RACKET BASICS, ORDER OF EVALUATION, RECURSION 1

COMPUTER SCIENCE 61AS

Functional Programming

1. What is functional programming? Give an example of a function below:

Solution: In functional programming, you do not do assignments, functions have no memory and leave no side effects, and each function always returns the same thing each time, given the same parameter. For example,

```
-> (square 3)
9
-> (square 3)
9
```

This may seem obvious, but later in this class, we will cover programming that is not functional.

2. Not all procedures are functions. Give an example of a procedure that is not functional. It doesn't have to be a procedure you've seen before!

Solution: The built-in `random` procedure is not functional.

```
-> (random 10)
3
-> (random 10)
8
```

The Substitution Model

We consider how Racket evaluates an expression by using the substitution model. For most students, the substitution model is the most intuitive way to think about expression evaluation. Note, however, that this is not how Racket actually works. As we'll find out later in the course, the substitution model eventually will become inadequate for evaluating our expressions or, more specifically, when we introduce assignments. But for now, while were still in the realm of functional programming, the substitution model will serve us nicely.

1. According to the substitution model, how should we evaluate an expression? Specifically, show how `(+ 20 (square 2))` is evaluated, assuming `square` is already defined.

Solution: First, lets review how an expression is evaluated again:

(a) Evaluate all subexpressions of the combination (including the procedure and the arguments) so that we know exactly what they are.

(b) Apply the evaluated procedure to the evaluated arguments that we obtained in the previous step.

To perform step 2, basically, we evaluate the body of the procedure after we replace the formal parameters with the given arguments.

Here's how the substitution model wants us to think:

(a) First, evaluate all subexpressions. `+` evaluates to the addition procedure, and 20 evaluates to 20.

(b) Before we can apply the + procedure, we need to evaluate `(square 2)`. We know square is a procedure and 2 is 2. Now it's time to substitute!

(c) `(square 2)` is replaced with the body of square, with a 2 substituted for x.

```
(* 2 2)
```

We would evaluate this expression in the same way as before, finally getting 4.

(d) Now we just need to evaluate `(+ 20 4)`, which gives us 24.

Applicative vs. Normal Order

1. Show how `(f (+ 2 1))` is evaluated in both applicative and normal order, given the definitions below.

```
(define (double x) (* x 2))
(define (square y) (* y y))
(define (f z) (+ (square (double z)) 1))
```

Solution: Applicative order:

```
(f (+ 2 1)) ==>
(f 3) ==>
(+ (square (double 3)) 1) ==>
(+ (square (* 3 2)) 1) ==>
(+ (square 6) 1) ==>
(+ (* 6 6) 1) ==>
37
```

Normal order:

```
(f (+ 2 1)) ==>
(+ (square (double (+ 2 1))) 1) ==>
(+ (square (* (+ 2 1) 2)) 1) ==>
(+ (* (* (+ 2 1) 2) (* (+ 2 1) 2)) 1) ==>
(+ (* (* 3 2) (* 3 2)) 1) ==>
(+ (* 6 6) 1) ==>
(+ 36 1) ==>
37
```

2. In Exercise 1 above, you should have found that applicative order is more efficient than normal order. Define a procedure where normal order is more efficient.

Solution: 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.

3. Evaluate this expression using both applicative and normal order: `(square (random 5))`. Do you get the same result? Why or why not?

Solution: Unless youre 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.

4. Consider a magical function `count` that takes in no arguments, and each time it is invoked 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.

Solution: 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
```

The Basics of Recursion

1. What is recursion?

Solution: Function calls itself
Simplifies a large problem
'Till there's a base case

2. What are three things you find in every recursive function?

Solution:

(a) One or more base cases

(b) Way(s) to make the problem into a smaller problem of the same type (so that it can be solved recursively).

(c) One or more recursive cases that solve the smaller problem and then uses that to solve the original (large) problem.

3. When you write a recursive function, you seem to call it before it has been fully defined. Why doesn't this break the Racket interpreter, or cause any infinite loops?

Solution: When you define a function, Racket does not evaluate the body of the function. The body of the function is evaluated only when we call the procedure.

Practice with Recursion

1. Consider a function `sum-every-other`, which sums every other number in a sentence, starting from the second element. It should work as shown below.

```
-> (sum-every-other '(1 2 3 4 5))
6
-> (sum-every-other '(7 2 7 4 7))
6
-> (sum-every-other '(1))
0
```

   a. What is the domain and range of `sum-every-other`?

   Solution: The domain (what the function takes in) is a sentence of numbers. The range (what the function outputs) is a single number.

   b. What's wrong with the following code? Fix all mistakes you find.

```
(define (sum-every-other sent)
    (if (empty? sent)
        0
        (+ (first sent) (sum-every-other (bf sent)))))
```

   Solution: The current code sums every number in the sentence. If you don't see why, run through a small example step-by-step.

   To instead sum every other element, the code should be as follows:

```
(define (sum-every-other sent)
    (if (<= (count sent) 1)
        0
        (+ (first (bf sent)) (sum-every-other (bf (bf sent))))))
```

2. Write a procedure `(expt base power)` which implements the exponents function. For example, `(expt 3 2)` returns 9, and `(expt 2 3)` returns 8.

Solution:

```
(define (expt base power)
    (if (= power 0)
        1
        (* base (expt base (- power 1)))))
```

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`.

```
-> (subsent '(6 4 2 7 5 8) 3)
(7 5 8)
```

Solution:

```
(define (subsent sent i)
    (if (= i 0)
        sent
        (subsent (bf sent) (- i 1))))
```

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

4. Consider the procedure `sum-of-sents` which sums the numbers in two sentences. Note, the sentences don't 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)
```

a. What is the domain and range of `sum-of-sents`? Be as specific as possible!

Solution: The domain is two sentences of numbers. The range is a sentence of numbers.

b. What should your base case(s) be? Write in code a condition that tells us when our base case is reached as well as what it should return.

Solution: We'll have two base cases:

```
(cond ((empty? s1) s2)
      ((empty? s2) s1)
      (...))
```

c. Define `sum-of-sents`.

Solution:

```
(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))))))
```