# 13 Planning

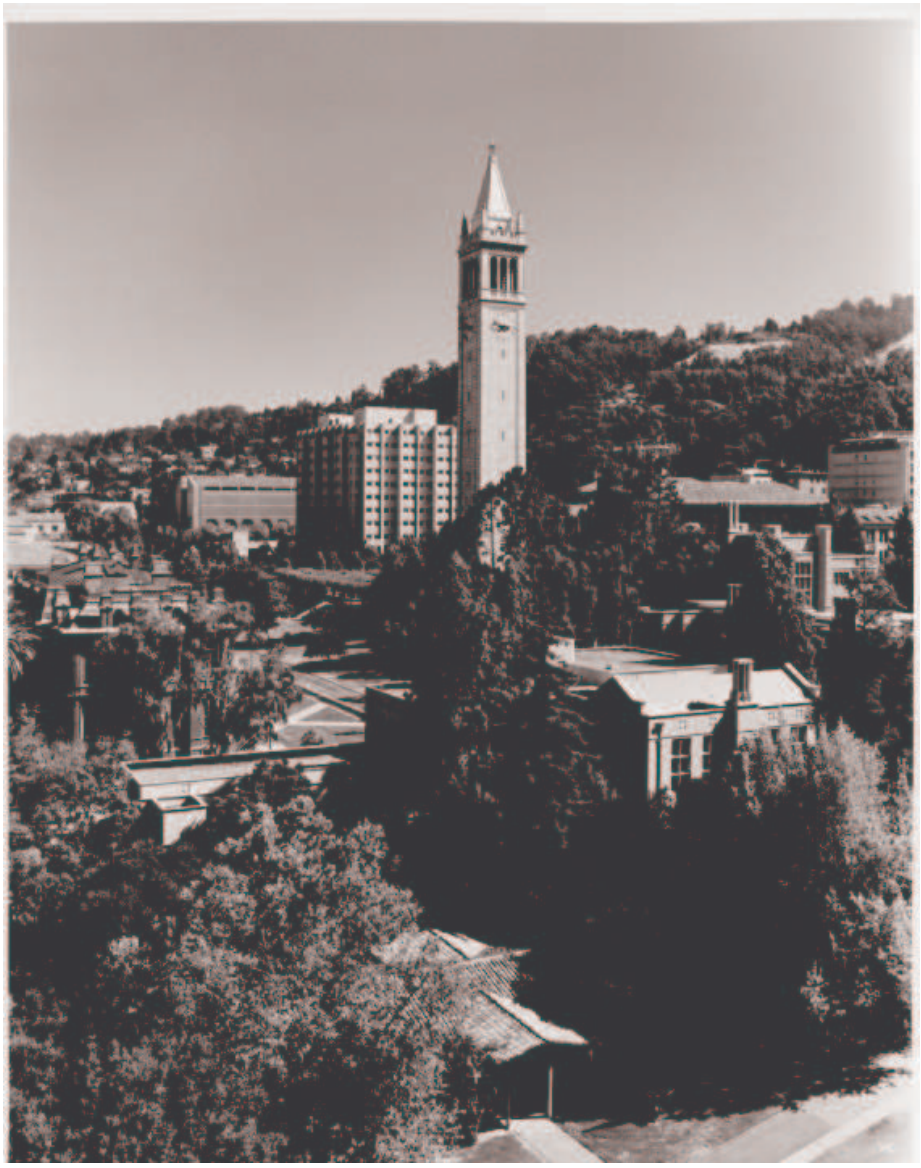
Program file for this chapter: `poker`

The picture on page 234 shows some of the architecture on the University of California campus at Berkeley. At the left of the picture is South Hall, one of the original campus buildings, with red brick, ivy, and many chimneys. The white brick clock tower that dominates the center of the picture is Sather Tower, popularly called the Campanile after the building in Venice, Italy, on which it is modeled. Just to its left is Evans Hall, the concrete fortress that houses the Mathematics Department. Andrews Hall, at the very front of the picture, is a small, one-floor building with an unusually shaped roof. Stephens Hall, mostly hidden by the trees behind Andrews, is a yellow-green zigzag.

Page 235 shows a similar view of the Stanford University campus in Palo Alto, California. Compared to the Berkeley buildings, the ones you see here look very uniform. At the left in the first photo is a corner of the Quadrangle, the central building complex of the campus. The School of Education, down the path on the left, follows the same pattern of rough tan stone with a sloping orange roof. The Meyer Library, at the rear of the photo, follows the same color scheme, even though it's obviously a more recent building. The second photo shows the new School of Law. In this building the architect has clearly worked to combine the same tan stone, orange roof theme with more modern details: the texture of the stone is more uniform and the arches are less ornate.

Both of these campuses are the result of architectural planning, but they illustrate two different *styles* of planning. The Stanford campus was planned *top-down;* first came an overall concept and then the details to fill in that concept. The Berkeley campus was planned *bottom-up;* each new building was designed to fit its architect's idea of the immediate, local situation. Some of the individual buildings are quite beautiful, but it's those buildings, rather than the campus as a unit, that attract attention.

(I'm oversimplifying, of course. In a strictly top-down approach, the entire campus would be laid out on paper before any building was built. Adding new buildings later,



University of California, Berkeley

Stanford University

even if they're made to fit in with the old ones, means that the original plan was defective. Instead of patching it up, a top-down purist would have the architects begin all over again, allowing for more buildings from the beginning of the design process. And in fact the original Berkeley campus was much more uniform than the campus today, but very rapid growth led to widespread changes in the original plan. Still, the difference in architectural planning styles is striking and suggestive.)

The same two planning strategies are possible in computer programming. Suppose you want to write a program to play tic-tac-toe, as I did in Chapter 6. You can start by saying, "Let's see if I can draw the board." You'd write a procedure to draw the four lines that make up the tic-tac-toe grid. Then you might write procedures to draw an X and an O in the right size for the boxes you've made. And so on. That would be a bottom-up design. Alternatively, you might start by deciding on the major tasks that your program will have to carry out. You might then write a top-level procedure like this:

```
to ttt
drawboard
choose.x.o
playgame
end

to playgame
move "x
if winp "x [stop]
move "o
if winp "o [stop]
playgame
end
```

In writing `ttt` and `playgame`, I've freely used subprocedures that I haven't written yet. Later I could fill in the gaps, writing procedures that will do exactly what's needed to fill their places in the main procedure.

## Structured Programming

In recent years the majority of computer scientists have adopted a school of thought called structured programming. This phrase—the title of a 1972 book by O. J. Dahl, Edsger Dijkstra, and C. A. R. Hoare—describes an uncompromising top-down philosophy of programming. Structured programming is more than just the top-down idea, though; it also includes rules for each step in the program development process. For example, one potential problem with top-down programming is that it's hard to test a procedure

you've written until its subprocedures are written also. (By contrast, a subprocedure can be tested before its superprocedures are written.) Structured programming solves this problem by recommending the use of *stubs*—preliminary versions of the subprocedures that don't really do the job but provide some result that allows the higher-level procedures to be tested. For example, an operation that hasn't been written yet might be replaced by a stub that always outputs zero, or the empty list, or some other simple, appropriate value.

More importantly, the structured programming approach tells us not to write any procedures at all until we've first written a detailed specification of the how the program should behave, and then a detailed plan of how it will be organized to achieve that goal.

The programming language Pascal was designed by Niklaus Wirth in 1970 to promote a programming style and philosophy like that of structured programming. Pascal is meant to teach a top-down structured style by providing just the tools needed for that approach but making it hard to program in other styles. The widespread use of Pascal in college programming courses reflects the popularity of the structured programming approach.

(As I am preparing the second edition of this book in 1995, Pascal is just beginning to lose ground as a teaching language; several competing schools of thought about programming have led to diverse language choices. The best known right now is the language C++, which exemplifies an *object oriented* approach to program structure. Others are Ada and Modula, two languages more or less in the Pascal tradition, and Scheme, which is, like Logo, a dialect of Lisp and represents the artificial intelligence tradition.)

## Critique of Structured Programming

One area of computer science in which the top-down approach has not been accepted so enthusiastically is artificial intelligence. AI researchers try to program computers to carry out ill-defined, complex tasks (playing chess is a prototypical example) for which there is no single, obvious method. In that kind of research project you can't start by writing down on paper a complete specification of how the finished program will be organized. Instead you start with a more or less vague idea, you try programming it, and then you play around with it to try to improve the results. That's one reason why the majority of AI programs are written in Lisp, a language that is interactive, so it encourages you to "program at the keyboard." Pascal, on the other hand, was designed to be a *compiled* language, in which you must write an entire program before you can carry out a single instruction.

Logo, a dialect of Lisp, was developed by artificial intelligence researchers. Their idea was to see if they could use some of their experience with the problem of trying to

get computers to think in order to help human beings learn to think more effectively—at least about certain kinds of problems. You shouldn't be surprised, therefore, to learn that Logoites tend not to be enthusiastic about structured programming.

It's not that we're against planning. On the contrary! Planning is one of the most fundamental problem-solving skills. But there are many kinds of planning. The kind in which every part of your program's behavior is written down before you begin programming isn't very realistic in many contexts. Even in the large-scale business or government projects that structured programmers like to talk about, it's very common that the ultimate users of a program change their minds about how it should work, once they have some experience with using it. The wise programmer will anticipate these changing requirements in the original planning process. Still, one never anticipates everything; a sensible person faced with an unexpected change in requirements will be flexible enough to modify the initial plan, not start all over again. And it's even more true for people like you, who are just learning to program, that the "goal" of a programming project is exploratory rather than predetermined.

Sometimes human lives depend on the correct operation of a computer program. In one famous example, just about the time that the first edition of this book was published, one person died and others were injured because the program controlling a medical X-ray machine gave patients massive overdoses of radiation. Certainly, any programming techniques that can help prevent such accidents are valuable. Still, the techniques applicable to life-or-death programming situations are not necessarily the best techniques for beginning learners, nor even for experienced researchers who are exploring a new area rather than writing production programs.

## A Sample Project: Counting Poker Hands

To make all this more concrete, I'd like to show you an actual planning process for a programming project. I'm going to write a Logo program and tell you what I'm doing as I go along. I am sitting at a rather crowded desk; on my left is a microcomputer running Logo, and on my right is the terminal with which I call up the large computer I use for text editing. I'll switch back and forth as I work.* Please understand that I'm not showing

* Home computers have become more powerful since I wrote that in 1984. I can now run Logo in one window and edit the book in another window on the same computer.

you the Official Logo Programming Style. I'm showing you one way in which one Logo programmer approaches a particular project.

The project I have in mind is to announce the value of a poker hand. That is, the program should behave something like this:

```
? pokerhand [3s qc 3d 3h qd]
full house (threes and queens)
? pokerhand [4c 7h 5d 3c 6d]
straight (seven high)
? pokerhand [2h 10d 5s 6s 10s]
pair of tens
?
```

In imagining this sample dialogue, I'm doing a kind of top-down planning. I've specified, for example, the form in which I intend to represent a hand (a list of five cards) and a card (a word combining a rank from

```
[a 2 3 4 5 6 7 8 9 10 j q k]
```

with a suit from

```
[h s d c]
```

standing for hearts, spades, diamonds, and clubs). I suppose that means that I've decided we're playing five-card draw poker rather than seven-card stud. But later I may want to think again about that choice. I've also written down a few of the specific messages the program can print, although I'm much less certain about these. I may or may not actually bother with the details in parentheses, for example.

Okay, how will the program work? I envision a series of tests for particular kinds of poker hands. So in my head there is a vague procedure template like this:

```
to pokerhand :cards
if royal.flushp :cards [print [royal flush] stop]
if fourp :cards [print [four of a kind] stop]
if straight.flushp :cards [print [straight flush] stop]
...
print [I suggest you fold.]
end
```

This isn't something I'm ready to type into my computer. I'm still thinking about how the details are likely to work out. One thing that comes to mind is that, as it stands, there will be a great duplication of effort. The test for a royal flush is just like the test for a straight flush, plus a particular special condition (ace high). I shouldn't really make that test twice. For that matter the test for a straight flush is the test for a straight combined with the test for a flush. I shouldn't have another instruction starting `if straightp` repeating the same test.

## An Initialization Procedure

I also shouldn't read through the list representing the hand a million times, each time pulling out the rank without the suit or vice versa. It seems that I should begin by going through the hand *once,* extracting various kinds of information into a bunch of variables. I'll probably have `ranks` and `suits`, along with things like `pairs`, which will list the ranks that appear twice in the hand. I'm not sure exactly what variables I'll need, but I am now impatient to start programming. What I'm going to do is write an initialization procedure to set up all this information.

In revising this chapter for the second edition, I find that I have very different ideas about how to write this initialization procedure. But I think that it's worthwhile, since this is a chapter about planning a program and not about the finished product, to preserve my original version and the reasoning that led me to write it. In the next section I'll show another approach.

```
to poker.init :cards                    ;; first edition version
make "ranks []
make "suits []
make "pairs []
make "threes []
make "fours []
read.cards :cards
end
```

`Read.cards`, when I write it, will insert new members into the lists that `poker.init` sets up as empty lists. Why `pairs`, `threes`, and `fours` but not, for example, `straights` and `flushes`? Pairhood is a property of just part of a hand, whereas straighthood is a property of the entire hand. It doesn't make sense to say that three of the five cards form a straight. But the lists `:ranks` and `:suits` will help in determining whether the hand is a straight or a flush, respectively. For instance, a flush is a hand in which there is only

one suit, so if `:suits` turns out to be a list of length one, the hand is a flush. A full house is a hand with one rank listed in `:threes` and another listed in `:pairs`.

I seem to be violating my own rules here, with all these explicit assignments to variables that are not made `local`. But of course the whole point of an initialization procedure is that the variables will be used later by some other procedure, not this one or one of its subprocedures. In a large project, it's typical for an initialization procedure to assign values to nonlocal variables. If I'm being careful, when I get around to writing the top-level `pokerhand` I'll probably put `local` instructions for these variables there.

I can write `read.cards` without thinking about it at all, and I hope you can too. It's one of the standard templates: "Do something to each member of a list."

```
to read.cards :cards
foreach :cards "read.card
end
```

It's not obvious what goes inside `read.card`, but I can imagine some of the instructions. So I'll start writing it anyway.

```
to read.card :card
make "ranks fput butlast :card :ranks
make "suits fput last :card :suits
...
```

Okay, time to do some thinking. I can see that there are going to be a lot of those `make`, `fput` instructions. I should have a subprocedure to handle it.

```
to read.card :card
insert butlast :card "ranks
insert last :card "suits
...
```

(By the way, do you see why I extract the rank of a card with `butlast` rather than `first`? It wouldn't matter, except for the tens where the rank is two digits. That is, the ten of spades is represented by the word `10s`. The `first` of that word is the single digit `1`; its `butlast` is `10`, the number we really want.)

I know that the second input to `insert` has to be the name of the variable (like `"ranks`) and not the value of the variable (like `:ranks`) because I've used techniques like this before. That input is going to be used, among other things, as the first input to a `make` invocation. `Make` needs the *name* of the variable in order to be able to change its

value. Although this particular notation is specific to Logo, most programming languages have some way to distinguish between *call by value* (`:ranks`) and *call by name* (`"ranks`) or some similar mechanism to handle the special cases in which a subprocedure must be able to modify a superprocedure's variable.

What about `pairs` and so on? The idea is that if I've seen this particular rank before, I should insert it in `pairs`:

```
if memberp butlast :card :ranks [insert butlast :card "pairs]
```

But there's a bug. If I put that instruction after the ones I've already written, the rank will *always* be found in `:ranks` because I've just put it there! Instead I have to put this instruction *before* the one that inserts into `:ranks`. In fact, the same problem will arise with the other lists. I have to start by testing `:threes` and inserting into `:fours`, and work my way down to `:ranks`. This illustrates a general rule: *Always make the most restrictive test first.* I learned that rule through hours of debugging earlier projects; now I recognize the situation right away. Here's the finished procedure:

```
to read.card :card
insert last :card "suits
if memberp butlast :card :threes [insert butlast :card "fours stop]
if memberp butlast :card :pairs [insert butlast :card "threes stop]
if memberp butlast :card :ranks [insert butlast :card "pairs stop]
insert butlast :card "ranks
end
```

The `stop` commands are just for efficiency. Suppose I've found a particular rank three times already in this poker hand, and the card I'm looking at now is the fourth of the same rank. Then the first `if` will succeed, since the rank was already a member of `:threes`. If the `stop` command were omitted, I'd go on to the next `if` instruction, which would find the rank in `:pairs` and therefore insert it into `:threes`. But that's unnecessary; if I've found the rank in `:threes`, there is no need to insert it there again! In other words, if I'm about to insert this card in the list of `fours`, there is no need to check to see if it's in the lists of smaller runs of the same rank. (Of course, it's sort of funny having `threes` and `fours` as lists, since there can't be more than one of them in a five-card poker hand! But this structure makes the instructions pleasingly similar.)

I notice another potential bug. When I add a rank to, for example, `:threes`, I don't remove it from `:pairs`. So my data base will claim that I have a pair as well as three of a kind. I could write a `remove` procedure analogous to `insert`, but my guess is that it won't be necessary. If I follow the "most restrictive test first" principle later in the

program, I'll know I have three of a kind before I ever look at `:pairs`. If it turns out to be a problem later, I'll fix it then.

I'm slightly annoyed that this procedure computes `butlast :card` so many times. Perhaps it should be

```
to read.card :card
local "rank
make "rank butlast :card
...
```

But in fact I haven't bothered making that change.

Finally, here is the missing subprocedure `insert`:

```
to insert :item :list
if memberp :item thing :list [stop]
make :list fput :item thing :list
end
```

The first instruction is there to ensure that nothing is added to the same list twice. The `stop` commands I mentioned earlier ought to ensure the same thing, except for the list `:suits`. But since I need the instruction for that case anyway, I'll take a "belt and suspenders" approach for all the lists.

The input that I've called `item` here used to be called `thing`, because I was thinking, "Insert a thing into a list." But I found that using the procedure `thing` next to the variable `thing` looked too confusing to me, even though it wouldn't have bothered the Logo interpreter.

I hope you noticed that the second instruction starts with `make :list` rather than `make "list`. This is the indirect assignment technique that I mentioned briefly in Chapter 3. Remember that the variable `list` contains the *name* of another variable, such as `threes`. It is that second variable whose value is changed. For example,

```
insert butlast :card "fours
```

invokes `insert` with an input whose name is `list` and whose value is `fours`. In this case, the `make` instruction inside `insert` is equivalent to

```
make "fours fput :item thing "fours
```

or

```
make "fours fput :item :fours
```

## Second Edition Second Thoughts

I wrote the first edition using versions of Logo without higher order functions. These functions can be written in Logo, and in fact I did write them later in the book, but I wasn't using them in this chapter. But in retrospect, the style of creating a variable named `ranks` whose value is an empty list, and then adding the rank of each card by reassigning a new value to the variable, seems much harder to understand than this:

```
to poker.init :cards
make "ranks map "butlast :cards
make "suits remdup map "last :cards
...
```

`Remdup` is an operation, primitive in Berkeley Logo, whose output is the same as its input, but with duplicate members removed.

As for `pairs`, `threes`, and `fours`, I think they are most easily replaced by an array that keeps track of the number of times each rank appears in the hand.

```
to poker.init :cards                          ;; second edition version
make "ranks map [ranknum butlast ?] :cards
make "suits remdup map "last :cards
make "rankarray {0 0 0 0 0 0 0 0 0 0 0 0 0}
foreach :ranks [setitem ? :rankarray (item ? :rankarray)+1]
end

to ranknum :rank                              ;; turn rank to number
if :rank = "a [output 1]
if :rank = "j [output 11]
if :rank = "q [output 12]
if :rank = "k [output 13]
output :rank
end
```

Since I want to use the card's rank as an index into an array, I have to use a number from 1 to 13 to represent the ranks inside the program, even though the person using the program will still represent a rank in the more human-readable form of A for ace and so on.

Where the first version of the program would test for four of a kind with

```
if not emptyp :fours ...
```

this new version will say

```
if memberp 4 :rankarray ...
```

Notice that my second thoughts are about low-level details of the program. I haven't changed my mind about the big idea, which is to have a procedure `poker.init` that examines the hand and converts the information into a format in which the rest of the program can use it more easily. This is the same idea I used in the tic-tac-toe program of Chapter 6, in which I converted a human-readable "position" such as

```
{x o 3 x x 6 7 8 o}
```

into an internal list of "triples":

```
[xo3 xx6 78o xx7 ox8 36o xxo 3x7]
```

From now on, I won't show two versions of every procedure. I'll use the revised data representation, even though the chapter tells the story of how I wrote the older version of the program.

## Planning and Debugging

Ideally, according to structured programming, you should never have to do any debugging. You should start with a complete, clear program specification. Then you should use the approved style to translate that specification into a program. Then you should be able to *prove* mathematically that your program is correct! Debugging is a relic of the dark ages.

That's not the Logo approach. I've already done some debugging in this project. Programming is sort of like real life: you don't always get it right the first time. Structured programmers don't get it right the first time either; the difference is that Logoites aren't embarrassed about it. We think of debugging as part of the process of solving problems in general.

If you're a student in a school, the odds are that you aren't often encouraged to accept debugging as valuable. When you hand in a paper or a quiz, the teacher doesn't

point out errors and invite you to try again. Instead he marks your errors in red ink and takes off points for them. You're taught that your work has to be perfect the first time. One of the strong contributions that computer programming in general, and Logo in particular, has made to education is to provide one context in which you are shown a more realistic approach to making and correcting mistakes.

## Classifying Poker Hands

The main thing remaining to be done in my project is the collection of predicates like `fourp` and `royal.flushp` to check for particular kinds of poker hands. I decided to write some of the easy ones, namely the ones for multiples of the same rank.

```
to fourp
output memberp 4 :rankarray
end

to threep
output memberp 3 :rankarray
end

to pairp
output memberp 2 :rankarray
end

to full.housep
output and threep pairp
end
```

These are all pretty obvious. Notice, though, that one thing has changed since my initial idea: these procedures don't take `:cards` as an input. They don't examine the poker hand directly; they examine the variables set up by the initialization procedure.

Now I want to start putting all these pieces together, so I'm going to write a preliminary version of `pokerhand`.

```
to pokerhand :cards
poker.init :cards
if fourp [print [four of a kind] stop]
if full.housep [print [full house] stop]
if threep [print [three of a kind] stop]
if pairp [print ifelse paircount = 1 [one pair] [two pairs] stop]
print [something else]
end
```

```
to paircount
output count locate 2 1
end

to locate :number :index
if :index > 13 [output []]
if (item :index :rankarray) = :number ~
   [output fput :index (locate :number :index+1)]
output locate :number :index+1
end
```

If there's a pair, I can't simply use `memberp` to find out how many pairs are in the hand. Instead, the procedure `locate` looks at each member of `:rankarray` and outputs a list of all the ranks of which there are exactly two cards in the hand. For this purpose I could have had `locate` output the number of pairs, which would be a little easier than computing the list of ranks of pairs. But I recall that I want to be able to say things like "pair of sevens," and for that I'll need the actual ranks.

Let's try it:

```
? pokerhand [ah 2c 4d 2s 6h]
I don't know how  to one  in pokerhand
[if pairp [print ifelse paircount = 1 [one pair] [two pairs] stop]]
```

Looks like a bug. (This really happened; I'm not just making it up to be able to talk about debugging!) The first step in solving a problem like this is to read the error message carefully. This message tells me that when the error happened, the immediately active procedure was `pokerhand`. So that's where I should look for a mistake. (The exact form of the message will be different in different versions of Logo, but they'll all give you that piece of information. In Berkeley Logo, the error message also includes the instruction line in which the error occurred.) I then edited `pokerhand` and looked for the word `one`. I found it in the list

```
[one pair]
```

which is one of the inputs to an `ifelse` operation. Aha! The trouble is that `ifelse` *evaluates* whichever input is selected by its predicate input, so it's trying to evaluate that list as a Logo expression. What I meant was this:

```
if pairp [print ifelse paircount = 1 [[one pair]] [[two pairs]] stop]
```

Now it should evaluate `[[one pair]]` and come up with the value `[one pair]` to use as the input to `print`. Let's try again:

```
? pokerhand [ah 2c 4d 2s 6h]
one pair
? pokerhand [2h 5d 2s 2c 7d]
three of a kind
? pokerhand [2h 5d 2s 2c 5h]
full house
? pokerhand [3h 4h 5h 6h 7h]
something else
```

So far so good, but of course there is more work to do. We need to write `straightp`, `flushp`, and their combinations: straight flush and royal flush. I think I shouldn't have an instruction in `pokerhand` testing `royal.flushp` as I originally planned; instead I should test for `straightp` and, if that's true, look for special cases within that.

```
to flushp
output emptyp butfirst :suits
end
```

It's not so obvious how to write `straightp`. Here's my plan: First, find the lowest-rank card in the hand. Then, in order to have a straight, the next four ranks must also be present in the hand.

☞ This isn't the only possible way to test for a straight; can you think of, and implement, another?

```
to straightp
output nogap (reduce "min :ranks) 5
end

to min :a :b
output ifelse :a < :b [:a] [:b]
end

to nogap :smallest :howmany
if :howmany=0 [output "true]
if not equalp (item :smallest :rankarray) 1 [output "false]
output nogap :smallest+1 :howmany-1
end
```

`Nogap` starts with the smallest rank in the hand and checks that there is exactly one card in each of that and the next four ranks. It takes advantage of the fact that I'm representing ranks internally as numbers; it can just add 1 to a rank to get the next one in sequence. If `:howmany` reaches zero, it means that we have indeed found all five consecutive ranks in the hand. If one of the five desired ranks isn't in the hand, or if the hand has more than one card in any of the ranks, then the hand isn't a straight.

There is one problem with this approach. The ace can be used either high card (10-J-Q-K-A) or low card (A-2-3-4-5) in a straight. `Straightp` thinks that the ace can only be the low card. We'll fix that later.

Now let's try some other cases. I've just added the line

```
if straightp [print [straight] stop]
```

to `pokerhand`. It doesn't much matter where I put that line, because there is no danger of a straight also being found as a multiple of any one rank. This instruction will be changed, eventually, because we want to test for straight flush and so on. But for now this will make it possible to debug `straightp`.

```
? pokerhand [3h 6d 7h 5c 4d]
straight
? pokerhand [3h 6d 7h 5c 8d]
something else
```

I picked those examples pretty much at random. It's a good idea, when testing a procedure, to pick test cases "near the boundaries" of what the program is supposed to accept. For example, what about an ace-low straight, or a king-high? What about a hand in which "the next four ranks" don't exist, because the lowest card is a Jack?

```
? pokerhand [ah 2d 3c 4c 5h]
straight
? pokerhand [9d 10c jh qh kh]
straight
? pokerhand [js jh qs qh kd]
two pairs
```

(Actually, that last example may never invoke `straightp` at all, if the test for `pairp` comes first in `pokerhand`.) Anyway, it looks okay. I could try more examples but I think I believe it. I now decide that the instruction I just put into `pokerhand` should be

```
if straightp [print ifelse flushp [[straight flush]] [[straight]] stop]
```

and that it should be followed by

```
if flushp [print [flush] stop]
```

(The `if flushp` instruction has to come second because of the principle of "most restrictive first." If that test came first, a straight flush would be reported as just a flush.) Time for more tests:

```
? pokerhand [3h 6h ah kh 7h]
flush
? pokerhand [3h 6h ad kh 7h]
something else
? pokerhand [3h 6h 4h 5h 7h]
straight flush
? pokerhand [3h 6h 4h 5s 7h]
straight
```

Now it's time to solve the problem of the ace-high straight. It turns out to be easy; if the hand has an ace, then I can use `nogap`, the subprocedure of `straightp` that checks for consecutive ranks, to check for the four ranks from 10 to king.

```
to ace.highp
if not equalp (item 1 :rankarray) 1 [output "false]
output nogap 10 4
end
```

That's the end of the categories of poker hands, but to put it all together requires a little editing of `pokerhand`:

```
to pokerhand :cards
local [ranks suits rankarray]
poker.init :cards
if fourp [print [four of a kind] stop]
if full.housep [print [full house] stop]
if threep [print [three of a kind] stop]
if pairp [print ifelse paircount = 1 [[one pair]] [[two pairs]] stop]
if ace.highp [print ifelse flushp [[royal flush]] [[straight]] stop]
if straightp [print ifelse flushp [[straight flush]] [[straight]] stop]
if flushp [print [flush] stop]
print [nothing!]
end
```

## Embellishments

I've now done more or less what I set out to do. It took 14 procedures. I hope you have a feeling for the process of switching back and forth between thinking about a particular subproblem and thinking about the overall structure of the program.

I haven't done every detail of what I first suggested. In particular, I don't have the information about particular ranks in what I print. I think perhaps that's more effort than this project seems worth to me. (I'm not just being cute by saying "to me"; the point is that a real poker enthusiast might want to spend a lot of time on this program and make it as beautiful as possible.) But just to show how a completed program can be modified, I'll make it print things like `pair of sixes` instead of just `one pair`.

First I have to be able to find words like "`sixes`" starting with a rank indicator like `6`.

```
to plural :rank
output item :rank [aces twos threes fours fives sixes
                   sevens eights nines tens jacks queens kings]
end
```

The next step is to change one instruction in `pokerhand` to use this new tool:

```
if pairp [print ifelse paircount = 1
                       [sentence [pair of] plural first locate 2 1]
                       [[two pairs]]
          stop]
```

(If you were confused about the double square brackets around `one pair` and `two pairs` before, seeing this new version in which one of the possibilities is the output from a procedure, not a literal list, might help.)

```
? pokerhand [ah 7s 3d 10c 7c]
pair of sevens
```

☞ If you're motivated, you can modify the messages for other categories to include the specific rank information. You might want to change "`nothing`" to "`queen high`," for example.

☞ What if you wanted to use this program on a seven-card-stud hand? In other words, instead of a list of five cards, you'd be given a list of seven, from which you'd have to pick the best five. The main thing I can think of is that you'd have to be more careful about the order of the `if` instructions in `pokerhand`. I've said that you can test `threep` either

before or after `straightp` because they can't both be true. But that's not the case for a seven-card hand:

```
[3h 3s 3d 4d 5s 6h 7c]
```

If you try this challenge, make sure your program announces

```
[8s 9s 10s js qs kh ad]
```

as a straight flush, not as an ace-high straight.

## Putting the Project in a Context

I wrote this program because I was looking for an example for this book that would be not too long, not too short. That's kind of an artificial reason for starting a project. In real life, if I wrote a program like this one, it would be part of a larger program that would actually *play* poker.

In that context the problem would become very different. We wouldn't want merely to print the designation of a hand; we'd want to be able to compare several hands and announce a winner. To do that, we'd have to attach something like a numerical ranking to the hand, which might become the output from `pokerhand`. But it can't be just a single number; there are too many possible hands to have a list of all of them in rank order. Instead, the ranking of a hand might be a list of numbers. `[5 7 10]` might mean that the hand is a full house (I'm guessing that that would rank about fifth in value), with three sevens and two tens. To compare two lists of numbers, compare their `firsts`; if those are equal, go on to compare the next members.

The point is that I'm now back to something approaching top-down planning. As the scale of the project becomes a lot bigger, that kind of advance planning seems necessary. But this isn't really top-down because comparing two hands is just one subproblem of playing poker. Really, according to the top-down view, I should start by designing the top-level procedure `poker`. Perhaps a first attempt might look like this:

```
to poker
deal.cards
bid
draw.more.cards
bid
pokerhand
end
```

But it would be premature to type this into a computer. We have to think about issues like these: Is the computer a player or does it just deal and bank for the other players? How many people can play? What is a good strategy for bidding?

In the end it might turn out that the `pokerhand` we've just written wouldn't fit into the larger project; it might have to be rewritten for that context. To a structured programmer, the effort we've put in would then be wasted. But I think that even if every procedure had to be edited, I'd benefit from having taken the time to understand how to solve this subproblem.

## Program Listing

```
to pokerhand :cards
local [ranks suits rankarray]
poker.init :cards
if fourp [print [four of a kind] stop]
if full.housep [print [full house] stop]
if threep [print [three of a kind] stop]
if pairp [print ifelse paircount = 1 [[one pair]] [[two pairs]] stop]
if ace.highp [print ifelse flushp [[royal flush]] [[straight]] stop]
if straightp [print ifelse flushp [[straight flush]] [[straight]] stop]
if flushp [print [flush] stop]
print [nothing!]
end

to poker.init :cards
make "ranks map [ranknum butlast ?] :cards
make "suits remdup map "last :cards
make "rankarray {0 0 0 0 0 0 0 0 0 0 0 0 0}
foreach :ranks [setitem ? :rankarray (item ? :rankarray)+1]
end

to ranknum :rank
if :rank = "a [output 1]
if :rank = "j [output 11]
if :rank = "q [output 12]
if :rank = "k [output 13]
output :rank
end

to fourp
output memberp 4 :rankarray
end
```

```
to threep
output memberp 3 :rankarray
end

to pairp
output memberp 2 :rankarray
end

to full.housep
output and threep pairp
end

to paircount
output count locate 2 1
end

to locate :number :index
if :index > 13 [output []]
if (item :index :rankarray) = :number ~
   [output fput :index (locate :number :index+1)]
output locate :number :index+1
end

to flushp
output emptyp butfirst :suits
end

to straightp
output nogap (reduce "min :ranks) 5
end

to min :a :b
output ifelse :a < :b [:a] [:b]
end

to nogap :smallest :howmany
if :howmany=0 [output "true]
if not equalp (item :smallest :rankarray) 1 [output "false]
output nogap :smallest+1 :howmany-1
end

to ace.highp
if not equalp (item 1 :rankarray) 1 [output "false]
output nogap 10 4
end
```