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

Assorted Scheme Basics

1. **The `(` is the most important character in Scheme.** If you have coded in other languages such as C or Java, you may have noticed that they do not treat this character with the same reverence: you can add as many left parentheses as you would like, and the only thing you would change is the precedence with which an expression is evaluated: for instance, does `3 + 4 * 5` mean `(3 + 4) * 5` or `3 + (4 * 5)`? We know that the latter is not different from `3 + (((4 * 5)))`. In Scheme, however, **when you see the (character, you know that you are about to invoke a procedure**. This is true almost **always**, and requires that we think carefully before using a left parenthesis. So, for instance, if we have the expression `(f 3 4)`, we had better be sure that `f` is a procedure that has already been defined.

2. **Everything in Scheme is true, except for `#f`.** We have already seen the `if` statement, and we will revisit it later in this discussion. In using the `if` statement, we checked if a condition was true, and based on the *Boolean value* of the condition—whether the condition returned true and false—we determined what we needed to do. Our version of Scheme has two special Boolean values: `#t`, which signifies "true"-ness, and `#f`, which signifies "false"-ness: so long as our condition returns one of these two values, our `if` statement will work properly.

   But wait: it gets better! It turns out that the condition does not have to explicitly return `#t`; any value **besides `#f`** is considered equivalent to `#t`, in conditional statements. *Woah.* We'll revisit this point when we talk about special forms, so if you're still a bit confused and/or concerned for our sanities, hang on.

3. **Words and Sentences.** Words and sentences are among the basic types of data we will be playing with in our version of Scheme, along with numbers, symbols, and later on, lists. A word is declared in Scheme with the quote `'`. So, for example, `'nom` is a word. If you were to type a word at the Scheme interpreter prompt, Scheme will not try to evaluate whatever is after the quote, and will return it as is. (This is a good, though not foolproof, way of understanding how words work.) There is a *primitive* (built-in) procedure `word` that can also be used to construct words for you.

   Sentences are similar: they are collections of words enclosed within parentheses, *and quoted*. So, for example, `'(om nom)` is a sentence with two words. Notice that this is one of the few times that a left parenthesis does not precede a procedure invocation, because of the quote right in front of it. Er, left in front of it. Analogous to the `word` procedure, there is a primitive procedure called `sentence`.

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

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

   > nom
   > (om 'nom)
   > (om nom)
   > ('om 'nom)
   > (word 'a 'bc)
   > (word 'a '(bc))
   > (sentence 'a 'bc)
   > (sentence 'a '(bc d))
   > (sentence '(a b) '(c d))
   ```

Are Your Applications In Order: The Substitution Model

We consider how Scheme evaluates an expression by using the *substitution model*, which is a good model to start out with. First, let's review how an expression is evaluated:

1. Evaluate all subexpressions of the combination (including the procedure name and arguments) so that we know exactly what they are.
2. Apply the evaluated procedure to the evaluated arguments that we obtained in the previous step.

To perform step 2, we evaluate the body of the procedure after we replace the *formal parameters* (the names given to the arguments) with the actual arguments given. Let's dive right in. First, a few functions to play with:

```
(define (double x) (+ x x))
(define (square y) (* y y))
(define (f z) (+ (square (double z)) 1))
```

Now, what happens when we evaluate `(f (+ 2 1))`? Here's how the substitution model wants us to think:

1. Let's evaluate the subexpression: what is `f`? It is a procedure. What is `(+ 2 1)`? `3`.
   `(f` **`3`**`)`
2. Now, let's take the body of `f`, and *substitute* `3` for `z`:
   `(+ (square (double 3)) 1)`
3. Now, we evaluate the expression: `+` is `+`, `1` is `1`, but what is `(square (double 3))`? Well, `square` is a procedure; what is `(double 3)`? `double` is a procedure too, and `3` is `3`! `1` and `3` are *self-evaluating*, because evaluating them produces themselves. So now that we've evaluated everything, we can start applying the procedures. First, we evaluate the call to `double` by substituting the body in again:
   `(+ (square` **`(+ 3 3)`**`) 1)`
4. Now, what's `(+ 3 3)`?
   `(+ (square` **`6`**`) 1)`
5. How do we apply `square`? Let's substitute in the body of `square`:
   `(+` **`(* 6 6)`** `1)`
6. So what's `(* 6 6)`?
   `(+` **`36`** `1)`
7. And what's `(+ 36 1)`?
   `37`

If the above seems a little obvious, it's because it is – the substitution model is the most intuitive way to think about expression evaluation. Note, however, that this is not how Scheme actually works, and as we'll find out later in the course, the substitution model will eventually become inadequate for evaluating our expressions – more specifically, when we introduce assignments. But for now, while we're still in the realm of functional programming, the substitution model will serve us nicely.

By the way, notice that in step 2, we evaluate the arguments before we substitute. This way of doing things – evaluation before substitution – is known as *applicative order*. Yes, you guessed it: there is another way of doing this, but we'll see that many weeks later.

You're So Special: Special Forms

Remember how we said that arguments are always evaluated before a procedure is called? Well, we lied. Sorry. There are certain *special forms* whose arguments Scheme does not necessarily evaluate before calling. You have already seen a few of them: the ones we will consider here are `define`, `if`, `cond`, `and`, `or`, `quote`.

1. `define`: `define` is a special form, because if used to create a procedure, the variable names and the body are not evaluated. So, for instance, in:

   ```
   (define (foo x y)
       (+ x y))
   ```

   the subexpressions `(foo x y)` and `(+ x y)` are, in a sense, arguments to `define`. However, we can't (and don't) "evaluate" `(foo x y)`, because we don't yet have values for `x` and `y`.

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)`?

2. `if`, `cond`: `if` and `cond` are *control structures*: they determine which expressions are to be evaluated, based on certain conditions. Think of them as forks in a road.

   The syntax for `if` is

   ```
   (if <condition> <consequent> <alternative>)
   ```

   where `condition`, `consequent`, `alternative` are expressions. `if` is a special form because the three expressions are not all evaluated at once: first, the `condition` expression is evaluated, and if this condition is true, then we evaluate the `consequent`; else, we evaluate the `alternative`. You can see why this might be desirable from an efficiency perspective: you only evaluate the expressions you need. Sometimes, however, we need to be able to branch more than once, which is where `cond` proves useful.

   The syntax for `cond` is

   ```
   (cond (<first condition> <first consequent>)
         (<second condition> <second consequent>)
         ...
         (else <alternative>))
   ```

   `cond` allows us to check multiple conditions (or *clauses*), and the conditions are checked sequentially: if the first condition is true, then the first consequent is evaluated; otherwise, the second condition is checked, and so on. If none of the conditions are satisfied, then the alternative expression under `else` is evaluated. (Note that it is not necessary to have an `else` case, although it is good practice.)

   The left parentheses in bold are another exception to the rule that left parentheses precede function invocations. Here, they only serve to separate and isolate the different clauses: the conditions should be expressions that evaluate to `#t` or `#f`. Every `cond` statement can be converted into a corresponding series of nested `if` statements.

   QUESTION:

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

   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))
      ```

   Remember that we had earlier claimed everything in Scheme is true, except for `#f`? Consider the following chunk of code:

   ```
   (cond ((= x 3) 'garply)
         (4 'foo)
         (else 'bar))
   ```

When `x` is 3, this statement will return the word garply. Before proceeding, think of your gut reaction of the result when `x` is not 3.

It turns out that the result is the word `'foo` and the else case is never hit. This is because 4 is considered to be “true” by virtue of not being `#f`. Everything that is not `#f` will be considered to be true by `if` and `cond`.

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

3. `and`, `or`: `and` and `or` statements are useful when you want to consider several conditions at once, for example.

The syntax of an `and` statement is

```
(and <expression1> <expression2> … <expressionN>)
```

It can take any number of expressions as arguments. An `and` statement is “true” if all of the constituent subexpressions are “true”; if any one of them is “false”, then the whole statement is “false”. `and` is a special form because the expressions are evaluated from left to right: if any one of the expressions turn out to be false, the whole statement bails and returns `#f`; if, however, all of the expressions are true, the whole statement returns the value of the last “true” subexpression. So, for example,

```
(and (= 3 3) (= 3 4) (= 3 5))
```

returns `#f`, but the subexpression `(= 3 5)` is never evaluated. Alternatively,

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

does not return `#t` even though all subexpressions are “true”. The statement returns 4, which is the last true subexpression. However, for the purposes of `if` and `cond`, this return value is still true.

QUESTION:
1. **What does the statement `(and (= 3 3) (= 4 4))` return?**

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

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

3. **How about this code?**

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

An `or` statement has the following similar syntax:

```
(or <expression1> <expression2> … <expressionN>)
```

and returns `#f` if all of the expressions are “false”. If at least one of the expressions are “true”, then it returns the value of the first “true” expression. For instance, the expression

```
(or (= 2 3) (= 3 4) 'foo)
```

returns the word `'foo`.

4. `quote`: Earlier, we had said that when faced with `'foo`, Scheme does not evaluate `foo`. This should now intrigue you: is it possible that `'foo` is a special form? Why yes: internally, Scheme converts `'foo` to `(quote foo)`, where `quote` is a special form that does not evaluate its argument, but instead converts it into a word and a sentence.

In Order To Understand Recursion, You Must First Understand Recursion

A *recursive procedure* is one that calls itself in its body. The classic example is finding the factorial of a number:

```
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
```

Note that, in order to calculate the factorial of n, we tried to first calculate the factorial of `(- n 1)`. This is because of the observation that `5! = 5 * 4!`, and that, if we know what `4!` is, we could find out what `5!` is. If this makes you a bit suspicious, it's okay – it takes a little getting used to. The trick is to use the procedure *as if it is already correctly defined for another value*. This will become easier with practice. The mantra to repeat to yourself is:

TRUST THE RECURSION!

However, for recursive procedures, you need to be careful to include a *base case*: a recursive function will keep on calling itself, and since we need an answer at the end of the day, we need it to stop somewhere. Usually, this is a trivial case for which we know the answer. In the example of the factorial, the base case was when `n` was equal to zero.

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`.**

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.**

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)`

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.

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)
```

Secrets to Success in CS61A

- **Ask questions.** When you encounter something you don't know, ask. That's what we're here for. (Though this is not to say you should raise your hand impulsively; some usage of the brain first is preferred.) You're going to see a lot of challenging stuff in this class, and you can always come to us for help.
- **Go to office hours.** Office hours give you time with the professor or TAs by themselves, and you'll be able to get some (nearly) one-on-one instruction to clear up confusion. You're NOT intruding; the professors LIKE to teach, and the TAs…well, the TAs get paid. Remember that, if you cannot make office hours, you can always make separate appointments with us!
- **Do the readings (on time!).** There's a reason why they're assigned. And it's not because we're evil; that's only partially true.
- **Do all the homeworks.** We don't give many homework problems, but those we do give are challenging, time-consuming, but very rewarding as well.
- **Do all the labwork.** Most of them are simple and take no more than an hour or two after you're used to using emacs and Scheme. This is a great time to get acquainted with new material. If you don't finish, work on it at home, and come to office hours if you need more guidance!
- **Study in groups.** Again, this class is not trivial; you might feel overwhelmed going at it alone. Work with someone, either on homework, on lab, or for midterms (as long as you don't violate the cheating policy!).