

CS61A Notes – Week 2: Order of evaluation, recursion, procedures as data (solutions)

Applicative vs. Normal Order

QUESTIONS

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

Anything where not evaluating the arguments will save time works. Most trivially,

```
(define (f x) 3) ;; a function that always returns 3
```

When you call `(f (fib 10000))`, applicative order would choke, but normal order would just happily drop `(fib 10000)` and just return 3.

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

Unless you're lucky, the result will be quite different. Expanding to normal order, you have `(* (random x) (random x))`, and the two separate calls to `random` will probably return different values.

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.

For applicative order, `(count)` is only called once - returns 1 - and is squared twice. So you have `(square (square 1))`, which evaluates to 1.

For normal order, `(count)` is called FOUR times:

```
(* (square (count)) (square (count))) =>
(* (* (count) (count)) (* (count) (count))) =>
(* (* 1 2) (* 3 4)) =>
24
```

Yoshimi Battles the Pink Recursive Robots

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.

```
(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.

```
(define (count-ways n)
  (cond ((= n 0) 1)
        ((< n 0) 0)
        (else (+ (count-ways (- n 1))
                  (count-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 i)
  (cond ((= i 0) sent)
        (else (subsent (bf sent) (- i 1)))))
```

Note that we're assuming `i` is valid (or, not larger than length of the sentence).

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.

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

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)`

```
(define (sum-of-sents s1 s2)
  (cond ((empty? s1) s2)
        ((empty? s2) s1)
        (else (se (+ (first s1) (first s2))
                    (sum-of-sents (bf s1) (bf s2))))))
```

What in the World is `lambda`?

QUESTIONS: What do the following evaluate to?

```
(lambda (x) (* x 2))
#[closure arglist=(x) e16fd0]

((lambda (b) (* 10 ((lambda (c) (* c b)) b))) ((lambda (e) (+ e 5)) 5))
1000

((lambda (a) (a 3)) (lambda (z) (* z z)))
9
```

Procedures as Arguments

QUESTIONS

- What does this guy evaluate to?
`((lambda (x) (x x)) (lambda (y) 4))`
 4
- What about his new best friend?
`((lambda (y z) (z y)) * (lambda (a) (a 3 5)))`

15

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

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

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

```
(define (bar x y) x)
```

Procedures as Return Values

QUESTIONS

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 doesn'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)))))
```

Recall that Brian said to keep numbers less than 6, this wouldn't work: `(keep (< 6) '(4 5 6 7 8))`

1. Why doesn't this work?

`(< 6)` evaluates to `#t`, not a procedure. Since `keep` requires a procedure, it fails miserably.

2. Of course, this being Berkeley, and us being rebels, we're going to promptly prove the authority figure – the Professor himself – 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.

The insight is that `(lessthan 6)` must return a procedure. In fact, it must return a procedure that checks if a given number is less than 6.

```
(define (lessthan n)
  (lambda (x) (< x n)))
```

3. Now, how would we go about making this legal?

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

The tricky thing here is that `(< 6)` must also return a procedure as we did up there. That requires us to redefine what `'<'` is, since `'<'` the primitive procedure obviously doesn't return a procedure.

```
(define (< n)
  (lambda (x) (> n x)))
```

Note also that we can't use `'<'` in the body as a primitive!