

Iterating Over Lists

Summary: We examine techniques for doing computation using each element of a list.

Contents:

- Introduction
- `map` - Building New Lists from Old
- Using `map` with Lists of Spots
- `foreach!` - Doing Something with Each Element of a List
- Detour: Sectioning
- Quick Reference

Introduction

As you may recall from our initial discussions of algorithms, there are four primary kinds of control that help us write algorithms: We sequence operations, we choose between operations, we encapsulate groups of operations into functions, and we repeat operations. At this point, you've learned at least one mechanism for each kind of control.

However, the only mechanisms you've learned for repeating actions apply only to images. As you may recall, `image.map` computes a new image, each of whose pixels is computed by applying a function to the corresponding pixel of another image; `image.map!` updates an image using a similar technique; and `region.compute-pixels!` creates an image by applying a procedure to each position. In each case, we iterate over positions in the image (and call a function based on either the color at that position or the position itself).

What if we want to iterate over other kinds of values? In a few days, you'll learn about *recursion*, Scheme's most general mechanism for repeating actions. For now, let's consider a variant of what we've already learned. That is, we will learn techniques for iterating over lists.

To ground these explorations, we will consider the ways in which these techniques might be useful for working with lists of spots, the representation of images we considered in previous readings.

`map` - Building New Lists from Old

Scheme provides one standard procedure for doing something with each element of a list, `map`. Just as `image.map` creates a new image, `map` creates a new list. In particular, `(map fun lst)` creates a new list of the same length as `lst` by applying the function to each element of `lst`.

For example, we can add 1 to every number in a list of numbers with

```
(map (lambda (num) (+ 1 num)) numbers)
```

Let's see that code in a sample sequence from the interactions pane.

```
> (define numbers (list 11 2 1 8 16))
> numbers
(11 2 1 8 16)
> (map (lambda (num) (+ 1 num))) numbers)
(12 3 2 9 17)
> numbers
(11 2 1 8 16)
```

As the last command suggests, `map` is much like `image.map` in that it does not modify the original list. Rather, it computes a new list. And, as was the case with `image.map`, you can call `map` with either an anonymous function (as above) or with a named function (as in the following).

```
> (define numbers (list 11 2 1 8 16))
> (map square numbers)
(121 4 1 64 256)
```

Using map with Lists of Spots

The `map` procedure can be quite useful with lists of spots. For example, if we've created one list of spots, we can map a procedure onto that list to change the color of the spots, move the spots elsewhere, or even rotate the spots around some point.

For example, in the lab on lists of spots, you wrote a procedure that translated a spot horizontally. We can translate a whole list of spots horizontally by 20 units with

```
(map (lambda (spot) (spot.htrans spot 20)) spots)
```

In effect, once you've designed an image using lists of spots, you can use it as a kind of *rubber stamp*, drawing it again and again at different places in the image.

foreach! - Doing Something with Each Element of a List

Of course, for `map` to be useful with lists of spots, we need a way to render all of the spots in a list, and not just one. Of course, `map` provides a solution. We can draw all the spots with something like the following:

```
(map (lambda (spot) (spot.draw canvas spot)) spots)
```

This solution will certainly work. However, it's also a bit awkward. While we are doing something with each element in the list, our goal is not to create a new list. In this case, and many others, we iterate through the list not to create a list, but to do things with the values in the list, with no goal of creating new values.

This technique of iterating a list for the side effect, and not to create a new list, is common enough that most Scheme programmers define a procedure for just that purpose. That procedure is not in Standard Scheme, but it's common enough that it has a common name, `foreach!`.

So, here's the typical way to define a procedure that draws each spot in a list of spots.

```
(define spot-list.draw
  (lambda (spots image)
    (foreach! (lambda (spot) (spot.draw spot image)) spots)))
```

We can write a similar procedure to draw a bigger version of each spot.

```
(define spot-list.decadraw
  (lambda (spots image)
    (foreach! (lambda (spot) (spot.decadraw spot image)) spots)))
```

We're now ready to draw lots of copies of our sample image.

```
(spot-list.draw spots image)
(spot-list.draw (map (lambda (spot) (spot.htrans 20 spot)) spot) image)
(spot-list.draw (map (lambda (spot) (spot.vtrans 25 spot)) spot) image)
```

Detour: Sectioning

The calls that end the preceding section are both nicely concise, in that the `map` lets us repeat, and also a bit verbose, in that the lambda expressions seem fairly long, particularly since all we're really saying is “fill in one parameter of `spot.htrans` or `spot.vtrans`”. We even saw the same issue when our first call to `map`, when we wrote `(lambda (num) (+ 1 num))`.

To provide a more concise way to write those procedures, many Scheme programmers use the procedures `left-section` (typically written `l-s`) and `right-section` (typically written `r-s`), whose purpose is to fill in one parameter of a two-parameter procedure. The `left-section` procedure takes two parameters, a two-parameter function and a value, and creates a new function by filling in the first (left) parameter of the function. The `right-section` procedure takes two parameters, a two-parameter function and a value, and creates a new function by filling in the second (right) parameter of the function.

For example,

- `(l-s + 2)` means “add two”, and is shorthand for `(lambda (x) (+ 2 x))`
- `(r-s - 3)` means “subtract three”, and is shorthand for `(lambda (x) (- x 3))`
- `(l-s - 5)` means “subtract from five”, and is shorthand for `(lambda (x) (- 5 x))`
- `(r-s expt 3)` means “raise to the third power”, and is shorthand for `(lambda (x) (expt x 3))`

Here are some examples of these procedures in action:

```

> (define numbers (list 6 9 12 5 1))
> (map (l-s + 2) numbers)
(8 11 14 7 3)
> (map (r-s - 3) numbers)
(3 6 9 2 -2)
> (map (r-s expt 3) numbers)
(216 729 1728 125 1)
> (map (l-s expt 2) numbers)
(64 512 4096 32 2)

```

Once we have `l-s` and `r-s` in our toolbox, the previous examples are much more concise.

```

(spot-list.draw (map (r-s spot.htrans 20) spots) image)
(spot-list.draw (map (r-s spot.vtrans 25) spots) image)

```

We can also use these techniques to simplify our definitions from the previous section.

```

(define spot-list.draw
  (lambda (spots image)
    (foreach! (r-s spot.draw image) spots)))
(define spot-list.decadraw
  (lambda (spots image)
    (foreach! (r-s spot.decadraw image) spots)))

```

Some programmers find this more concise form easier to read, and a bit more elegant. Others find it puzzling. While you will probably find it puzzling at first, after a bit it will become more natural (or so we hope).

Quick Reference

```
(foreach! func list)
```

Traditional List Procedure. Evaluate *func* on each element of the given list. Called primarily for side effects.

```
(l-s proc left)
```

Traditional Procedure. Given a two-parameter procedure, *proc*, and a value, *left*, create a new one-parameter procedure that applies *proc* to *left* and the parameter. That is, the result is `(lambda (x) (proc left x))`.

```
(map fun lst)
```

Standard List Procedure. Create a new list, each of whose elements is computed by applying *fun* to the corresponding element of *lst*.

```
(r-s proc left)
```

Traditional Procedure. Given a two-parameter procedure, *proc*, and a value, *right*, create a new one-parameter procedure that applies *proc* to that parameter and *right*. That is, the result is `(lambda (x) (proc x right))`.

Copyright © 2007 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do

not necessarily reflect the views of the National Science Foundation. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.