# 3 Variables

In the last chapter I suggested that you would find yourself limited in writing new procedures because your procedures don't take inputs, so they do exactly the same thing every time you use them. In this chapter we'll overcome that limitation.

## User Procedures with Inputs

As a first example I'm going to write a very simple command named `greet`, which will take a person's name as its one input. Here's how it will work:

```
? greet "Brian
Hello, Brian
Pleased to meet you.
? greet "Emma
Hello, Emma
Pleased to meet you.
```

This procedure will be similar to the `hello` command in the last chapter, except that what it prints will depend on the input we give it.

Each time we *invoke* `greet`, we want to give it an input. So that Logo will expect an input, we must provide for one when we *define* `greet`. (Each procedure has a definite number of inputs; if `greet` takes one input once, it must take one input every time it's invoked.) Also, in order for the instructions inside `greet` to be able to use the input, we must give the input a *name*. Both of these needs are met in the `to` command that supplies the title line for the procedure:

```
? to greet :person
```

You are already familiar with the use of the `to` command, the need for a word like `greet` to name the procedure, and the appearance of the greater-than prompt instead of the question mark. What's new here is the use of `:person` after the procedure name. This addition tells Logo that the procedure `greet` will require one input and that the name of the input will be `person`. It may help to think of the input as a container; when the procedure `greet` is invoked, something (such as the word `Brian` or the word `Emma`) will be put into the container named `person`.

Why is the colon used in front of the name `person`? Remember that the inputs to `to`, unlike the inputs to all other Logo procedures, are *not* evaluated before `to` is invoked. Later we'll see that a colon has a special meaning to the Logo evaluator, but that special meaning is *not* in effect in a title line. Instead, the colon is simply a sort of mnemonic decoration to make a clear distinction between the word `greet`, which is a *procedure* name, and the word `person`, which is an *input* name. Some versions of Logo don't even require the colon; you can experiment with yours if you're curious. (By the way, if you want to sound like a Logo maven, you should pronounce the colon "dots," as in "to greet dots person.")

To see why having a name for the input is helpful, look at the rest of the procedure definition:

```
> print sentence "Hello, thing "person
> print [Pleased to meet you.]
> end
?
```

You already know about `print` and `sentence` and about quoting words with the quotation mark and quoting lists with square brackets. What's new here is the procedure `thing`.

`Thing` is an operation. It takes one input, which must be a word that's the name of a container. The output from `thing` is whatever datum is in the container.

The technical name for what I've been calling a "container" is a *variable*. Every variable has a *name* and a *thing* (or *value*). The name and the thing are both *parts of* the variable. We'll sometimes speak loosely of "the variable `person`," but you should realize that this *is* speaking loosely; what we should say is "the variable named `person`." `Person` itself is a *word,* which is different from a variable.

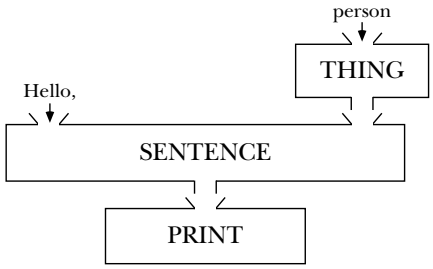
When I type the instruction

```
greet "Brian
```

the Logo interpreter starts with the first word on the line, `greet`. As usual, Logo takes this to be the name of a procedure. Logo discovers that `greet` requires one input, so it continues to the next thing on the line. This is a quoted word, `"Brian`. Since it's quoted, it requires no further interpretation. The word `Brian` itself becomes the input to `greet`.*

Logo is now ready to invoke `greet`. The first step, before evaluating the instruction lines in `greet`, is to create a variable to hold the input. This variable is given the word `person` as its *name,* and the word `Brian` as its *thing.* (Please notice that I don't have to know the name of `greet`'s input in order to use it. All I have to know is what *type of thing*—a person's name—`greet` expects as its input. What are the names of the inputs to a primitive like `sentence`? We don't know and we don't need to know.)

Logo now evaluates the first instruction in `greet`. The process is just like the ones we went through in such detail in Chapter 2. In the course of this evaluation Logo invokes the procedure `thing` with the word `person` as its input. The output from `thing` is the thing in the variable named `person`, namely the word `Brian`. That's how the word `Brian` becomes one of the inputs to `se`. Here's a plumbing diagram.




* While reading the definition of `greet`, it's easy to say "the input is `person`"; then, while reading an invocation of `greet`, it's easy to say "the input is `Brian`." To avoid confusion between the input's name and its value, there are more precise technical terms that we can use when necessary. The name of the input, given in the title line of the procedure definition, is called a *formal parameter.* The value of the input, given when the procedure is invoked, is called an *actual argument.* In case the actual argument is the result of a more complicated subexpression, as in the instruction

```
greet first [Brian Harvey]
```

we might want to distinguish between the *actual argument expression,* `first [Brian Harvey]`, and the *actual argument value,* which is the word `Brian`.

## What Kind of Container?

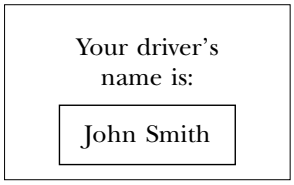
One of the favorite activities that Logo experts use to while away the time when the computer is down is to argue about the best metaphor to use for variables. A variable is a container, but what kind of container?

One popular metaphor is a mailbox. The mailbox has a *name* painted on it, like "The Smiths." Inside the mailbox is a piece of mail. The person from the Post Office assigns a *value* to the box by putting a letter in it. Reading a letter is like invoking `thing` on the mailbox.

I don't like this metaphor very much, and if I explain why not, it may help illuminate for you some details about how variables work. The first problem is that a real mailbox can contain several letters. A variable can only contain *one* thing or value. (I should say "one thing at a time," since we'll see that it's possible to replace the thing in a variable with a different thing.)

Another problem with the mailbox metaphor is that to read a letter, you take it out of the mailbox and tear it open. Then it isn't in the mailbox any more. When you invoke `thing` to look at the thing in a variable, on the other hand, it's still in the variable. You could use `thing` again and get the same answer.

There are two metaphors that I like. The one I like best won't make sense for a while, until we talk about scope of variables. But here is the one I like second best: Sometimes when you take a bus or a taxi, there is a little frame up in front that looks like this:




The phrase "your driver's name is" is like a label for this frame, and it corresponds to the *name* of a variable. Each bus driver has a metal or plastic plate that says "John Smith" or whoever it is. The driver inserts this plate, which corresponds to the *value* of the variable, into the frame. You can see why this is a closer metaphor than the mailbox. There is only one plate in the frame at a time. To find out who's driving the bus, you just have to look inside the frame; you don't have to remove the plate.

(To be strictly fair I should tell you that some Logoites don't like the whole idea of containers. They have a completely different metaphor, which involves sticking labels on things. But I think it would only confuse you if I explained that one right now.)

## An Abbreviation

Examining the value of a variable is such a common thing to do in a Logo procedure that there is a special abbreviation for it. Instead of the expression

```
thing "person
```

you can simply say

```
:person
```

So in the `greet` procedure, we could have said

```
print sentence "hello :person
```

Please note that the colon is *not* just an abbreviation for the word `thing` but rather for the combination `thing`-quote.

When drawing plumbing diagrams, treat `:narf` as if it were spelled out as `thing "narf`.

## More Procedures

It's time to invent more procedures. I'll give you a couple of examples and you should make up more on your own.

```
to primer :name
print (sentence first :name [is for] word :name ".)
print (sentence "Run, word :name ", "run.)
print (sentence "See :name "run.)
end

? primer "Paul
P is for Paul.
Run, Paul, run.
See Paul run.
```

`Primer` uses the extra-input kludge I mentioned near the end of Chapter 2. It also shows how the operations `word` and `sentence` can be used in combination to punctuate a sentence properly.

With all of these examples, incidentally, you should take the time to work through each instruction line to make sure you understand what is the input to what.

```
to soap.opera :him :her :it
print (sentence :him "loves word :her ".)
print (sentence "However, :her [doesn't care for] :him "particularly.)
print (sentence :her [is madly in love with] word :it ".)
print (sentence :him [doesn't like] :it [very much.])
end

? soap.opera "Bill "Sally "Fred
Bill loves Sally.
However, Sally doesn't care for Bill particularly.
Sally is madly in love with Fred.
Bill doesn't like Fred very much.
```

In this example you see that a procedure can have more than one input. `Soap.opera` has three inputs. You can also see why each input must have a name, so that the instructions inside the procedure have a way to refer to the particular input you want to use. You should also notice that `soap.opera` has a period in the middle of its name, not a space, because the name of a procedure must be a single Logo word.

For the next example I'll show how you can write an *interactive* procedure, which reads something you type on the keyboard. For this we need a new tool. `Readlist` is an operation with no inputs. Its output is always a list, containing whatever you type on a single line (up to a RETURN). `Readlist` waits for you to type a line, then outputs what you type.

```
to converse
print [Please type your full name.]
halves readlist
end

to halves :name
print sentence [Your first name is] first :name
print sentence [Your last name is] last :name
end
```

```
? converse
please type your full name.
Brian Harvey
Your first name is Brian
Your last name is Harvey
```

This program includes two procedures, `converse` and `halves`. (A *program* is a bunch of procedures that work together to achieve a common goal.) `Converse` is the *top-level procedure*. In other words, `converse` is the procedure that you invoke at the question-mark prompt to set the program in motion. `Halves` is a *subprocedure* of `converse`, which means that `halves` is invoked by an instruction inside `converse`. Similarly, `converse` is a *superprocedure* of `halves`.

There are two things you should notice about the terminology "subprocedure" and "superprocedure." The first thing is that these are *relative* terms. It doesn't mean anything to say "`Halves` is a subprocedure." Any procedure can be used as part of a larger program. `Converse`, for example, is a superprocedure of `halves`, but `converse` might at the same time be a subprocedure of some higher-level procedure we haven't written yet. The second point is that primitive procedures can also be considered as subprocedures. For example, `sentence` is a subprocedure of `halves`.

(Now that we're dealing with programs containing more than one defined procedure, it's a good time for me to remind you that the commands that act on procedures can accept a list as input as well as a single word. For example, you can say

```
po [converse halves]
```

and Logo will print out the definitions of both procedures.)

Why are two procedures necessary for this program? When the program reads your full name, it has to remember the name so that it can print two parts of it separately. It wouldn't work to say

```
to incorrect.converse
print [Please type your full name.]
print sentence [Your first name is] first readlist
print sentence [Your last name is] last readlist
end
```

because each invocation of `readlist` would read a separate line from the keyboard instead of using the same list for both first and last names. We solve this problem by using

the output from `readlist` as the input to a subprocedure of `converse` and letting the subprocedure do the rest of the work.

One of the examples in Chapter 1 was this procedure:

```
to hi
print [Hi. What's your name?]
print sentence [How are you,] word first readlist "?
ignore readlist
print [That's nice.]
end
```

`Hi` uses a procedure called `ignore` that we haven't yet discussed. `Ignore` is predefined in Berkeley Logo but would be easy enough to define yourself:

```
to ignore :something
end
```

That's not a misprint; `ignore` really has no instructions in its definition. `Ignore` is a command that takes one input and has no effect at all! Its purpose is to ignore the input. In `hi`, the instruction

```
ignore readlist
```

waits for you to type a line on the keyboard, then just ignores whatever you type. (We couldn't just use `readlist` as an instruction all by itself because a complete instruction has to begin with a command, not an operation. That is, since `readlist` outputs a value, there must be a command to tell Logo what to do with that value. In this case, we want to `ignore` it.)

☞ Write a procedure to conjugate the present tense of a regular first-conjugation (-er) French verb. (Never mind if you don't know what any of that means! You're about to see.) That is, the letters `er` at the end of the verb should be replaced by a different ending for each pronoun:

```
? conj "jouer
je joue
tu joues
il joue
nous jouons
vous jouez
elles jouent
```

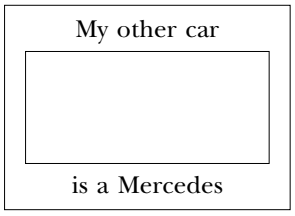
The verb `jouer` (to play) consists of the root `jou` combined with the infinitive ending `er`. Print six lines, as shown, in which the ending is changed to `e`, `es`, etc. Try your procedure on `monter` (to climb), `frapper` (to hit), and `garder` (to keep).

By the way, in a practical program we would have to deal with the fact that French contains many irregular verbs. In addition to wildly irregular ones like être (to be, irregular even in English) there are ones like manger, to eat, which are almost regular except that the first and second person plural forms keep the letter e: nous mangeons. Many issues in natural language programming (that is, getting computers to speak or understand human language) turn out like this—90% of the cases are trivial, but most of your effort goes into the other 10%.

## An Aside on Variable Naming

In my metaphor about the frame containing the bus driver's name, the inscription on the frame tells you what to expect inside the frame. Variable names like `person` and `name` serve a similar purpose. (You might argue that the `it` in the group of names `him`, `her`, and `it` is a little misleading. But it serves to keep the story straight, probably better than an alternative like `him1` and `him2`.)

Another kind of frame is the one you sometimes see around a car's license plate:




I know it's pedantic to pick apart a joke, but just the same I want to make the point that this one works only because the car itself provides enough clues that what belongs in the frame is indeed a license plate. If you were unfamiliar with the idea of license plates, that frame wouldn't help you.

The computer equivalent of this sort of joke is to give your variables names that don't reflect their purpose in the procedure. Some people like to name variables after their boyfriends or girlfriends or relatives. That's okay if you're writing simple programs, like the ones in this chapter, in which it's very easy to read the program and figure out what it does. But when you start writing more complicated programs, you'll need all the help you can get in remembering what each piece of the program does. I recommend starting early on the habit of using sensible variable names.

## Don't Call It `x`

Another source of trouble in variable naming is lazy fingers. When I'm teaching programming classes, a big part of my job is reading program listings that students bring to me, saying, "I just can't find the bug in this program." I have an absolute rule that I refuse to read any program in which there is a variable named `x`.

My students always complain about this arbitrary rule at first. But more often than not, a student goes through a program renaming all the variables and then finds that the bug has disappeared! This magical result comes about because when you use variable names like `x`, you run the risk of using the same name for two different purposes at the same time. When you pick reasonable names, you'll pick two different names for the two purposes.

It is people who've programmed in BASIC who are most likely to make this mistake. For reasons that aren't very important any more, BASIC used to *require* single-letter variable names. Even now there are limits on longer names in most versions of BASIC that make it risky to use more than two or three letters in a name. So if you're a BASIC programmer, you've probably gotten into bad habits, which you should make a point of correcting.

## Writing New Operations

So far all the procedures we've written have been commands. That is, our procedures have had an *effect* (like printing something) rather than an *output* to be used with other procedures. You can also write operations, once you know how to give your procedure an output. Here is an example:

```
to second :thing
output first butfirst :thing
end

? print second [the red computer]
red
```

`Second` is an operation with one input. Like the primitive operation `first`, it extracts a component of its input, either a character from a word or a member from a list. However, it outputs the second component instead of the first one.

What is new in this procedure definition is the use of the primitive command `output`. `Output` can be used only inside a procedure definition, not at top level. (In

other words, not when you are typing in response to a question-mark prompt.) It takes one input, which can be any datum. The effect of `output` is to make the datum you supply as its input be the output from your procedure.

Some people find it confusing that `output` itself is a *command,* even though a procedure that uses `output` is an *operation.* But it makes sense for `output` to be the head of a complete instruction. The effect of the instruction is to inform Logo what output you want your procedure (the procedure named `second` in this case) to supply.

Another possible confusion is between `output` and `print`. The problem is that people talk about "computer output" while waving a stack of paper at you, so you think of "output" as meaning "stuff the computer printed." But in Logo, "output" is something one procedure hands to another procedure, not something that is printed.

I chose the name `thing` for the input to `second` to remind myself that the input can be anything, word or list. `Thing` is also, as you know, the name of a primitive procedure. This is perfectly okay. The same word can name both a procedure and a variable. Logo can tell which you mean by the context. A word that is used in an instruction without punctuation is a procedure name. A word that is used as an input to the procedure `thing` is a variable name. (This can happen because you put dots in front of the word as an abbreviation or because you explicitly typed `thing` and used the word as its input.) The expression `:thing` is an abbreviation for

```
thing "thing
```

in which the first `thing` names a procedure, and the second `thing` names a variable.

☞ Write an operation `query` that takes a sentence as input and that outputs a question formed by swapping the first two words and adding a question mark to the last word:

```
? print query [I should have known better]
should I have known better?
? print query [you are experienced]
are you experienced?
```

## Scope of Variables

This is going to be a somewhat complicated section, and an important one, so slow down and read it carefully.

When one procedure with inputs invokes another procedure with inputs as a subprocedure, it's possible for them to share variables and it's also possible for them to

have separate variables. The following example isn't meant to do anything particularly interesting, just to make explicit what the rules are.

```
to top :outer :inner
print [I'm in top.]
print sentence [:outer is] :outer
print sentence [:inner is] :inner
bottom "x
print [I'm in top again.]
print sentence [:outer is] :outer
print sentence [:inner is] :inner
end

to bottom :inner
print [I'm in bottom.]
print sentence [:outer is] :outer
print sentence [:inner is] :inner
end

? top "a "b
I'm in top.
:outer is a
:inner is b
I'm in bottom.
:outer is a
:inner is x
I'm in top again.
:outer is a
:inner is b
```

First, concentrate on the variable named `outer`. This name is used for the first input to `top`. `Bottom` doesn't have an input named `outer`. When `bottom` refers to `:outer`, since it doesn't have one of its own, the reference is to the variable `outer` that belongs to its superprocedure, `top`. That's why `a` is printed as the value of `outer` in both procedures.

> **If a procedure refers to a variable that does not belong to that procedure, Logo looks for a variable of that name in the superprocedure of that procedure.**

Suppose procedure `a` invokes procedure `b`, and `b` invokes `c`. Suppose an instruction in procedure `c` refers to a variable `v`. First Logo tries to find a variable named `v` that belongs to `c`. If that fails, Logo looks for a variable named `v` that belongs to procedure

b. Finally, if neither c nor b has a variable named v, Logo looks for such a variable that belongs to procedure a.

Now look at inner. The important thing to understand is that *there are two variables named* inner, one belonging to each procedure. When top is invoked, its input named inner gets the word b as its value. When top invokes bottom, bottom's input (which is also named inner) gets the value x. But when bottom finishes, and top continues, the name inner once again refers to the variable named inner that belongs to top. The one that belongs to bottom has disappeared.

> **Variables that belong to a procedure are temporary. They exist only so long as that procedure is active. If one procedure has a variable with the same name as one belonging to its superprocedure, the latter is temporarily "hidden" while the subprocedure is running.**

Because each procedure has its own variable named inner, we refer to the procedure input variables as *local* to a particular procedure. Inputs are always local in Logo. There is also a name for the fact that a procedure can refer to variables belonging to its superprocedures. If you want to show off, you can explain to people that Logo has *dynamic scope,* which is what that rule is called.

## The Little Person Metaphor

Earlier I told you my second favorite metaphor about variables. My very favorite is an old one, which Logo teachers have been using for years. It is a metaphor about procedures as well as variables, which is why I didn't present it earlier. Now that you're thinking about the issue of variable scope, you can see that to have a full understanding of variables, you have to be thinking about procedures at the same time.

The metaphor is that inside the computer there is a large community of little people. Each person is a specialist at a particular procedure. So there are print people and butfirst people and bottom people and greet people. I like to think of these people as elves, because I started teaching Logo on a computer called a PDP-11, and I like the pun of an elf inside an 11. But if you find elves too cute or childish, perhaps you should think of these people as doctors in white coats, specializing in dermatology or ophthalmology or whatever. Another terminology for the same idea, one which is becoming more and more widely used in advanced computer science circles, is to call the little people *actors* and to call their procedures *scripts.* Each actor has only one script, but several actors can have the same script.

In any case, what's important is that when a procedure is invoked, a little person who is an expert on that procedure goes to work. (It's important that the person is *an expert in* the procedure, and not the procedure *itself;* we'll see later that there can be two little people carrying out the same procedure at the same time. This is one of the more complicated ideas in Logo, so I think the expert metaphor will help you later.)

You may be wondering where the variables come in. Well, each elf is wearing a jerkin, a kind of vest, with a bunch of pockets. (If your people are doctors, the pockets are in those white lab coats.) A person has as many pockets as the procedure he or she knows has inputs. A `print` expert has one pocket; a `sentence` expert has two. Each pocket can contain a datum, the value of the variable. (The pockets are only big enough for a single datum.) Each pocket also has a name tag sewn on the inside, which contains the name of the variable.

The name tags are on the inside to make the point that other people don't need to know the names of an expert's variables. Other experts only need to know how many pockets someone has and what kind of thing to put in them.

When I typed

```
top "a "b
```

the Chief Elf (whose name is Evaluator) found an elf named Theresa, who is a `top` expert, and put an `a` in her first pocket and a `b` in her second pocket.

Theresa's first instruction is

```
print [I'm in top.]
```

To carry out that instruction, she handed the list `[I'm in top.]` to another elf named Peter, a `print` expert.

Theresa's second instruction is

```
print sentence [:outer is] :outer
```

To carry out this instruction, Theresa wanted to hire Peter again, but before she could give him his orders, she first had to deal with Sally, a `sentence` expert. (This is the old evaluation story from Chapter 2 again.) But Theresa didn't know what to put in Sally's second pocket until she got the information from Tom, a `thing` expert. (Remember that `:outer` is an abbreviation for `thing "outer`.)

What's important right now is how Tom does his job. Tom is a sort of pickpocket. He doesn't steal anything; he just sneaks looks in other people's pockets. There are lots of people inside the computer, but the only ones with things in their pockets are the ones who are actually employed at a given moment. Aside from Tom himself, the only person who was employed at the time was Theresa, so Tom could only look in her pockets for a name tag saying `outer`. (Theresa is *planning* to hire Sally and then Peter, to finish carrying out her instruction, but she can't hire them until she gets the information she needs from Tom.)

Later Theresa will hire Bonnie, a `bottom` specialist, to help with the instruction

```
bottom "x
```

Theresa will give Bonnie the word `x` to put in her pocket. Bonnie also has an instruction

```
print sentence [:outer is] :outer
```

As part of the process of carrying out this instruction, Bonnie will hire Tom to look for something named `outer`. In that case Tom first looks in the pockets of Bonnie, the person who hired him. Not finding a pocket named `outer`, Tom can *then* check the pockets of Theresa, the person who hired Bonnie. (If you're studying Logo in a class with other people, it can be both fun and instructive to act this out with actual people and pockets.)


Theresa Bonnie Tom

An appropriate aspect of this metaphor is that it's slightly rude to look in someone else's pockets, and you shouldn't do it unnecessarily. This corresponds to a widely

accepted rule of Logo style: most of the time, you should write procedures so that they don't have to look at variables belonging to their superprocedures. Whatever information a procedure needs should be given to it explicitly, as an input. You'll find situations in which that rule seems very helpful, and other situations in which taking advantage of dynamic scope seems to make the program easier to understand.

☞ The `conj` procedure you wrote earlier deals only with the present tense of the verb. In French, many other tenses can be formed by a similar process of replacing the endings, but with different endings for different tenses. Also, second conjugation (-ir) and third conjugation (-re) verbs have different endings even in the present tense. You don't want to write dozens of almost-identical procedures for each of these cases. Instead, write a single procedure `superconj` that takes two inputs, a verb and a list of six endings, and performs the conjugation:

```
? superconj "jouer [ais ais ait ions iez aient]      ; imperfect tense
je jouais
tu jouais
il jouait
nous jouions
vous jouiez
elles jouaient
? superconj "finir [is is it issons issez issent]    ; 2nd conj present
je finis
tu finis
il finit
nous finissons
vous finissez
elles finissent
```

You can save some typing and take advantage of dynamic scope if you use a helper procedure. My `superconj` looks like this:

```
to superconj :verb :endings
sc1 "je 1
sc1 "tu 2
sc1 "il 3
sc1 "nous 4
sc1 "vous 5
sc1 "elles 6
end
```

Write the helper procedure `sc1` to finish this.

## Changing the Value of a Variable

It is possible for a procedure to change the thing in a variable by using the `make` command. `Make` takes two inputs. The first input must be a word that is the name of a variable, just like the input to `thing`. `Make`'s second input can be any datum. The effect of `make` is to make the variable named by its first input contain as its value the datum that is its second input, instead of whatever used to be its value. For example,

```
make "inner "y
```

would make the variable named `inner` have the word `y` as its value. (If there are two variables named `inner`, as is the case while `bottom` is running, it is the one in the lower-level procedure that is changed. This is the same as the rule for `thing` that we have already discussed.)

Suppose a procedure has variables named `old` and `new` and you want to copy the thing in `old` into `new`. You could say

```
make "new thing "old
```

or use the abbreviation

```
make "new :old
```

People who don't understand evaluation sometimes get very upset about the fact that a quotation mark is used to refer to `new` and a colon is used to refer to `old`. They think this is just mumbo-jumbo because they don't understand that a quotation mark is part of what the colon abbreviates! In both cases we are referring to the name of a variable. A variable name is a Logo word. To refer to a word in an instruction and have it evaluate to itself, not invoke a procedure named `new` or `old`, the word must be quoted. The difference is that the first input to `make` is the *name* of the variable we want to change (`new`), while the second input to `make` is, in this example, the *value* of a variable (`old`), which we get by invoking `thing`. Since you understand all this, you won't get upset. You also won't resort to magic formulas like "always use quote for the first variable and dots for the second" because you understand that the inputs to `make` can be computed with any expression you want! For example, we could copy `old`'s value into `new` this way:

```
make first [new old] thing last [new old]
```

This instruction contains neither a quotation mark nor a colon, but the inputs to `make` are exactly the same as they were in the earlier version.

Earlier I mentioned that it is considered slightly rude for a procedure to read its superprocedures' variables. It is *extremely* rude for a procedure to change the values of other procedures' variables! Perhaps you can see why that's so. If you're trying to read the definition of a procedure, and part way through that procedure it invokes a subprocedure, there is no clue to the fact that the subprocedure changes a variable. If you break this rule, it makes your program very hard to read because you have to read all the procedures at once. If each procedure deals only with its own variables, you have written a *modular* program, in which each piece can be understood separately.

## Global and Local Variables

What if the first input to `make` isn't the name of an input to an active procedure? In other words, what if you try to assign a value to a variable that doesn't exist? What happens is that a new variable is created that is *not* local to any procedure. The name for this kind of variable is a *global* variable. `Thing` looks at global variables if it can't find a local variable with the name you want.

A local variable disappears when the procedure it belongs to finishes. Global variables don't belong to any procedure, so they stay around forever. This can be convenient, when you have a permanent body of information that several procedures must use. But it can also lead to problems if you are careless about what's in which variable. Local variables come and go with the procedures they belong to, so it's easy to avoid clutter when you use them. Global variables are more like old socks under the bed.

If you are a BASIC programmer, you've become accustomed to a language in which all variables are global. I've learned over the years that it's impossible, at this point in your career, for you to appreciate the profound effect that's had on your style of programming. Only after you've used procedural languages like Logo for quite a while will you understand. Meanwhile there is only one hope for you: you are not allowed to use global variables *at all* for the next few months. Please take my word for it.

Sometimes it's convenient for a procedure to use a variable that is not an input, but which could just as well be local. To do this, you can use the `local` command. This command takes one input, a word. It creates a variable, local to the procedure that invoked `local`, with that word as its name. For example, we can use `local` to rewrite the earlier `converse` example without needing the `halves` subprocedure:

```
to new.converse
local "name
print [Please type your full name.]
make "name readlist
print sentence [Your first name is] first :name
print sentence [Your last name is] last :name
end
```

The instruction that invokes `local` can be anywhere in the procedure before the variable is given a value with `make`. It's traditional, though, to put `local` instructions at the beginning of a procedure.

The same procedure would work even without the `local`, but then it would create a global variable named `name`. It's much neater if you can avoid leaving unnecessary global variables around, so you should use `local` unless there is a reason why you really need a global variable.

## Indirect Assignment

Earlier I showed you the example

```
make first [new old] thing last [new old]
```

in which the first input to `make` was the result of evaluating a complex expression rather than an explicit quoted word in the instruction. But the example was kind of silly, used only to make the point that such a thing is possible.

Here are a couple of examples in which the use of a computed first input to `make` really makes sense. These are tricky examples; it may take a couple of readings before you see what I'm doing here. The technique I'm using is an advanced part of Logo programming. First is the procedure `increment`:

```
to increment :variable
make :variable (thing :variable)+1
end
```

To *increment* a variable means to add something to it, usually (as in this procedure) to add one to it. The input to `increment` is the name of a variable. The procedure adds 1 to that variable:

```
? make "count 12
? print :count
12
? increment "count
? print :count
13
```

You may wonder what the point is. Why couldn't I just say

```
make "count :count+1
```

instead of the obscure `make` instruction I used? The answer is that if we have several variables in the program, each of which sometimes gets incremented, this technique allows a single procedure to be able to increment any variable. It's a kind of shorthand for something we might want to do repeatedly.

In the definition of `increment`, the first input to `make` is not `"variable` but rather `:variable`. Therefore, the word `variable` itself is not the name of the variable that is incremented. (To say that more simply, the variable named `variable` isn't incremented.) Instead the variable named `variable` contains as its value the name of *another* variable. (In the example the value of `variable` is the word `count`.) It is that second variable whose value is changed. (In the example `:count` was 12 and becomes 13.)

While reading `increment`, remember that in the second input to `make`,

```
thing :variable
```

is really an abbreviation for

```
thing thing "variable
```

In other words this expression asks for the value of the variable whose name is itself the value of `variable`.

As a second example suppose you're writing a program to play a game of Tic-Tac-Toe. The computer will play one side and a person can play the other side. The person gets to choose X or O (that is, going first or second). The choice might be made with procedures like these:

```
to computer.first
make "computer "X
make "person "O
end

to person.first
make "person "X
make "computer "O
end
```

Elsewhere in the program there will be a procedure that asks the person where he or she wants to move. Suppose the squares on the board are numbered 1 through 9, and suppose we have two variables, `Xsquares` and `Osquares`, which contain lists of numbers corresponding to the squares marked X and O. Look at this procedure:

```
to person.move :square
make word :person "squares sentence :square thing word :person "squares
end
```

The input to `person.move` is the number of the square into which the person has asked to move. The first input to `make` is the expression

```
word :person "squares
```

If the person has chosen to move first, then `:person` is the word `X`, and the value of this expression is the word `Xsquares`. If the person has chosen to move last, then `:person` is the word `O`, and the value of the expression is the word `Osquares`. Either way, the expression evaluates to the name of the appropriate variable, into which the newly chosen square is appended.

These are examples of *indirect assignment,* which means assigning a value to a variable whose name is computed by the program. This is an unusual, advanced technique. Most of the time you'll use an explicit quoted word as the first input to `make`. But the technique is a powerful one; many programming languages don't have this capability at all. In Logo it isn't something that had to be invented specially; it is a free consequence of the fact that the inputs to any procedure (including `make`) are evaluated before the procedure is invoked.

## Functional Programming

But don't get carried away with the flexibility of `make`. *Another* advanced Logo technique avoids the whole idea of changing the value of a variable. Any procedure that uses `make`

can be rewritten to use an input to a subprocedure instead; compare the two versions of the `converse` program in this chapter.

Why would you want to avoid `make`? One reason is that if the value of a variable changes partway through a procedure, then the sequence of steps within the procedure is very important. One hot area in computer science research is *parallel* computation: What if, instead of a computer that can only do one thing at a time, we build a computer that can do many things at once? It's hard to take advantage of that ability if each step of our program depends on the results of previous steps, and if later steps depend on the result of this one.

A procedure is *functional* if it always gives the same output when invoked with the same input(s). We need a few more Logo tools before we can write interesting functional programs, but we'll come back to this idea soon.