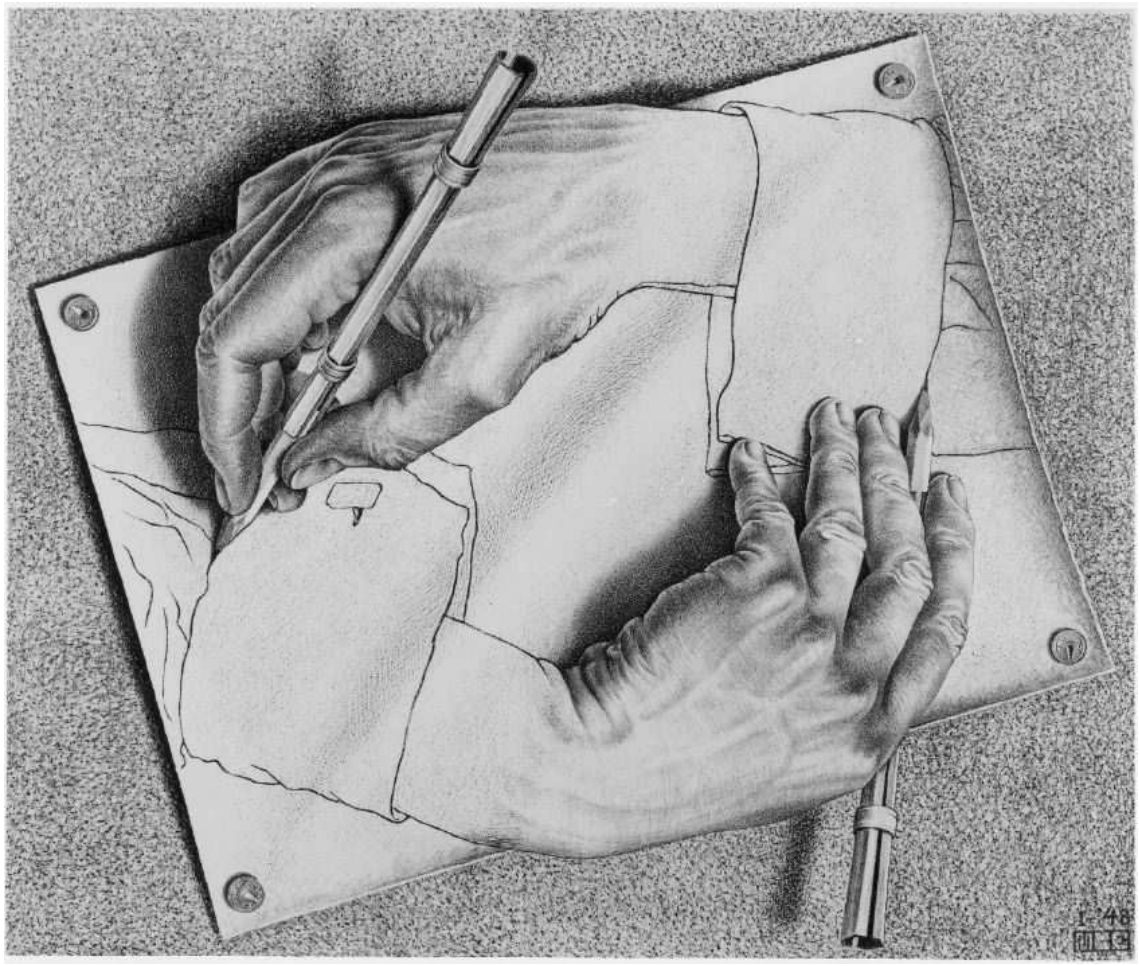

Drawing Hands, by M. C. Escher (lithograph, 1948)

12 The Leap of Faith

In the combining method, we build up to a recursive procedure by writing a number of special-case nonrecursive procedures, starting with small arguments and working toward larger ones. We find a generalizable way to use a smaller version in writing a larger one. As a result, all our procedures end up looking nearly the same, so we combine them into one procedure.

The combining method is a good way to begin thinking about recursion because each step of a solution is clearly justified by earlier steps. The sequence of events by which we get from a problem statement to a Scheme procedure is clear and straightforward. The disadvantage of the combining method, though, is that it involves a lot of drudgery, not all of which really helps toward the ultimate solution. In this chapter we're going to develop a new method called *the leap of faith* that overcomes this difficulty.

From the Combining Method to the Leap of Faith

Let's look again at the way we developed the `letter-pairs` procedure in the last chapter. We went through several steps:

- We wrote specific versions for zero-, one-, two-, and three-letter words.
- We wrote `letter-pairs4`, decided it was too complicated, and looked for a way to use `letter-pairs3` to help.
- Having rewritten `letter-pairs4`, we tried to write `letter-pairs5` using the same pattern. Since it didn't quite work, we revised the pattern.
- We generalized the pattern to write an unnumbered, recursive `letter-pairs`.

- We checked to make sure that the recursive pattern would work for two-letter and three-letter words.
- Since the pattern doesn't work for zero- or one-letter words, we made those the base cases.

Although we needed the lowest numbered procedures in order to make the entire collection of numbered procedures work, those low-numbered ones didn't contribute to the critical step of finding a generalizable pattern. Once you understand the idea of recursion, writing the individual procedures is wasted effort.

In the leap of faith method, we short-circuit this process in two ways. First, we don't bother thinking about small examples; we begin with, for example, a seven-letter word. Second, we don't use our example to write a particular numbered procedure; we write the recursive version directly.

Example: `Reverse`

We're going to write, using the leap of faith method, a recursive procedure to reverse the letters of a word:

```
> (reverse 'beatles)
SELTAEB
```

Is there a `reverse` of a smaller argument lurking within that return value? Yes, many of them. For example, `LTA` is the `reverse` of the word `ATL`. But it will be most helpful if we find a smaller subproblem that's only *slightly* smaller. (This idea corresponds to writing `letter-pairs7` using `letter-pairs6` in the combining method.) The closest smaller subproblem to our original problem is to find the `reverse` of a word one letter shorter than `beatles`.

```
> (reverse 'beatle)
ELTAEB
```

This result is pretty close to the answer we want for `reverse` of `beatles`. What's the relationship between `ELTAEB`, the answer to the smaller problem, and `SELTAEB`, the answer to the entire problem? There's one extra letter, `S`, at the beginning. Where did

the extra letter come from? Obviously, it's the last letter of `beatles`.*

This may seem like a sequence of trivial observations leading nowhere. But as a result of this investigation, we can translate what we've learned directly into Scheme. In English: "the `reverse` of a word consists of its last letter followed by the `reverse` of its `butlast`." In Scheme:

```
(define (reverse wd)                    ;; unfinished
  (word (last wd)
        (reverse (bl wd))))
```

The Leap of Faith

If we think of this Scheme fragment merely as a statement of a true fact about `reverse`, it's not very remarkable. The amazing part is that this fragment is *runnable!*** It doesn't *look* runnable because it invokes itself as a helper procedure, and—if you haven't already been through the combining method—that looks as if it can't work. "How can you use `reverse` when you haven't written it yet?"

The leap of faith method is the assumption that the procedure we're in the middle of writing already works. That is, if we're thinking about writing a `reverse` procedure that can compute `(reverse 'paul)`, we assume that `(reverse 'aul)` will work.

Of course it's not *really* a leap of faith, in the sense of something accepted as miraculous but not understood. The assumption is justified by our understanding of the combining method. For example, we understand that the four-letter `reverse` is relying on the three-letter version of the problem, not really on itself, so there's no circular reasoning involved. And we know that if we had to, we could write `reverse1` through `reverse3` "by hand."

The reason that our technique in this chapter may seem more mysterious than the combining method is that this time we are thinking about the problem top-down. In the combining method, we had already written `whatever3` before we even raised the question of `whatever4`. Now we start by thinking about the larger problem and assume

* There's also a relationship between `(reverse 'eatles)` and `(reverse 'beatles)`, with the extra letter `b` at the end. We could take either of these subproblems as a starting point and end up with a working procedure.

** Well, almost. It needs a base case.

that we can rely on the smaller one. Again, we're entitled to that assumption because we've gone through the process from smaller to larger so many times already.

The leap of faith method, once you understand it, is faster than the combining method for writing new recursive procedures, because we can write the recursive solution immediately, without bothering with many individual cases. The reason we showed you the combining method first is that the leap of faith method seems too much like magic, or like "cheating," until you've seen several believable recursive programs. The combining method is the way to learn about recursion; the leap of faith method is the way to write recursive procedures once you've learned.

The Base Case

Of course, our definition of `reverse` isn't finished yet: As always, we need a base case. But base cases are the easy part. Base cases transform simple arguments into simple answers, and you can do that transformation in your head.

For example, what's the simplest argument to `reverse`? If you answered "a one-letter word" then pick a one-letter word and decide what the result should be:

```
> (reverse 'x)
X
```

`reverse` of a one-letter word should just be that same word:

```
(define (reverse wd)
  (if (= (count wd) 1)
      wd
      (word (last wd)
            (reverse (bl wd)))))
```

Example: `Factorial`

We'll use the leap of faith method to solve another problem that we haven't already solved with the combining method.

The factorial of a number n is defined as $1 \times 2 \times \cdots \times n$. So the factorial of 5 (written "5!") is $1 \times 2 \times 3 \times 4 \times 5$. Suppose you want Scheme to figure out the factorial of some large number, such as 843. You start from the definition: 843! is $1 \times 2 \times \cdots \times 842 \times 843$. Now you have to look for another factorial problem whose answer will help us find the answer to 843!. You might notice that 2!, that is, 1×2, is part of 843!, but that doesn't

help very much because there's no simple relationship between 2! and 843!. A more fruitful observation would be that 842! is $1 \times \cdots \times 842$—that is, all but the last number in the product we're trying to compute. So $843! = 843 \times 842!$. In general, $n!$ is $n \times (n-1)!$. We can embody this idea in a Scheme procedure:

```
(define (factorial n)                          ;; first version
  (* n (factorial (- n 1))))
```

Asking for $(n-1)!$ is the leap of faith. We're expressing an answer we don't know, 843!, in terms of another answer we don't know, 842!. But since 842! is a smaller, similar subproblem, we are confident that the same algorithm will find it.*

Remember that in the `reverse` problem we mentioned that we could have chosen either the `butfirst` or the `butlast` of the argument as the smaller subproblem? In the case of the `factorial` problem we don't have a similar choice. If we tried to subdivide the problem as

$$6! = 1 \times (2 \times 3 \times 4 \times 5 \times 6)$$

then the part in parentheses would *not* be the factorial of a smaller number.**

---

* What makes us confident? We imagine that we've worked on this problem using the combining method, so that we've written procedures like these:

```
(define (factorial1 n)
  1)

(define (factorial2 n)
  (* 2 (factorial1 (- n 1))))

(define (factorial3 n)
  (* 3 (factorial2 (- n 1))))

;; ...

(define (factorial842 n)
  (* 842 (factorial841 (- n 1))))
```

and therefore we're entitled to use those lower-numbered versions in finding the factorial of 843. We haven't actually written them, but we could have, and that's what justifies using them. The reason we can take 842! on faith is that 842 is smaller than 843; it's the smaller values that we're pretending we've already written.

** As it happens, the part in parentheses does equal the factorial of a number, 6 itself. But expressing the solution for 6 in terms of the solution for 6 doesn't lead to a recursive procedure; we have to express this solution in terms of a *smaller* one.

As the base case for `factorial`, we'll use 1! = 1.

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Likely Guesses for Smaller Subproblems

To make the leap of faith method work, we have to find a smaller, similar subproblem whose solution will help solve the given problem. How do we find such a smaller subproblem?

In the examples so far, we've generally found it by finding a smaller, similar *return value* within the return value we're trying to achieve. Then we worked backward from the smaller solution to figure out what smaller argument would give us that value. For example, here's how we solved the `reverse` problem:

original argument	`beatles`
desired return value	`SELTAEB`
smaller return value	`ELTAEB`
corresponding argument	`beatle`
relationship of arguments	`beatle` is `(bl 'beatles)`
relationship of return values	`SELTAEB` is `(word 's 'ELTAEB)`
Scheme expression	`(word (last arg)` `(reverse (bl arg)))`

Similarly, we looked at the definition of 843! and noticed within it the factorial of a smaller number, 842.

But a smaller return value won't necessarily leap out at us in every case. If not, there are some likely guesses we can try. For example, if the problem is about integers, it makes sense to try $n-1$ as a smaller argument. If the problem is about words or sentences, try the `butfirst` or the `butlast`. (Often, as in the `reverse` example, either will be helpful.) Once you've guessed at a smaller argument, see what the corresponding return value should be, then compare that with the original desired return value as we've described earlier.

In fact, these two argument-guessing techniques would have suggested the same subproblems that we ended up using in our two examples so far. The reason we didn't teach these techniques from the beginning is that we don't want you to think they're

essential parts of the leap of faith method. These are just good guesses; they don't always work. When they don't, you have to be prepared to think more flexibly.

Example: `Downup`

Here's how we might rewrite `downup` using the leap of faith method. Start by looking at the desired return value for a medium-sized example:

```
> (downup 'paul)
(PAUL PAU PA P PA PAU PAUL)
```

Since this is a procedure whose argument is a word, we guess that the `butfirst` or the `butlast` might be helpful.

```
> (downup 'aul)
(AUL AU A AU AUL)

> (downup 'pau)
(PAU PA P PA PAU)
```

This is a case in which it matters which we choose; the solution for the `butfirst` of the original argument doesn't help, but the solution for the `butlast` is most of the solution for the original word. All we have to do is add the original word itself at the beginning and end:

```
(define (downup wd)                          ;; no base case
  (se wd (downup (bl wd)) wd))
```

As before, this is missing the base case, but by now you know how to fill that in.

Example: `Evens`

Here's a case in which mindlessly guessing `butfirst` or `butlast` doesn't lead to a very good solution. We want a procedure that takes a sentence as its argument and returns a sentence of the even-numbered words of the original sentence:

```
> (evens '(i want to hold your hand))
(WANT HOLD HAND)
```

We look at `evens` of the `butfirst` and `butlast` of this sentence:

```
> (evens '(want to hold your hand))
(TO YOUR)

> (evens '(i want to hold your))
(WANT HOLD)
```

`Butfirst` is clearly not helpful; it gives all the wrong words. `Butlast` looks promising. The relationship between `evens` of the bigger sentence and `evens` of the smaller sentence is that the last word of the larger sentence is missing from `evens` of the smaller sentence.

```
(define (losing-evens sent)                    ;; no base case
  (se (losing-evens (bl sent))
      (last sent)))
```

For a base case, we'll take the empty sentence:

```
(define (losing-evens sent)
  (if (empty? sent)
      '()
      (se (losing-evens (bl sent))
          (last sent))))

> (losing-evens '(i want to hold your hand))
(I WANT TO HOLD YOUR HAND)
```

This isn't quite right.

It's true that `evens` of `(i want to hold your hand)` is the same as `evens` of `(i want to hold your)` plus the word `hand` at the end. But what about `evens` of `(i want to hold your)`? By the reasoning we've been using, we would expect that to be `evens` of `(i want to hold)` plus the word `your`. But since the word `your` is the fifth word of the argument sentence, it shouldn't be part of the result at all. Here's how `evens` should work:

```
> (evens '(i want to hold your))
(WANT HOLD)

> (evens '(i want to hold))
(WANT HOLD)
```

When the sentence has an odd number of words, its `evens` is the same as the `evens` of its `butlast`.* So here's our new procedure:

```
(define (evens sent)                          ;; better version
  (cond ((empty? sent) '())
        ((odd? (count sent))
         (evens (bl sent)))
        (else (se (evens (bl sent))
                  (last sent)))))
```

This version works, but it's more complicated than necessary. What makes it complicated is that on each recursive call we switch between two kinds of problems: even-length and odd-length sentences. If we dealt with the words two at a time, each recursive call would see the same kind of problem.

Once we've decided to go through the sentence two words at a time, we can reopen the question of whether to go right-to-left or left-to-right. It will turn out that the latter gives us the simplest procedure:

```
(define (evens sent)                          ;; best version
  (if (<= (count sent) 1)
      '()
      (se (first (bf sent))
          (evens (bf (bf sent))))))
```

Since we go through the sentence two words at a time, an odd-length argument sentence always gives rise to an odd-length recursive subproblem. Therefore, it's not good enough to check for an empty sentence as the only base case. We need to treat both the empty sentence and one-word sentences as base cases.

Simplifying Base Cases

The leap of faith is mostly about recursive cases, not base cases. In the examples in this chapter, we've picked base cases without talking about them much. How do you pick a base case?

* It may feel strange that in the case of an odd-length sentence, the answer to the recursive subproblem is the same as the answer to the original problem, rather than a smaller answer. But remember that it's the argument, not the return value, that has to get smaller in each recursive step.

In general, we recommend using the smallest possible base case argument, because that usually leads to the simplest procedures. For example, consider using the empty word, empty sentence, or zero instead of one-letter words, one-word sentences, or one.

How can you go about finding the simplest possible base case? Our first example in this chapter was `reverse`. We arbitrarily chose to use one-letter words as the base case:

```
(define (reverse wd)
  (if (= (count wd) 1)
      wd
      (word (last wd)
            (reverse (bl wd)))))
```

Suppose we want to consider whether a smaller base case would work. One approach is to pick an argument that would be handled by the current base case, and see what would happen if we tried to let the recursive step handle it instead. (To go along with this experiment, we pick a smaller base case, since the original base case should now be handled by the recursive step.) In this example, we pick a one-letter word, let's say `m`, and use that as the value of `wd` in the expression

```
(word (last wd)
      (reverse (bl wd)))
```

The result is

```
(word (last 'm)
      (reverse (bl 'm)))
```

which is the same as

```
(word 'm
      (reverse ""))
```

We want this to have as its value the word `M`. This will work out provided that `(reverse "")` has the empty word as its value. So we could rewrite the procedure this way:

```
(define (reverse wd)
  (if (empty? wd)
      ""
      (word (last word)
            (reverse (bl word)))))
```

We were led to this empty-word base case by working downward from the needs of the one-letter case. However, it's also important to ensure that the return value used for the empty word is the correct value, not only to make the recursion work, but for an empty word in its own right. That is, we have to convince ourselves that `(reverse "")` should return an empty word. But it should; the `reverse` of any word is a word containing the same letters as the original word. If the original has no letters, the `reverse` must have no letters also. This exemplifies a general principle: Although we choose a base case argument for the sake of the recursive step, we must choose the corresponding return value for the sake of the argument itself, not just for the sake of the recursion.

We'll try the base case reduction technique on `downup`:

```
(define (downup wd)
  (if (= (count wd) 1)
      (se wd)
      (se wd (downup (bl wd)) wd)))
```

If we want to use the empty word as the base case, instead of one-letter words, then we have to ensure that the recursive case can return a correct answer for a one-letter word. The behavior we want is

```
> (downup 'a)
(A)
```

But if we substitute `'a` for `wd` in the recursive-case expression we get

```
(se 'a (downup "") 'a)
```

which will have two copies of the word `A` in its value no matter what value we give to `downup` of the empty word. We can't avoid treating one-letter words as a base case.

In writing `factorial`, we used `1` as the base case.

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

Our principle of base case reduction suggests that we try for `0`. To do this, we substitute `1` for `n` in the recursive case expression:

```
(* 1 (factorial 0))
```

We'd like this to have the value 1; this will be true only if we define 0! = 1. Now we can say

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

In this case, the new procedure is no simpler than the previous version. Its only advantage is that it handles a case, 0!, that mathematicians find useful.

Here's another example in which we can't reduce the base case to an empty word. In Chapter 11 we used the combining method to write `letter-pairs`:

```
(define (letter-pairs wd)
  (if (<= (count wd) 1)
      '()
      (se (first-two wd)
          (letter-pairs (bf wd)))))

(define (first-two wd)
  (word (first wd) (first (bf wd))))
```

It might occur to you that one-letter words could be handled by the recursive case, and the base case could then handle only the empty word. But if you try to evaluate the expression for the recursive case as applied to a one-letter word, you find that

```
(first-two 'a)
```

is equivalent to

```
(word (first 'a) (first (bf 'a)))
```

which is an error. There is no second letter of a one-letter word. As soon as you see the expression `(first (bf wd))` within this program, you know that one-letter words must be part of the base case. The same kind of reasoning can be used in many problems; the base case must handle anything that's too small to fit the needs of the recursive case.

Pitfalls

⇒ One possible pitfall is a recursive case that doesn't make progress, that is, one that doesn't reduce the size of the problem in the recursive call. For example, let's say we're trying to write the procedure `down` that works this way:

```
> (down 'town)
(TOWN TOW TO T)
```

Here's an incorrect attempt:

```
(define (down wd)                              ;; wrong!
  (if (empty? wd)
      '()
      (se wd (down (first wd)))))
```

The recursive call looks as if it reduces the size of the problem, but try it with an actual example. What's `first` of the word `splat`? What's `first` of that result? What's `first` of *that* result?

⇒ A pitfall that sounds unlikely in the abstract but is actually surprisingly common is to try to do the second step of the procedure "by hand" instead of trusting the recursion to do it. For example, here's another attempt at that `down` procedure:

```
(define (down wd)                              ;; incomplete
  (se wd ...))
```

You know the first word in the result has to be the argument word. Then what? The next thing is the same word with its last letter missing:

```
(define (down wd)                              ;; wrong!
  (se wd (bl wd) ...))
```

Instead of taking care of the entire rest of the problem with a recursive call, it's tempting to take only one more step, figuring out how to include the second word of the required solution. But that approach won't get you to a general recursive solution. Just take the first step and then trust the recursion for the rest:

```
(define (down wd)
  (if (empty? wd)
      '()
      (se wd (down (bl wd)))))
```

⇒ The value returned in the base case of your procedure must be in the range of the function you are representing. If your function is supposed to return a number, it must return a number all the time, even in the base case. You can use this idea to help you check the correctness of the base case expression.

For example, in `downup`, the base case returns `(se wd)` for the base case argument of a one-letter word. How did we think to enclose the word in a sentence? We know that in the recursive cases `downup` always returns a sentence, so that suggests to us that it must return a sentence in the base case also.

⇒ If your base case doesn't make sense in its own right, it probably means that you're trying to compensate for a mistake in the recursive case.

For example, suppose you've fallen into the pitfall of trying to handle the second word of a sentence by hand, and you've written the following procedure:

```
(define (square-sent sent)                    ;; wrong
  (if (empty? sent)
      '()
      (se (square (first sent))
          (square (first (bf sent)))
          (square-sent (bf sent)))))

> (square-sent '(2 3))
ERROR: Invalid argument to FIRST:  ()
```

After some experimentation, you find that you can get this example to work by changing the base case:

```
(define (square-sent sent)                    ;; still wrong
  (if (= (count sent) 1)
      '()
      (se (square (first sent))
          (square (first (bf sent)))
          (square-sent (bf sent)))))

> (square-sent '(2 3))
(4 9)
```

The trouble is that the base case doesn't make sense on its own:

```
> (square-sent '(7))
()
```

In fact, this procedure doesn't work for any sentences of length other than two. The moral is that it doesn't work to correct an error in the recursive case by introducing an absurd base case.

Boring Exercises

12.1 Here is a definition of a procedure that returns the sum of the numbers in its argument sentence:

```
(define (addup nums)
  (if (empty? (bf nums))
      (first nums)
      (+ (first nums) (addup (bf nums)))))
```

Although this works, it could be simplified by changing the base case. Do that.

12.2 Fix the bug in the following definition:

```
(define (acronym sent)                          ;; wrong
  (if (= (count sent) 1)
      (first sent)
      (word (first (first sent))
            (acronym (bf sent)))))
```

12.3 Can we reduce the `factorial` base case argument from `0` to `-1`? If so, show the resulting procedure. If not, why not?

12.4 Here's the definition of a function f:

$$f(\mathit{sent}) = \begin{cases} \mathit{sent}, & \text{if } \mathit{sent} \text{ is empty;} \\ \texttt{sentence}(f(\texttt{butfirst}(\mathit{sent})), \texttt{first}(\mathit{sent})), & \text{otherwise.} \end{cases}$$

Implement f as a Scheme procedure. What does f do?

Real Exercises

Solve all of the following problems with recursive procedures. If you've read Part III, do not use any higher-order functions such as `every`*,* `keep`*, or* `accumulate`*.*

12.5 [8.8] Write an `exaggerate` procedure which exaggerates sentences:

```
> (exaggerate '(i ate 3 potstickers))
(I ATE 6 POTSTICKERS)

> (exaggerate '(the chow fun is good here))
(THE CHOW FUN IS GREAT HERE)
```

It should double all the numbers in the sentence, and it should replace "good" with "great," "bad" with "terrible," and anything else you can think of.

12.6 [8.11] Write a GPA procedure. It should take a sentence of grades as its argument and return the corresponding grade point average:

```
> (gpa '(A A+ B+ B))
3.67
```

Hint: write a helper procedure `base-grade` that takes a grade as argument and returns 0, 1, 2, 3, or 4, and another helper procedure `grade-modifier` that returns –.33, 0, or .33, depending on whether the grade has a minus, a plus, or neither.

12.7 Write a procedure `spell-number` that spells out the digits of a number:

```
> (spell-number 1971)
(ONE NINE SEVEN ONE)
```

Use this helper procedure:

```
(define (spell-digit digit)
  (item (+ 1 digit)
        '(zero one two three four five six seven eight nine)))
```

12.8 Write a procedure `numbers` that takes a sentence as its argument and returns another sentence containing only the numbers in the argument:

```
> (numbers '(76 trombones and 110 cornets))
(76 110)
```

12.9 Write a procedure `real-words` that takes a sentence as argument and returns all the "real" words of the sentence, using the same rule as the `real-word?` procedure from Chapter 1.

12.10 Write a procedure `remove` that takes a word and a sentence as arguments and returns the same sentence, but with all copies of the given word removed:

```
> (remove 'the '(the song love of the loved by the beatles))
(SONG LOVE OF LOVED BY BEATLES)
```

12.11 Write the procedure `count`, which returns the number of words in a sentence or the number of letters in a word.

12.12 Write a procedure `arabic` which converts Roman numerals into Arabic numerals:

```
> (arabic 'MCMLXXI)
1971

> (arabic 'MLXVI)
1066
```

You will probably find the `roman-value` procedure from Chapter 6 helpful. Don't forget that a letter can *reduce* the overall value if the letter that comes after it has a larger value, such as the `C` in `MCM`.

12.13 Write a new version of the `describe-time` procedure from Exercise 6.14. Instead of returning a decimal number, it should behave like this:

```
> (describe-time 22222)
(6 HOURS 10 MINUTES 22 SECONDS)

> (describe-time 4967189641)
(1 CENTURIES 57 YEARS 20 WEEKS 6 DAYS 8 HOURS 54 MINUTES 1 SECONDS)
```

Can you make the program smart about saying `1 CENTURY` instead of `1 CENTURIES`?