CS61A Notes
Week 1A: Basics, order of evaluation, special forms, recursion

Assorted Scheme Basics

What do you think Scheme will print for the following? Assume this definition of the `om` procedure:

```
(define (om foo)
    (word foo 'nom)
```

```
> nom
```

This will result in an error because `nom` has not been defined yet. If you wanted to print the word `nom`, you would type in `'nom`.

```
> (om 'nom)
```
This will result in the value `nomnom`, where the variable `foo` in the body of the procedure `om` was replaced by the word `nom`.
```
> (om nom)
```
This will attempt to replace the variable `foo` with the value of the variable `nom`, resulting in an error, since `nom` was not defined.
```
> ('om 'nom)
```
This will attempt to call whatever is to the right of the first left parenthesis. However, since `om` is a word and not a procedure, this statement will fail.
```
> (word 'a 'bc)
```
This will result in the word `abc`.
```
> (word 'a '(bc))
```
This will result in an error, because the arguments to the word procedure can only be words.
```
> (sentence 'a 'bc)
```
This will result in the sentence `'(a bc)`.
```
> (sentence 'a '(bc d))
```
This will result in the sentence `'(a bc d)`. The arguments to the sentence procedure can be words or sentences.
```
> (sentence '(a b) '(c d))
```
This will result in the sentence `'(a b c d)`.

You're So Special: Special Forms

1. `define`: This would be a good time to recapitulate the two versions of `define` that we have seen. What is the difference between `(define x 5)` and `(define (x) 5)`? The first call defines a *variable* named `x` and gives it a value of `5`. The second call defines a *procedure* named `x` with **no** arguments. This newly defined procedure will always return the value `5`.

2. `if`, `cond`:

   QUESTION:

   1. **Write the `abs` procedure, which implements the absolute value function. Use a `cond` statement.**

      ```
      (define (abs x)
          (cond ((= x 0) 0)
                ((< x 0) (- x))
                (else x)))
      ```

   2. **Convert the following `cond` statement into its equivalent `if` statement:**

      ```
      (cond ((= area-code 415) (word 'san 'francisco))
      ```

```
          ((= area-code 510) 'berkeley)
          (else 'where-you-from))

    (if (= area-code 415)
        (word 'san 'francisco)
        (if (= area-code 510)
            'berkeley
            'where-you-from))
```

Notice that the whole of the second `if` statement is the alternative case of the first `if` statement.

QUESTION: What are the results of the following expressions?
1. **`(if #t 3 4)`:** `3`
2. **`(if #f 3 4)`:** `4`
3. **`(if 'false 3 4)`:** `3` (because the *word* `false` is not `#f`)

3. `and`, `or`:

QUESTION:
1. **What does the statement `(and (= 3 3) (= 4 4))` return?**
   Since both statements are true, the `and` statement returns the last true value, which in this case happens to be `#t`, the result of `(= 4 4)`.

2. **What does the following code do?**

   ```
   (define (foo) (foo))
   (and (= 3 3) (foo))
   ```

   The `foo` procedure yields an infinite loop when it is called. (Why?) Also, an `and`-statement processes its arguments in order; as soon as it finds an argument that evaluates to false, the `and`-statement stops evaluating its arguments. In this case, the `(= 3 3)` expression evaluates to true, and the and-statement attempts to evaluate `(foo)`. However, since `(foo)` results in an infinite loop, the statement never returns a value – true *or* false.

3. **How about this code?**

   ```
   (define (foo) (foo))
   (and (= 3 4) (foo))
   ```

   Since the first expression `(= 3 4)` evaluates to false, the `and`-statement immediately returns `#f`, without having to evaluate `(foo)`.

In Order To Understand Recursion, You Must First Understand Recursion

QUESTIONS

1. **Write a procedure `(expt base power)` which implements the exponents function. For example, `(expt 3 2)` returns `9`, and `(expt 2 3)` returns `8`.**

   Usually, it is easier to think of the base case(s) before thinking about the recursive call. A good base case for this problem is when `power` is equal to zero: in this case, the answer is `1`. Now, in order to determine the recursive call, we realize that, if we were given `(expt base (- power 1))`, all we would need to do is multiply this answer by `base` to get the required answer.

```
(define (expt base power)
    (if (= power 0) 1 (* base (expt base (- power 1)))))
```

2. **I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many ways can I go up this flight of stairs? Write a procedure `count-stair-ways` that solves this for me.**

   Say I have `n` steps (`n > 3` for now). I can split the number of ways that I can get to the `n`th step into two separate cases:

   Case (1): Those that pass through the `(n - 1)`th step (after which you only need to climb 1 step), and
   Case (2): Those that pass through the `(n - 2)`th step (after which you only need to climb 2 steps).

   No single way can fall into both cases. Pause here and convince yourself that this is true before moving on, because this is the key insight.

   Then, we already know how many ways there are that pass through the `(n - 1)`th step: this is `(count-stair-ways (- n 1))`. We also know how many ways there are that pass through the `(n - 2)`th step: this is `(count-stair-ways (- n 2))`. So, ignoring base cases, we have

   ```
   (count-stair-ways n) = (count-stair-ways (- n 1)) + (count-stair-ways (- n 2)).
   ```

   There's a distinction to be made between a *way* and a *step*: a *way* is composed of *steps*. We're not counting steps: we're counting ways.

   ```
   (define (count-stair-ways n)
       (cond ((= n 1) 1)
             ((= n 2) 2)
             (else (+ (count-stair-ways (- n 1))
                      (count-stair-ways (- n 2))))))
   ```

3. **Define a procedure `subsent` that takes in a sentence and a parameter `i`, and returns a sentence with elements starting from position `i` to the end. The first element has `i = 0`. In other words,**
   `(subsent '(6 4 2 7 5 8) 3) => (7 5 8)`

   ```
   (define (subsent sent num)
       (cond ((empty? sent) '())
             ((= num 0) sent)
             (else (subsent (bf sent) (- num 1)))))
   ```

   The empty sentence is usually a good base case for sentence-based recursive problems. Also, it prevents the possibility of trying to find the `first/butfirst/last/butlast` of an empty sentence, which results in an error.

4. **I'm standing at the origin of some x-y coordinate system for no reason when a pot of gold dropped onto the point (x, y). I would love to go get that gold, but because of some arbitrary constraints or (in my case) mental derangement, I could only move right or up one unit at a time on this coordinate system. I'd like to find out how many ways I can reach (x, y) from the origin in this fashion (because, umm, my mother asked). Write `count-ways` that solves this for me.**

   The reasoning behind this question is similar to the reasoning behind `count-stair-ways` in question 2.

   ```
   (define (count-ways x y)
       (cond ((or (< x 0) (< y 0)) 0)
             ((and (= x 0) (= y 0)) 1)
             (else (+ (count-ways (- x 1) y)
                      (count-ways x (- y 1))))))
   ```

5. **Define a procedure `sum-of-sents` that takes in two sentences and outputs a sentence containing the sum of respective elements from both sentences. The sentences do not have to be the same size!**
`(sum-of-sents '(1 2 3) '(6 3 9)) => (7 5 12)`
`(sum-of-sents '(1 2 3 4 5) '(8 9)) => (9 11 3 4 5)`

What's a good base case here? There are two: if either (or both) of the sentences are empty, then we know we're done, because all we would have to do is return the other sentence. For instance, `(sum-of-sents '() '(3 4 5))` will return `'(3 4 5)`. Otherwise, we need to take the first element of each sentence, add them together, and prepend this sum to the new sentence.

```
(define (sum-of-sents sent1 sent2)
    (cond ((empty? sent1) sent2)
          ((empty? sent2) sent1)
          (else (se (+ (first sent1) (first sent2))
                    (sum-of-sents (bf sent1) (bf sent2))))))
```