RACKET BASICS, ORDER OF EVALUATION, RECURSION 1

COMPUTER SCIENCE 61AS

Functional Programming

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

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!

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.

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))
```

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.

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

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.

The Basics of Recursion

1. What is recursion?

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

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?

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`?

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)))))
```

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.

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)
```

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!

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.

c. Define `sum-of-sents`.