

## Writing Recursive Procedures

**Summary:** As you've started exploring the technique of recursion, some of you have asked (explicitly or implicitly), "So, how do I write a recursive procedure from scratch?" In this reading, we consider some common techniques for thinking about the design and implementation of recursive algorithms.

### Contents:

- Introduction
- Rely on Similar Procedures
- Writing Straightforward Recursive Procedures
- Writing Recursive Helpers

## Introduction

As we've seen in the reading on recursion, recursion is a powerful technique for writing procedures that can repeat work and can deal with parameters whose size we do not know in advance (such as lists of unknown length).

We have also seen a few basic techniques for structuring recursive procedures. In the most general formulation, we write a recursive procedure as follows.

```
(define recursive-procedure
  (lambda (parameters)
    (if test-for-base-case
        base-case
        recursive-case)))
```

When we consider the parts in a bit more detail, we see that the *base-case* may involve the parameters (e.g., we may take the car of a one-element list) or it may not (e.g., we may return 0). We also note that the recursive case involves recursing (obviously), simplifying the parameters before recursing, and, often, doing something after recursing.

```
(define recursive-proc
  (lambda (params)
    (if (base-case-test)
        (base-case params)
        (combine (partof params)
                  (recursive-proc (simplify params))))))
```

In many cases, the recursive part really had two alternatives - one in which some test held and one in which it failed to hold. For example, we often check whether a list starts with a particular value. In that case, we might generalize to something a bit more like the following.

```
(define recursive-proc
  (lambda (params)
    (cond
      ((base-case-test)
       (base-case params))
      ((special-case-test)
       (combine (partof params)
                (recursive-proc (simplify params))))
      (else (recursive-proc (simplify params))))))
```

We also saw a different pattern of recursion, one in which we use a helper procedure that takes additional parameters, often parameters to help us “accumulate” an answer.

```
(define recursive-proc
  (lambda (params)
    (recursive-proc-helper initial-value-of-accumulator params)))

(define recursive-proc-helper
  (lambda (computed-so-far remaining-params)
    (if (base-case-test)
        (modify computed-so-far)
        (recursive-proc-helper (update computed-so-far)
                               (simplify remaining-params)))))
```

But how do you choose how to simplify, to update, to modify, and how else to deal with the specifics of a particular procedure? Let us consider some common strategies.

## Rely on Similar Procedures

For both kinds of recursive procedures (those with recursive helpers and those without), the first strategy you should often employ is to see if you’ve solved (or seen a solution of) a similar procedure. If you have, then see if you can modify that similar procedure for this case. For example, if we are asked to compute the product of the values in a list, we might begin with `sum`.

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

What do we need to change in moving from `sum` to `product`?

- We need to change the name of the procedure. (Replace `sum` by `product`.)
- We need to consider whether or not to change the base case. What is the product of no numbers? In Scheme, the answer seems to be “1”, the multiplicative identity. Hence, we will need to change the base case from 0 to 1.
- We need to consider whether the operation is appropriate. Since we’re computing a product, we should replace addition by multiplication.

Putting that all together, we get the following:

```
(define product
  (lambda (vals)
    (if (null? vals)
        1
        (* (car vals) (product (cdr vals))))))
```

We can do something in writing `closest-to-zero`. Since `closest-to-zero` involves finding the “best” value in a list, we rely on a procedure we’ve seen before that finds a different kind of best value, `largest-in-list`.

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

In this case, we observe that `max` is used to select between two values, and choose the better (larger) one. We’ll need to replace this with something that finds the value closer to zero.

```
(define closest-to-zero
  (lambda (numbers)
    (if (null? (cdr numbers))
        (car numbers)
        (closer-to-zero (car numbers) (closest-to-zero (cdr numbers))))))
```

Of course, `closer-to-zero` does not exist. So, we’ll have to write it. Fortunately, it is fairly straightforward.

```
(define closer-to-zero
  (lambda (a b)
    (if (< (abs a) (abs b))
        a
        b)))
```

That’s it. We’re done.

## Writing Straightforward Recursive Procedures

But what happens when you don’t have a model on which you can base your solution? Then, you need to start “from scratch”, as it were.

I find it most useful to begin with the base case. Ask yourself *What is the simplest case for which I can directly compute an answer?* The answer to this question should give you a test for the base case. For lists, the simplest case is often an empty list or a single-element list.

Next, ask yourself *What is the answer in this simple case?* If you do not find that the answer is relatively obvious, then your test for the base case may be wrong. Note that it is important to *ensure that the type of the value of the base case matches the expected type*. That is, if you are returning a list, the base case should be a list, if you are returning a number, the base case should be a number, and so on and so forth.

Next, ask yourself *How do I simplify my parameters?* For lists, the typical case is to take the `cdr` of the list. For natural numbers, the typical case is to subtract some number. (A common, though less typical, case is to divide by some number, often two.)

Finally, ask yourself *Suppose I had a solution to this simplified version. What can I do with the result to compute my desired value?*

## Writing Recursive Helpers

The strategy we use for recursive helper procedures is similar, but differs in a few key ways. Recall that such procedures look something like the following:

```
(define recursive-proc-helper
  (lambda (computed-so-far remaining-params)
    (if (base-case-test)
        (modify computed-so-far)
        (recursive-proc-helper (update computed-so-far)
                               (simplify remaining-params))))))
```

You begin, again, by asking yourself *How do I know that I have no values left to process?* For lists, you often know that you're done when the list parameter (which we're fond of calling *remaining*) is empty. For numbers, you often know that you're done when one of the numeric parameters reaches 0.

You continue by asking yourself *What is the relationship of the value I computed along the way to my desired result?* The value computed along the way is often the desired result. At times, though, it may be slightly different (a reversed list, for example). Sometimes, the most straightforward thing to do is to return *computed-so-far* as a test, and then see the relationship.

You continue by asking yourself *How do I simplify the parameters?* Again, for lists the answer is often to take the `cdr`, and for numbers, the answer is often to subtract or divide.

You conclude by asking yourself *How should I update my intermediate result, given the particular information I've discarded by simplifying?* The answer should give you the *update* procedure.

---

Copyright © 2007 Samuel A. Rebelsky. 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.