2 Procedures

Logo is one of the most powerful programming languages around. In order to take advantage of that power, you must understand Logo's central ideas: *procedures* and *evaluation*. It is with these ideas that our exploration of Logo programming begins.

Procedures and Instructions

In response to Logo's question-mark prompt, type this instruction:

```
print 17
```

Logo will respond to this instruction by printing the number 17 and then printing another question mark, to indicate that it's ready for another instruction:

```
? print 17
17
```

(Remember, the things in **boldface** are the ones *you* should type; what's in lightface is what the computer prints.)

This instruction doesn't do much, but it's important to understand how it's put together. The word `print` is the name of a *procedure,* which is a piece of a computer program that has a particular specialized task. The procedure named `print`, for example, has the task of printing things on your screen.

If you have previously used some other programming language, you may be accustomed to the idea of different *statement types* making up the repertoire of the language. For example, BASIC has a `print` statement, a `let` statement, an `input` statement, etc. Pascal has an assignment statement, an `if` statement, a `while` statement, etc. Each kind

of statement has its own *syntax,* that is, its own special punctuation and organization. Logo is very different. It does not have different kinds of instructions; *everything* in Logo is done by the use of procedures. If Logo is your first programming language, you don't have to worry about this. But for people with previous experience in another language, it's a common source of misunderstanding.

When you first start up Logo, it "knows" about 200 procedures. These initial procedures are called *primitive* procedures. Your task as a Logo programmer is to add to Logo's repertoire by defining new procedures of your own. You do this by putting together procedures that already exist. We'll see how this is done later in this chapter.

The procedure `print`, although it has a specific task, doesn't always do *exactly* the same thing; it can print anything you want, not always the number 17. (You've seen several examples in Chapter 1.) This may seem like an obvious point, but later you will see that the *flexibility* of procedures is an important part of what makes them so powerful. To control this flexibility, we need a way to tell a procedure exactly what we want it to do. Therefore, each procedure can accept a particular number of *inputs.* An input is a piece of information. It can be a number, as in the example we're examining, but there are many other kinds of information that Logo procedures can handle. The procedure named `print` requires one input. Other procedures will require different numbers of inputs; some don't require any.

Technical Terms

In ordinary conversation, words such as *instruction* and *procedure* have pretty much the same meaning—they refer to any process, recipe, or method for carrying out some task. That's not the situation when we're talking about computer programming. Each of these words has a specific technical meaning, and it's very important for you to keep them straight in your head while you're reading this chapter. (Soon we'll start using more words, such as *command* and *operation,* which also have similar meanings in ordinary use but very different meanings for us.)

An *instruction* is what you type to Logo to tell it to do something. `Print 17` is an example of an instruction. We're about to see some more complicated instructions, made up of more pieces. An instruction has to contain enough information to specify *exactly* what you want Logo to do. To make an analogy with instructing human beings, "Read Chapter 2 of this book" is an instruction, but "read" isn't one, because it doesn't tell you what to read.

A *procedure* is like a recipe or a technique for carrying out a certain kind of task. `Print` is the name of a procedure just as "lemon meringue pie" is the name of a recipe.

(The recipe itself, as distinct from its name, is a bunch of instructions, such as "Preheat the oven to 325 degrees.") A procedure contains information about how to do something, but the procedure doesn't take action itself, just as a recipe in a book can't bake a pie by itself. Someone has to carry out the recipe. In the Logo world something has to *invoke* a procedure. To "invoke" a procedure means to carry it out, to do what the procedure says. Procedures are invoked by instructions. The instruction you gave just now invoked the procedure named `print`.

If an instruction is made up of names of procedures, and if the procedures invoked by the instruction are made up of more instructions, why doesn't the computer get caught in a vicious circle, always finding more detailed procedures to invoke and never actually doing anything? This question is a lot like the one about dictionaries: When you look up the definition of a word, all you find is more words. How do you know what *those* words mean? For words in the dictionary this turns out to be a very profound and difficult question. For Logo programming the answer is much simpler. In the end, your instructions and the procedures they invoke must be defined in terms of the primitive procedures. Those procedures are not made up of Logo instructions. They're the things that Logo just knows how to do in the first place.

Evaluation

Now try this instruction:

```
print sum 2 3
```

If everything is going according to plan, Logo didn't print the words "`sum 2 3`"; it printed the number 5. The input to `print` was the expression `sum 2 3`, but Logo *evaluated* the input before passing it to the `print` procedure. This means that Logo invoked the necessary procedures (in this case, `sum`) to compute the value of the expression (5).

In this instruction the word `sum` is also the name of a procedure. `Sum` requires two inputs. In this case we gave it the numbers 2 and 3 as inputs. Just as the task of procedure `print` is to print something, the task of procedure `sum` is to add two numbers. It is the result of this addition, the *output* from `sum`, that becomes the *input* to `print`.

Don't confuse *output* with *printing*. In Logo the word "output" is one of those technical terms I mentioned before. It refers to a value that one procedure computes and hands on to another procedure that needs an input. In this example `sum` outputs the number 5 to `print`, but `print` doesn't output anything to another procedure. When

`print` prints the 5, that's the end of the story. There are no more procedures waiting for inputs.

See if you can figure out what this instruction will do before you try it:

```
print sum 4 product 10 2
```

Here are the steps Logo takes to evaluate the instruction:

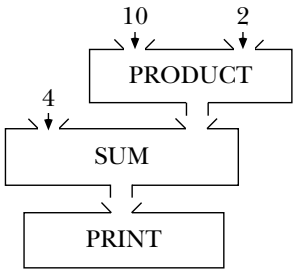
1. The first thing in the instruction is the name of the procedure `print`. Logo knows that `print` requires one input, so it continues reading the instruction line.
2. The next thing Logo finds is the word `sum`. This, too, is the name of a procedure. This tells Logo that the *output* from `sum` will be the *input* to `print`.
3. Logo knows that `sum` takes two inputs, so `sum` can't be invoked until Logo finds `sum`'s inputs.
4. The next thing in the instruction is the number 4, so that must be the first input to `sum`. This input, too, must be evaluated. Fortunately, a number simply evaluates to itself, so the value of this input is 4.
5. Logo still needs to find the second input to `sum`. The next thing in the instruction is the word `product`. This is, again, the name of a procedure. Logo must carry out that procedure to evaluate `sum`'s second input.
6. Logo knows that `product` requires two inputs. It must now look for the first of those inputs. (Meanwhile, `print` and `sum` are both "on hold" waiting for their inputs to be evaluated. `print` is waiting for its single input; `sum`, which has found one input, is waiting for its second.) The next thing on the line is the number 10. This number evaluates to itself, so the first input to `product` is 10.
7. Logo still needs another input for `product`, so it continues reading the instruction. The next thing it finds is the number 2. This number evaluates to itself, so the second input to `product` has the value 2.
8. Logo is now ready to invoke the procedure `product`, with inputs 10 and 2. The output from `product` is 10 times 2, or 20.
9. This output, 20, is the value of the second input to `sum`. Logo is now ready to invoke `sum`, with inputs 4 and 20. The output from `sum` is 24.
10. The output from `sum`, 24, is the input to `print`. Logo is now ready to invoke `print`, which prints 24. (You were only waiting for this moment to arise.)

That's a lot of talking about a pretty simple instruction! I promise not to do it again in quite so much detail. It's important, though, to be able to call upon your understanding of these details to figure out more complicated situations later. Using the output from one procedure as an input to another procedure is called *composition of functions*.

Some people find it helpful to look at a pictorial form of this analysis. We can represent each procedure as a kind of tank, with input hoppers on top and perhaps an output pipe at the bottom. (This organization makes sense because gravity will pull the information downward.) For example:

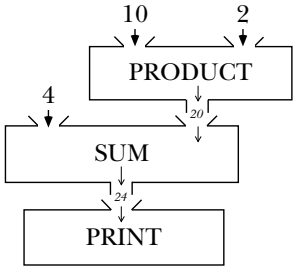
`Print` has one input, which is represented by the hopper above the tank. It doesn't have an output, so there is no pipe coming out the bottom. `Sum` has two inputs, shown at the top, and an output, shown at the bottom.

We can put these parts together to form a kind of "plumbing diagram" of the instruction:



In that diagram the output pipes from one procedure are connected to the input hoppers of another. Every pipe must be connected to something. The inputs that are explicitly given as numbers in the instruction are shown with arrows pointing into the hoppers.

You can annotate the diagram by indicating the actual information that flows through each pipe. Here's how that would look for this instruction:



By the way, I've introduced the procedures `print`, `sum`, and `product` so casually that you might think it's a law of nature that every programming language must have procedures with these names. Actually the details of Logo's repertoire of primitive procedures are quite arbitrary. It would be hard to avoid having a way to add numbers, but it might have been named `plus` or `add` instead of `sum`. For some primitives there are additional arbitrary details; for noncommutative operations such as `remainder`, for example, the rule about which input comes first was an arbitrary choice for Logo's designers. (☞ Experiment with `remainder` and see if you can describe it well enough that someone else can use it without needing to experiment.) I am making a point of the arbitrary nature of these details because people who are learning to program sometimes think they're doing badly if they don't *figure out* how a primitive procedure works in advance. But these rules aren't things you work out; they're things someone has to tell you, like the capital of Kansas.

Error Messages

We've observed that Logo knows in advance how many inputs a particular procedure needs. (`Print` needs one; `sum` and `product` each need two.) What if you give a procedure the wrong number of inputs? Try this:

```
print
```

(That is, the word `print` as an instruction all by itself, with no input.) You should see something like this:

```
? print
Not enough inputs to print
```

This gentle complaint from Logo tells you two things. First, it indicates the general *kind* of thing that went wrong (not enough inputs to some procedure). Second, it names the *particular* procedure that complained (`print`). In this case it was pretty obvious which procedure was involved, since we only used one procedure. But try this:

```
? print remainder product 4 5
Not enough inputs to remainder
```

In this case Logo's message is helpful in pinpointing the fact that it was `remainder`, not `print` or `product`, that lacked an input.

The reason I'm beating this error message to death is that one of the most common mistakes made by beginning programmers is to ignore what an error message says. Some people get very upset at seeing this kind of message and just give up without trying to figure out the problem. Other people make the opposite mistake, breezing past the message without taking advantage of the detailed help it offers. Some smart people at M.I.T. put a lot of effort into designing Logo's error messages, so please pay attention to them.

What if you give a procedure too many inputs? Try this:

```
? print 2 3
2
You don't say what to do with 3
```

(The exact text of the message, by the way, may be slightly different in some versions of Logo.) What happened here is that Logo carried out the instruction `print 2`, and then found the extra number `3` on the line. It would have been okay if we'd done something with the 3:

```
? print 2 print 3
2
3
```

It's okay to have more than one instruction on the same line, as long as they are complete instructions.

Commands and Operations

What's a "complete instruction"? Before I can answer that question, you have to understand that in Logo there are two kinds of procedures: commands and operations.

An *operation* is a procedure that computes a value and outputs it. `Sum` and `product` are operations, for example.

A *command* is a procedure that does *not* output a value but instead has some *effect* such as printing something on the screen, moving a turtle, or making a sound. `Print`, then, is a command. Some commands have effects that are not apparent on the outside but instead change something inside the computer that might become important later in the program.

A complete instruction consists of the name of a command, followed by as many expressions as necessary to provide its inputs. An *expression* is something like `sum 3 2`

or 17. Operations are used to construct expressions. More formally, an expression is one of two things: either an explicitly provided value such as a number, or else the name of an operation, followed by as many expressions as necessary to provide its inputs. For example, the expression sum 3 2 consists of the operation name sum followed by two expressions, the number 3 and the number 2. Numbers are the only values we've seen how to provide explicitly, but that's about to change.

Words and Lists

So far, our examples have been about numbers and arithmetic. Many people think that computers just do arithmetic, but actually it's much more interesting to use computers with other kinds of information. You've seen examples of text processing in Chapter 1, but this time we're going to do it *carefully!*

Suppose you want Logo to print the word Hello. You might try this:

```
? print Hello
I don't know how  to Hello
```

Logo interpreted the word Hello as the name of a procedure, just as in the examples with print sum earlier. The error message means that there is no procedure named hello in Logo's repertoire.

When Logo is evaluating instructions, it always interprets unadorned words such as print or sum or hello as names of procedures. In order to convince Logo to treat a word simply as itself, you must type a quotation mark (") in front of it:

```
? print "Hello
Hello
```

Here is why the quotation mark is used for this purpose in Logo: In computer science, to *quote* something means *to prevent it from being evaluated.* (Another way to say the same thing is that *the thing evaluates to itself* or that its value *after* evaluation is the same as what it is *before* evaluation.) For example, we have already seen that in Logo, numbers are automatically quoted. (It doesn't hurt to use the quotation mark with numbers, however.

```
? print sum "2 "3
5
```

Logo is perfectly happy to add the quote-marked numbers.)

(People who have programmed in some other language should note that quotation marks are not used in pairs in Logo. This is not just an arbitrary syntactic foible; it reflects the fact that a Logo *word* is a different idea from what would be called a *character string* in other languages. I urge you not only to program in Logo but even to think in Logo terminology.)

What if you want to print more than one word? You can combine several words to form a *list*. The easiest way to do this is to enclose the words in square brackets, which tells Logo to quote the list. That is, a list in brackets evaluates to the list itself:

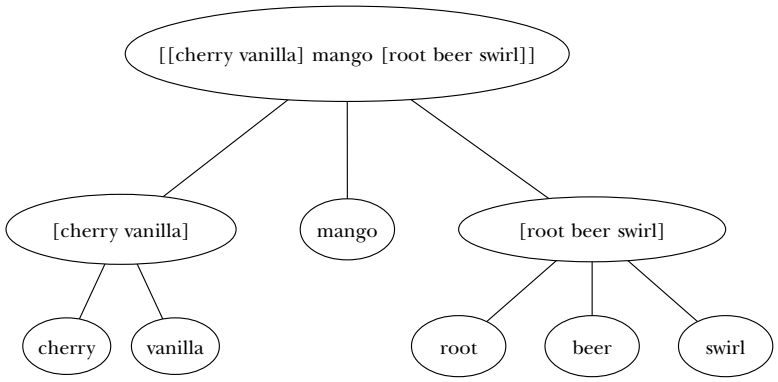
```
? print [How are you?]
How are you?
```

(If square brackets quote a list, what does it mean to evaluate a list? Well, every instruction line you type to Logo is actually a list, which is evaluated by invoking the procedures it names. Most of the time you don't have to remember that an instruction is a list, but that fact will become very useful later on.)

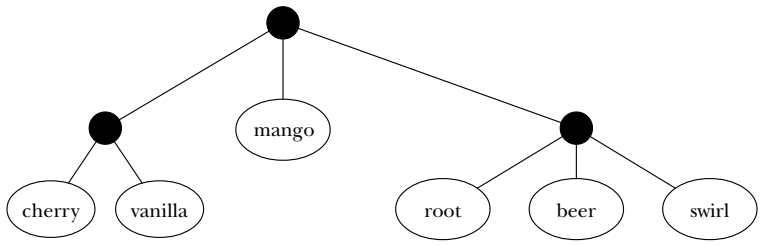
The list in the example above contains three *members*. In this example each member is a word. For example, the first member is the word `How`. But the members of a list aren't required to be words; they can also be lists. The fact that a list can have another list as a member makes lists very flexible as a way of grouping information. For example, the list

```
[[cherry vanilla] mango [root beer swirl]]
```

contains three members. The first and third members are themselves lists, while the second member is the word `mango`. A list like this can be represented using a *tree diagram:*



This diagram has the name "tree" because it resembles an upside-down tree, with a trunk at the top and branches extending downward. Often a tree diagram is drawn with only the *leaves* labeled—the words that make up the smallest sublists:



Keep in mind that the square brackets in Logo serve two purposes at once: they *delimit* a list—that is, they show where the list begins and ends—and they also *quote* the list, so that Logo's evaluator interprets the list as representing itself and not as requesting the invocation of procedures. The brackets surround the list; they are not *part of* the list. (Similarly, the quotation mark that indicates a quoted word is not part of the word.)

Words and lists are the two kinds of information that Logo can process. (Numbers are a special case of words.) The name I'll use for "either a word or a list" is a *datum*.* A list of words, such as `[How are you?]`, is called a *sentence* or a *flat list*. (It's called "flat" because the tree diagram only has one level, not counting the "root" at the top.) The name "sentence" is meant to suggest that flat lists are often, although not always, used to represent English sentences. A sentence is a special kind of list, just as a number is a special kind of word. We'll see other kinds of lists later.

How to Describe a Procedure

My high school U.S. history teacher was very fussy about what he considered the proper way to color in outline maps. He would make us do them over if we used colors or shading techniques he didn't like. We humored him because he was a very good teacher in other ways; for example, he gave us original historical documents to read instead of boring textbooks.

I hope you will humor me when I tell you that there is a right way and a wrong way to talk about procedures. If I were teaching you in person, I'd be very understanding

* Later we'll use a third kind of datum, called an "array."

about mistakes in your *programs,* but I'd hit you over the head (gently, of course) if you were sloppy about your *descriptions.*

Here is an example of the wrong way: "`Sum` adds up two numbers." It's not that this description isn't true but that it's inadequate. It leaves out too much.

Here is an example of the right way: "`Sum` is an operation. It has two inputs. Both inputs must be numbers. The output from `sum` is a number, the result of adding the two inputs."

Here are the ingredients in the right way:

1. Command or operation?
2. How many inputs?
3. What *type* of datum must each input be?
4. If the procedure is an operation, what is its *output?* If a command, what is its *effect?*

Another example: "The command `print` has one input. The input can be any datum. The effect of `print` is to print the input datum on the screen."

Manipulating Words and Lists

Logo provides several primitive operations for taking data apart and putting data together. Words come apart into *characters,* such as letters or digits or punctuation marks. (A character is not a third kind of datum. It's just a word that happens to be one character long.) Lists come apart into whatever data are the *members* of the list. A sentence, which is a list of words, comes apart into words.

`First` is an operation that takes one input. The input can be any nonempty datum. (In a moment you'll see what an empty datum is.) The output from `first` is the first member of the input if the input is a list, or the first character if the input is a word. Try these examples:

```
? print first "Hello
H
? print first [How are you?]
How
```

`Butfirst` is also an operation that takes one input. The input can be any nonempty datum. The output from `butfirst` is a list containing all but the first member of the

input if the input is a list, or a word containing all but the first character of the input if it's a word:

```
? print butfirst "Hello
ello
? print butfirst [How are you?]
are you?
```

Notice that the `first` of a list can be a word, but the `butfirst` of any datum is always another datum of the same type. Also notice what happens when you take the `butfirst` of a datum with only one thing in it:

```
? print butfirst "A

? print butfirst [Hello]

?
```

In each case Logo printed a blank line. In the first case that blank line represents an empty word, a word with no characters in it. The second blank line represents an empty list, a list with no members. You can indicate the empty word in an instruction by using a quotation mark with a space (or the RETURN key to end the instruction) after it. To indicate an empty list, use brackets with nothing inside them:

```
? print " print []


?
```

Do you understand why it doesn't make sense to use the empty word or the empty list as input to `first` or `butfirst`? Try it and see what happens.

You should also notice that the list `[Hello]` is not the same as the word `"Hello`. They look the same when you print them, but they act differently when you take their `first` or `butfirst`.

There are also primitive operations `last` and `butlast`. I'm sure you'll have no trouble guessing what they do. Try them out, then practice describing them properly.

This is probably a good place to mention that there are *abbreviations* for some Logo primitive procedures. For example, `bf` is an abbreviation for `butfirst`. `Pr` is an abbreviation for `print`. There isn't any abbreviation for `first`.

If you want to extract a piece of a word or list that isn't at the beginning or end, you can use the more general operation `item` with two inputs: a positive integer to indicate which member to select, and a word or list. For example:

```
? print item 3 "Yesterday
s
? print item 2 [Good Day Sunshine]
Day
```

`First`, `last`, `butfirst`, `butlast`, and `item` are taking-apart operations, or *selectors*. Logo also provides putting-together operations, or *constructors*.

`Sentence` is a constructor. It takes two inputs, which can be any data at all. Its output is always a list.

Describing the output from `sentence` is a little tricky because the same procedure serves two different purposes. The first purpose is the one suggested by its name: constructing sentences. If you use only words and sentences (flat lists) as inputs, then the output from `sentence` is a sentence concatenating (stringing together) the words contained in the inputs. Here are some examples:

```
? print sentence "hello "goodbye
hello goodbye
? print sentence [this is] [a test]
this is a test
? print sentence "this [is one too]
this is one too
? print sentence [] [list of words]
list of words
```

On the other hand, `sentence` can also be used to append two lists (flat or not). With lists as inputs, the output from `sentence` is a list in which the *members* of the first input and the *members* of the second input are concatenated:

```
? print sentence [[list 1a] [list 1b]] [[list 2a] [list 2b]]
[list 1a] [list 1b] [list 2a] [list 2b]
? print sentence [flat list] [[not flat] [list]]
flat list [not flat] [list]
```

In the second example the output is a list with four members: two words and two lists.

Using a word as input to `sentence` is equivalent to using a list with that word as its single member. `Sentence` is the only primitive operation that treats words the same as

single-word lists; you've seen from the earlier examples that `first` and `butfirst` treat the word `hello` and the list `[hello]` differently.

Another constructor for lists is `list`. Its inputs can be any data; its output is a list whose members are the inputs—not the members of the inputs, as for `sentence`.

```
? print list [this is] [a test]
[this is] [a test]
? print list "this [is one too]
this [is one too]
? print list [] [list of words]
[] [list of words]
```

`Word` is an operation that takes two inputs. Both inputs must be words. (They may be the empty word.) The output from `word` is a word formed by concatenating the characters in the input words:

```
? print word "hello "goodbye
hellogoodbye
? print word "now "here
nowhere
? print word "this [is a test]
word doesn't like [is a test] as input
```

Selectors and constructors can be composed, in the same way we composed `sum` and `product` earlier. See if you can work out what this example will do before you try it with the computer:*

---

* The tilde (~) at the end of the first line is the notation used by Berkeley Logo to indicate that this and the following line should be understood as a single, long instruction line. It's somewhat analogous to the way a hyphen (-) is used in English text when a single word must be split between two lines. Berkeley Logo will also continue an instruction to the next line if a line ends inside parentheses or brackets, so another way to indicate a long instruction line is to enclose the entire instruction in parentheses, like this:

```
(print word word last "awful first butfirst "computer
   first [go to the store, please.])
```

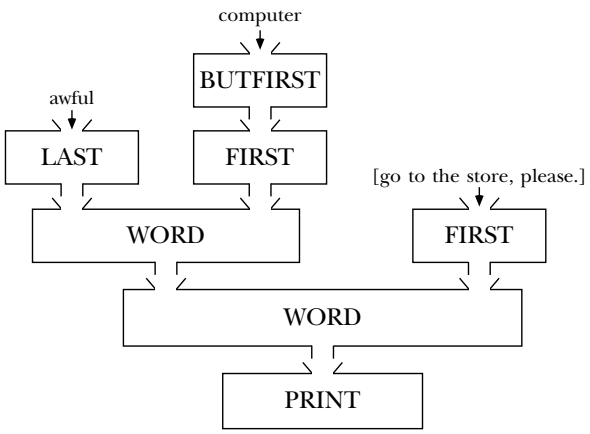
Other Logo dialects have other rules for line continuation. (In some dialects everything you type is automatically taken as one big line, so you don't have to think about this.) In the book, I'll indent continuation lines, as above, to make it quite clear that they are meant to be part of the same instruction as the line above. But Logo doesn't pay attention to the indentation.

```
print word word last "awful first butfirst "computer ~
   first [go to the store, please.]
```

Here is how I'd analyze it.

- The input to `print` is the output from `word`.
- The first input to `word` is the output from `word`.
- The first input to (the second) `word` is the output from `last`.
- The input to `last` is the quoted word `awful`.
- The output from `last` is the word `l`, which becomes the first input to the second `word`.
- The second input to the second `word` is the output from `first`.
- The input to `first` is the output from `butfirst`.
- The input to `butfirst` is the quoted word `computer`.
- The output from `butfirst` is the word `omputer`, which becomes the input to `first`.
- The output from `first` is the word `o`, which becomes the second input to the second `word`.
- The output from the second `word` is the word `lo`, which becomes the first input to the first `word`.
- The second input to (the first) `word` is the output from (the second) `first`.
- The input to `first` is the sentence `[go to the store, please.]`.
- The output from `first` is the word `go`, which becomes the second input to the first `word`.
- The output from `word` is the word `logo`, which becomes the input to `print`.
- Finally, `print` prints the word `logo`.

And here is the plumbing diagram:



☞ If you made it through that, you should find it easy to predict what these instructions will do:

```
print butlast "tricky
print butlast [tricky]
print se bl "farm bl bl bl "output
print first butfirst "hello
print first butfirst [abc def ghi]
(print word bl "hard word bl bl first [very hard]
   last first [extremely hard])
```

Remember that numbers are words, so you can combine arithmetic operations with these word and list operations:

```
? print word sum 2 3 product 2 3
56
? print sum word 2 3 product 2 3
29
? print sentence sum 2 3 word 2 3
5 23
```

`Count` is an operation that takes one input. The input can be any datum. The output from `count` is a number, indicating the length of the input. If the input is a word, the output is the number of characters in the word. If the input is a list, the output is the number of members in the list.

```
? print count "hello
5
? print count [hello]
1
? print count "
0
? print count []
0
? print word count "hello count "goodbye
57
? print sum count "hello count "goodbye
12
```

`Print` and `Show`

Because lists are often used to represent English sentences in conversational programs like the `hi` procedure of Chapter 1, `print` prints only the members of a list, without enclosing brackets. This behavior could be confusing if a list contains only one member:

```
? print [aardvark]
aardvark
? print "aardvark
aardvark
```

There is no visible difference between a word and a one-word list. But the two values are actually quite different, as we can see if we use them as inputs to `first`:

```
? print first [aardvark]
aardvark
? print first "aardvark
a
```

The `first` of a sentence is its first word, even if it has only one word, but the `first` of a word is its first letter.

To help distinguish words from lists, Logo has another printing command called `show` that displays brackets around lists:

```
? show [aardvark]
[aardvark]
? show "aardvark
aardvark
? show sentence [this is] [an example]
[this is an example]
? show list [this is] [an example]
[[this is] [an example]]
```

Use `print` if your program wants to carry on a conversation with the user in English. Use `show` if you are using lists to represent some structure other than a sentence.

Order of Evaluation

You may hear people say something like this: "Logo evaluates from right to left." What they mean is that in an instruction such as

```
print first butfirst butfirst [print the third word]
```

Logo first evaluates

```
butfirst [print the third word]
```

and next evaluates

```
butfirst [the third word]
```

and then

```
first [third word]
```

and finally

```
print "third
```

In other words, the procedures named toward the right end of the instruction line must be invoked *before* Logo can know the appropriate input values for the procedures farther to the left.

This right-to-left idea can be a useful way of helping you understand evaluation in Logo. But you should realize that it's not quite true. It only works out that way

if the instruction line contains only one instruction and each procedure used in that instruction takes only one input. If you look back at one of the examples in which two-input procedures such as `word` or `sum` are used, you'll see that Logo really does read the instruction line from left to right. And if there are two instructions on the same line, the one on the left is evaluated first.

The reason for the seeming right-to-left evaluation is that Logo can't *finish* evaluating a procedure invocation until it has collected and evaluated the inputs to the procedure. But Logo *starts* evaluating an instruction line by looking at the first word on the line. In the example just above, the evaluation of `first` and `butfirst` is *part of* the evaluation of `print`.

Special Forms of Evaluation

So far, the evaluation process has been very uniform. Logo looks at the first word of an instruction and interprets that word as the name of a procedure. Logo knows how many inputs each procedure requires. It then evaluates as many expressions as necessary to assign values to those inputs. The expressions are evaluated the same way: Logo looks at the first word... and so on.

Although this evaluation process is perfectly general, Logo also provides a couple of special forms of evaluation to make certain things easier to type. (The computer science terminology for such a special case is a "kludge." The letter "u" in this word is pronounced as in "rude," not as in "sludge.")

One special case is that Logo provides *infix arithmetic* as well as the *prefix arithmetic* we've used so far. That is, you can say

```
print 2+3
```

instead of

```
print sum 2 3
```

When you use infix operations, the usual rules of precedence apply: multiplications and divisions are done before additions and subtractions unless you use parentheses. In other words, `2+3*4` (the asterisk represents multiplication) means `2+(3*4)`, while `2*3+4` means `(2*3)+4`. You should take note that this issue of precedence doesn't arise when prefix operations are used.

☞ For example, look at these expressions:

```
sum 2 product 3 4
product sum 2 3 4
sum product 2 3 4
product 2 sum 3 4
```

Each of these indicates precisely what order of operations is desired. The first, for example, is equivalent to `2+3*4`. Try converting the others to infix form. Which ones require parentheses?

The second special form of evaluation is that certain primitive procedures can be given extra inputs, or fewer inputs than usual, by using parentheses around the procedure name and all its inputs. Here are some examples:

```
? print sum 2 3 4
5
You don't say what to do with 4
? print (sum 2 3 4)
9
? show (list "one)
[one]
? show (list)
[]
```

`Sum`, `product`, `word`, `list`, `sentence`, and `print` can be used with any number of inputs.

By the way, it is always permitted to enclose a procedure name and its inputs (the correct number of them!) in parentheses, even when it's not necessary, to make the instruction more readable. One of the earlier illustrations, for example, might be easier to read in this form:

```
print word (word (last "awful) (first butfirst "computer)) ~
   (first [go to the store, please.])
```

Notice that Logo's placement of parentheses is different from the function notation used in algebra. In algebra you say $f(x)$. In Logo you would express the same idea as `(f x)`.

Writing Your Own Procedures

With these tools, you are ready to begin writing new procedures. Type this:

```
to hello
```

`To` is a command, but it's a very special one. It's the only one that does not evaluate its inputs. Remember earlier when we said

```
print Hello
```

and Logo complained that it didn't know how to `Hello`? Well, `to` doesn't make that kind of complaint. Instead it prepares to have you *teach it how* `to hello`. (That's why `to` is called `to`!) What you should see on the screen is something like this:

```
? to hello
>
```

Instead of a question mark, Logo has printed a greater-than symbol as the prompt. This special prompt warns you that whatever instructions you type won't be carried out immediately, as usual. Instead Logo remembers what you type as part of the procedure named `hello`. Continue like this:

```
> print "Hello
> print [This is Logo speaking.]
> print [What's new?]
> end
?
```

The word `end` isn't the name of a procedure. It's a special signal to Logo that you're finished defining the procedure `hello`.*

Now you can try out your new procedure:

```
? hello
Hello
This is Logo speaking.
What's new?
```

* Why can't we simply think of `end` as the name of a procedure, just as `print` is? This is a minor point, but one that you can use to test your understanding of what's going on while you are defining a procedure. When you see the greater-than prompt, Logo *does not evaluate* the lines you type. It simply remembers those lines as part of the procedure you're defining. If `end` were a procedure, it wouldn't be evaluated right away, just as those `print` instructions aren't evaluated right away. It, too, would be remembered as part of the definition of `hello`. Instead, typing `end` has an *immediate* effect: It ends the procedure definition and returns to the question-mark prompt that allows interactive evaluation.

You can also examine the procedure itself by asking Logo to print it out. The command `po` (for Print Out) takes one input, a word or a list. The input is either the name of a procedure (if a word) or a list of names of procedures. The effect of `po` is to print out the definition(s) of the procedure(s) named by the input. Here is an example:

```
? po "hello
to hello
print "Hello
print [This is Logo speaking.]
print [What's new?]
end
?
```

Unlike `to`, but like all other Logo procedures, `po` *does* evaluate its input. That's why the word `hello` must be quoted in this example.

In a procedure definition the line starting `to` is called the *title line*. The lines containing instructions are, naturally, called *instruction lines*. We won't have many occasions to talk about the line containing only the word `end`, but just in case, we'll call it the *end line*.

The command `pops` (for Print Out ProcedureS) takes no inputs. Its effect is to print out the definitions of all the procedures you've defined. The command `pots` (for Print Out TitleS) also takes no inputs and prints out only the title lines of all the procedures you've defined.

Some writers and teachers reserve the word "procedure" to refer only to ones you write yourself, such as `hello`. They use the word "primitive" as a noun, to mean things like `print` and `butfirst`. They say things like "Logo instructions are made up of procedures and primitives." This is a big mistake. The procedures you write are *just like* the procedures Logo happens to know about in the first place. It's just that somebody else wrote the primitive procedures. But you use your own procedures in exactly the same way that you use primitive procedures: you type the name of the procedure and Logo evaluates that name by invoking the procedure. It's okay to say "`Last` is a primitive" as an abbreviation for "`Last` is a primitive procedure," as long as you know what you're talking about.

☞ Try defining more procedures. You'll find that you don't have quite enough tools yet to make your procedures very interesting; the main problem is that yours don't take inputs, so they do exactly the same thing every time you use them. We'll solve that problem in the next chapter.

Editing Your Procedures

As you may remember from earlier experiences, Logo includes an *editor,* a program that allows you to make corrections to a procedure you've defined. You can also use the editor to write procedure definitions in the first place. The editor works slightly differently in each version of Logo, so you should consult the manuals for your own computer (or Appendix A, for Berkeley Logo) to review the details.

By the way, when you're learning about the `edit` command, don't forget that it can accept a list of procedure names as input, not only a single word. By listing several procedures in the input to `edit`, you can have them all visible at once while you're editing, and you can copy instructions from one to another. This is a powerful capability of the Logo editor, which beginners often neglect.

Once you've gotten familiar with the Logo editor, you'll probably find yourself wanting to use it all the time, and you'll rarely choose to define a procedure by invoking `to` directly. (Don't get confused about that last sentence; of course you type `to` when you're using the editor, but you don't type it as a command to the Logo interpreter in response to a question mark prompt.) The editor makes it much easier to correct typing mistakes. Nevertheless, if you need to define a short procedure in the middle of doing something else, you may occasionally find it simpler to use `to` rather than wait for an editor to start up.

Syntax and Semantics

Except for the special case of `to`, all Logo instructions follow the same rules about the meaning of punctuation and about which subexpression provides an input to which procedure call. These are called *syntax* rules. The rules pay no attention to what any particular procedure means, or what inputs might or might not be sensible for that procedure; those aspects of a program are called its *semantics,* which is a fancy word for "meaning." You might say that Logo's plumber, the part of Logo that hooks up the plumbing diagrams, doesn't know anything about semantics. So, for example, if you make a mistake like

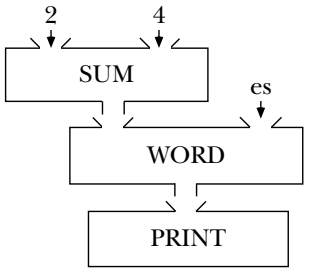
```
print item [john paul george ringo] 2
```

and get a Logo error message, you might feel that it's obvious what you meant—and it would be, to another person—and so Logo should have figured it out and done the right thing. But computers aren't as smart as people, and so you can rely only on Logo's syntax rules, not on the semantics of your program, to help Logo make sense of what you write.

To illustrate the difference between syntax and semantics, we'll start by examining the following Logo instruction:

```
? print word sum 2 4 "es
6es
```

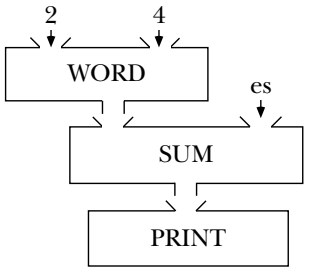
Here's its plumbing diagram:



The connections in a plumbing diagram depend only on the numbers of inputs and outputs for each procedure used. Logo "connects the plumbing" *before* invoking any of the procedures named in the instruction. The plumbing is connected regardless of whether the specified inputs actually make sense to the procedures in question. For example, suppose we make a slight change to the instruction given just now:

```
print sum word 2 4 "es
```

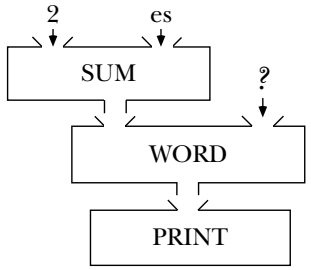
The only change is that `word` and `sum` have been interchanged. Since these are both two-input operations, the shape of the plumbing diagram is unchanged.



The plumbing connections are syntactically fine, so Logo can work out which expression provides the input to which procedure call. However, when Logo gets around to invoking the procedure `sum` with inputs `24` and `es`, an error message will result because the second input isn't a number. This is a *semantic* error.

By contrast, the following instruction shows a *syntactic* error, in which Logo is unable to figure out a plumbing diagram in which all the pieces connect up.

```
print word sum 2 "es
```



The question mark in the diagram indicates a missing input. In this example, the programmer intended the word `es` to be the second input to `word`; from the programmer's point of view, it is a number, the desired second input to `sum`, that's "really" missing. But Logo doesn't know about the programmer's intentions, and Logo's plumber follows uniform rules in deciding which input goes with which procedure call.

The rule is that Logo starts by looking for an input to `print`. The first thing it finds is `word`, so the output from `word` is hooked up to the input for `print`. Now Logo is looking for two inputs to `word`. The next thing it finds is `sum`, so the output from `sum` is hooked up to the first input for `word`. Now Logo is looking for two inputs to `sum`, and the syntax rules say that Logo must find those two inputs before it can continue with the still-pending task of finding a second input for `word`. Logo's plumber isn't smart enough to say, "Hey, here's a non-number as input to `sum`, and I happen to remember that we still need another input for `word`, so that must be what the programmer meant."

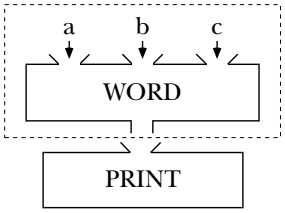
There are really only two kinds of plumbing errors. In the one shown here, too few expressions are included in the instruction, so that the message `not enough inputs` results. The other error is that too many expressions appear inside the instruction. This may result in the message `you don't say what to do with` something, or, if the extra expressions are within parentheses, by `too much inside ()'s`.

Parentheses and Plumbing Diagrams

Parentheses can be used in a Logo instruction for three reasons: for readability, to show the precedence of infix operators, or to include a nonstandard number of inputs for certain primitives. In all three cases, the syntax rule is that everything inside the

parentheses must form one single complete expression. In plumbing diagram terms, this means that the stuff inside the parentheses must correspond to a subdiagram with no inputs and with exactly one output (unless an entire instruction is parenthesized, in which case the diagram will have no outputs):

```
print (word "a "b "c)
```

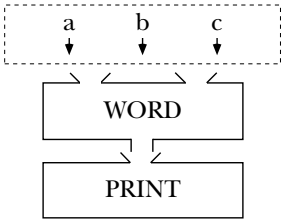


The dotted rectangle indicates the subdiagram corresponding to the expression inside the parentheses. That rectangle has no inputs; there are three inputs *within* the rectangle, but in each case the source of the input and the recipient of the input are both inside. There is no recipient inside the rectangle that needs a source from outside. The rectangle has one output; the entire expression within the rectangle provides the input to `print`.

The mathematical function notation $f(x)$ used in algebra often tempts beginning Logo programmers to write the above example as

```
print word ("a "b "c)          ; (wrong)
```

but by thinking about the plumbing diagram we can see that that would not put one single expression inside the parentheses:

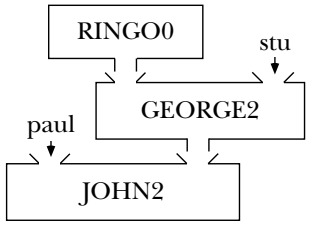


The part of the instruction inside the parentheses is trying to provide three outputs, not just one. This violates the rules. Also, since the word `word` isn't inside the parentheses, that procedure follows its ordinary rules and expects only two inputs.

Nonsense Plumbing Diagrams

To emphasize the point that the plumbing diagram depends only on the number of inputs expected by each procedure, and not on the purpose or meaning of the procedure, we can draw plumbing diagrams for nonsense instructions using unknown procedures. The rule of this game is that each procedure name includes a number indicating how many inputs it accepts. For example, `garply2` is a procedure that requires two inputs. If a procedure can accept extra inputs when used with parentheses, we put an `x` after the number; `baz3x` ordinarily takes three inputs, but can be given any number of inputs by using parentheses around the subexpression that invokes it.

```
john2 "paul george2 ringo0 "stu
```



We don't have to know what any of these procedures do. The only information we need is that some words in the instruction are quoted, while others are names of procedures that take a known number of inputs. This is a syntactically correct instruction because each procedure has been given exactly as many inputs as it requires.

☞ Try these:

```
baz3x 1 2 foo3x foo3x 4 5 6 (foo3x 7) 8
baz3x 1 [2 foo3x foo3x 4 5 6 (foo3x 7)] 8
if2 test3 [a b] [c d] [e f] [g h]
if2 try0 [foo3x 8 9]
```