CS 61A Fall 2017 — Final Review

December 4, 2017

Instructions

Form a small group. Start on the first problem. Check off with a helper or discuss your *solution process* with another group once everyone understands *how to solve* the first problem and then repeat for the second problem . . .

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is.* You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

1 Scheme

1.1 What would Scheme display?

(a)
```
> '(1 2 3)
```

(b)
```
> '(1 . (2 . (3 . ())))
```

(c)
```
> '(((1 . 2) . 3) 4 . (5 . 6))
```

(d)
```
> (cons 1 2)
```

(e)
```
> (cons 2 '())
```

(f)
```
> (cons 1 (cons 2 '()))
```

(g)
```
> (cons 1 (cons 2 3))
```

(h)
```
> (cons (cons (car '(1 2 3))
              (list 2 3 4))
        (cons 2 3))
```

(i)
```
> (car (cdr (car '((1 2) 3 (4 5)))))
```

(j)
```
> (cddr '((1 2) 3 (4 5)))
```

1.2 Define `sixty-ones`. Return the number of times that `1` follows `6` in the list.

```
> (sixty-ones '(4 6 1 6 0 1))
1
> (sixty-ones '(1 6 1 4 6 1 6 0 1))
2
> (sixty-ones '(6 1 6 1 4 6 1 6 0 1))
3
```

1.3 Identify the bug(s) in this program.

```
> (sum-every-other '(1 2 3))
4
> (sum-every-other '())
0
> (sum-every-other '(1 2 3 4))
4
> (sum-every-other '(1 2 3 4 5))
9

(define (sum-every-other lst)
    (cond ((null? lst) lst)
          (else (+ (cdr lst)
                   (sum-every-other (caar lst)) ))))
```

1.4 (a) Implement `add-to-all`.

```
> (add-to-all 'foo '((1 2) (3 4) (5 6)))
((foo 1 2) (foo 3 4) (foo 5 6))
```

(b) Rewrite `add-to-all` tail-recursively.

1.5 Define `sublists`. Hint: use `add-to-all`.

```
> (sublists '(1 2 3))
(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

1.6 (a) Define `reverse`. Hint: use `append`.

```
> (reverse '(1 2 3))
(3 2 1)
```

(b) Define `reverse` tail-recursively. Hint: use a helper function and `cons`.

2 Interpreters

2.1 Circle the number of calls to `scheme_eval` and `scheme_apply` for the code below.

(a)
```
scm> (+ 1 2)
3
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

(b)
```
scm> (if 1 (+ 2 3) (/ 1 0))
5
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

(c)
```
scm> (or #f (and (+ 1 2) 'apple) (- 5 2))
apple
```

`scheme_eval`	6	8	9	10
`scheme_apply`	1	2	3	4

(d)
```
scm> (define (add x y) (+ x y))
add
scm> (add (- 5 3) (or 0 2))
2
```

`scheme_eval`	12	13	14	15
`scheme_apply`	1	2	3	4

2.2 Identify the number of calls to `scheme_eval` and `scheme_apply`.

(a)
```
scm> (define pi 3.14)
pi
scm> (define (hack x)
          (cond
            ((= x pi) pwned)
            ((< x 0) (hack pi))
            (else (hack (- x 1)))))
hack
```

(b)
```
scm> (hack 3.14)
pwned
```

(c)
```
scm> ((lambda (x) (hack x)) 0)
pwned
```

3 Streams

3.1 Implement `merge`, which takes two streams `s1` and `s2` whose elements are ordered. `merge` returns a stream that contains elements from `s1` and `s2` in sorted order, eliminating repetition. You may assume `s0` and `s1` themselves do not contain repeats. `s1` and `s2` may or may not be infinite streams.

```
(define (merge s0 s1)
  (cond ((null? s0) s1)
        ((null? s1) s0)









  )
)
```

3.2 A famous problem, first raised by Richard Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. These are called regular numbers. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them fit the requirement.

As an alternative, we can write a function that returns an infinite stream of such numbers. Let us call the stream of numbers `s` and notice the following facts about it.

- `s` begins with 1.
- The elements of `(scale-stream s 2)` are also elements of `s`.
- The same is true for `(scale-stream s 3)` and $(scale - streams5)$.
- These are all of the elements of s.

Now all we have to do is combine elements from these sources. Use the `merge` function you defined previously to fill in the definition of `make-s`:

```
(define (make-s)





)
```

4 Iterators

4.1 Define a generator that yields the sequence of perfect squares. The sequence of perfect squares looks like: $1, 4, 9, 16 \ldots$

```
def perfect_squares():
```

4.2 Implement `zip`, which yields a series of lists, each containing the nth items of each iterable. It should stop when the smallest iterable runs out of elements.

```
def zip(*iterables):
    """
    >>> z = zip_generator([1, 2, 3], [4, 5, 6], [7, 8])
    >>> for i in z:
    ...     print(i)
    ...
    [1, 4, 7]
    [2, 5, 8]
    """
```

4.3 Implement `generate_subsets` that returns all subsets of the positive integers from 1 to `n`. Each call to this generator's `next` method will return a list of subsets of the set $\{1, 2, \ldots, n\}$, where n is the number of previous calls to `next`.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```

5 SQL

`pizzas` defines the name, open, and close hours of pizzarias. `meals` defines typical meal times. A pizzaria is open for a meal if the meal time is within `open` and `close`.

```
create table pizzas as
  select "Pizzahhh" as name, 12 as open, 15 as close union
  select "La Val's"        , 11        , 22          union
  select "Sliver"          , 11        , 20          union
  select "Cheeseboard"     , 16        , 23          union
  select "Emilia's"        , 13        , 18;
create table meals as
  select "breakfast" as meal, 11 as time union
  select "lunch"            , 13         union
  select "dinner"           , 19         union
  select "snack"            , 22;
```

5.1 There's nothing wrong with going to the same pizza place for meals greater than 6 hours apart, right? Create a table `double` with the earlier meal, the later meal, and the name of the pizza place. Only include rows that describe two meals that are *more than 6 hours apart* and a pizza place that is open for both of the meals.

```
create table double as
```

```
> select * from double where name="Sliver";
breakfast|dinner|Sliver
```

5.2 For each meal, list all the pizza options. Create a table `options` that has one row for every meal and three columns. The first column is the meal, the second is the total number of pizza places open for that meal, and the last column is a comma-separated list of open pizza places *in alphabetical order*. Assume that there is at least one pizza place open for every meal. Order the resulting rows by meal time.

Hint: Define a recursive table in a `with` statement that includes all partial lists of options, then use the **max** aggregate function to pick the full list for each meal.

```
create table options as
```

```
> select * from options where meal="dinner";
dinner|3|Cheeseboard, La Val's, Sliver
```