CS61A Notes 03 – Efficiency (v1.0) (and Applicative vs. Normal)

Applicative vs. Normal Order

The substitution model example presented above was done using "applicative order", where Scheme first evaluates completely all the procedures and arguments before executing the function call. Another way to do this – in "normal order" – is to expand out complex expressions involving defined procedures into one involving only primitive operators and self-evaluating values, and then perform the evaluation. In other words, we *defer* evaluation of the arguments; we substitute the *unevaluated* expression (as opposed to evaluated expression for applicative order) into the body of a procedure.

So for the same problem `(f (+ 2 1))`:

1. We'll expand `f` into its body first, leaving `(+ 2 1)` unevaluated:
   `(+ (square (double (+ 2 1))) 1)`
2. Now, we want to expand `square`; let's write in the body of `square`, and substitute `(double (+ 2 1))` for `y`, leaving `(double (+ 2 1))` unevaluated:
   `(+ (* (double (+ 2 1)) (double (+ 2 1))) 1)`
3. Now, we want to expand `double`; let's write the body of `double`, and substitute `(+ 2 1)` for `x`:
   `(+ (* (+ (+ 2 1) (+ 2 1)) (+ (+ 2 1) (+ 2 1))) 1)`
4. Finally we've expanded everything to just primitive operators and self-evaluating values. We'll do these one by one…

```
(+ (* (+ 3 (+ 2 1)) (+ (+ 2 1) (+ 2 1))) 1)
(+ (* (+ 3 3) (+ (+ 2 1) (+ 2 1))) 1)
(+ (* (+ 3 3) (+ 3 (+ 2 1))) 1)
(+ (* (+ 3 3) (+ 3 3)) 1)
(+ (* 6 (+ 3 3)) 1)
(+ (* 6 6) 1)
(+ 36 1)
37
```

Well okay, that seems harmless enough. It's just two ways of thinking about the problem: for applicative order, we evaluate completely all subexpressions first before we apply the current procedure. For normal order, we expand all the defined procedures into their bodies until everything is expressed in terms of primitive operations and self-evaluating values; then we find the answer.

So what's the difference? Some of you will raise concerns about efficiency (obviously, in this case, normal order causes us to perform `(+ 2 1)` four times, whereas we only performed the same calculation once for applicative order!). But in terms of correctness, the two are the same. That is, **in *functional programming*, applicative and normal order of evaluation will *always* return the same answer!**

QUESTIONS

1. **Above, applicative order was more efficient. Define a procedure where normal order is more efficient.**

2. **Evaluate this expression using both applicative and normal order: `(square (random x))`. Will you get the same result? Why or why not?**

3. **Consider a magical function `count` that takes in no arguments, and each time it is invoked, it returns `1` more than it did before, starting with `1`. Therefore, `(+ (count) (count))` will return `3`. Evaluate `(square (square (count)))` with both applicative and normal order; explain your result.**

Recursive vs. Iterative Processes

The Professor did a good job explaining this, so just a few brief points:

- Don't confuse "recursive procedures" and "recursive processes". A "recursive procedure" refers to the simple fact that a procedure calls itself somewhere within its body. A "recursive process" refers to the fact that the space a procedure occupies *as it runs* grows – it needs to remember additional state as it recurses. The former certainly does not imply the latter; therefore, a recursive procedure can generate an iterative process.
- The basic difference between recursive and iterative processes is that, for an iterative process, after some recursive call reaches the base case and returns, **there is nothing left to be done**. For a recursive process, after some recursive call reaches the base case and returns, **there is still work to do**. For example, for `(factorial 4)`, after reaching the base case, you still need to apply the chain of multiplications `(* 4 (* 3 (* 2 1)))`.
- Here's how you can tell: if the **last thing** a function does is make the recursive call, then the function should be an **iterative process**. If the function still has things to do after the recursive call, then it is a **recursive process**.
- The Big Simulation Of Recursive/Iterative Processes (the fun never ends)
- Understand the `fib-iter` procedure provided in SICP on page 39, and be able to tell why it is more efficient than the way we used to do `fib`.

QUESTIONS: Will the following generate a recursive or iterative process?

1.
```
(define (foo x)
        (* (- (+ (/ x 3) 4) 6) 2))
```

2.
```
(define (foo x)
        (if (= x 0) 0 (+ x (foo (- x 1)))))
```

3.
```
(define (helper1 x)
        (if (= x 0) 1 (helper1 (- x 1))))

    (define (helper2 x)
        (if (= x 0) 1 (+ 1 helper2 (- x 1))))
```

   a.
```
(define (bar x)
                (if (even? x) (helper1 (- x 1)) (helper1 (- x 2)))))
```

   b.
```
(define (bar x)
                (if (even? x) (helper2 (- x 1)) (helper2 (- x 2)))))
```

```
c. (define (bar x)
                              (if (= x 0) (helper2 x) (helper1 x)))

d. (define (bar x)
                              (if (= x 0) (helper1 x) (helper2 x)))

e. (define (bar x)
                              (cond ((= x 0) 1)
                                    ((= (helper2 x) 3) 5)
                                    (else (helper1 x))))

f. (define (bar x)
     (helper2 (helper1 x)))
```

Yoshimi Battles The Recursive Robots, Pt. 2

1. **There is something called a "falling factorial". `(falling n k)` means that `k` consecutive numbers should be multiplied together, starting from `n` and working downward. For example, `(falling 7 3)` means `7 * 6 * 5`. Write the procedure falling that generates an iterative process.**

2. **Write a version of `(expt base power)` that works with negative powers as well.**

3. **Implement `(ab+c a b c)` that takes in values `a, b, c` and returns `(a*b) + c`. However, you cannot use *. Make it a recursive process. (The problem ripped from Greg's notes)**

4. **Implement `(ab+c a b c)` as an iterative process. Don't define helper procedures.**

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 exists 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 irregardless 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:
  1. count the number of recursive calls there will be given input n
  2. count how much time it takes to process the input *per recursive call*

  The answer is usually the product of the above 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 – and it is – 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))))
```