QUESTIONS

In Class:

1. Evaluate this expression using both applicative and normal order: \( (\text{square} \ (\text{random} \ x)) \). 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 \(* \ (\text{random} \ x) \ (\text{random} \ x)\), and the two separate calls to random will probably return different values.

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

For applicative order, \( (\text{count}) \) is only called once – returns 1 – and is squared twice. So you have \( (\text{square} \ (\text{square} \ 1)) \), which evaluates to 1.

For normal order, \( (\text{count}) \) is called FOUR times:

\[ (* \ (\text{square} \ (\text{count}))) \ (\text{square} \ (\text{count}))) \Rightarrow\]
\[ (* \ (* \ (\text{count}) \ (\text{count})) \ (* \ (\text{count}) \ (\text{count}))) \Rightarrow\]
\[ (* \ (* \ 1 \ 2) \ (* \ 3 \ 4)) \Rightarrow\]
\[ 24 \]

Extra Practice:

3. 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,

\( (\text{define} \ (f \ x) \ 3) ;; \text{a function that always returns 3} \)

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

---

Yoshimi Battles the Pink Recursive Robots

TRUST THE RECURSION!

QUESTIONS

In Class:

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

\( (\text{define} \ (\text{expt} \ \text{base} \ \text{power}) \)
\( \ (\text{if} \ (= \ \text{power} \ 0) \)
\( \ 1 \)
\( \ (* \ \text{base} \ (\text{expt} \ \text{base} \ (- \ \text{power} \ 1)))))) \)
2. There is something called a “falling factorial”. \((\text{falling } n \ k)\) means \(k\) consecutive numbers should be multiplied together, starting from \(n\) and working downward. For example, \((\text{falling } 7 \ 3)\) means \(7 \times 6 \times 5\). Write the procedure \texttt{falling} that generates an iterative process.

\[
(\text{define (falling b n)}
\text{(define (helper b n ans)}
\text{(if (= n 1)}
\text{(* b ans)}
\text{(helper (- b 1) (- n 1) (* b ans)))})
\text{(helper b n 1))}
\]

Extra Practice:

3. Define a procedure \texttt{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, \((\text{subsent } '(6 4 2 7 5 8) \ 3) \Rightarrow (7 5 8)\)

\[
(\text{define (subsent sent i)}
\text{(cond ((= i 0) sent)}
\text{(else (subsent (bf sent) (- i 1)))))})
\]

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

4. Write a version of \texttt{(expt base power)} that works with negative powers as well.

\[
(\text{define (expt base power)}
\text{(cond ((= power 0) 1)}
\text{((> power 0) (* base (expt base (- power 1)))})
\text{(else (/ (expt base (+ power 1)) base))))})
\]

5. Define a procedure \texttt{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!

\[
(\text{sum-of-sents } '(1 2 3) \ ,(6 3 9) \ ) \Rightarrow (7 5 12)
(\text{sum-of-sents } '(1 2 3 4 5) \ ,(8 9)) \Rightarrow (9 11 3 4 5)
\]

\[
(\text{define (sum-of-sents s1 s2)}
\text{(cond ((empty? s1) s2)}
\text{((empty? s2) s1)}
\text{(else (se (+ (first s1) (first s2)}
\text{(sum-of-sents (bf s1) (bf s2)))))))})
\]

What in the World is lambda?

QUESTIONS: What do the following evaluate to?

\[
(\text{lambda (x) } (* \ x \ 2))
#\text{[closure arglist=(x) e16fd0]}
((\text{lambda (a) } (a \ 3)) \ (\text{lambda (z) } (* \ z \ z)))
9
\]
Procedures as Arguments

QUESTIONS

In Class:

1. What does this guy evaluate to?
\[ ((\text{lambda} (x) (x x)) (\text{lambda} (y) 4)) \]
   
\[ 4 \]

2. What about his new best friend?
\[ ((\text{lambda} (y z) (z y)) * (\text{lambda} (a) (a 3 5))) \]
   
\[ 15 \]

Extra Practice:

3. Write a procedure, foo, that, given the call below, will evaluate to 10.
   \[ ((\text{foo} \text{foo} \text{foo}) \text{foo} 10) \]
   
\[ (\text{define} (\text{foo} x y) y) \]

4. Write a procedure, bar, that, given the call below, will evaluate to 10 as well.
   \[ (\text{bar} (\text{bar} (\text{bar} 10 \text{bar}) \text{bar}) \text{bar}) \]
   
\[ (\text{define} (\text{bar} x y) x) \]

Procedures as Return Values

QUESTIONS

1. Why doesn’t this work?
\[ (< 6) \text{ evaluates to } \#t, \text{ not a procedure. Since } \text{keep requires a procedure, it fails miserably.} \]

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:
\[ (\text{keep} (\text{lessthan} 6) \ '(4 5 6 7 8)) \]

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.

\[ (\text{define} \ (\text{lessthan} n) \ (\text{lambda} (x) (< x n))) \]

3. Now, how would we go about making this legal?
\[ (\text{keep} (< 6) \ '(4 5 6 7 8)) \]

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.

\[ (\text{define} \ (< n) \ (\text{lambda} (x) (> n x))) \]

Note also that we can’t use ‘<’ in the body as a primitive!