SQL, TAIL RECURSION, STREAMS, AND LOGIC

COMPUTER SCIENCE 61A

August 8 to August 12, 2015

1 SQL

Name	Genre	Rating	Type
Antman	Action	7.9	Live action
Minions	Comedy	6.7	Animated
Inside Out	Animation	8.6	Animated
Pixels	Comedy	5.4	Live action
Mission Impossible	Action	8.1	Live action

1. Create a table called `Summer Movies` which contains the five rows of the table above.

Solution:

```
create table summer movies as
    select 'Antman' as name, 'Action' as genre, 7.9 as
       rating, 'Live action' as type union
    select 'Minions', 'Comedy', 6.7, 'Animated' union
    select 'Inside Out', 'Animation', 8.6, 'Animated' union
    select 'Pixels', 'Comedy', 5.4, 'Live action' union
    select 'Mission Impossible', 'Action', 8.1, 'Live
       action'
```

2. Write a query to select the name and the rating of all live-action movies that are action movies and order them by rating.

Solution:
```
select name, rating from Summer Movies where genre =
    A c t i o n   order by rating
```

3. Write a query to select the names of all movies which have the same genre. Make sure and get rid of duplicates.

Solution:
```
select m1.name,m2.name from Summer Movies as m1, Summer
   Movies as m2 where m1.genre = m2.genre and m1.name > m2.
   name
```

4. Write a query to select all movies which score above a 7.0, ordered by their rating as well.

Solution:
```
select name from Summer Movies where rating > 7.0 order by
   rating
```

2 Tail Recursion

Consider the function sum-list:

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

1. Rewrite sum-list using tail recursion.

Solution: Solution 1: (must add 0 as a second argument)

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

Solution 2: (allow only a list as argument)

```
(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)
)
```

3 Streams

1. Why do we use streams? Why don't we just use linked lists instead?

   Solution: Lazy Evaluation, the elements of a stream are only evaluated when they are needed. We also have the added benefit of infinite streams.

   Steams represented in scheme have very specific functions associated with them.
   Stream creation: `cons-stream`
   First element of a stream: `car`
   Rest of the stream: `stream-cdr`
   Empty Stream: `nil`
   To check for emptiness: `null?`

2. Define a function called `integers` that returns a stream of integers starting from `first`

   Solution: (define (integers first)
   ```
   (cons-stream n (integers (+ first 1))))
   ```

3. What would Scheme Print?

```
scm> (define ints (integers 1))
```

Solution: `ints`

```
scm> (car (stream-cdr ints))
```

Solution: `2`

```
scm> (car ints)
```

Solution: `1`

```
scm> (car (stream-cdr (stream-cdr (stream-cdr ints))))
```

Solution: `4`

How many times did the stream have to compute a new value of rest for the last input?

Solution: `2`

```
scm> (define s (cons-stream (car ints)
                            (cons-stream (car (stream-cdr ints))
                                         nil)))
```

Solution: `s`

```
scm> (stream-cdr s)
```

Solution: `(2 . #[promise (not forced)])`

4. Write `conditional_map_stream`, a scheme function which goes through every element of a stream of numbers and returns a new stream which has either the original element if the function applied to the number was non-negative, or the value of the function applied to the original number otherwise.

```
scm> (define (f x) (- x 1))
f
scm> (define s (cons-stream 1
                            (cons-stream 3
                                         (cons-stream 12)))
s
scm> (define new (conditional_map_stream s f))
new
scm> (car new)
1
scm> (car (stream-cdr new))
2
```

```
(define (conditional_map_stream s f)
```

Solution:

```
    (cond ((null? s) s)
          ((> (f (car s)) 0)
                (cons-stream (f (car s))
                             (conditional_map_stream (
                               stream-cdr s) f)
          (else (cons-stream (car s) (
             conditional_map_stream (stream-cdr s) f)))))))
```

4 Logic

1. Define a set of facts to model the table of data below:

Name	Genre	Rating	Type
Antman	Action	7.9	Live-action
Minions	Comedy	6.7	Animated

Solution:

```
(fact (movie (name Antman) (genre action) (rating 7.9) (
   type live-action)))

(fact (movie (name Minions) (genre comedy) (rating 6.7) (
   type animated)))
```

2. Write facts for odd-length, as shown below:

```
logic> (odd-length (Minions are adorable))
Success!
logic> (odd-length (61a rocks))
Failed
```

Solution:

```
(fact (odd-length (?x)))
(fact (odd-length (?x ?y . ?z))
    (odd-length ?z))
```

3. Write facts for reverse, a relation between two lists that is satisfied if and only if the second list is the reverse of the first list. Hint: use append (given below), which was defined in lecture.

```
(fact (append () ?lst2 ?lst2))
(fact (append (?elem . ?rest1) ?lst2 (?elem . ?rest2))
      (append ?rest1 ?lst2 ?rest2))
```

Solution:

```
(fact reverse () ()))
(fact (reverse (?first . ?rest) ?result)
      (reverse ?rest ?new-rest)
      (append ?new-rest (?first) ?result))
```