6 Example: Tic-Tac-Toe

Program file for this chapter: `ttt`

This chapter is the first application of the ideas we've explored to a sizable project. The primary purpose of the chapter is to introduce the techniques of *planning* a project, especially the choice of how to organize the information needed by the program. This organization is called the *data structure* of the program. Along the way, we'll also see a new data type, the array, and a few other details of Logo programming.

The Project

Tic-tac-toe is not a very challenging game for human beings. If you're an enthusiast, you've probably moved from the basic game to some variant like three-dimensional tic-tac-toe on a larger grid.

If you sit down right now to play ordinary three-by-three tic-tac-toe with a friend, what will probably happen is that every game will come out a tie. Both you and your friend can probably play perfectly, never making a mistake that would allow your opponent to win.

But can you *describe* how you know where to move each turn? Most of the time, you probably aren't even aware of alternative possibilities; you just look at the board and instantly know where you want to move. That kind of instant knowledge is great for human beings, because it makes you a fast player. But it isn't much help in writing a computer program. For that, you have to know very explicitly what your strategy is.

By the way, although the example of tic-tac-toe strategy is a relatively trivial one, this issue of instant knowledge versus explicit rules is a hot one in modern psychology. Some cognitive scientists, who think that human intelligence works through mechanisms similar to computer programs, maintain that when you know how to do something without knowing *how* you know, you have an explicit set of rules deep down inside. It's just that the rules have become a habit, so you don't think about them deliberately.

They're "compiled," in the jargon of cognitive psychology. On the other hand, some people think that your implicit how-to knowledge is very different from the sort of lists of rules that can be captured in a computer program. They think that human thought is profoundly different from the way computers work, and that a computer cannot be programmed to simulate the full power of human problem-solving. These people would say, for example, that when you look at a tic-tac-toe board you immediately grasp the strategic situation as a whole, and your eye is drawn to the best move without any need to examine alternatives according to a set of rules. (You might like to try to be aware of your own mental processes as you play a game of tic-tac-toe, to try to decide which of these points of view more closely resembles your own experience—but on the other hand, the psychological validity of such introspective evidence is *another* hotly contested issue in psychology!)

☞ Before you read further, try to write down a set of strategy rules that, if followed consistently, will never lose a game. Play a few games using your rules. Make sure they work even if the other player does something bizarre.

I'm going to number the squares in the tic-tac-toe board this way:

1	2	3
4	5	6
7	8	9

Squares 1, 3, 7, and 9 are *corner squares*. I'll call 2, 4, 6, and 8 *edge squares*. And of course number 5 is the *center square*. I'll use the word *position* to mean a specific partly-filled-in board with X and O in certain squares, and other squares empty.

One way you might meet my challenge of describing your strategy explicitly is to list all the possible sequences of moves up to a certain point in the game, then say what move you'd make next in each situation. How big would the list have to be? There are nine possibilities for the first move. For each first move, there are eight possibilities for the second move. If you continue this line of reasoning, you'll see that there are nine factorial, or 362880, possible sequences of moves. Your computer may not have enough memory to list them all, and you certainly don't have enough patience!

Fortunately, not all these sequences are interesting. Suppose you are describing the rules a computer should use against a human player, and suppose the human being moves first. Then there are, indeed, nine possible first moves. But for each of these, there is only *one* possible computer move! After all, we're programming the computer. We get to decide which move it will choose. Then there are seven possible responses by the opponent, and so on. The number of sequences when the human being plays first is 9 times 7 times 5 times 3, or 945. If the computer plays first, it will presumably

always make the single best choice. Then there are eight possible responses, and so on. In this case the number of possible game sequences is 8 times 6 times 4 times 2, or 384. Altogether we have 1329 cases to worry about, which is much better than 300,000 but still not an enjoyable way to write a computer program.

In fact, though, this number is still too big. Not all games go for a full nine moves before someone wins. Also, many moves force the opponent to a single possible response, even though there are other vacant squares on the board. Another reduction can be achieved by taking advantage of *symmetry*. For example, if X starts in square 5, any game sequence in which O responds in square 1 is equivalent to a sequence in which O responds in square 3, with the board rotated 90 degrees. In fact there are only two truly different responses to a center-square opening: any corner square, or any edge square.

With all of these factors reducing the number of distinct positions, it would probably be possible to list all of them and write a strategy program that way. I'm not sure, though, because I didn't want to use that technique. I was looking for rules expressed in more general terms, like "all else being equal, pick a corner rather than an edge."

Why should I prefer a corner? Each corner square is part of three winning combinations. For example, square 1 is part of `123`, `147`, and `159`. (By expressing these winning combinations as three-digit numbers, I've jumped ahead a bit in the story with a preview of how the program I wrote represents this information.) An edge square, on the other hand, is only part of two winning combinations. For example, square 2 is part of `123` and `258`. Taking a corner square makes three winning combinations available to me and unavailable to my opponent.

Since I've brought up the subject of winning combinations, how many of *them* are there? Not very many: three horizontal, three vertical, and two diagonal. Eight altogether. That *is* a reasonable amount of information to include in a program, and in fact there is a list of the eight winning combinations in this project.

You might, at this point, enjoy playing a few games with the program, to see if you can figure out the rules it uses in its strategy. If you accepted my earlier challenge to write down your own set of strategy rules, you can compare mine to yours. Are they the same? If not, are they equally good?

The top-level procedure in this project is called `ttt`. It takes no inputs. When you invoke this procedure, it will ask you if you'd like to play first (X) or second (O). Then you enter moves by typing a digit 1–9 for the square you select. The program draws the game board on the Logo graphics screen.

I'm about to start explaining my strategy rules, so stop reading if you want to work out your own and haven't done it yet.

Strategy

The highest-priority and the lowest-priority rules seemed obvious to me right away. The highest-priority are these:

1. If I can win on this move, do it.
2. If the other player can win on the next move, block that winning square.

Here are the lowest-priority rules, used only if there is nothing suggested more strongly by the board position:

$n-2$. Take the center square if it's free.

$n-1$. Take a corner square if one is free.

n. Take whatever is available.

The highest priority rules are the ones dealing with the most urgent situations: either I or my opponent can win on the next move. The lowest priority ones deal with the least urgent situations, in which there is nothing special about the moves already made to guide me.

What was harder was to find the rules in between. I knew that the goal of my own tic-tac-toe strategy was to set up a *fork*, a board position in which I have two winning moves, so my opponent can only block one of them. Here is an example:

X	O	
	X	
X		O

X can win by playing in square 3 or square 4. It's O's turn, but poor O can only block one of those squares at a time. Whichever O picks, X will then win by picking the other one.

Given this concept of forking, I decided to use it as the next highest priority rule:

3. If I can make a move that will set up a fork for myself, do it.

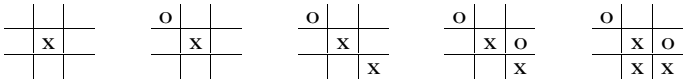
That was the end of the easy part. My first attempt at writing the program used only these six rules. Unfortunately, it lost in many different situations. I needed to add something, but I had trouble finding a good rule to add.

My first idea was that rule 4 should be the defensive equivalent of rule 3, just as rule 2 is the defensive equivalent of rule 1:

4a. If, on the next move, my opponent can set up a fork, block that possibility by moving into the square that is common to his two winning combinations.

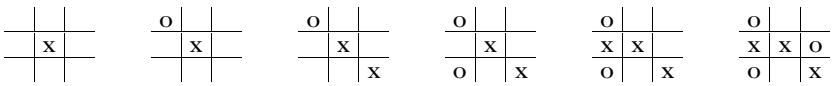
In other words, apply the same search technique to the opponent's position that I applied to my own.

This strategy works well in many cases, but not all. For example, here is a sequence of moves under this strategy, with the human player moving first:



In the fourth grid, the computer (playing O) has discovered that X can set up a fork by moving in square 6, between the winning combinations `456` and `369`. The computer moves to block this fork. Unfortunately, X can also set up a fork by moving in squares 3, 7, or 8. The computer's move in square 6 has blocked one combination of the square-3 fork, but X can still set up the other two. In the fifth grid, X has moved in square 8. This sets up the winning combinations `258` and `789`. The computer can only block one of these, and X will win on the next move.

Since X has so many forks available, does this mean that the game was already hopeless before O moved in square 6? No. Here is something O could have done:



In this sequence, the computer's second move is in square 7. This move also blocks a fork, but it wasn't chosen for that reason. Instead, it was chosen *to force X's next move*. In the fifth grid, X has had to move in square 4, to prevent an immediate win by O. The advantage of this situation for O is that square 4 was *not* one of the ones with which X could set up a fork. O's next move, in the sixth grid, is also forced. But by then the board is too crowded for either player to force a win; the game ends in a tie, as usual.

This analysis suggests a different choice for an intermediate-level strategy rule, taking the offensive:

4b. If I can make a move that will set up a winning combination for myself, do it.

Compared to my earlier try, this rule has the benefit of simplicity. It's much easier for the program to look for a single winning combination than for a fork, which is two such combinations with a common square.

Unfortunately, this simple rule isn't quite good enough. In the example just above, the computer found the winning combination `147` in which it already had square 1, and the other two were free. But why should it choose to move in square 7 rather than square 4? If the program did choose square 4, then X's move would still be forced, into square 7.

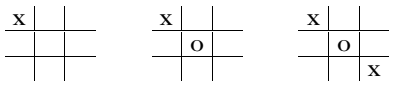
We would then have forced X into creating a fork, which would defeat the program on the next move.

It seems that there is no choice but to combine the ideas from rules 4a and 4b:

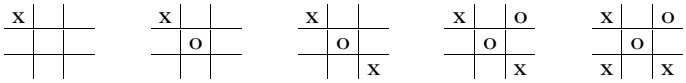
4. If I can make a move that will set up a winning combination for myself, do it. But ensure that this move does not force the opponent into establishing a fork.

What this means is that we are looking for a winning combination in which the computer already owns one square and the other two are empty. Having found such a combination, we can move in either of its empty squares. Whichever we choose, the opponent will be forced to choose the other one on the next move. If one of the two empty squares would create a fork for the opponent, then the computer must choose that square and leave the other for the opponent.

What if *both* of the empty squares in the combination we find would make forks for the opponent? In that case, we've chosen a bad winning combination. It turns out that there is only one situation in which this can happen:



Again, the computer is playing O. After the third grid, it is looking for a possible winning combination for itself. There are three possibilities: `258`, `357`, and `456`. So far we have not given the computer any reason to prefer one over another. But here is what happens if the program happens to choose `357`:



By this choice, the computer has forced its opponent into a fork that will win the game for the opponent. If the computer chooses either of the other two possible winning combinations, the game ends in a tie. (All moves after this choice turn out to be forced.)

This particular game sequence was very troublesome for me because it goes against most of the rules I had chosen earlier. For one thing, the correct choice for the program is any edge square, while the corner squares must be avoided. This is the opposite of the usual priority.

Another point is that this situation contradicts rule 4a (prevent forks for the other player) even more sharply than the example we considered earlier. In that example, rule 4a wasn't enough guidance to ensure a correct choice, but the correct choice was at least *consistent* with the rule. That is, just blocking a fork isn't enough, but threatening a win

and *also* blocking a fork is better than just threatening a win alone. This is the meaning of rule 4. But in this new situation, the corner square (the move we have to avoid) *does* block a fork, while the edge square (the correct move) *doesn't* block a fork!

When I discovered this anomalous case, I was ready to give up on the idea of beautiful, general rules. I almost decided to build into the program a special check for this precise board configuration. That would have been pretty ugly, I think. But a shift in viewpoint makes this case easier to understand: What the program must do is force the other player's move, and force it in a way that helps the computer win. If one possible winning combination doesn't allow us to meet these conditions, the program should try another combination. My mistake was to think either about forcing alone (rule 4b) or about the opponent's forks alone (rule 4a).

As it turns out, the board situation we've been considering is the only one in which a possible winning combination could include two possible forks for the opponent. What's more, in this board situation, it's a diagonal combination that gets us in trouble, while a horizontal or vertical combination is always okay. Therefore, I was able to implement rule 4 in a way that only considers one possible winning combination by setting up the program's data structures so that diagonal combinations are the last to be chosen. This trick makes the program's design less than obvious from reading the actual program, but it does save the program some effort.

Program Structure and Modularity

Most game programs—in fact, most interactive programs of any kind—consist of an initialization section followed by a sequence of steps carried out repeatedly. In the case of the tic-tac-toe game, the overall program structure will be something like this:

```
to ttt
initialize
forever [
   if game.is.over [stop]
   record.human.move get.human.move
   if game.is.over [stop]
   record.program.move compute.program.move
]
end
```

The parts of this structure shown in *italics* are just vague ideas. At this point in the planning, I don't know what inputs these procedures might need, for example. In fact, there may not be procedures exactly like this in the final program. One example is that

the test that I've called *game.is.over* here will actually turn out to be two separate tests `already.wonp` and `tiedp` (using a final letter `p` to indicate a predicate, following the convention established by the Logo primitive predicates).

This half-written procedure introduces a Logo primitive we haven't used before: `forever`. It takes a list of Logo instructions as its input, and carries out those instructions repeatedly, much as `repeat`, `for`, and `foreach` do. But the number of repetitions is unlimited; the repetition stops only if, as in this example, the primitive `stop` or `output` is invoked within the repeated instructions. `Forever` is useful when the ending condition can't be predicted in advance, as in a game situation in which a player might win at any time.

It may not be obvious why I've planned for one procedure to figure out the next move and a separate procedure to record it. (There are two such pairs of procedures, one for the program's moves and the other for the human opponent's moves.) For one thing, I expect that the recording of moves will be much the same whether it's the program or the person moving, while the decision about where to move will be quite different in the two cases. For the program's move we must apply strategy rules; for the human player's moves we simply ask the player. Also, I anticipate that the selection of the program's moves, which will be the hardest part of the program, can be written in functional style. The strategy procedure is a function that takes the current board position as its input, always returning the same chosen square for any given input position.

This project contains 28 procedures. These procedures can be divided into related groups like this:

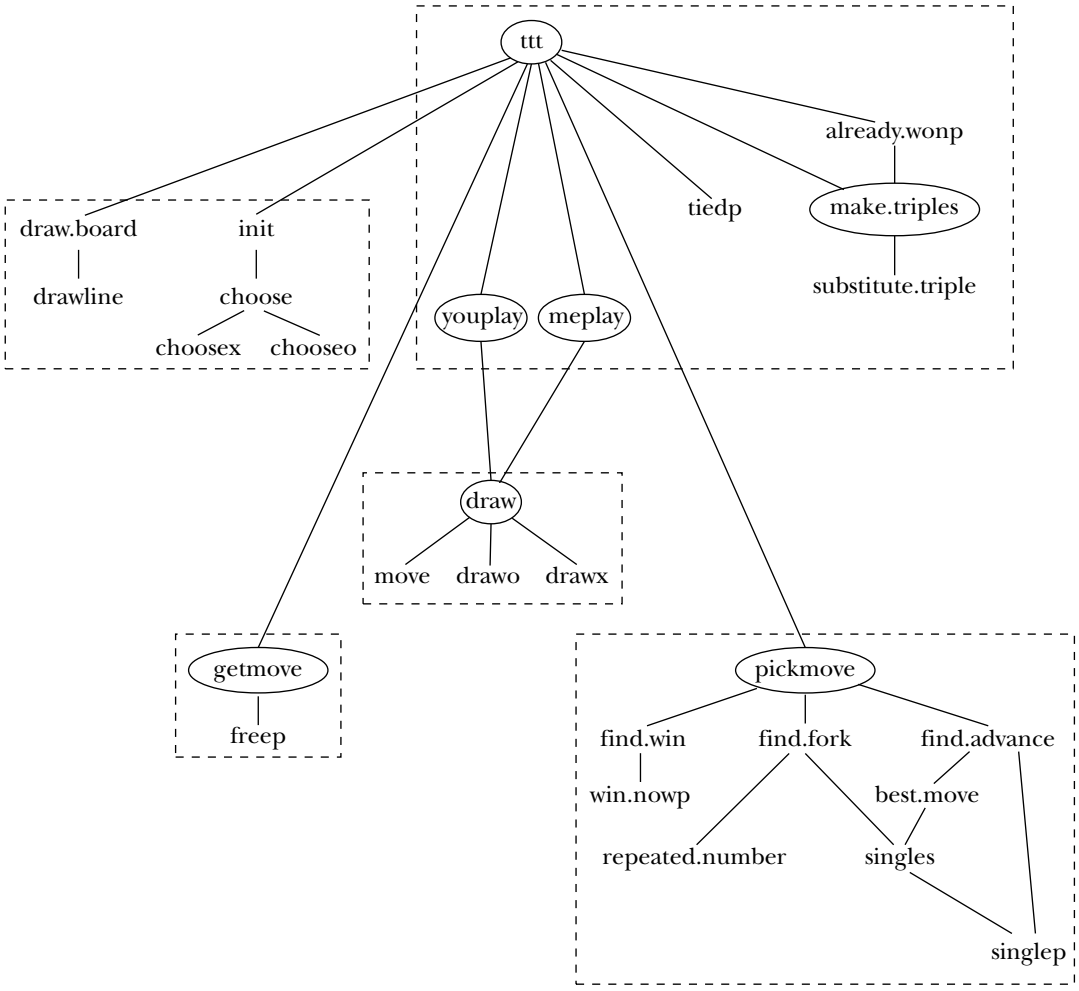
7 overall orchestration
6 initialization
2 get opponent's moves
9 compute program's moves
4 draw moves on screen

As you might expect, figuring out the computer's strategy is the most complex part of the program's job. But this strategic task is still only about a third of the complete program.

The five groups are quite cleanly distinguishable in this project. There are relatively few procedure invocations between groups, compared to the number within a group. It's easy to read the procedures within a group and understand how they work without having to think about other parts of the program at the same time.

The following diagram shows the subprocedure/superprocedure relationships within the program, and indicates which procedures are in each of the five groups listed above.

Some people find diagrams like this one very helpful in understanding the structure of a program. Other people don't like these diagrams at all. If you find it helpful, you may want to draw such diagrams for your own projects.



In the diagram, I've circled the names of seven procedures. If you understand the purpose of each of these, then you will understand the general structure of the entire program. (Don't turn to the end and read the actual procedures just now. Instead, see if you can understand the following paragraphs just from the diagram.)

`Ttt` is the top-level procedure for which I gave a rough outline earlier. It calls initialization procedures (`draw.board` and `init`) to set up the game, then repeatedly

alternates between the human opponent's moves and the program's moves. It calls `getmove` to find out the next move by the opponent, `youplay` to record that move, then `pickmove` to compute the program's next move and `meplay` to record it.

`Make.triples` translates from one representation of the board position to another. The representation used within `ttt` is best suited for display and for user interaction, while the representation output by `make.triples` is best for computing the program's strategy. We'll look into data representation more closely later.

`Getmove` invites the opponent to type in a move. It ensures that the selected move is legal before accepting it. The output from `getmove` is a number from 1 to 9 representing the chosen square.

`Pickmove` figures out the program's next move. It is the "smartest" procedure in the program, embodying the strategy rules I listed earlier. It, too, outputs a number from 1 to 9.

`Youplay` and `meplay` are simple procedures that actually carry out the moves chosen by the human player and by the program, respectively. Each contains only two instructions. The first invokes `draw` to draw the move on the screen. The second modifies the `position` array to remember that the move has been made.

`Draw` moves the turtle to the chosen square on the tic-tac-toe board. Then it draws either an X or an O. (We haven't really talked about Logo's turtle graphics yet. If you're not familiar with turtle graphics from earlier Logo experience, you can just take this part of the program on faith; there's nothing very interesting about it.)

Notice, in the diagram, that the lines representing procedure calls come into a box only at the top. This is one sign of a well-organized program: The dashed boxes in the diagram truly do represent distinct parts of the program that don't interact very much.

Data Representation

I've written several tic-tac-toe programs, in different programming languages. This experience has really taught me about the importance of picking a good data representation. For my first tic-tac-toe program, several years ago, I decided without much prior thought that a board position should be represented as three lists of numbers, one with X's squares, one with O's squares, and one with the free squares. So this board position

x	o	
x	x	
		o

could be represented like this:

```
make "xsquares [1 4 5]
make "osquares [2 9]
make "free [3 6 7 8]
```

These three variables would change in value as squares moved from `:free` to one of the others. This representation was easy to understand, but not very helpful for writing the program!

What questions does a tic-tac-toe program have to answer about the board position? If, for example, the program wants to print a display of the position, it must answer questions of the form "Who's in square 4?" With the representation shown here, that's not as easy a question as we might wish:

```
to occupant :square                          ;; old program
if memberp :square :xsquares [output "x]
if memberp :square :osquares [output "o]
output "free
end
```

On the other hand, this representation isn't so bad when we're accepting a move from the human player and want to make sure it's a legal move:

```
to freep :square                             ;; old program
output memberp :square :free
end
```

Along with this representation of the board, my first program used a constant list of winning combinations:

```
make "wins [[1 2 3] [4 5 6] [7 8 9] [1 4 7] [2 5 8]
            [3 6 9] [1 5 9] [3 5 7]]
```

It also had a list of all possible forks. I won't bother trying to reproduce this very long list for you, since it's not used in the current program, but the fork set up by X in the board position just above was represented this way:

```
[4 [1 7] [5 6]]
```

This indicates that square 4 is the pivot of a fork between the winning combinations `[1 4 7]` and `[4 5 6]`. Each member of the complete list of forks was a list like this sample.

The list of forks was fairly long. Each edge square is the pivot of a fork. Each corner square is the pivot of three forks. The center square is the pivot of six forks. This adds up to 22 forks altogether.

Each time the program wanted to choose a move, it would first check all eight possible winning combinations to see if two of the squares were occupied by the program and the third one free. Since any of the three squares might be the free one, this is a fairly tricky program in itself:

```
to checkwin :candidate :mysquares :free      ;; old program
if memberp first :candidate :free ~
   [output check1 butfirst :candidate :mysquares]
if memberp last :candidate :free ~
   [output check1 butlast :candidate :mysquares]
if memberp first butfirst :candidate :free ~
   [output check1 list first :candidate last :candidate :mysquares]
output "false
end

to check1 :sublist :mysquares                ;; old program
output and (memberp first :sublist :mysquares) ~
           (memberp last :sublist :mysquares)
end
```

This procedure was fairly slow, especially when invoked eight times, once for each possible win. But the procedure to check each of the possible forks was even worse!

In the program that I wrote for the first edition of *Computer Science Logo Style,* a very different approach is used. This approach is based on the realization that, at any moment, a particular winning combination may be free for anyone (all three squares free), available only to one player, or not available to anyone. It's silly for the program to go on checking a combination that can't possibly be available. Instead of a single list of wins, the new program has three lists:

`mywins`	wins available to the computer
`yourwins`	wins available to the opponent
`freewins`	wins available to anyone

Once I decided to organize the winning combinations in this form, another advantage became apparent: for each possible winning combination, the program need only remember the squares that are free, not the ones that are occupied. For example, the board position shown above would contain these winning combinations, supposing the computer is playing X:

```
make "mywins [[7] [6] [3 7]]
make "yourwins [[3 6] [7 8]]
make "freewins []
```

The sublist `[7]` of `:mywins` indicates that the computer can win simply by filling square 7. This list represents the winning combination that was originally represented as `[1 4 7]`, but since the computer already occupies squares 1 and 4 there is no need to remember those numbers.

The process of checking for an immediate win is streamlined with this representation for two reasons, compared with the `checkwin` procedure above. First, only those combinations in `:mywins` must be checked, instead of all eight every time. Second, an immediate win can be recognized very simply, because it is just a list with one member, like `[7]` and `[6]` in the example above. The procedure `single` looks for such a list:

```
to single :list                                  ;; old program
output find [equalp (count ?) 1] :list
end
```

The input to `single` is either `:mywins`, to find a winning move for the computer (rule 1), or `:yourwins`, to find and block a winning move for the opponent (rule 2).

Although this representation streamlines the strategy computation (the `pickmove` part of the program), it makes the recording of a move quite difficult, because combinations must be moved from one list to another. That part of the program was quite intricate and hard to understand.

Arrays

This new program uses *two* representations, one for the interactive part of the program and one for the strategy computation. The first of these is simply a collection of nine words, one per square, each of which is the letter X, the letter O, or the number of the square. With this representation, recording a move means changing one of the nine words. It would be possible to keep the nine words in a list, and compute a new list (only slightly different) after each move. But Logo provides another data type, the *array,* which allows for changing one member while keeping the rest unchanged.

If arrays allow for easy modification and lists don't, why not always use arrays? Why did I begin the book with lists? The answer is that each data type has advantages and disadvantages. The main disadvantage of an array is that you must decide in advance how big it will be; there aren't any constructors like `sentence` to lengthen an array.

In this case, the fixed length of an array is no problem, because a tic-tac-toe board has nine squares. The `init` procedure creates the position array with the instruction

```
make "position {1 2 3 4 5 6 7 8 9}
```

The braces `{}` indicate an array in the same way that brackets indicate a list.

If player X moves in square 7, we can record that information by saying

```
setitem 7 :position "x
```

(Of course, the actual instruction in procedures `meplay` and `youplay` uses variables instead of the specific values 7 and X.) `Setitem` is a command with three inputs: a number indicating which member of the array to change, the array itself, and the new value for the specified member.

To find out who owns a particular square, we could write this procedure:

```
to occupant :square
output item :square :position
end
```

(The `item` operation can select a member of an array just as it can select a member of a list or of a word.) In fact, though, it turns out that I don't have an `occupant` procedure in this program. But the parts of the program that examine the board position do use `item` in a similar way, as in this example:

```
to freep :square
output numberp item :square :position
end
```

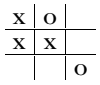
To create an array without explicitly listing all of its members, use the operation `array`. It takes a number as argument, indicating how many members the array should have. It returns an array of the chosen size, in which each member is the empty list. Your program can then use `setitem` to assign different values to the members.

The only primitive operation to select a member of an array is `item`. Word-and-list operations such as `butfirst` can't be used with arrays. There are operations `arraytolist` and `listtoarray` to convert a collection of information from one data type to the other.

Triples

The position array works well as a long-term representation for the board position, because it's easy to update; it also works well for interaction with the human player, because it's easy to find out the status of a particular square. But for computing the program's moves, we need a representation that makes it easy to ask questions such as "Is there a winning combination for my opponent on the next move?" That's why, in the first edition of these books, I used the representation with three lists of possible winning combinations.

When Matthew Wright and I wrote the book *Simply Scheme,* we decided that the general idea of combinations was a good one, but the three lists made the program more complicated than necessary. Since there are only eight possible winning combinations in the first place, it's not so slow to keep one list of all of them, and use that list as the basis for all the questions we ask in working out the program's strategy. If the current board position is



we represent the three horizontal winning combinations with the words `xo3`, `xx6`, and `78o`. Each combination is represented as a three-"letter" word containing an `x` or an `o` for an occupied square, or the square's number for a free square. By using words instead of lists for the combinations, we make the entire set of combinations more compact and easier to read. Each of these words is called a *triple.* The job of procedure `make.triples` is to combine the information in the position array with a list of the eight winning combinations:

```
? show make.triples
[xo3 xx6 78o xx7 ox8 36o xxo 3x7]
```

`Make.triples` takes no inputs because the list of possible winning combinations is built into it, and the position array is in `ttt`'s local variable `position`:

```
to make.triples
output map "substitute.triple [123 456 789 147 258 369 159 357]
end

to substitute.triple :combination
output map [item ? :position] :combination
end
```

This short subprogram will repay careful attention. It uses `map` twice, once in `make.triples` to compute a function of each possible winning combination, and once in `substitute.triple` to compute a function of each square in a given combination. (That latter function is the one that looks up the square in the array `:position`.)

Once the program can make the list of triples, we can use that to answer many questions about the status of the game. For example, in the top-level `ttt` procedure we must check on each move whether or not a certain player has already won the game. Here's how:

```
to already.wonp :player
output memberp (word :player :player :player) (make.triples)
end
```

If we had only the `position` array to work with, it would be complicated to check all the possible winning combinations. But once we've made the list of triples, we can just ask whether the word `xxx` or the word `ooo` appears in that list.

Here is the actual top-level procedure definition:

```
to ttt
local [me you position]
draw.board
init
if equalp :me "x [meplay 5]
forever [
  if already.wonp :me [print [I win!] stop]
  if tiedp [print [Tie game!] stop]
  youplay getmove                          ;; ask person for move
  if already.wonp :you [print [You win!] stop]
  if tiedp [print [Tie game!] stop]
  meplay pickmove make.triples             ;; compute program's move
]
end
```

Notice that `position` is declared as a local variable. Because of Logo's dynamic scope, all of the subprocedures in this project can use `position` as if it were a global variable, but Logo will "clean up" after the game is over.

Two more such quasi-global variables are used to remember whether the computer or the human opponent plays first. The value of `me` will be either the word `x` or the word `o`, whichever letter the program itself is playing. Similarly, the value of `you` will be `x` or `o`

to indicate the letter used by the opponent. All of these variables are given their values by the initialization procedure `init`.

This information could have been kept in the form of a single *flag variable,* called something like `mefirst`, that would contain the word `true` if the computer is X, or `false` if the computer is O. (A flag variable is one whose value is always `true` or `false`, just as a predicate is a procedure whose output is `true` or `false`.) It would be used something like this:

```
if :mefirst [draw "x :square] [draw "o :square]
```

But it turned out to be simpler to use two variables and just say

```
draw :me :square
```

One detail in the final program that wasn't in my first rough draft is the instruction

```
if equalp :me "x [meplay 5]
```

just before the `forever` loop. It was easier to write the loop so that it always gets the human opponent's move first, and then computes a move for the program, rather than having two different loops depending on which player goes first. If the program moves first, its strategy rules would tell it to choose the center square, because there is nothing better to do when the board is empty. By checking for that case before the loop, we are ready to begin the loop with the opponent as the next to move.

Variables in the Workspace

There are nine global variables that are part of the workspace, entered directly with top-level `make` instructions rather than set up by `init`, because their values are never changed. Their names are `box1` through `box9`, and their values are the coordinates on the graphics screen of the center of each square. For example, `:box1` is `[-40 50]`. These variables are used by `move`, a subprocedure of `draw`, to know where to position the turtle before drawing an X or an O.

The use of variables loaded with a workspace file, rather than given values by an initialization procedure, is a practice that Logo encourages in some ways and discourages in others. Loading variables in a workspace file makes the program start up faster, because it decreases the amount of initialization required. On the other hand, variables are sort of second-class citizens in workspace files. In many versions of Logo the `load`

command lists the names of the procedures in the workspace file, but not the names of the variables. Similarly, `save` often reports the number of procedures saved, but not the number of variables. It's easy to create global variables and forget that they're there.

Certainly preloading variables makes sense only if the variables are really constants; in other words, a variable whose value may change during the running of a program should be initialized explicitly when the program starts. Otherwise, the program will probably give incorrect results if you run it a second time. (One of the good ideas in the programming language Pascal is that there is a sort of thing in the language called a *constant*; it has a name and a value, like a variable, but you can't give it a new value in mid-program. In Logo, you use a global variable to hold a constant, and simply refrain from changing its value. But being able to *say* that something is a constant makes the program easier to understand.)

One reason the use of preloaded variables is sometimes questioned as a point of style is that when people are sloppy in their use of global variables, it's hard to know which are really meant to be preloaded and which are just left over from running the program. That is, if you write a program, test it by running it, and then save it on a diskette, any global variables that were created during the program execution will still be in the workspace when you load that diskette file later. If there are five intentionally-loaded variables along with 20 leftovers, it's particularly hard for someone to understand which are which. This is one more reason not to use global variables when what you really want are variables local to the top-level procedure.

The User Interface

The only part of the program that really interacts with the human user is `getmove`, the procedure that asks the user where to move.

```
to getmove
local "square
forever [
  type [Your move:]
  make "square readchar
  print :square
  if numberp :square
     [if and (:square > 0) (:square < 10)
         [if freep :square [output :square]]]
  print [not a valid move.]
]
end
```

There are two noteworthy things about this part of the program. One is that I've chosen to use `readchar` to read what the player types. This primitive operation, with no inputs, waits for the user to type any single character on the keyboard, and outputs whatever character the user types. This "character at a time" interaction is in contrast with the more usual "line at a time" typing, in which you can type characters, erase some if you make a mistake, and finally use the RETURN or ENTER key to indicate that the entire line you've typed should be made available to your program. (In Chapter 1 you met Logo's `readlist` primitive for line at a time typing.) Notice that if tic-tac-toe had ten or more squares in its board I wouldn't have been able to make this choice, because the program would have to allow the entry of two-digit numbers.

`Readchar` was meant for fast-action programs such as video games. It therefore does not display (or *echo*) the character that you type on the computer screen. That's why `getmove` includes a `print` instruction to let the user see what she or he has typed!

The second point to note in `getmove` is how careful it is to allow for the possibility of a user error. Ordinarily, when one procedure uses a value that was computed by another procedure, the programmer can assume that the value is a legitimate one for the intended purpose. For example, when you invoke a procedure that computes a number, you assume that you can add the output to another number; you don't first use the `number?` predicate to double-check that the result was indeed a number. But in `getmove` we are dealing with a value that was typed by a human being, and human beings are notoriously error-prone! The user is *supposed* to type a number between 1 and 9. But perhaps someone's finger might slip and type a zero instead of a nine, or even some character that isn't a number at all. Therefore, `getmove` first checks that what the user typed is a number. If so, it then checks that the number is in the allowed range. (We'd get a Logo error message if `getmove` used the `<` operation with a non-numeric input.) Only if these conditions are met do we use the user's number as the square-selecting input to `freep`.

Implementing the Strategy Rules

To determine the program's next move, `ttt` invokes `pickmove`; since many of the strategy rules will involve an examination of possible winning combinations, `pickmove` is given the output from `make.triples` as its input.

The strategy I worked out for the program consists of several rules, in order of importance. So the structure of `pickmove` should be something like this:

```
to pickmove :triples
if first.rule.works [output first.rule's.square]
if second.rule.works [output second.rule's.square]
...
end
```

This structure would work, but it would be very inefficient, because the procedure to determine whether a rule is applicable does essentially the same work as the procedure to choose a square by following the rule. For example, here's a procedure to decide whether or not the program can win on this move:

```
to can.i.win.now
output not emptyp find "win.nowp :triples
end

to win.nowp :triple
output equalp (filter [not numberp ?] :triple) (word :me :me)
end
```

The subprocedure `win.nowp` decides whether or not a particular triple is winnable on this move, by looking for a triple containing one number and two letters equal to whichever of X or O the program is playing. For example, `3xx` is a winnable triple if the program is playing X.

The procedure to pick a move if there is a winnable triple also must apply `win.nowp` to the triples:

```
to find.winning.square
output filter "numberp find "win.nowp :triples
end
```

If there is a winnable triple `3xx`, then the program should move in square 3. We find that out by looking for the number within the first winnable triple we can find.

It seems inelegant to find a winnable triple just to see if there are any, then find the same triple again to extract a number from it. Instead, we take advantage of the fact that the procedure I've called `find.winning.square` will return a distinguishable value—namely, an empty list—if there is no winnable triple. We say

```
to pickmove :triples
local "try
make "try find.winning.square
if not emptyp :try [output :try]
...
end
```

In fact, instead of the procedure `find.winning.square` the actual program uses a similar `find.win` procedure that takes the letter X or O as an input; this allows the same procedure to check both rule 1 (can the computer win on this move) and rule 2 (can the opponent win on the following move).

`Pickmove` checks each of the strategy rules with a similar pair of instructions:

```
make "try something
if not emptyp :try [output :try]
```

Here is the complete procedure:

```
to pickmove :triples
local "try
make "try find.win :me                ; rule 1: can computer win?
if not emptyp :try [output :try]
make "try find.win :you               ; rule 2: can opponent win?
if not emptyp :try [output :try]
make "try find.fork                   ; rule 3: can computer fork?
if not emptyp :try [output :try]
make "try find.advance                ; rule 4: can computer force?
if not emptyp :try [output :try]
output find [memberp ? :position] [5 1 3 7 9 2 4 6 8]   ; rules 5-7
end
```

The procedures that check for each rule have a common flavor: They all use `filter` and `find` to select interesting triples and then to select an available square from the chosen triple. I won't go through them in complete detail, but there's one that uses a Logo feature I haven't described before. Here is `find.fork`:

```
to find.fork
local "singles
make "singles singles :me                    ; find triples like 14x, x23
if emptyp :singles [output []]
output repeated.number reduce "word :singles ; find square in two triples
end
```

Suppose the computer is playing X and the board looks like this:

`Find.fork` calls `singles` (a straightforward procedure that you can read in the complete listing at the end of this chapter) to find all the triples containing one X and two vacant squares. It outputs

```
[4x6 x47 3x7]
```

indicating that the middle row, the left column, and one of the diagonals meet these conditions. To find a fork, we must find a vacant square that is included in two of these triples. The expression

```
reduce "word :singles
```

strings these triples together into the word `4x6x473x7`. The job of `repeated.number` is to find a digit that occurs more than once in this word. Here is the procedure:

```
to repeated.number :squares
output find [memberp ? ?rest] filter "numberp :squares
end
```

The expression

```
filter "numberp :squares
```

gives us the word `464737`, which is the input word with the letters removed. We use `find` to find a repeated digit in this number. The new feature is the use of `?rest` in the predicate template

```
[memberp ? ?rest]
```

`?rest` represents the part of the input to `find` (or any of the other higher-order functions that understand templates) to the right of the value being used as `?`. So in this example, `find` first computes the value of the expression

```
memberp 4 64737
```

This happens to be `true`, so `find` returns the value 4 without looking at the remaining digits. But if necessary, `find` would have gone on to compute

```
memberp 6 4737
memberp 4 737
memberp 7 37
memberp 3 7
memberp 7 "
```

(using the empty word as `?rest` in the last line) until one of these turned out to be true.

Further Explorations

The obvious first place to look for improvements to this project is in the strategy.

At the beginning of the discussion about strategy, I suggested that one possibility would be to make a complete list of all possible move sequences, with explicit next-move choices recorded for each. How many such sequences are there? If you write the program in a way that considers rotations of the board as equivalent, perhaps not very many. For example, if the computer moves first (in the center, of course) there are really only two responses the opponent can make: a corner or an edge. Any corner is equivalent to any other. From that point on, the entire sequence of the game can be forced by the computer, to a tie if the opponent played a corner, or to a win if the opponent played an edge. If the opponent moves first, there are three cases, center, corner, or edge. And so on.

An intermediate possibility between the complete list of cases and the more general rules I used would be to keep a complete list of cases for, say, the first two moves. After that, general rules could be used for the "endgame." This is rather like the way people, and some computer programs, play chess: they have the openings memorized, and don't really have to start thinking until several moves have passed. This book-opening approach is particularly appealing to me because it would solve the problem of the anomalous sequence that made such trouble for me in rule 4.

A completely different approach would be to have no rules at all, but instead to write a *learning* program. The program might recognize an immediate win (rule 1) and the threat of an immediate loss (rule 2), but otherwise it would move randomly and record the results. If the computer loses a game, it would remember the last unforced choice it made in that game, and keep a record to try something else in the same situation next time. The result, after many games, would be a complete list of all possible sequences, as I suggested first, but the difference is that you wouldn't have to do the figuring out

of each sequence. Such learning programs are frequently used in the field of artificial intelligence.

It is possible to combine different approaches. A famous checkers-playing program written by Arthur Samuel had several general rules programmed in, like the ones in this tic-tac-toe program. But instead of having the rules arranged in a particular priority sequence, the program was able to learn how much weight to give each rule, by seeing which rules tended to win the game and which tended to lose.

If you're tired of tic-tac-toe, another possibility would be to write a program that plays some other game according to a strategy. Don't start with checkers or chess! Many people have written programs in which the computer acts as dealer for a game of Blackjack; you could reverse the roles so that you deal the cards, and the computer tries to bet with a winning strategy. Another source of ideas is Martin Gardner, author of many books of mathematical games.

Program Listing

```
;; Overall orchestration

to ttt
local [me you position]
draw.board
init
if equalp :me "x [meplay 5]
forever [
  if already.wonp :me [print [I win!] stop]
  if tiedp [print [Tie game!] stop]
  youplay getmove                          ;; ask person for move
  if already.wonp :you [print [You win!] stop]
  if tiedp [print [Tie game!] stop]
  meplay pickmove make.triples             ;; compute program's move
]
end

to make.triples
output map "substitute.triple [123 456 789 147 258 369 159 357]
end

to substitute.triple :combination
output map [item ? :position] :combination
end
```

```
to already.wonp :player
output memberp (word :player :player :player) (make.triples)
end

to tiedp
output not reduce "or map.se "numberp arraytolist :position
end

to youplay :square
draw :you :square
setitem :square :position :you
end

to meplay :square
draw :me :square
setitem :square :position :me
end

;; Initialization

to draw.board
splitscreen clearscreen hideturtle
drawline [-20 -50] 0 120
drawline [20 -50] 0 120
drawline [-60 -10] 90 120
drawline [-60 30] 90 120
end

to drawline :pos :head :len
penup
setpos :pos
setheading :head
pendown
forward :len
end

to init
make "position {1 2 3 4 5 6 7 8 9}
print [Do you want to play first (X)]
type [or second (O)? Type X or O:]
choose
print [For each move, type a digit 1-9.]
end
```

```
to choose
local "side
forever [
  make "side readchar
  pr :side
  if equalp :side "x [choosex stop]
  if equalp :side "o [chooseo stop]
  type [Huh? Type X or O:]
]
end

to chooseo
make "me "x
make "you "o
end

to choosex
make "me "o
make "you "x
end

;; Get opponent's moves

to getmove
local "square
forever [
  type [Your move:]
  make "square readchar
  print :square
  if numberp :square [
     [if and (:square > 0) (:square < 10)
         [if freep :square [output :square]]]
  print [not a valid move.]
]
end

to freep :square
output numberp item :square :position
end
```

```
;; Compute program's moves

to pickmove :triples
local "try
make "try find.win :me
if not emptyp :try [output :try]
make "try find.win :you
if not emptyp :try [output :try]
make "try find.fork
if not emptyp :try [output :try]
make "try find.advance
if not emptyp :try [output :try]
output find [memberp ? :position] [5 1 3 7 9 2 4 6 8]
end

to find.win :who
output filter "numberp find "win.nowp :triples
end

to win.nowp :triple
output equalp (filter [not numberp ?] :triple) (word :who :who)
end

to find.fork
local "singles
make "singles singles :me
if emptyp :singles [output []]
output repeated.number reduce "word :singles
end

to singles :who
output filter [singlep ? :who] :triples
end

to singlep :triple :who
output equalp (filter [not numberp ?] :triple) :who
end

to repeated.number :squares
output find [memberp ? ?rest] filter "numberp :squares
end

to find.advance
output best.move filter "numberp find [singlep ? :me] :triples
end
```

```
to best.move :my.single
local "your.singles
if emptyp :my.single [output []]
make "your.singles singles :you
if emptyp :your.singles [output first :my.single]
ifelse (count filter [? = first :my.single]
                     reduce "word :your.singles) > 1 ~
       [output first :my.single] ~
       [output last :my.single]
end

;; Drawing moves on screen

to draw :who :square
move :square
ifelse :who = "x [drawx] [drawo]
end

to move :square
penup
setpos thing word "box :square
end

to drawo
pendown
arc 360 18
end

to drawx
setheading 45
pendown
repeat 4 [forward 25.5 back 25.5 right 90]
end

make "box1 [-40 50]
make "box2 [0 50]
make "box3 [40 50]
make "box4 [-40 10]
make "box5 [0 10]
make "box6 [40 10]
make "box7 [-40 -30]
make "box8 [0 -30]
make "box9 [40 -30]
```