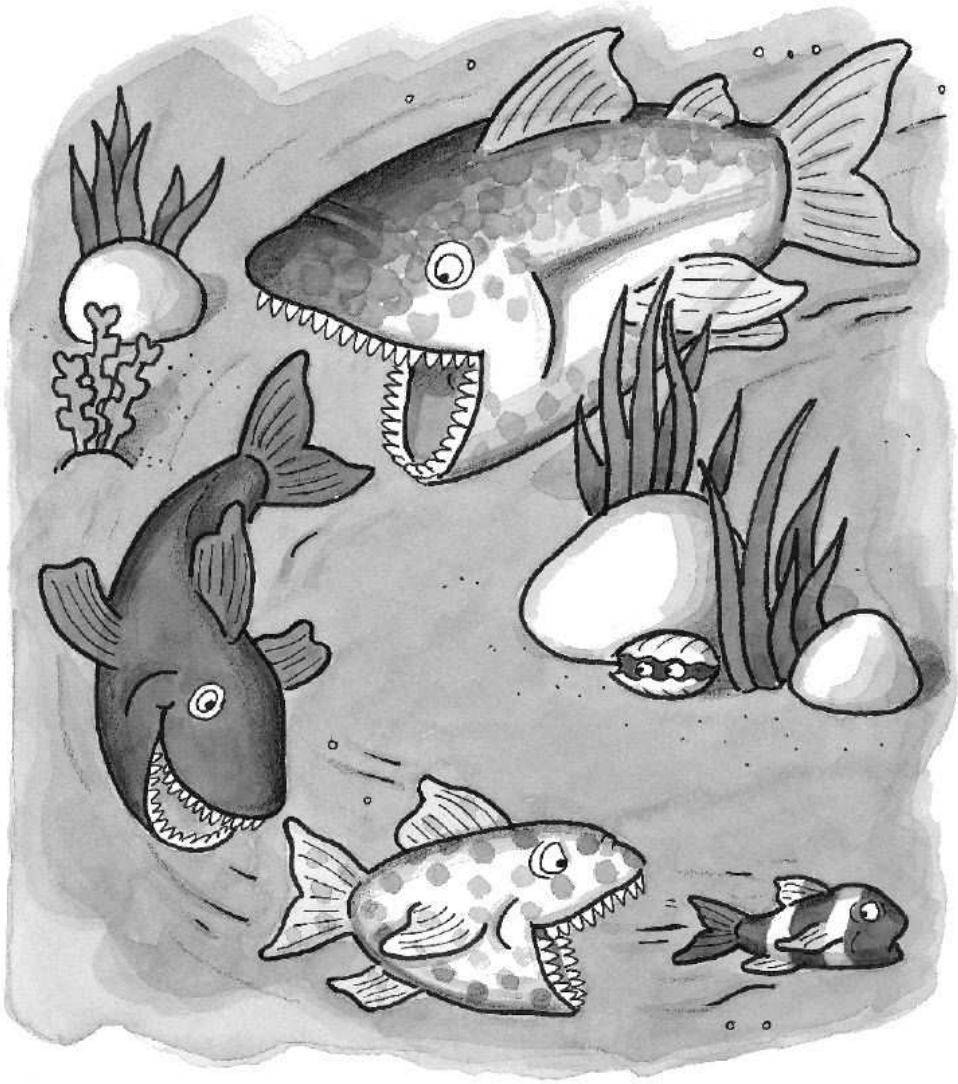

14 Common Patterns in Recursive Procedures

There are two ideas about how to solve programming problems.* One idea is that programmers work mostly by recognizing categories of problems that come up repeatedly and remembering the solution that worked last time; therefore, programming students should learn a lot of program *patterns,* or *templates,* and fill in the blanks for each specific problem. Another idea is that there are a few powerful principles in programming, and that if a learner understands the principles, they can be applied to *any* problem, even one that doesn't fit a familiar pattern.

Research suggests that an expert programmer, like an expert at any skill, *does* work mainly by recognizing patterns. Nevertheless, we lean toward the powerful-principle idea. The expert's memory is not full of arbitrary patterns; it's full of *meaningful* patterns, because the expert has gone through the process of struggling to reason out how each procedure works and how to write new procedures.

Still, we think it's worth pointing out a few patterns that are so common that you'll have seen several examples of each before you finish this book. Once you learn these patterns, you can write similar procedures almost automatically. But there's an irony in learning patterns: In Scheme, once you've identified a pattern, you can write a general-purpose procedure that handles all such cases without writing individual procedures for each situation. Then you don't have to use the pattern any more! Chapter 8 presents several general pattern-handling procedures, called *higher-order* procedures. In this chapter we'll consider the patterns corresponding to those higher-order procedures, and we'll use the names of those procedures to name the patterns.

* That's because there are two kinds of people: those who think there are two kinds of people, and those who don't.

What's the point of learning patterns if you can use higher-order procedures instead? There are at least two points. The first, as you'll see very soon, is that some problems *almost* follow one of the patterns; in that case, you can't use the corresponding higher-order procedure, which works only for problems that *exactly* follow the pattern. But you can use your understanding of the pattern to help with these related problems. The second point is that in Chapter 19 we'll show how the higher-order functions are themselves implemented using these recursive patterns.

This chapter isn't an official list of all important patterns; as you gain programming experience, you'll certainly add more patterns to your repertoire.

The `Every` Pattern

Here's a procedure to square every number in a sentence of numbers:

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

Here's a procedure to translate every word of a sentence into Pig Latin:

```
(define (pigl-sent sent)
  (if (empty? sent)
      '()
      (se (pigl (first sent))
          (pigl-sent (bf sent)))))
```

The pattern here is pretty clear. Our recursive case will do something straightforward to the `first` of the sentence, such as `square`ing it or `pigl`ing it, and we'll combine that with the result of a recursive call on the `butfirst` of the sentence.

The `letter-pairs` procedure that we wrote in Chapter 11 is an example of a procedure that follows the `every` pattern pretty closely, but not exactly. The difference is that `letter-pairs` looks at its argument sentence two words at a time.

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

Compare this with the earlier definition of `square-sent`. The recursive case still uses `se` to combine one part of the result with a recursive call based on the `butfirst` of the argument, but here both the first letter and the second letter of the argument contribute to the first word of the result. That's why the base case also has to be different; the recursive case requires at least two letters, so the base case is a one-letter word.*

Let's solve a slightly different problem. This time, we want to break the word down into *non-overlapping* pairs of letters, like this:

```
> (disjoint-pairs 'tripoli)                    ;; the new problem
(TR IP OL I)

> (letter-pairs 'tripoli)                      ;; compare the old one
(TR RI IP PO OL LI)
```

The main difference between these two functions is that in `disjoint-pairs` we eliminate two letters at once in the recursive call. A second difference is that we have to deal with the special case of odd-length words.

```
(define (disjoint-pairs wd)
  (cond ((empty? wd) '())
        ((= (count wd) 1) (se wd))
        (else (se (word (first wd) (first (bf wd)))
                  (disjoint-pairs (bf (bf wd)))))))
```

The `Keep` Pattern

In the `every` pattern, we collect the results of transforming each element of a word or sentence into something else. This time we'll consider a different kind of problem: choosing some of the elements and forgetting about the others. First, here is a procedure to select the three-letter words from a sentence:

```
(define (keep-three-letter-words sent)
  (cond ((empty? sent) '())
        ((= (count (first sent)) 3)
         (se (first sent) (keep-three-letter-words (bf sent))))
        (else (keep-three-letter-words (bf sent)))))
```

* If you've read Chapter 8, you know that you could implement `square-sent` and `pigl-sent` without recursion, using the `every` higher order function. But try using `every` to implement `letter-pairs`; you'll find that you can't quite make it work.

```
> (keep-three-letter-words '(one two three four five six seven))
(ONE TWO SIX)
```

Next, here is a procedure to select the vowels from a word:

```
(define (keep-vowels wd)
  (cond ((empty? wd) "")
        ((vowel? (first wd))
         (word (first wd) (keep-vowels (bf wd))))
        (else (keep-vowels (bf wd)))))

> (keep-vowels 'napoleon)
AOEO
```

Let's look at the differences between the `every` pattern and the `keep` pattern. First of all, the `keep` procedures have three possible outcomes, instead of just two as in most `every`-like procedures. In the `every` pattern, we only have to distinguish between the base case and the recursive case. In the `keep` pattern, there is still a base case, but there are *two* recursive cases; we have to decide whether or not to keep the first available element in the return value. When we do keep an element, we keep the element itself, not some function of the element.

As with the `every` pattern, there are situations that follow the `keep` pattern only approximately. Suppose we want to look for doubled letters within a word:

```
> (doubles 'bookkeeper)
OOKKEE

> (doubles 'mississippi)
SSSSPP
```

This isn't a pure `keep` pattern example because we can't decide whether to keep the first letter by looking at that letter alone; we have to examine two at a time. But we can write a procedure using more or less the same pattern:

```
(define (doubles wd)
  (cond ((= (count wd) 1) "")
        ((equal? (first wd) (first (bf wd)))
         (word (first wd) (first (bf wd)) (doubles (bf (bf wd)))))
        (else (doubles (bf wd)))))
```

As in the `evens` example of Chapter 12, the base case of `doubles` is unusual, and one of the recursive calls chops off two letters at once in forming the smaller subproblem.

But the structure of the `cond` with a base case clause, a clause for keeping letters, and a clause for rejecting letters is maintained.

The `Accumulate` Pattern

Here are two recursive procedures for functions that combine all of the elements of the argument into a single result:

```
(define (addup nums)
  (if (empty? nums)
      0
      (+ (first nums) (addup (bf nums)))))

(define (scrunch-words sent)
  (if (empty? sent)
      ""
      (word (first sent) (scrunch-words (bf sent)))))

> (addup '(8 3 6 1 10))
28

> (scrunch-words '(ack now ledge able))
ACKNOWLEDGEABLE
```

What's the pattern? We're using some combiner (`+` or `word`) to connect the word we're up to with the result of the recursive call. The base case tests for an empty argument, but the base case return value must be the identity element of the combiner function.

If there is no identity element for the combiner, as in the case of `max`, we modify the pattern slightly:*

```
(define (sent-max sent)
  (if (= (count sent) 1)
      (first sent)
      (max (first sent)
           (sent-max (bf sent)))))
```

* Of course, if your version of Scheme has $-\infty$, you can use it as the return value for an empty sentence, instead of changing the pattern.

Combining Patterns

```
(define (add-numbers sent)
  (cond ((empty? sent) 0)
        ((number? (first sent))
         (+ (first sent) (add-numbers (bf sent))))
        (else (add-numbers (bf sent)))))

> (add-numbers '(if 6 were 9))
15
```

This procedure combines aspects of `keep` with aspects of `accumulate`. We want to do two things at once: get rid of the words that aren't numbers and compute the *sum* of those that are numbers. (A simple `keep` would construct a sentence of them.) `Add-numbers` looks exactly like the `keep` pattern, except that there's a funny combiner and a funny base case, which look more like `accumulate`.*

Here's an example that combines `every` and `keep`. We want a procedure that takes a sentence as its argument and translates every word of the sentence into Pig Latin, but leaves out words that have no vowels, because the Pig Latin translator doesn't work for such words. The procedure `safe-pigl` will be like a `keep` pattern in that it keeps only words that contain vowels, but like an `every` in that the result contains transformed versions of the selected words, rather than the words themselves.

```
(define (safe-pigl sent)
  (cond ((empty? sent) '())
        ((has-vowel? (first sent))
         (se (pigl (first sent)) (safe-pigl (bf sent))))
        (else (safe-pigl (bf sent)))))

(define (has-vowel? wd)
  (not (empty? (keep-vowels wd))))
```

* Here's the higher-order function version, from Chapter 8:

```
(define (add-numbers sent)
  (accumulate + (keep number? sent)))
```

The higher-order function version is more self-documenting and easier to write. The recursive version, however, is slightly more efficient, because it avoids building up a sentence as an intermediate value only to discard it in the final result. If we were writing this program for our own use, we'd probably choose the higher-order function version; but if we were dealing with sentences of length 10,000 instead of length 10, we'd pay more attention to efficiency.

```
> (safe-pigl '(my pet fly is named xyzzy))
(ETPAY ISAY AMEDNAY)
```

Finally, here's an example that combines all three patterns. In Chapter 1 we wrote (using higher-order procedures) the `acronym` procedure, which selects the "real" words of a sentence (the `keep` pattern), takes the first letter of each word (the `every` pattern), and combines these initial letters into a single word (the `accumulate` pattern). In a recursive procedure we can carry out all three steps at once:

```
(define (acronym sent)
  (cond ((empty? sent) "")
        ((real-word? (first sent))
         (word (first (first sent))
               (acronym (bf sent))))
        (else (acronym (bf sent)))))
```

Don't become obsessed with trying to make every recursive problem fit one of the three patterns we've shown here. As we said at the beginning of the chapter, what's most important is that you understand the principles of recursion in general, and understand how versatile recursion is. The patterns are just special cases that happen to come up fairly often.

Helper Procedures

Let's say we want a procedure `every-nth` that takes a number *n* and a sentence as arguments and selects every *n*th word from the sentence.

```
> (every-nth 3 '(with a little help from my friends))
(LITTLE MY)
```

We get in trouble if we try to write this in the obvious way, as a sort of `keep` pattern.

```
(define (every-nth n sent)                    ;; wrong!
  (cond ((empty? sent) '())
        ((= n 1)
         (se (first sent) (every-nth n (bf sent))))
        (else (every-nth (- n 1) (bf sent)))))
```

The problem is with the **`n`** that's in boldface. We're thinking that it's going to be the `n` of the *original* invocation of `every-nth`, that is, 3. But in fact, we've already counted `n` down so that in this invocation its value is 1. (Check out the first half of the same `cond` clause.) This procedure will correctly skip the first two words but will keep all the

words after that point. That's because we're trying to remember two different numbers: the number we should always skip between kept words, and the number of words we still need to skip this time.

If we're trying to remember two numbers, we need two names for them. The way to achieve this is to have our official `every-nth` procedure call a helper procedure that takes an extra argument and does the real work:

```
(define (every-nth n sent)
  (every-nth-helper n n sent))

(define (every-nth-helper interval remaining sent)
  (cond ((empty? sent) '())
        ((= remaining 1)
         (se (first sent)
             (every-nth-helper interval interval (bf sent))))
        (else (every-nth-helper interval (- remaining 1) (bf sent)))))
```

This procedure always calls itself recursively with the same value of `interval`, but with a different value of `remaining` each time. `Remaining` keeps getting smaller by one in each recursive call until it equals 1. On that call, a word is kept for the return value, and we call `every-nth-helper` recursively with the value of `interval`, that is, the *original* value of `n`, as the new `remaining`. If you like, you can think of this combination of an *initialization* procedure and a *helper* procedure as another pattern for your collection.

How to Use Recursive Patterns

One way in which recursive patterns can be useful is if you think of them as templates with empty slots to fill in for a particular problem. Here are template versions of the `every`, `keep`, and `accumulate` patterns as applied to sentences:

```
(define (every-something sent)
  (if (empty? sent)
      '()
      (se (__________ (first sent))
          (every-something (bf sent)))))

(define (keep-if-something sent)
  (cond ((empty? sent) '())
        ((__________? (first sent))
         (se (first sent) (keep-if-something (bf sent))))
        (else (keep-if-something (bf sent)))))
```

```
(define (accumulate-somehow sent)
  (if (empty? sent)
      ___________
      (___________ (first sent)
                   (accumulate-somehow (bf sent)))))
```

Suppose you're trying to write a procedure `first-number` that takes a sentence as its argument and returns the first number in that sentence, but returns the word `no-number` if there are no numbers in the argument. The first step is to make a guess about which pattern will be most useful. In this case the program should start with an entire sentence and select a portion of that sentence, namely one word. Therefore, we start with the `keep` pattern.

```
(define (first-number sent)                    ;; first guess
  (cond ((empty? sent) '())
        ((___________ (first sent))
         (se (first sent) (first-number (bf sent))))
        (else (first-number (bf sent)))))
```

The next step is to fill in the blank. Obviously, since we're looking for a number, `number?` goes in the blank.

The trouble is that this procedure returns *all* the numbers in the given sentence. Now our job is to see how the pattern must be modified to do what we want. The overall structure of the pattern is a `cond` with three clauses; we'll consider each clause separately.

What should the procedure return if `sent` is empty? In that case, there is no first number in the sentence, so it should return `no-number`:

```
((empty? sent) 'no-number)
```

What if the first word of the sentence is a number? The program should return just that number, ignoring the rest of the sentence:

```
((number? (first sent)) (first sent))
```

What if the first word of the sentence isn't a number? The procedure must make a recursive call for the `butfirst`, and whatever that recursive call returns is the answer. So the `else` clause does not have to be changed.

Here's the whole procedure:

```
(define (first-number sent)
  (cond ((empty? sent) 'no-number)
        ((number? (first sent)) (first sent))
        (else (first-number (bf sent)))))
```

After filling in the blank in the `keep` pattern, we solved this problem by focusing on the details of the procedure definition. We examined each piece of the definition to decide what changes were necessary. Instead, we could have focused on the *behavior* of the procedure. We would have found two ways in which the program didn't do what it was supposed to do: For an argument sentence containing numbers, it would return all of the numbers instead of just one of them. For a sentence without numbers, it would return the empty sentence instead of `no-number`. We would then have finished the job by *debugging* the procedure to fix each of these problems. The final result would have been the same.

Problems That Don't Follow Patterns

We want to write the procedure `sent-before?`, which takes two sentences as arguments and returns `#t` if the first comes alphabetically before the second. The general idea is to compare the sentences word by word. If the first words are different, then whichever is alphabetically earlier determines which sentence comes before the other. If the first words are equal, we go on to compare the second words.*

```
> (sent-before? '(hold me tight) '(sun king))
#T

> (sent-before? '(lovely rita) '(love you to))
#F

> (sent-before? '(strawberry fields forever)
                '(strawberry fields usually))
#T
```

Does this problem follow any of the patterns we've seen? It's not an `every`, because the result isn't a sentence in which each word is a transformed version of a word in the arguments. It's not a `keep`, because the result isn't a subset of the words in the arguments. And it's not exactly an `accumulate`. We do end up with a single true or false result, rather than a sentence full of results. But in a typical `accumulate` problem,

* Dictionaries use a different ordering rule, in which the sentences are treated as if they were single words, with the spaces removed. By the dictionary rule, "a c" is treated as if it were "ac" and comes after "ab"; by our rule, "a c" comes before "ab" because we compare the first words ("a" and "ab").

every word of the argument contributes to the solution. In this case only one word from each sentence determines the overall result.

On the other hand, this problem does have something in common with the `keep` pattern: We know that on each invocation there will be three possibilities. We might reach a base case (an empty sentence); if not, the first words of the argument sentences might or might not be relevant to the solution.

We'll have a structure similar to the usual `keep` pattern, except that there's no `se` involved; if we find unequal words, the problem is solved without further recursion. Also, we have two arguments, and either of them might be empty.

```
(define (sent-before? sent1 sent2)
  (cond ((empty? sent1) #t)
        ((empty? sent2) #f)
        ((before? (first sent1) (first sent2)) #t)
        ((before? (first sent2) (first sent1)) #f)
        (else (sent-before? (bf sent1) (bf sent2)))))
```

Although thinking about the `keep` pattern helped us to work out this solution, the result really doesn't look much like a `keep`. We had to invent most of the details by thinking about this particular problem, not by thinking about the pattern.

In the next chapter we'll look at examples of recursive procedures that are quite different from any of these patterns. Remember, the patterns are a shortcut for many common problems, but don't learn the shortcut at the expense of the general technique.

Pitfalls

Review the pitfalls from Chapter 12; they're still relevant.

⇒ How do you test for the base case? Most of the examples in this chapter have used `empty?`, and it's easy to fall into the habit of using that test without thinking. But, for example, if the argument is a number, that's probably the wrong test. Even when the argument is a sentence or a non-numeric word, it may not be empty in the base case, as in the Pig Latin example.

⇒ A serious pitfall is failing to recognize a situation in which you need an extra variable and therefore need a helper procedure. If at each step you need the entire original argument as well as the argument that's getting closer to the base case, you probably need a helper procedure. For example, write a procedure `pairs` that takes a word as argument and returns a sentence of all possible two-letter words made of letters from the argument word, allowing duplicates, like this:

```
> (pairs 'toy)
(TT TO TY OT OO OY YT YO YY)
```

⇒ A simple pitfall, when using a helper procedure, is to write a recursive call in the helper that calls the main procedure instead of calling the helper. (For example, what would have happened if we'd had `every-nth-helper` invoke `every-nth` instead of invoking itself?)

⇒ Some recursive procedures with more than one argument require more than one base case. But some don't. One pitfall is to leave out a necessary base case; another is to include something that looks like a base case but doesn't fit the structure of the program.

For example, the reason `sent-before?` needs two base cases is that on each recursive call, both `sent1` and `sent2` get smaller. Either sentence might run out first, and the procedure should return different values in those two cases.

On the other hand, Exercise 11.7 asked you to write a procedure that has two arguments but needs only one base case:

```
(define (copies num wd)
  (if (= num 0)
      '()
      (se wd (copies (- num 1) wd))))
```

In this example, the `wd` argument *doesn't* get smaller from one invocation to the next. It would be silly to test for `(empty? wd)`.

A noteworthy intermediate case is `every-nth-helper`. It does have two `cond` clauses that check for two different arguments reaching their smallest allowable values, but the `remaining` clause isn't a base case. If `remaining` has the value 1, the procedure still invokes itself recursively.

The only general principle we can offer is that you have to think about what base cases are appropriate, not just routinely copy whatever worked last time.

Exercises

Classify each of these problems as a pattern (`every`, `keep`, or `accumulate`), if possible, and then write the procedure recursively. In some cases we've given an example of invoking the procedure we want you to write, instead of describing it.

14.1

```
> (remove-once 'morning '(good morning good morning))
(GOOD GOOD MORNING)
```

(It's okay if your solution removes the other `MORNING` instead, as long as it removes only one of them.)

14.2

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

14.3

```
> (remdup '(ob la di ob la da))              ;; remove duplicates
(OB LA DI DA)
```

(It's okay if your procedure returns `(DI OB LA DA)` instead, as long as it removes all but one instance of each duplicated word.)

14.4

```
> (odds '(i lost my little girl))
(I MY GIRL)
```

14.5 [8.7] Write a procedure `letter-count` that takes a sentence as its argument and returns the total number of letters in the sentence:

```
> (letter-count '(fixing a hole))
11
```

14.6 Write `member?`.

14.7 Write `differences`, which takes a sentence of numbers as its argument and returns a sentence containing the differences between adjacent elements. (The length of the returned sentence is one less than that of the argument.)

```
> (differences '(4 23 9 87 6 12))
(19 -14 78 -81 6)
```

14.8 Write `expand`, which takes a sentence as its argument. It returns a sentence similar to the argument, except that if a number appears in the argument, then the return value contains that many copies of the following word:

```
> (expand '(4 calling birds 3 french hens))
(CALLING CALLING CALLING CALLING BIRDS FRENCH FRENCH FRENCH HENS)

> (expand '(the 7 samurai))
(THE SAMURAI SAMURAI SAMURAI SAMURAI SAMURAI SAMURAI SAMURAI)
```

14.9 Write a procedure called `location` that takes two arguments, a word and a sentence. It should return a number indicating where in the sentence that word can be found. If the word isn't in the sentence, return `#f`. If the word appears more than once, return the location of the first appearance.

```
> (location 'me '(you never give me your money))
4
```

14.10 Write the procedure `count-adjacent-duplicates` that takes a sentence as an argument and returns the number of words in the sentence that are immediately followed by the same word:

```
> (count-adjacent-duplicates '(y a b b a d a b b a d o o))
3

> (count-adjacent-duplicates '(yeah yeah yeah))
2
```

14.11 Write the procedure `remove-adjacent-duplicates` that takes a sentence as argument and returns the same sentence but with any word that's immediately followed by the same word removed:

```
> (remove-adjacent-duplicates '(y a b b a d a b b a d o o))
(Y A B A D A B A D O)

> (remove-adjacent-duplicates '(yeah yeah yeah))
(YEAH)
```

14.12 Write a procedure `progressive-squares?` that takes a sentence of numbers as its argument. It should return `#t` if each number (other than the first) is the square of the number before it:

```
> (progressive-squares? '(3 9 81 6561))
#T

> (progressive-squares? '(25 36 49 64))
#F
```

14.13 What does the `pigl` procedure from Chapter 11 do if you invoke it with a word like "frzzmlpt" that has no vowels? Fix it so that it returns "frzzmlptay."

14.14 Write a predicate `same-shape?` that takes two sentences as arguments. It should return `#t` if two conditions are met: The two sentences must have the same number of words, and each word of the first sentence must have the same number of letters as the word in the corresponding position in the second sentence.

```
> (same-shape? '(the fool on the hill) '(you like me too much))
#T

> (same-shape? '(the fool on the hill) '(and your bird can sing))
#F
```

14.15 Write `merge`, a procedure that takes two sentences of numbers as arguments. Each sentence must consist of numbers in increasing order. `Merge` should return a single sentence containing all of the numbers, in order. (We'll use this in the next chapter as part of a sorting algorithm.)

```
> (merge '(4 7 18 40 99) '(3 6 9 12 24 36 50))
(3 4 6 7 9 12 18 24 36 40 50 99)
```

14.16 Write a procedure `syllables` that takes a word as its argument and returns the number of syllables in the word, counted according to the following rule: the number of syllables is the number of vowels, except that a group of consecutive vowels counts as one. For example, in the word "soaring," the group "oa" represents one syllable and the vowel "i" represents a second one.

Be sure to choose test cases that expose likely failures of your procedure. For example, what if the word ends with a vowel? What if it ends with two vowels in a row? What if it has more than two consecutive vowels?

(Of course this rule isn't good enough. It doesn't deal with things like silent "e"s that don't create a syllable ("like"), consecutive vowels that don't form a diphthong ("cooperate"), letters like "y" that are vowels only sometimes, etc. If you get bored, see whether you can teach the program to recognize some of these special cases.)

Project: Spelling Names of Huge Numbers

Write a procedure `number-name` that takes a positive integer argument and returns a sentence containing that number spelled out in words:

```
> (number-name 5513345)
(FIVE MILLION FIVE HUNDRED THIRTEEN THOUSAND THREE HUNDRED FORTY FIVE)

> (number-name (factorial 20))
(TWO QUINTILLION FOUR HUNDRED THIRTY TWO QUADRILLION NINE HUNDRED TWO
 TRILLION EIGHT BILLION ONE HUNDRED SEVENTY SIX MILLION SIX HUNDRED
 FORTY THOUSAND)
```

There are some special cases you will need to consider:

- Numbers in which some particular digit is zero
- Numbers like 1,000,529 in which an entire group of three digits is zero.
- Numbers in the teens.

Here are two hints. First, split the number into groups of three digits, going from right to left. Also, use the sentence

```
'(thousand million billion trillion quadrillion quintillion
  sextillion septillion octillion nonillion decillion)
```

You can write this bottom-up or top-down. To work bottom-up, pick a subtask and get that working before you tackle the overall structure of the problem. For example, write a procedure that returns the word `FIFTEEN` given the argument `15`.

To work top-down, start by writing `number-name`, freely assuming the existence of whatever helper procedures you like. You can begin debugging by writing *stub* procedures that fit into the overall program but don't really do their job correctly. For example, as an intermediate stage you might end up with a program that works like this:

```
> (number-name 1428425)                    ;; intermediate version
(1 MILLION 428 THOUSAND 425)
```