I haven’t talked much, until now, about how to find and fix mistakes in the programs you write. Except for the chapter-length examples in Chapters 6, 12, and 14, it hasn’t been much of a problem because the sample programs I’ve shown you have been so small. That doesn’t mean you can’t make a mistake in a small program! But mistakes are relatively easy to find when the entire program is one procedure with just a few instruction lines. In a real programming project, which might have 20 or 200 procedures, it’s harder to locate an error.

**Using Error Messages**

At one point in Chapter 13 I saw the error message

I don’t know how to one in pokerhand

Logo’s error messages were deliberately designed to use an informal, smooth, low-key style so that beginning programmers won’t find them intimidating. But there is a lot of information in that message if you learn how to find it. The message tells me three things. First, it tells me what kind of error is involved. In this particular message, the phrase “I don’t know how” suggests that a procedure is missing, and the words “to one” subtly suggest how the problem could be fixed. Second, the message tells me the specific expression that was in error: the word one. Third, it tells me that the error was detected while Logo was carrying out the procedure named pokerhand.

The precise form of the message may be different in different situations. If you make a mistake in a top-level instruction (that is, one that you type to a question mark prompt, not inside a procedure), the part about in pokerhand won’t be included.
One very important thing to remember is that the place where an error is found may not be the place where the error really is. That’s a little vague, so let’s think about the I don’t know how error. All the Logo interpreter knows is that it has been asked to invoke a procedure that doesn’t exist. But there can be several possible reasons for that. The most common reason is that you’ve just misspelled the name of a procedure. When the message is

I don’t know how to forward in poly

you can be pretty sure, just from reading the message, that the problem is a misspelling of forward. In this case the mistake is in poly, just as the message tells you.

On the other hand you might get a message like this about a procedure that really should exist. For example, I might have seen

I don’t know how to straight in pokerhand

If I had been confronted with that message, I might have looked at pokerhand, and indeed I would have found an instruction that invokes a procedure named straight. But that’s not an error; there should be such a procedure. One of two things would be wrong: either I’d forgotten to define straight altogether or else I made a spelling mistake in the title line of straight rather than in an instruction line of pokerhand. To find out, I would type the command pots (which, as you recall, stands for Print Out TitleS) and look for a possible misspelling of straight.

Another way to get the same error message is to write a program using one version of Logo and then transfer it to another version with somewhat different primitives. For example, Berkeley Logo includes higher order functions such as map that are not primitive in most other Logo dialects. If you write a program that uses map and then try to run it in another version of Logo, you’ll get a message saying I don’t know how to map. In that case you’d have to write your own version of map or rewrite the program to avoid using it—for example, by using a recursive operation instead.

The mistake I actually made in Chapter 13 wasn’t a misspelling, a missing definition, or a nonexistent primitive. Instead, I failed to quote a list with square brackets. The particular context in which I did it, in an input to ifelse, is a fairly obscure one. But here is a common beginner’s mistake, especially for people who are accustomed to other programming languages:
? print "How are you?"
How
i don’t know how to are

The moral of all this is that the error message does give you some valuable help in finding your bug, but it doesn’t tell you the whole story. You have to read the message intelligently.

Invalid Data

I’ve spent a lot of time on the I don’t know how message because it’s probably the most common one. Another very common kind of message, which will merit some analysis here, is

procedure doesn’t like datum as input

In general, this means that you’ve violated the rules about the kinds of data that some primitive procedure requires as input. (Recall that the type of input is one of the things I’ve been insisting that you mention as part of the description of a procedure.) For example, word requires words as inputs, so:

? print word "hello, [old buddy]
word doesn’t like [old buddy] as input

There are several special cases, however, that come up more often than something as foolish as using a list as an input to word. The most common message of this form is this one:

butfirst doesn’t like [] as input

This almost invariably means that you’ve left out the stop rule in a recursive procedure. The offending input to butfirst isn’t an explicit empty list but instead is the result of evaluating a variable, usually an input to the procedure you’re writing, that’s butfirsted in the recursive invocation. This is a case where the error isn’t really in the instruction that caused the message. Usually there is nothing wrong with the actual invocation of butfirst; the error is a missing instruction earlier in the procedure. If the input is a word instead of a list, this message will take the possibly confusing form

butfirst doesn’t like as input
That’s an invisible empty word between like and as!

I said that this message is almost always caused by a missing stop rule. You have to be careful about the “almost.” For example, recall this practical joke procedure from Chapter 1:

to process :instruction
test emptyp :instruction
  iftrue [type "|? | process readlist stop]
  iffalse [print sentence [I don’t know how to|] first :instruction]
end

This is not a recursive procedure, and the question of stop rules doesn’t arise. But its input might be empty, because the victim enters a blank line. If I hadn’t thought of that, and had written

to process :instruction
  print sentence [I don’t know how to|] first :instruction
end

the result would be

first doesn’t like [] as input in process

Another case that sometimes comes up in programs that do arithmetic is

/ doesn’t like 0 as input

For example, if you write a program that takes the average of a bunch of numbers and you try to use the program with an empty list of numbers as input, you’ll end up trying to divide zero by zero. The solution is to insert an instruction that explicitly tests for that possibility.

As always, the procedure that provokes the error message may not actually be the procedure that is in error. Consider this short program:

to second :thing
  output first butfirst :thing
end

to swap :list
  output list (second :list) (first :list)
end

286  Chapter 15  Debugging
? print swap [watch pocket]
pocket watch
? print swap [farewell]
first doesn’t like [] as input in second
[output first butfirst :thing]

Although the error was caught during the invocation of second, there is nothing wrong with second itself. The error was in the top-level instruction, which provided a bad input to swap. That instruction doesn’t even include an explicit reference to second. In this small example it’s easy to see what happened. But in a more complicated program it can be hard to find errors like this one.

There are two ways you can protect yourself against this kind of difficulty. The first is defensive programming. I could have written the program this way:

to swap :list
  if emptyp :list [pr [empty input to swap] stop]
  if emptyp butfirst :list [pr [singleton input to swap] stop]
  output list (second :list) (first :list)
end

This version checks for bad inputs and gives a more helpful error message.* It would also be possible to figure out an appropriate output for these cases and not consider them errors at all:

to swap :list
  if emptyp :list [output []]
  if emptyp butfirst :list [output :list]
  output list (second :list) (first :list)
end

This version manages to produce an output for any input at all. How should you choose between these two defensively written versions? It depends on the context in which you’ll be using swap. If you are writing a program in which swap should always get a particular kind of list as input, which should always have two members, then you should use the first defensive version, which will let you know if you make an error in the input to swap.

---

* Actually, when you invoke this version of swap with a bad input, you’ll see two error messages. The procedure itself will print an error message. Then, since it stops instead of outputting something to its superprocedure, you’ll get a didn’t output error message from the Logo interpreter.
But if **swap** is intended as a general tool, which might be used in a variety of situations, it might be better to accept any input.

The second protective technique, besides defensive programming, is tracing, the technique we used in Chapter 9. If you get an error message from a utility procedure like **second** and you have no idea how it was invoked, you can find out by tracing the entry into all of your procedures.

Another way to get the **doesn’t like** message is to forget the order of inputs to a procedure, either a primitive or one that you’ve written. For example, **lput** is a primitive operation that requires two inputs. The first input can be any datum, but the second must be a list. The output from **lput** is a list that contains all the members of the second input, plus one more member at the end equal to the first input.

```
? show lput "c [a b]
[a b c]
```

**lput** takes its inputs in the same order as **fput**, with the new member first and then the old list. But you might get confused and want the inputs to appear left-to-right as they appear in the result:

```
? show lput [a b] "c
lput doesn’t like c as input
```

### Incorrect Results

Beginning programmers are often dismayed when they see an error message, but more experienced programmers are relieved. They know that the bugs that cause such messages are the easy ones to find! Much harder are the bugs that allow a program to run to completion but produce the wrong answer. In that kind of situation you don’t have the advantage of knowing which procedure tickled the error message, so it’s hard to know where to begin looking.

Here’s a short program with a couple of bugs in it. **Arabic** is an operation that takes one input, a word that is a Roman numeral. The output from **arabic** is the number represented by that Roman numeral in ordinary (Arabic numeral) notation.

```lisp
  to arabic :num
  output addup map "digit :num
  end
```
to digit :digit
output lookup :digit [[I 1] [V 5] [X 10] [L 50] [C 100] [D 500] [M 1000]]
end

to lookup :word :dictionary
if emptyp :dictionary [output ""]
if equalp :word first first :dictionary [output last first :dictionary]
output lookup :word bf :dictionary
end

to addup :list
if emptyp :list [output 0]
if emptyp bf :list [output first :list]
if (first :list) < (first bf :list) ~
    [output sum ((first bl :list)-(first :list)) addup bf bf :list]
output sum first :list addup bf :list
end

Arabic uses two non-primitive subprocedures, dividing its task into two parts. First
digit translates each letter of the Roman numeral into the number it represents: C into
100, M into 1000. The result is a list of numbers. Then addup translates that list into a
single number, adding or subtracting each member as appropriate. The rule is that the
numbers are added, except that a smaller number that appears to the left of a larger one
is subtracted from the total. For example, in the Roman numeral CLIV all the letters are
added except for the I, which is to the left of the V. Since I represents 1 and V represents
5, and 1 is less than 5, the I is subtracted. The result is 100 + 50 + 5 − 1 or 154.

Here’s what happened the first time I tried arabic:

? print arabic "MLXVI"
13

This is a short enough program that you may be able to find the bug just by reading
it. But even if you do, let’s pretend that you don’t, because I want to use this example to
talk about some ways of looking for bugs systematically.

The overall structure of the program is that digit is invoked for each letter, and
the combined output from all the calls to digit is used as the input to addup. The first
step is to try to figure out which of the two is at fault. Which should we try first? Since
addup depends on the work of digit, whereas digit doesn’t depend on addup, it’s
probably best to start with digit. So let’s try looking at the output from digit directly.
? print digit "M
1000
? print digit "V
5

So far so good. Perhaps the problem is in the way map is used to combine the results from digit:

? show map "digit "MLXVI
1000501051

Aha! I wanted a list of numbers, one for each Roman digit, but instead I got all the numbers combined into one long word. I had momentarily forgotten that if the second input to map is a word, its output will be a word also. As soon as I see this, the solution is apparent to me: I should use map.se instead of map.

? show map.se "digit "MLXVI
[1000 50 10 5 1]
to arabic :num
output addup map.se "digit :num
end

? print arabic "MLXVI
1066

This time I got the answer I expected. On to more test cases:

? print arabic "III
3
? print arabic "XVII
17
? print arabic "CLV
155
? print arabic "CLIV
150
?

Another error! The result was 150 instead of the correct 154. Since the other three examples are correct, the program is not completely at sea; it’s a good guess that the bug has to do with the case of subtracting instead of adding. Trying a few more examples will help confirm that guess.
Indeed, numbers that involve subtraction seem to fail, while ones that are purely additive seem to work. If you look carefully at exactly how the program fails, you may notice that the letter that should be subtracted and the one after it are just ignored. So in the numeral MCMLXXXIV, which represents 1984, the CM and the IV don’t contribute to the program’s result.

Once again, we must find out whether the bug is in digit or in addup, and it makes sense to start by checking the one that’s called first. (If you read the instructions in the definitions of digit and addup, you’ll see that digit handles each digit in isolation, whereas addup is the one that looks at two consecutive digits to decide whether or not to subtract. But at first I’m not reading the instructions at all; I’m trying to be sure that I understand the behavior of each procedure before I look inside any of them. For a simple problem like this one, the approach I’m using is more ponderous than necessary. But it would pay off for a larger program with more subtle bugs.)

I’ve started with Roman numerals for which the overall program works. Why not just concentrate on the cases that fail? Because I want to see what the correct output from mapping digit over the Roman numeral is supposed to look like. It turns out to be a list of numbers, one for each letter in the Roman numeral.

You may wonder why I need to investigate the correct behavior of digit experimentally. If I’ve planned the program properly in the first place, I should know what it’s supposed to do. There are several reasons why I might feel a need for this sort of experiment. Perhaps it’s someone else’s program I’m debugging, and I don’t know what the plan was. Perhaps it’s a program I wrote a long time ago and I’ve forgotten. Finally, since there is a bug after all, perhaps my understanding is faulty even if I do think I know what digit is supposed to do.
Now let's try digit for some of the buggy cases.

? show map.se "digit "IV
[1 5]

? show map.se "digit "MCMLXXXIV
[1000 100 1000 50 10 10 10 1 5]

Digit still does the right thing: It outputs the number corresponding to each letter. The problem must be in addup.

Now it's time to take a look at addup. There are four instructions in its definition. Which is at fault? It must be one that comes into play only for the cases in which subtraction is needed. That's a clue that it will be one of the if instructions, although instructions that aren't explicitly conditional can, in fact, depend on earlier if tests. (In this procedure, for example, the last instruction doesn't look conditional. But it is carried out only if none of the earlier instructions results in an output being evaluated.)

Rather than read every word of every line carefully, we should start by knowing the purpose of each instruction. The first one is an end test, detecting an empty numeral. The second is also an end test, detecting a single-digit numeral. (Why are two end tests necessary? How would the program fail if each one were eliminated?) The third instruction deals with the subtraction case, and the fourth with the addition case. The bug, then, is probably in the third instruction. Here it is again:

\[
\text{if } (\text{first :list}) < (\text{first bf :list}) - \\
\quad \{ \text{output sum } ((\text{first bl :list})-(\text{first :list})) \text{ addup } bf bf :list\}
\]

At this point a careful reading of the instruction will probably make the error obvious. If not, look at each of the expressions used within the instruction, like

\[
\text{first :list}
\]

and

\[
\text{bf bf :list}
\]

What number or list does each of them represent?

(If you'd like to take time out for a short programming project now, you might try writing roman, an operation to translate in the opposite direction, from Arabic to Roman numerals. The rules are that I can be subtracted from V or X; X can be subtracted from...
Tracing and Stepping

In Chapter 9 we used the techniques of tracing and stepping to help you understand how recursive procedures work. The same techniques can be very valuable in debugging. Tracing a procedure means making it print an indication of when it starts and stops. Stepping a procedure means making it print each of its instructions and waiting for you to type something before evaluating the instruction.

Berkeley Logo provides primitive commands trace and step that automatically trace or step procedures for you. Trace and step take one input, which can be either a word or a list. If the input is a word, it must be the name of a procedure. If a list, it must be a list of words, each of which is the name of a procedure. The effect of trace is to modify the procedure or procedures named in the input to identify the procedure and its inputs when it is invoked. The effect of step is to modify the named procedure(s) so that each instruction is printed before being evaluated.

Tracing a procedure is particularly useful in the annoying situation in which a program just sits there forever, never stopping, but never printing anything either. This usually means that there is an error in a recursive procedure, which invokes itself repeatedly with no stop rule or with an ineffective one. If you trace recursive procedures, you can find out how you got into that situation.

Pausing

When a program fails, either with an error message or by printing the wrong result, it can be helpful to examine the values of the variables used within the program. Of course, you understand by now that “the variables used within the program” may be a complicated idea; if there are recursive procedures with local variables, there may be several variables with the same name, one for each invocation of a procedure.

Once a program is finished running, the local variables created by the procedures within the program no longer exist. You can examine global variables individually by printing their values or all at once with the pons command. (Pons stands for Print Out NameS; it takes no inputs and prints the names and values of all current variables.) But it’s too late to examine local variables after a program stops.
To get around this problem, Berkeley Logo provides a **pause** command. This command takes no inputs. Its effect is to stop, temporarily, the procedure in which it appears. (Like **stop** and **output**, **pause** is meaningless at top level.) Logo prints a question mark prompt (along with the name of the paused procedure to remind you that it’s paused), and you can enter instructions to be evaluated as usual. But the paused procedure is **still active**; its local variables still exist. (Any superprocedures of the paused procedure, naturally, are also still active.) The instructions you type while the procedure is paused can make use of local variables, just as if the instructions appeared within the procedure definition.

The main use of **pause** is for debugging. If your program dies with an error message you don’t understand, you can insert a **pause** command just before the instruction that gets the error. Then you can examine the variables that will be used by that instruction.

Better yet, you can ask Logo to pause automatically whenever an error occurs. In fact, you can ask Logo to carry out any instructions you want, whenever an error occurs, by creating a variable named **erract** (short for error action) whose value is an instruction list. If you want your program to pause at any error, say

```logo
? make "erract [pause]
```

before you run the program. To undo this request, you can erase the variable name **erract** with the **ern** (erase name) command:

```logo
? ern "erract
```

Once you’ve examined the relevant variables, you may want to continue running the program. You’ll certainly want to continue if this pause wasn’t the one you’re waiting for, just before the error happens. Logo provides the command **continue** (abbreviated **co**) for this purpose. If you type **continue** with no input, Logo will continue the evaluation of the paused procedure where it left off.

It is also possible to use **continue** with an input, turning the **pause** command into an operation by providing a value for it to output. Whether or not that’s appropriate depends on which error message you get. If the message complains about a missing value, you may be able to provide one to allow the program to continue:

```logo
to demo.error
  print first :nonesuch
end
```
? make "erract [pause]
? demo.error
nonesuch has no value in demo.error
[print first :nonesuch]
Pausing...
demo.error? continue "hello
h

If, after examining variables, you figure out the reason for the bug, you may not want to bother continuing the buggy procedure. Instead you'll want to forget about it, edit the definition to fix the bug, and try again. But you shouldn't just forget about it because the procedure is still active. If you don't want to continue it, you should stop it instead, to get back to the "real" top level with no procedures active. (Instead of stop, a more definitive way to stop all active procedures is with the instruction

throw "toplevel

For now just think of this as a magic incantation; we'll talk more about throw in the second volume.)

Berkeley Logo also has a special character that you can type on the keyboard to cause an immediate pause. The character depends on which computer you're using; see Appendix A. This is not as useful a capability as you might think because it's hard to synchronize your typing with the activity of the program so that it gets paused in the right context (that is, with the right procedures active and the right local variables available). But it can be useful if you can see that the program is repeating the same activities over and over, for example; pausing just about anywhere during that kind of loop is likely to give you useful information.

---

Final Words of Wisdom

You may be feeling a frustrating sense of incompleteness about this chapter. After the chapter on variables, for example, you really knew everything there is to know about variables. (I suppose that's not strictly true, since you hadn't thought about recursion yet, but it's true enough.) But you certainly don't know everything there is to know about debugging. That's because there isn't a complete set of rules that will get you through every situation. You just have to do a lot of programming, meet a lot of bugs, and develop an instinct for them.

As a beginner, you'll probably meet bugs with a different flavor from the ones I've been discussing. You'll put a space after a quotation mark or a colon, before the word...
to which it should be attached. You’ll leave out a left or right parenthesis or bracket.
(Perhaps you’ll get confused about when to use parentheses and when brackets!) All
of these simple errors will quickly get you error messages, and you can probably find
your mistake just by reading the offending instruction. Later, as your programs get more
complicated, you’ll start having the more interesting bugs that require analysis to find
and fix.

It’s a good idea to program with a partner. Sometimes you can find someone
else’s bugs more easily than your own—when you read your own program, you know
too well what you meant to say. This advice is not just for beginners; even experienced
programmers often benefit from sharing their bugs with a friend. Another advantage of
such a partnership is that trying to explain your program to someone else will often help
you understand it more clearly yourself. I’ve often discovered a persistent bug halfway
through explaining the problem to someone.

The main point, I think, is one I’ve made in earlier chapters: there is nothing
shameful about a bug in your program. As a teacher, I’ve been astonished to see students
react to a simple bug by angrily erasing an entire program, which they’d spent hours
writing! Teach yourself to expect bugs and approach them with a good-natured spirit.

On the other hand, you can minimize your debugging time by writing the program
in a reasonable style in the first place. If your program is one long procedure, you should
know that you’re making it harder to locate an offending instruction. If all your variables
are named \( x \) and \( y \), you deserve whatever happens to you! And if you can’t figure out,
yourself, which procedure does what, then perhaps you should stop typing in procedures
and spend a little time with paper and pencil listing the tasks each procedure needs to
carry out.