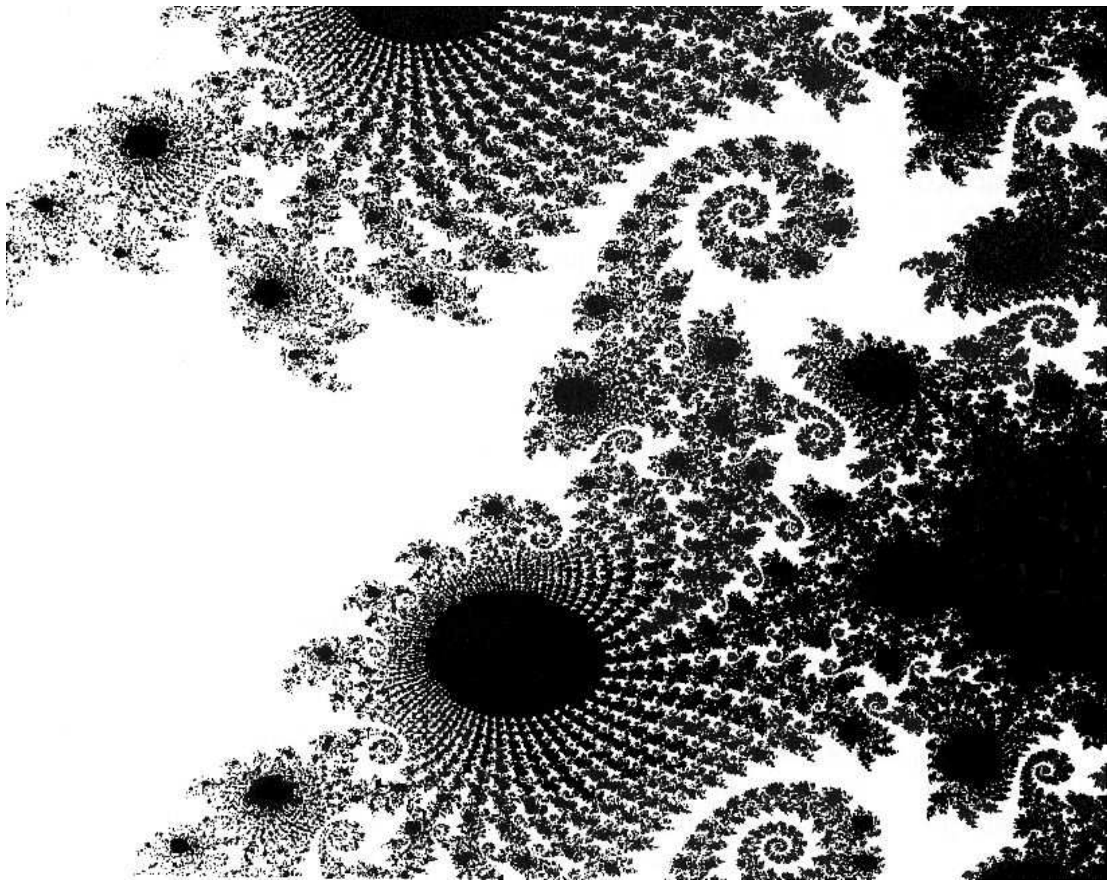

Zoom in on some parts of a fractal and you'll see a miniature version of the whole thing.

15 Advanced Recursion

By now you've had a good deal of experience with straightforward recursive problems, and we hope you feel comfortable with them. In this chapter, we present some more challenging problems. But the same leap of faith method that we used for easier problems is still our basic approach.

Example: `Sort`

First we'll consider the example of sorting a sentence. The argument will be any sentence; our procedure will return a sentence with the same words in alphabetical order.

```
> (sort '(i wanna be your man))
(BE I MAN WANNA YOUR)
```

We'll use the `before?` primitive to decide if one word comes before another word alphabetically:

```
> (before? 'starr 'best)
#F
```

How are we going to think about this problem recursively? Suppose that we're given a sentence to sort. A relatively easy subproblem is to find the word that ought to come first in the sorted sentence; we'll write `earliest-word` later to do this.

Once we've found that word, we just need to put it in front of the sorted version of the rest of the sentence. This is our leap of faith: We're going to assume that we can already sort this smaller sentence. The algorithm we've described is called *selection* sort.

Another subproblem is to find the "rest of the sentence"—all the words except for the earliest. But in Exercise 14.1 you wrote a function `remove-once` that takes a word and a sentence and returns the sentence with that word removed. (We don't want to use `remove`, which removes all copies of the word, because our argument sentence might include the same word twice.)

Let's say in Scheme what we've figured out so far:

```
(define (sort sent)                          ;; unfinished
  (se (earliest-word sent)
      (sort (remove-once (earliest-word sent) sent))))
```

We need to add a base case. The smallest sentence is `()`, which is already sorted.

```
(define (sort sent)
  (if (empty? sent)
      '()
      (se (earliest-word sent)
          (sort (remove-once (earliest-word sent) sent)))))
```

We have one unfinished task: finding the earliest word of the argument.

```
(define (earliest-word sent)
  (earliest-helper (first sent) (bf sent)))

(define (earliest-helper so-far rest)
  (cond ((empty? rest) so-far)
        ((before? so-far (first rest))
         (earliest-helper so-far (bf rest)))
        (else (earliest-helper (first rest) (bf rest)))))*
```

For your convenience, here's `remove-once`:

```
(define (remove-once wd sent)
  (cond ((empty? sent) '())
        ((equal? wd (first sent)) (bf sent))
        (else (se (first sent) (remove-once wd (bf sent))))))
```

* If you've read Part III, you might instead want to use `accumulate` for this purpose:

```
(define earliest-word sent)
  (accumulate (lambda (wd1 wd2) (if (before? wd1 wd2) wd1 wd2))
              sent))
```

Example: `From-Binary`

We want to take a word of ones and zeros, representing a binary number, and compute the numeric value that it represents. Each binary digit (or *bit*) corresponds to a power of two, just as ordinary decimal digits represent powers of ten. So the binary number 1101 represents $(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 13$. We want to be able to say

```
> (from-binary 1101)
13

> (from-binary 111)
7
```

Where is the smaller, similar subproblem? Probably the most obvious thing to try is our usual trick of dividing the argument into its `first` and its `butfirst`. Suppose we divide the binary number `1101` that way. We make the leap of faith by assuming that we can translate the butfirst, `101`, into its binary value 5. What do we have to add for the leftmost `1`? It contributes 8 to the total, because it's three bits away from the right end of the number, so it must be multiplied by 2^3. We could write this idea as follows:

```
(define (from-binary bits)                   ;; incomplete
  (+ (* (first bits) (expt 2 (count (bf bits))))
     (from-binary (bf bits))))
```

That is, we multiply the `first` bit by a power of two depending on the number of bits remaining, then we add that to the result of the recursive call.

As usual, we have written the algorithm for the recursive case before figuring out the base case. But it's pretty easy; a number with no bits (an empty word) has the value zero.*

```
(define (from-binary bits)
  (if (empty? bits)
      0
      (+ (* (first bits) (expt 2 (count (bf bits))))
         (from-binary (bf bits)))))
```

Although this procedure is correct, it's worth noting that a more efficient version can be written by dissecting the number from right to left. As you'll see, we can then avoid the calls to `expt`, which are expensive because we have to do more multiplication than should be necessary.

* A more straightforward base case would be a one-bit number, but we've reduced that to this more elegant base case, following the principle we discussed on page 197.

Suppose we want to find the value of the binary number `1101`. The `butlast` of this number, `110`, has the value six. To get the value of the entire number, we double the six (because `1100` would have the value 12, just as in ordinary decimal numbers 430 is ten times 43) and then add the rightmost bit to get 13. Here's the new version:

```
(define (from-binary bits)
  (if (empty? bits)
      0
      (+ (* (from-binary (bl bits)) 2)
         (last bits))))
```

This version may look a little unusual. We usually combine the value returned by the recursive call with some function of the current element. This time, we are combining the current element itself with a function of the recursive return value. You may want to trace this procedure to see how the intermediate return values contribute to the final result.

Example: `Mergesort`

Let's go back to the problem of sorting a sentence. It turns out that sorting one element at a time, as in selection sort, isn't the fastest possible approach. One of the fastest sorting algorithms is called *mergesort,* and it works like this: In order to mergesort a sentence, divide the sentence into two equal halves and recursively sort each half. Then take the two sorted subsentences and *merge* them together, that is, create one long sorted sentence that contains all the words of the two halves. The base case is that an empty sentence or a one-word sentence is already sorted.

```
(define (mergesort sent)
  (if (<= (count sent) 1)
      sent
      (merge (mergesort (one-half sent))
             (mergesort (other-half sent)))))
```

The leap of faith here is the idea that we can magically `mergesort` the halves of the sentence. If you try to trace this through step by step, or wonder exactly what happens at what time, then this algorithm may be very confusing. But if you just *believe* that the recursive calls will do exactly the right thing, then it's much easier to understand this program. The key point is that if the two smaller pieces have already been sorted, it's pretty easy to merge them while keeping the result in order.

We still need some helper procedures. You wrote `merge` in Exercise 14.15. It uses the following technique: Compare the first words of the two sentences. Let's say the first word of the sentence on the left is smaller. Then the first word of the return value is the first word of the sentence on the left. The rest of the return value comes from recursively merging the `butfirst` of the left sentence with the entire right sentence. (It's precisely the opposite of this if the first word of the other sentence is smaller.)

```
(define (merge left right)
  (cond ((empty? left) right)
        ((empty? right) left)
        ((before? (first left) (first right))
         (se (first left) (merge (bf left) right)))
        (else (se (first right) (merge left (bf right))))))
```

Now we have to write `one-half` and `other-half`. One of the easiest ways to do this is to have `one-half` return the elements in odd-numbered positions, and have `other-half` return the elements in even-numbered positions. These are the same as the procedures `odds` (from Exercise 14.4) and `evens` (from Chapter 12).

```
(define (one-half sent)
  (if (<= (count sent) 1)
      sent
      (se (first sent) (one-half (bf (bf sent))))))

(define (other-half sent)
  (if (<= (count sent) 1)
      '()
      (se (first (bf sent)) (other-half (bf (bf sent))))))
```

Example: `Subsets`

We're now going to attack a much harder problem. We want to know all the subsets of the letters of a word—that is, words that can be formed from the original word by crossing out some (maybe zero) of the letters. For example, if we start with a short word like `rat`, the subsets are `r`, `a`, `t`, `ra`, `rt`, `at`, `rat`, and the empty word (`""`). As the word gets longer, the number of subsets gets bigger very quickly.*

As with many problems about words, we'll try assuming that we can find the subsets of the `butfirst` of our word. In other words, we're hoping to find a solution that will include an expression like

```
(subsets (bf wd))
```

* Try writing down all the subsets of a five-letter word if you don't believe us.

Let's actually take a four-letter word and look at its subsets. We'll pick `brat`, because we already know the subsets of its `butfirst`. Here are the subsets of `brat`:

```
"" b r a t br ba bt ra rt at bra brt bat rat brat
```

You might notice that many of these subsets are also subsets of `rat`. In fact, if you think about it, *all* of the subsets of `rat` are also subsets of `brat`. So the words in `(subsets 'rat)` are some of the words we need for `(subsets 'brat)`.

Let's separate those out and look at the ones left over:

`rat` subsets:	`""`	`r`	`a`	`t`	`ra`	`rt`	`at`	`rat`
others:	`b`	`br`	`ba`	`bt`	`bra`	`brt`	`bat`	`brat`

Right about now you're probably thinking, "They've pulled a rabbit out of a hat, the way my math teacher always does." The words that aren't subsets of `rat` all start with `b`, followed by something that *is* a subset of `rat`. You may be thinking that you never would have thought of that yourself. But we're just following the method: Look at the smaller case and see how it fits into the original problem. It's not so different from what happened with `downup`.

Now all we have to do is figure out how to say in Scheme, "Put a `b` in front of every word in this sentence." This is a straightforward example of the `every` pattern:

```
(define (prepend-every letter sent)
  (if (empty? sent)
      '()
      (se (word letter (first sent))
          (prepend-every letter (bf sent)))))
```

The way we'll use this in `(subsets 'brat)` is

```
(prepend-every 'b (subsets 'rat))
```

Of course in the general case we won't have `b` and `rat` in our program, but instead will refer to the formal parameter:

```
(define (subsets wd)                         ;; first version
  (se (subsets (bf wd))
      (prepend-every (first wd) (subsets (bf wd)))))
```

We still need a base case. By now you're accustomed to the idea of using an empty word as the base case. It may be strange to think of the empty word as a set in the first place, let alone to try to find its subsets. But a set of zero elements is a perfectly good set, and it's the smallest one possible.

The empty set has only one subset, the empty set itself. What should `subsets` of the empty word return? It's easy to make a mistake here and return the empty word itself. But we want `subsets` to return a sentence, containing all the subsets, and we should stick with returning a sentence even in the simple case.* (This mistake would come from not thinking about the *range* of our function, which is sentences. This is why we put so much effort into learning about domains and ranges in Chapter 2.) So we'll return a sentence containing one (empty) word to represent the one subset.

```
(define (subsets wd)                         ;; second version
  (if (empty? wd)
      (se "")
      (se (subsets (bf wd))
          (prepend-every (first wd) (subsets (bf wd))))))
```

This program is entirely correct. Because it uses two identical recursive calls, however, it's a lot slower than necessary. We can use `let` to do the recursive subproblem only once:**

```
(define (subsets wd)
  (if (empty? wd)
      (se "")
      (let ((smaller (subsets (bf wd))))
        (se smaller
            (prepend-every (first wd) smaller)))))
```

Pitfalls

⇒ We've already mentioned the need to be careful about the value returned in the base case. The `subsets` procedure is particularly error-prone because the correct value, a sentence containing the empty word, is quite unusual. An empty subset isn't the same as no subsets at all!

⇒ Sometimes you write a recursive procedure with a correct recursive case and a reasonable base case, but the program still doesn't work. The trouble may be that the base case doesn't quite catch all of the ways in which the problem can get smaller. A

* We discussed this point in a pitfall in Chapter 12.

** How come we're worrying about efficiency all of a sudden? We really *did* pull this out of a hat. The thing is, it's a *lot* slower without the `let`. Adding one letter to the length of a word doubles the time required to find its subsets; adding 10 letters multiplies the time by about 1000.

second base case may be needed. For example, in `mergesort`, why did we write the following line?

```
(<= (count sent) 1)
```

This tests for two base cases, empty sentences and one-word sentences, whereas in most other examples the base case is just an empty sentence. Suppose the base case test were `(empty? sent)` and suppose we invoke `mergesort` with a one-word sentence, `(test)`. We would end up trying to compute the expression

```
(merge (mergesort (one-half '(test)))
       (mergesort (other-half '(test))))
```

If you look back at the definitions of `one-half` and `other-half`, you'll see that this is equivalent to

```
(merge (mergesort '(test)) (mergesort '()))
```

The first argument to `merge` is the same expression we started with! Here is a situation in which the problem doesn't get smaller in a recursive call. Although we've been trying to avoid complicated base cases, in this situation a straightforward base case isn't enough. To avoid an infinite recursion, we must have two base cases.

Another example is the `fib` procedure from Chapter 13. Suppose it were defined like this:

```
(define (fib n)                                ;; wrong!
  (if (= n 1)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

It would be easy to make this mistake, because everybody knows that in a recursion dealing with numbers, the base case is the smallest possible number. But in `fib`, each computation depends on *two* smaller values, and we discover that we need two base cases.

⇒ The technique of recursion is often used to do something repetitively, but don't get the idea that the word "recursion" *means* repetition. Recursion is a technique in which a procedure invokes itself. We do use recursion to solve repetitive problems, but don't confuse the method with the ends it achieves. In particular, if you've programmed in other languages that have special-purpose looping mechanisms (the ones with names like `for` and `while`), those aren't recursive. Conversely, not every recursive procedure carries out a repetition.

Exercises

15.1 Write a procedure `to-binary`:

```
> (to-binary 9)
1001

> (to-binary 23)
10111
```

15.2 A "palindrome" is a sentence that reads the same backward as forward. Write a predicate `palindrome?` that takes a sentence as argument and decides whether it is a palindrome. For example:

```
> (palindrome? '(flee to me remote elf))
#T

> (palindrome? '(flee to me remote control))
#F
```

Do not reverse any words or sentences in your solution.

15.3 Write a procedure `substrings` that takes a word as its argument. It should return a sentence containing all of the substrings of the argument. A *substring* is a subset whose letters come consecutively in the original word. For example, the word `bat` is a subset, but *not* a substring, of `brat`.

15.4 Write a predicate procedure `substring?` that takes two words as arguments and returns `#t` if and only if the first word is a substring of the second. (See Exercise 15.3 for the definition of a substring.)

Be careful about cases in which you encounter a "false start," like this:

```
> (substring? 'ssip 'mississippi)
#T
```

and also about subsets that don't appear as consecutive letters in the second word:

```
> (substring? 'misip 'mississippi)
#F
```

15.5 Suppose you have a phone number, such as 223-5766, and you'd like to figure out a clever way to spell it in letters for your friends to remember. Each digit corresponds to three possible letters. For example, the digit 2 corresponds to the letters A, B, and C. Write a procedure that takes a number as argument and returns a sentence of all the possible spellings:

```
> (phone-spell 2235766)
(AADJPMM AADJPMN ... CCFLSOO)
```

(We're not showing you all 2187 words in this sentence.) You may assume there are no zeros or ones in the number, since those don't have letters.

Hint: This problem has a lot in common with the subsets example.

15.6 Let's say a gladiator kills a roach. If we want to talk about the roach, we say "the roach the gladiator killed." But if we want to talk about the gladiator, we say "the gladiator that killed the roach."

People are pretty good at understanding even rather long sentences as long as they're straightforward: "This is the farmer who kept the cock that waked the priest that married the man that kissed the maiden that milked the cow that tossed the dog that worried the cat that killed the rat that ate the malt that lay in the house that Jack built." But even a short *nested* sentence is confusing: "This is the rat the cat the dog worried killed." Which rat was that?

Write a procedure `unscramble` that takes a nested sentence as argument and returns a straightforward sentence about the same cast of characters:

```
> (unscramble '(this is the roach the gladiator killed))
(THIS IS THE GLADIATOR THAT KILLED THE ROACH)

> (unscramble '(this is the rat the cat the dog the boy the
                      girl saw owned chased bit))
(THIS IS THE GIRL THAT SAW THE BOY THAT OWNED THE DOG THAT
      CHASED THE CAT THAT BIT THE RAT)
```

You may assume that the argument has exactly the structure of these examples, with no special cases like "that lay *in* the house" or "that *Jack* built."

Project: Scoring Poker Hands

The idea of this project is to invent a procedure `poker-value` that works like this:

```
> (poker-value '(h4 s4 c6 s6 c4))
(FULL HOUSE - FOURS OVER SIXES)

> (poker-value '(h7 s3 c5 c4 d6))
(SEVEN-HIGH STRAIGHT)

> (poker-value '(dq d10 dj da dk))
(ROYAL FLUSH - DIAMONDS)

> (poker-value '(da d6 d3 c9 h6))
(PAIR OF SIXES)
```

As you can see, we are representing cards and hands just as in the Bridge project, except that poker hands have only five cards.*

Here are the various kinds of poker hands, in decreasing order of value:

- Royal flush: ten, jack, queen, king, and ace, all of the same suit
- Straight flush: five cards of sequential rank, all of the same suit
- Four of a kind: four cards of the same rank
- Full house: three cards of the same rank, and two of a second rank
- Flush: five cards of the same suit, not sequential rank
- Straight: five cards of sequential rank, not all of the same suit
- Three of a kind: three cards of the same rank, no other matches

* Later on we'll think about seven-card variants of poker.

- Two pair: two pairs of cards, of two different ranks
- Pair: two cards of the same rank, no other matches
- Nothing: none of the above

An ace can be the lowest card of a straight (ace, 2, 3, 4, 5) or the highest card of a straight (ten, jack, queen, king, ace), but a straight can't "wrap around"; a hand with queen, king, ace, 2, 3 would be worthless (unless it's a flush).

Notice that most of the hand categories are either entirely about the ranks of the cards (pairs, straight, full house, etc.) or entirely about the suits (flush). It's a good idea to begin your program by separating the rank information and the suit information. To check for a straight flush or royal flush, you'll have to consider both kinds of information.

In what form do you want the suit information? Really, all you need is a true or false value indicating whether or not the hand is a flush, because there aren't any poker categories like "three of one suit and two of another."

What about ranks? There are two kinds of hand categories involving ranks: the ones about equal ranks (pairs, full house) and the ones about sequential ranks (straight). You might therefore want the rank information in two forms. A sentence containing all of the ranks in the hand, in sorted order, will make it easier to find a straight. (You still have to be careful about aces.)

For the equal-rank categories, what you want is some data structure that will let you ask questions like "are there three cards of the same rank in this hand?" We ended up using a representation like this:

```
> (compute-ranks '(q 3 4 3 4))
(ONE Q TWO 3 TWO 4)
```

One slightly tricky aspect of this solution is that we spelled out the numbers of cards, `one` to `four`, instead of using the more obvious `(1 Q 2 3 2 4)`. The reason, as you can probably tell just by looking at the latter version, is that it would lead to confusion between the names of the ranks, most of which are digits, and the numbers of occurrences, which are also digits. More specifically, by spelling out the numbers of occurrences, we can use `member?` to ask easily if there is a three-of-a-kind rank in the hand.

You may find it easier to begin by writing a version that returns only the name of a category, such as `three of a kind`, and only after you get that to work, revise it to give more specific results such as `three sixes`.

Extra Work for Hotshots

In some versions of poker, each player gets seven cards and can choose any five of the seven to make a hand. How would it change your program if the argument were a sentence of seven cards? (For example, in five-card poker there is only one possible category for a hand, but in seven-card you have to pick the best category that can be made from your cards.) Fix your program so that it works for both five-card and seven-card hands.

Another possible modification to the program is to allow for playing with "wild" cards. If you play with "threes wild," it means that if there is a three in your hand you're allowed to pretend it's whatever card you like. For this modification, your program will require a second argument indicating which cards are wild. (When you play with wild cards, there's the possibility of having five of a kind. This beats a straight flush.)