Exercise 7: Enumerating Continuations

In last week's lecture, we began discussing *continuations*, representations of the control flow at a particular point in the execution of a program. In imperative-style programming, we can think of a continuation as being the state of the *call stack* at a moment in time. This is very powerful, but also pretty complex. However, in the specific case of pure functions arranged in nested function calls, there's a much easier way of representing continuations. We'll explore this specific representation in this week's exercise.

Note: in both past and futures lectures we discuss how to directly access and manipulate continuations in Racket programs. This exercise is *not* about that; you won't be using `shift` or `reset`, for example. Instead, this exercise is about making sure you understand continuations as an abstract concept, and how we can represent them in familiar Racket syntax.

Deadline: November 5, 2019 before 10:00pm

Starter code

- `ex7.rkt`

Task 1: Enumerating continuations

You only have one task on this exercise: writing a function `continuations` that computes and displays (representations of) the continuations of every subexpression of an input datum.

We'll use a very simple grammar for our datum here: only numeric literals and nested `+` function calls are allowed. You may assume all inputs are both syntactically- and semantically-valid.

We'll represent a continuation by a Racket datum that uses the special symbol `'_` to represent where to put the value of the subexpression. For example, in the expression `(+ (+ 3 4) 9)`, we represent the continuation of the `4` as the Racket datum `'(+ (+ 3 _) 9)`.

Warning: even though there's only one task on this exercise, the nature of continuations adds enough complexity that a naive recursive approach runs into some trouble. We've provided a fairly detailed design in the starter code, and some discussion about the technical challenges. **However, you're welcome to use your own approach, as long as it adheres to the global restrictions for exercises in this course.**

Please note that you *may* use `eval` on this exercise. Our starter code has used it in one place , and you may choose to use it in other places as well, especially if you use a different approach.

As an aside, once you complete this exercise, you may find it interesting to attempt this problem in an imperative, mutating style/language—it wasn't immediately clear to us how this more familiar setting would actually make this problem easier!