# CS61A Notes 02 – Procedures as Data

## 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)`

*CS61A Summer 2010*
*George Wang, Jon Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng*
*Notes developed based on those of Chung Wu and Justin Chen*

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)
```

---

## What in the world is `lambda`?

No, `lambda` is not a chemistry term, nor does it refer to a brilliant computer game that spawned an overrated mod. `lambda`, in Scheme, means "a procedure". The syntax:

```
(lambda ([formal parameters])
        [body of the procedure])
```

So let's use a rather silly example:

```
(lambda (x)
        (* x x))
```

Read: *a procedure that takes in a single argument, and returns a value that is that argument multiplied by itself*. When you type this into Scheme, the expression will evaluate to something like:

```
#[closure arglist=(x) 14da38]
```

This is simply how Scheme prints out a "procedure": the last number is irrelevant (and can change),

while `arglist` tells us what arguments this procedure takes. Since `lambda` just gives us a procedure, we can use it the way we always use procedures – as the first item of a function call. So let's try squaring 3:

```
( (lambda (x) (* x x)) 3)
```

Read that carefully and understand what we just did. Our statement is a function call, whose function is "a procedure that takes in a single argument, and returns a value that is that argument multiplied by itself", and whose argument is the number `3`. Compare with `(+ 2 3)`; in that case, Scheme evaluates the first element – some procedure – and passes in the arguments 2 and 3. In the same way, Scheme evaluates the first element of our statement – some procedure – and passes in the argument 3.

A lot of you probably recognized the above `lambda` statement as our beloved `square`. Of course, it's rather annoying to have to type that `lambda` thing every time we want to square something. So we can bound it to any name of our desire. Oh, let's just be spontaneous, and use the name `square`:

```
(define square
   (lambda (x) (* x x)))
```

Note the similarities to `(define pi 3.1415926)`! For `pi`, we told Scheme to "bind to the symbol `pi` the value of the expression `3.1415926` (which is self-evaluating)". For `square`, we told Scheme to "bind to the symbol `square` the value of the `lambda` expression (which happens to be a procedure)". The two cases are not so different after all. If you type `square` into the Scheme prompt, it would print out something like this again:

```
#[closure arglist=(x) 14da38]
```

As we said before, this is how Scheme prints out a procedure. *Note that this is not an error*! It's simply what the symbol `square` evaluates to – a procedure. It is as natural as typing in `pi` and getting `3.1415926` back (if you've already defined `pi`, of course).

It's still a little annoying to type that `lambda` though, so we sugar-coat it a bit with some *syntactic sugar* to make it easier to read:

```
(define (square x) (* x x))
```

Ah, now that's what we're used to. The two ways of defining square are exactly the same to Scheme; the latter is just easier on the eyes, and it's probably the one you're more familiar with. Why bother with `lambda` at all, then? That is a question we will eventually answer, with vengeance.

**QUESTIONS**: What do the following evaluate to? Describe the result: if it is self-evaluating, determine the value; if it is a procedure, describe the number of arguments it takes.

1. `(lambda (x) (* x 2))`

2. `((lambda (y) (* (+ y 2) 8)) 10)`

3. `((lambda (b) (* 10 ((lambda (c) (* c b)) b)))`
   `((lambda (e) (+ e 5)) 5))`

4. `((lambda (a) (a 3)) (lambda (z) (* z z)))`

5. `((lambda (n) (+ n 10))`
   `((lambda (m) (m ((lambda (p) (* p 5)) 7))) (lambda (q) (+ q q))))`

---

## First-Class Procedures!

In programming languages, something is first-class if it doesn't chew with its mouth open. Also, as *SICP* points out, it can be bound to variables, passed as arguments to procedures, returned as results of procedures and used in data structures. Obviously we've seen that numbers are first-class – we've been passing them in and out of procedures since the beginning of time (which was about earlier this week), and have been putting them in data structures – sentences – since the dinosaurs went extinct (also earlier this week).

What is surprising – and exceedingly powerful – about Scheme, is that **procedures are also awarded first-class status**. That is, procedures are treated just like any other values! This is one of the more exotic features of LISP (compared to, say C or Java, where passing functions and methods around as arguments require much more work). And as you'll see, it's also one of the coolest.

---

## Procedures As Arguments

A procedure which takes other procedures as arguments is called a **higher-order procedure**. You've already seen a few examples, in the lecture notes, so we won't point them out again; let's work on something else instead. Suppose we'd like to square or double every word in the sentence:

```
(define (square-every-word sent)
     (if (empty? sent)
            '()
            (se (* (first sent) (first sent)) (square-every-word (bf sent)))))

(define (double-every-word sent)
     (if (empty? sent)
            '()
            (se (* 2 (first sent)) (double-every-word (bf sent)))))
```

Note that the only thing different about `square-every-word` and `double-every-word` is just what we do to `(first sent)`! Wouldn't it be nice to generalize procedures of this form into something more convenient? When we pass in the sentence, couldn't we specify, also, what we want to do to each word of the sentence?

To do that, we can define a higher-order procedure called `every`. `every` takes in the procedure you want to apply to each element *as an argument*, and applies it to each word in the sentence indiscriminately. So to write `square-every-word`, you can simply do:

```
(define (square-every-word sent) (every (lambda (x) (* x x)) sent))
```

Equivalently, to write `double-every-word`, you can write:

```
(define (double-every-word sent) (every (lambda (x) (* 2 x)) sent))
```

Now isn't that nice. Nicer still: the implementation of `every` is left as a homework exercise! You should be feeling all warm and fuzzy now.

The secret to working with procedures as values is to **keep the domain and range of procedures straight**! Keep careful track of what each procedure is supposed to take in and return, and you will be fine. Let's try a few:

**QUESTIONS**

1. What does this guy evaluate to?
   ```
   ((lambda (x) (x x)) (lambda (y) 4))
   ```

2. What about his new best friend?
   ```
   ((lambda (y z) (z y)) * (lambda (a) (a 3 5)))
   ```

3. Write a procedure `foo` that, given the call below, will evaluate to `10`.
   ```
   ((foo foo foo) foo 10)
   ```

4. Write a procedure, bar, that, given the call below, will evaluate to 10 as well.
   ```
   (bar (bar (bar 10 bar) bar) bar)
   ```

5. Something easy: write `first-satisfies` that takes in a predicate procedure and a sentence, and returns the first element that satisfies the predicate test. Return `#f` if none satisfies the predicate test. (A *predicate* is a special name for a procedure that only returns either `#t` or `#f`.) For example, `(first-satisfies even? '(1 2 3 4 5))` returns `2`, and `(first-satisfies number? '(a clockwork orange))` returns `#f`.

   ```
   (define (first-satisfies pred? sent)
   ```

## Procedures As Return Values

The problem is often: write a procedure that, given _____, **returns a procedure** that _____. The keywords – conveniently boldfaced – is that your procedure is supposed to return a procedure. This is often done through a call to `lambda`:

```
(define (my-wicked-procedure blah) (lambda (x y z) (...)))
```

Note that the above procedure, when called, will return a **procedure** that will take in three arguments. That is the common form for such problems (of course, never rely on these "common forms" as they'll just work against you on midterms!)

### QUESTIONS

1. In lecture, you were introduced to the procedure `keep`, which takes in a predicate procedure and a sentence, and throws away all words of the sentence that don't satisfy the predicate. The code for `keep` was:

```
(define (keep pred? sent)
  (cond ((empty? sent) '())
        ((pred? (first sent))
         (sentence (first sent) (keep pred? (bf sent))))
        (else (keep pred? (bf sent)))))
```

   To keep numbers less than 6, George and the TAs claim this wouldn't work:

```
(keep (< 6) '(4 5 6 7 8))
```

   a) Why won't the above work?

   b) Of course, this being Berkeley, and we being rebels, we're going to promptly prove the authoritative figures wrong. And just like some rebels, we'll do so by cheating. Let's do a simpler version; suppose we'd like this to do what we intended:

```
(keep (lessthan 6) '(4 5 6 7 8))
```

   Define procedure `lessthan` to make this legal.

   c) Now, how would we go about making this legal?

```
(keep (< 6) '(4 5 6 7 8))
```

---

## Orders of Growth

When we talk about the efficiency of a procedure (at least for now), we're often interested in how much more expensive it is to run the procedure with a larger input. That is, as the size of the input grows, how do the speed of the procedure and the space its process occupies grow?

For expressing all of these, we use what is called the Big-Theta notation. For example, if we say the running time of a procedure `foo` is in $\Theta(n^2)$, we mean that the time it takes to process the input grows as the square of the size of the input. More generally, we can say that `foo` is in some $\Theta$(`f(n)`) if there exist some constants `k1` and `k2` such that and some constants `c1` and `c2` such that,

```
k1 * f(n) < running time of foo for some n > c1, and
k2 * f(n) > running time of foo for some n > c2
```

To prove, then, that `foo` is in $\Theta$(`f(n)`), we only need to find constants `k1` and `k2` where the above holds. Fortunately for you, in 61A, we're not that concerned with rigor, and you probably won't need to know exactly how to do this (you will get the painful details in 61B!) What we want you to have in 61A, then, is the intuition of guessing the orders of growth for certain procedures.

---

## Kinds of Growth

Here are some common ones: $\Theta$`(1)` – constant time (takes the same amount of time regardless of input size); $\Theta$`(log n)` – logarithmic time; $\Theta$`(n)` – linear time; $\Theta(n^2)$, $\Theta(n^3)$, etc – polynomial time; $\Theta(2^n)$ – exponential time ("intractable"; these are really, really horrible).

---

## Orders of Growth in Space

"Space" refers to how much information a process must remember before it can complete. If you'll recall, the advantage of an iterative process is that it does not have to keep any information on the stack as it executes. So an iterative process always occupies a constant amount of space – $\Theta(1)$.
For a recursive process, the space it occupies tends to grow with the number of recursive calls. For example, for the `factorial` procedure, every time before we make a recursive call, we need to "remember" to multiply `(fact (- n 1))` by `n`. Since we'll make `n` recursive calls, we'll need to remember `n` such facts. So the orders of growth in space for the `factorial` function is $\Theta(n)$.

---

## Orders of Growth in Time

"Time", for us, basically refers to the number of recursive calls. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

A few things to note:

- If the function contains only primitive procedures like `+` or `*`, then it is constant time – $\Theta(1)$. An example would be `(define (plusone x) (+ x 1))`

- If the function is recursive, you need to:
  - count the number of recursive calls there will be, given input `n`, and
  - count how much time it takes to process the input per recursive call.

  The answer is usually the product of the two. For example, given a fruit basket with 10 apples, how long does it take for me to process the whole basket? Well, I'll recursively call my `eat` procedure which eats one apple at a time (so I'll call the procedure 10 times). Each time I eat an apple, it takes me 30 minutes. So the total amount of time is just 30*10 = 300 minutes!
- If the function contains calls of helper functions that are not constant-time, then you need to take the orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be factored into #2 above. (If this is confusing, try the examples below.)
- When we talk about orders of growth, we don't really care about constant factors. So if you get something like $\Theta(1000000n)$, this is really $\Theta(n)$. (Why? Can you use the definition of the Big-Theta notation given above to prove this?)
- We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3 + n^2 + 4n + 399)$, we take it to be $\Theta(n^3)$. (Again, why?)

**QUESTIONS**: What is the order of growth in time for:

1.
```
(define (fact x)
  (if (= x 0)
      1
      (* x (fact (- x 1)))))
```

2.
```
(define (fact-iter x answer)
        (if (= x 0)
            answer
            (fact-iter (- x 1) (* answer x))))
```

3.
```
(define (sum-of-facts x n)
    (if (= n 0)
        0
        (+ (fact x) (sum-of-facts x (- n 1)))))
```

4.
```
(define (fib n)
      (if (<= n 1)
         1
         (+ (fib (- n 1)) (fib (- n 2)))))
```

5.
```
(define (square n)
     (cond ((= n 0) 0)
           ((even? n) (* (square (quotient n 2)) 4))
           (else (+ (square (- n 1)) (- (+ n n) 1))) ) )
```

6.
```
(define (gcd x y)          ;; This is hard!
          (if (= y 0)
               x
               (gcd y (remainder x y))))
```

---