CS151.02 2010S *Functional Problem Solving*

Class 25: Recursion Basics

Held: Monday, March 8, 2010

Summary: We begin our exploration of *recursion*, the most general form of repetition available in Scheme. You can use recursion to both build and iterate over different kinds of values. Today, we explore how you use recursion to build simple numeric values from lists, and how to remove elements from lists.

Related Pages:

- EBoard.
- Lab: Recursion Basics.
- Reading: Recursion Basics.

Notes:

- Reading for tomorrow: Recursion with Helper Procedures.
- It's clearly time for another pop-tart day in 151.
- Are there questions on Assignment 6?
- Please try to finish today's lab on your own.

Overview:

- The idea of recursion.
- A sample recursive procedure: `sum`.
- Another example: Filtering.

Repetition

- You may recall that when we first considered algorithms we identified a number of key aspects of algortihms:
  - *variables*: the ability to name things;
  - *conditions*: the ability to choose between things;
  - *procedures*: the ability to name (and parameterize) collections of steps;
  - *repetition*: the ability to do something multiple times;
  - *input* and *output*: the ability to get and report values.
- We've already seen how to do all of these things.
- Examples of repetition from baking:
  - Stir the mix 50 times
  - Knead the bread dough until it feels like your earlobe
  - Bake until golden-brown.
- Examples of repetition from mathematics:
  - Sum these values

    - Find the smallest of these values
- Examples of repetition from everyday life:
    - Naively find a name in the phone book
    - Do I have a CD by Van Morrison?
- We know a few ways to repeat actions:
    - Using lists; `map` and `for-each`
    - Using images: `image-variant`, `image-transform!`, and `image-compute-pixels!`.
    - With side-effecting actions: `repeat`.
- Today we begin to consider more general forms of repetition.

Recursion

- In Scheme, the most common mechanism for repetition is *recursion*.
- To do something that involves repeated actions, you
    - Do one action
    - Repeat the rest
    - Combine the results if necessary.
- For example, to stir your cake mix 50 times, you stir it one time and then stir it 49 more times.
- More generally, to stir a cake mix *n* times, you stir it one time and then *n*-1 more times.
- Similarly, to knead dough until it's the right consistency, you knead it a little, check the consistency, and, if it's not the right consistency, knead it until its the right consistency.
- In the case of mathematics, to sum a list we might add the first value to the sum of the remaining values (or add the last value to the sum of the initial values).
- There are a few key aspects to recursive design:
    - You need to know when you're done (and what to do when you're done). This aspect of recursive design is called the *base case*.
    - You need to know what to do when you're not done. Here, you should do a little, try again, and then perhaps combine the results. This aspect of recursive design is called the *recursive case*.
    - You need to be sure that you're getting closer to the base case (otherwise you'll never stop).

Recursion in Scheme

- Here's the form of a typical recursive procedure:

```
(define proc
  (lambda (val)
    (if (base-case-test)
        (base-case val)
        (combine (partof val)
                 (proc (update val))))))
```

- When the value you're working with is a list and your base case is the null list, the form is somewhat simpler:

```
(define proc
  (lambda (lst)
    (if (null? lst)
        null-case
        (combine (onestep (car lst))
                 (proc (cdr lst))))))
```

- Sometimes it's useful to grab the recursive result first, particularly if you're going to use it in multiple ways.

```
(define proc
  (lambda (lst)
    (if (null? lst)
        null-cse
        (let ((recursive-result (proc (cdr lst))))
          (if (test)
              (combine1 (step1 (car lst)) recursive-result)
              (combine2 (step2 (car lst)) recursive-result))))))
```

Lab

- Start the lab on recursion.
- Continue it on your own.
- Be prepared to reflect and to ask questions.
