but a bunch of competition has sprung up! Through some cunning corporate espionage, we have determined that one such competitor plans to open shop with the following rates:

competitor

Species [species]	$/piece [price]
Salmon	2
Eel	3.4
Yellowtail	3.2
Tuna	2.6

Write a query that compares how much our hatchery will charge per fish versus the competitor. For example, the table should contain the following row:

```
Salmon | 60
```

Because we make 30 pieces at $4 a piece for $120, whereas the competitor will make 30 pieces at $2 a piece for $60. Finally, the difference is 60. Remember to do this for every species!

Solution:

```
SELECT fish.species, (fish.price - competitor.price) *
   pieces
    FROM fish, competitor
    WHERE fish.species = competitor.species;
```

11. **Recursive Select** Suppose these fish breed every day. The population of each fish gets multiplied by its breeding rate every year. Write a recursive select function that creates a table of fish 10 years from now.

Solution:

```
WITH
  yearly_pop(yearly_species, yearly_pop, N) AS (
     SELECT species, pop, 0 FROM fish UNION
     SELECT yearly_species, yearly_pop * rate, N + 1
         FROM yearly_pop, fish
         WHERE yearly_species = species AND N <= 10
  )
SELECT yearly_species, yearly_pop FROM yearly_pop WHERE N
   = 10;
```