Repetition Through Recursion

Summary: In many algorithms, you want to do things again and again and again. For example, you might want to do something with each value in a list. In general, the term used for doing things again and again is called *repetition*. In Scheme, the primary technique used for repetition is called *recursion*, and involves having procedures call themselves.

Contents:

- Introduction
- An Example: Summation
  - Watching Sum Work
- Base Cases
- An Alternative Sum
  - Watching New Sum
- Filtering Lists
- Singleton Base Cases
- A Common Form of Recursive Procedures
- Using And and Or

Introduction

As we've already seen, it is commonplace for the body of a procedure to include calls to another procedure, or even to several others. For example, we might write our "find one root of the quadratic equation" procedure as

```
(define root1
  (lambda (a b c)
    (/ (+ (- 0 b)
          (sqrt (- (* b b)
                   (* 4 a c))))
       (* 2 a))))
```

Here, there are calls to addition, subtraction, division, multiplication, and square root in the definition of `root1`.

Direct recursion is the special case of this construction in which the body of a procedure includes one or more calls to the very same procedure -- calls that deal with simpler or smaller arguments.

An Example: Summation

For instance, let's define a procedure called `sum` that takes one argument, a list of numbers, and returns the result of adding all of the elements of the list together:

```
> (sum (list 91 85 96 82 89))
443
> (sum (list -17 17 12 -4))
8
> (sum (list 19/3))
19/3
> (sum null)
0
```

Because the list to which we apply `sum` may have any number of elements, we can't just pick out the numbers using `list-ref` and add them up -- there's no way to know in general whether an element even exists at the position specified by the second argument to `list-ref`. One thing we do know about lists, however, is that every list is either *(a)* empty, or *(b)* composed of a first element and a list of the rest of the elements, which we can obtain with the `car` and `cdr` procedures.

Moreover, we can use the predicate `null?` to distinguish between the *(a)* and *(b)* cases, and conditional evaluation to make sure that only the expression for the appropriate case is chosen. So the structure of our definition is going to look something like this:

```
(define sum
  (lambda (numbers)
    (if (null? numbers)
        ; The sum of an empty list
        ; The sum of a non-empty list
    )))
```

The sum of the empty list is easy -- since there's nothing to add, the total is 0.

And we know that in computing the sum of a non-empty list, we can use `(car numbers)`, which is the first element, and `(cdr numbers)`, which is the rest of the list. So the problem is to find the sum of a non-empty list, given the first element and the rest of the list. Well, the rest of the list is one of those "simpler or smaller" arguments mentioned above. Since Scheme supports direct recursion, we can invoke the `sum` procedure within its own definition to compute the sum of the elements of the rest of a non-empty list. Add the first element to this sum, and we're done!

```
;;; Procedure:
;;;    sum
;;; Parameters:
;;;    numbers, a list of numbers.
;;; Purpose:
;;;    Find the sum of the elements of a given list of numbers
;;; Produces:
;;;    total, a number.
;;; Preconditions:
;;;    All the elements of numbers must be numbers.
;;; Postcondition:
;;;    total is the result of adding together all of the elements of numbers.
;;;    If all the values in numbers are exact, total is exact.
;;;    If any values in numbers are inexact, total is inexact.
(define sum
  (lambda (numbers)
    (if (null? numbers)
        0
        (+ (car numbers) (sum (cdr numbers))))))
```

At first, this may look strange or magical, like a circular definition: If Scheme has to know the meaning of `sum` before it can process the definition of `sum`, how does it ever get started?

The answer is what Scheme learns from a procedure definition is not so much the meaning of a word as the algorithm, the step-by-step method, for solving a problem. Sometimes, in order to solve a problem, you have to solve another, somewhat simpler problem of the same sort. There's no difficulty here as long as you can eventually reduce the problem to one that you can solve directly.

Another way to think about it is in terms of the way we normally write instructions. We often say "go back to the beginning and do the steps again". Given that we've named the steps in the algorithm, the recursive call is, in one sense, a way to tell the computer to go back to the beginning.

Watching Sum Work

The repeatedly solving simpler problems strategy is how Scheme proceeds when it deals with a call to a recursive procedure -- say, `(sum (cons 38 (cons 12 (cons 83 null))))`. First, it checks to find out whether the list it is given is empty. In this case, it isn't. So we need to determine the result of adding together the value of `(car ls)`, which in this case is 38, and the sum of the elements of `(cdr ls)` -- the rest of the given list.

The rest of the list at this point is the value of `(cons 12 (cons 83 null))`. How do we compute its sum? We call the `sum` procedure again. This list of two elements isn't empty either, so again we wind up in the alternate of the `if`-expression. This time we want to add 12, the first element, to the sum of the rest of the list. By "rest of the list", this time, we mean the value of `(cons 83 null)` -- a one-element list.

To compute the sum of this one-element list, we again invoke the `sum` procedure. A one-element list *still* isn't empty, so we head once more into the alternate of the `if`-expression, adding the car, 83, to the sum of the elements of the cdr, `null`. The "rest of the list" this time around is empty, so when we invoke `sum` yet one more time, to determine the sum of this empty list, the test in the `if`-expression succeeds and the consequent, rather than the alternate, is selected. The sum of `null` is 0.

We now have to work our way back out of all the procedure calls that have been waiting for arguments to be computed. The sum of the one-element list, you'll recall, is 83 plus the sum of `null`, that is, 83 + 0, or just 83. The sum of the two-element list is 12 plus the sum of the `(cons 83 null)`, that is, 12 + 83, or 95. Finally, the sum of the original three-element list is 38 plus the sum of `(cons 12 (cons 83 null))` that is, 38 + 95, or 133.

Here's a summary of the steps in the evaluation process.

```
   (sum (cons 38 (cons 12 (cons 83 null))))
=> (+ 38 (sum (cons 12 (cons 83 null)))))
=> (+ 38 (+ 12 (sum (cons 83 null))))
=> (+ 38 (+ 12 (+ 83 (sum null))))
=> (+ 38 (+ 12 (+ 83 0)))
=> (+ 38 (+ 12 83))
=> (+ 38 95)
=> 133
```

Talk about delayed gratification! That's a while to wait before we can do the first addition.

The process is exactly the same, by the way, regardless of whether we construct the three-element list using `cons`, as in the example above, or as `(list 38 12 83)` or `'(38 12 83)`. Since we get the same list in each case, `sum` takes it apart in exactly the same way no matter what mechanism was used to build it.

Base Cases

The method of recursion works in this case because each time we invoke the `sum` procedure, we give it a list that is a little shorter and so a little easier to deal with, and eventually we reach the *base case* of the recursion -- the empty list -- for which the answer can be computed immediately.

If, instead, the problem became harder or more complicated on each recursive invocation, or if it were impossible ever to reach the base case, we'd have a *runaway recursion* -- a programming error that shows up in DrScheme not as a diagnostic message printed in red, but as an endless wait for a result. The designers of DrScheme's interface provided a `Break` button above the definition window so that you can interrupt a runaway recursion: Move the mouse pointer onto it and click the left mouse button, and DrScheme will abandon its attempt to evaluate the expression it's working on.

As you may have noted, there are three basic parts to these kinds of recursive functions.

- A *recursive case* in which the function calls itself with a simpler or smaller parameter.
- A *base case* in which the function does not call itself.
- A *test* that decides which case holds.

You'll come back to these three parts for each function you write.

An Alternative Sum

As you may have noted, an odd thing about the `sum` procedure is that it works from right to left. Traditionally, we sum from left to right. Can we rewrite `sum` to work from left to right? Certainly, but we may need a *helper procedure* (another procedure whose primary purpose is to assist our current procedure) to do so.

If you think about it, when you're summing a list of numbers from left to right, you need to keep track of two different things:

- The sum of the values seen so far.
- The remaining values to add.

Hence, we'll build our helper procedure with two parameters, `sum-so-far` and `remaining`. We'll start the body with a template for recursive procedures (a test to determine whether to use the base case or recursive case, the base case, and the recursive case). We'll then fill in each part.

```
(define new-sum-helper
  (lambda (sum-so-far remaining)
    (if (test)
        base-case
        recursive-case)))
```

The recursive case is fairly easy. We add the first element of `remaining` to `sum-so-far` and continue with the new `sum-so-far` and the rest of remaining. To "continue", we simply call `new-sum-helper` again.

```
(define new-sum-helper
  (lambda (sum-so-far remaining)
    (if (test)
        base-case
        (new-sum-helper (+ sum-so-far (car remaining))
                        (cdr remaining)))))
```

The recursive case then gives us a clue as to what to use for the test. We need to stop when there are no elements left in the list.

```
(define new-sum-helper
  (lambda (sum-so-far remaining)
    (if (null? remaining)
        base-case
        (new-sum-helper (+ sum-so-far (car remaining))
                        (cdr remaining)))))
```

We're almost done. What should the base case be? In the previous version, it was 0. However, in this case, we've been keeping a running sum. When we run out of things to add, the value of the complete sum is the value of the running sum.

```
(define new-sum-helper
  (lambda (sum-so-far remaining)
    (if (null? remaining)
        sum-so-far
        (new-sum-helper (+ sum-so-far (car remaining))
                        (cdr remaining)))))
```

Now we're ready to write the primary procedure whose responsibility it is to call `new-sum-helper`. Like `sum`, `new-sum` will take a list as a parameter. That list will become `remaining`. What value should `sum-so-far` begin with? Since we have not yet added anything when we start, it begins at 0.

```
(define new-sum
  (lambda (numbers)
    (new-sum-helper 0 numbers)))
```

Putting it all together, we get the following.

```
;;; Procedure:
;;;   new-sum
;;; Parameters:
;;;   numbers, a list of numbers.
;;; Purpose:
;;;   Find the sum of the elements of a given list of numbers
```

```
;;; Produces:
;;;    total, a number.
;;; Preconditions:
;;;    All the elements of numbers must be numbers.
;;; Postcondition:
;;;    total is the result of adding together all of the elements of numbers.
;;;    If all the values in numbers are exact, total is exact.
;;;    If any values in numbers are inexact, total is inexact.
(define new-sum
  (lambda (numbers)
    (new-sum-helper 0 numbers)))

;;; Procedure:
;;;    new-sum-helper
;;; Parameters:
;;;    sum-so-far, a number.
;;;    remaining, a list of numbers.
;;; Purpose:
;;;    Add sum-so-far to the sum of the elements of a given list of numbers
;;; Produces:
;;;    total, a number.
;;; Preconditions:
;;;    All the elements of remaining must be numbers.
;;;    sum-so-far must be a number.
;;; Postcondition:
;;;    total is the result of adding together sum-so-far and all of the
;;;      elements of remaining.
;;;    If both sum-so-far and all the values in remaining are exact,
;;;      total is exact.
;;;    If either sum-so-far or any values in remaining are inexact,
;;;      total is inexact.
(define new-sum-helper
  (lambda (sum-so-far remaining)
    (if (null? remaining)
        sum-so-far
        (new-sum-helper (+ sum-so-far (car remaining))
                        (cdr remaining)))))
```

Watching New Sum

Does this change make a difference in the way in which the sum is evaluated? Let's watch.

```
   (new-sum (cons 38 (cons 12 (cons 83 null))))
=> (new-sum-helper 0 (cons 38 (cons 12 (cons 83 null))))
=> (new-sum-helper (+ 0 38) (cons 12 (cons 83 null)))
=> (new-sum-helper 38 (cons 12 (cons 83 null)))
=> (new-sum-helper (+ 38 12) (cons 83 null))
=> (new-sum-helper 50 (cons 83 null))
=> (new-sum-helper (+ 50 83) null)
=> (new-sum-helper 133 null)
=> 133
```

Note that the intermediate results for new-sum were different, primarily because new-sum operates from left to right.

6

Filtering Lists

Often the computation for a non-empty list involves making another test. Suppose, for instance, that we want to define a procedure that takes a list of integers and "filters out" the negative ones, so that if, for instance, we give it a list consisting of -13, 63, -1, 0, 4, and -78, it returns a list consisting of 63, 0, and 4. We can use direct recursion to develop such a procedure:

- If the given list is empty, there are no elements to filter out and also no elements to keep, so the correct result is the empty list.
- If the given list is not empty, we examine its car and its cdr. We can use a call to the very procedure that we're defining to filter negative elements out of the cdr. That gives a list comprising all of its non-negative elements.
  - If the car of the given list -- that is, its first element -- is negative, we ignore the car and just return the result of the recursive procedure call, without change.
  - Otherwise, we invoke `cons` to attach the car to the new list.

Translating this algorithm into Scheme yields the following definition:

```
(define filter-out-negatives
  (lambda (ls)
    (if (null? ls)
        null
        (if (negative? (car ls))
            (filter-out-negatives (cdr ls))
            (cons (car ls) (filter-out-negatives (cdr ls)))))))
```

Of course, when you see nested `if` expressions, you may instead prefer to use a `cond`. We can express the same idea as follows:

```
(define filter-out-negatives
  (lambda (ls)
    (cond
      ((null? ls) null)
      ((negative? (car ls)) (filter-out-negatives (cdr ls)))
      (else (cons (car ls) (filter-out-negatives (cdr ls)))))))
```

Singleton Base Cases

Sometimes the problem that we need an algorithm for doesn't apply to the empty list, even in a vacuous or trivial way, and the base case for a direct recursion instead involves *singleton lists* -- that is, lists with only one element. For instance, suppose that we want an algorithm that finds a largest element of a given non-empty list of real numbers. (The list must be non-empty because there is no "largest element" of an empty list.)

```
> (largest-of-list (list -17 38 62/3 -14/9 204/5 26 19))
204/5
```

The assumption that the list is not empty is a *precondition* for the meaningful use of this procedure, just as a call to Scheme's built-in `quotient` procedure requires that the second argument, the divisor, be non-zero. You should form the habit of noting and detailing such preconditions as you write the initial comment for a procedure:

```
;;; Procedure:
;;;   largest-of-list
;;; Parameters:
;;;   numbers, a list of real numbers.
;;; Purpose:
;;;   Find the largest element of a given list of real numbers
;;; Produces:
;;;   largest, a real number.
;;; Preconditions:
;;;   numbers is not empty.
;;;   All the values in numbers are real numbers.  That is, numbers
;;;      contains only numbers, and none of those numbers are complex.
;;; Postconditions:
;;;   largest is an element of numbers (and, by implication, is real).
;;;   largest is greater than or equal to every element of numbers.
```

If a list of real numbers is a singleton, the answer is trivial -- its only element is one of its largest elements. Otherwise, we can take the list apart into its car and its cdr, invoke the procedure recursively to find the largest element of the cdr, and use Scheme's built-in procedure `max` to compare the car to the largest element of the cdr, returning whichever is greater.

We can test whether the given list is a singleton by checking whether its cdr is an empty list. The value of the expression `(null? (cdr ls))` is `#t` if `ls` is a singleton, `#f` if `ls` has two or more elements.

Here, then, is the procedure definition:

```
(define largest-of-list
  (lambda (numbers)
    (if (null? (cdr numbers))
        (car numbers)
        (max (car numbers) (largest-of-list (cdr numbers))))))
```

If someone who uses this procedure happens to violate its precondition, applying the procedure to the empty list, DrScheme notices the error and prints out a diagnostic message:

```
> (largest-of-list null)
cdr: expects argument of type <pair>; given ()
```

A Common Form of Recursive Procedures

If you consider the examples above, you will see that there is a common form for most of the procedures. The form goes something like this

```
(define recursive-proc
  (lambda (val)
    (if (base-case-test?)
        (base-case-computation val)
        (combine (partof val)
                 (recursive-proc (simplify val))))))
```

For example, for the `largest-of-list` procedure,

- The *recursive-proc* is `largest-of-list`.
- The *val* is `numbers`, our list of numbers.
- The *base-case-test* is `(null? (cdr numbers))`, which checks whether `numbers` has only one element.
- The *base-case-computation* is `car`, which extracts the one number left in `numbers`.
- The *partof* procedure is also `car`, which extracts the first number in `numbers`.
- The *simplify* procedure is `cdr`, which drops the first element, thereby giving us a simpler (well, smaller) list.
- Finally, the *combine* procedure is `max`.

Similarly, consider the first complete version of `sum`.

```
(define sum
  (lambda (numbers)
    (if (null? numbers)
        0
        (+ (car numbers) (sum (cdr numbers))))))
```

In the `sum` procedure,

- The *recursive-proc* is `sum`.
- The *val* is again `numbers`, a list of numbers.
- The *base-case-test* is `(null? numbers)`, which checks if we have no numbers.
- The *base-case-computation* is `0`. (This computation does not quite match the form above, since we don't apply the 0 to `numbers`. As this example suggests, sometimes the base case does not involve the parameter.)
- The *partof* procedure is `car`, which extracts the first value in `numbers`.
- The *simplify* procedure is `cdr`, which drops the the first element.

Using And and Or

Of course, this common form is not the only way to define recursive procedures. In particular, when we define a predicate that uses direct recursion on a given list, the definition is usually a little simpler if we use `and`- and `or`-expressions rather than `if`-expressions. For instance, consider a predicate `all-even?` that takes a given list of integers and determines whether all of them are even. As usual, we consider the cases of the empty list and non-empty lists separately:

- Since the empty list has no elements, it is (as mathematicians say) "vacuously true" that all of its elements are even -- there is certainly no counterexample that one could use to refute the assertion. So `all-even?` should return #t when given the empty list.
- For a non-empty list, we separate the car and the cdr. If the list is to count as "all even", the car must clearly be even, and in addition the cdr must be an all-even list. We can use a recursive call to determine whether the cdr is all-even, and we can combine the expressions that test the car and cdr conditions with `and` to make sure that they are both satisfied.

Thus `all-even?` should return #t when the given list either is empty or has an even first element and all even elements after that. This yields the following definition:

```
;;; Procedure:
;;;   all-even?
;;; Parameters:
;;;   values, a list of integers.
;;; Purpose:
;;;   Determine whether all of the elements of a list of integers
;;;   are even.
;;; Produces:
;;;   result, a Boolean.
;;; Preconditions:
;;;   All the values in the list are integers.
;;; Postconditions:
;;;   result is #t if all of the elements of values are even.
;;;   result is #f if at least one element is not even.
(define all-even?
  (lambda (values)
    (or (null? values)
        (and (even? (car values))
             (all-even? (cdr values))))))
```

When `values` is the empty list, `all-even?` applies the first test in the `or`-expression, finds that it succeeds, and stops, returning #t. In any other case, the first test fails, so `all-even?` proceeds to evaluate the first test in the `and`-expression. If the first element of `values` is odd, the test fails, so `all-even?` stops, returning #f. However, if the first element of `values` is even, the test succeeds, so `all-even?` goes on to the recursive procedure call, which checks whether all of the remaining elements are even, and returns the result of this recursive call, however it turns out.