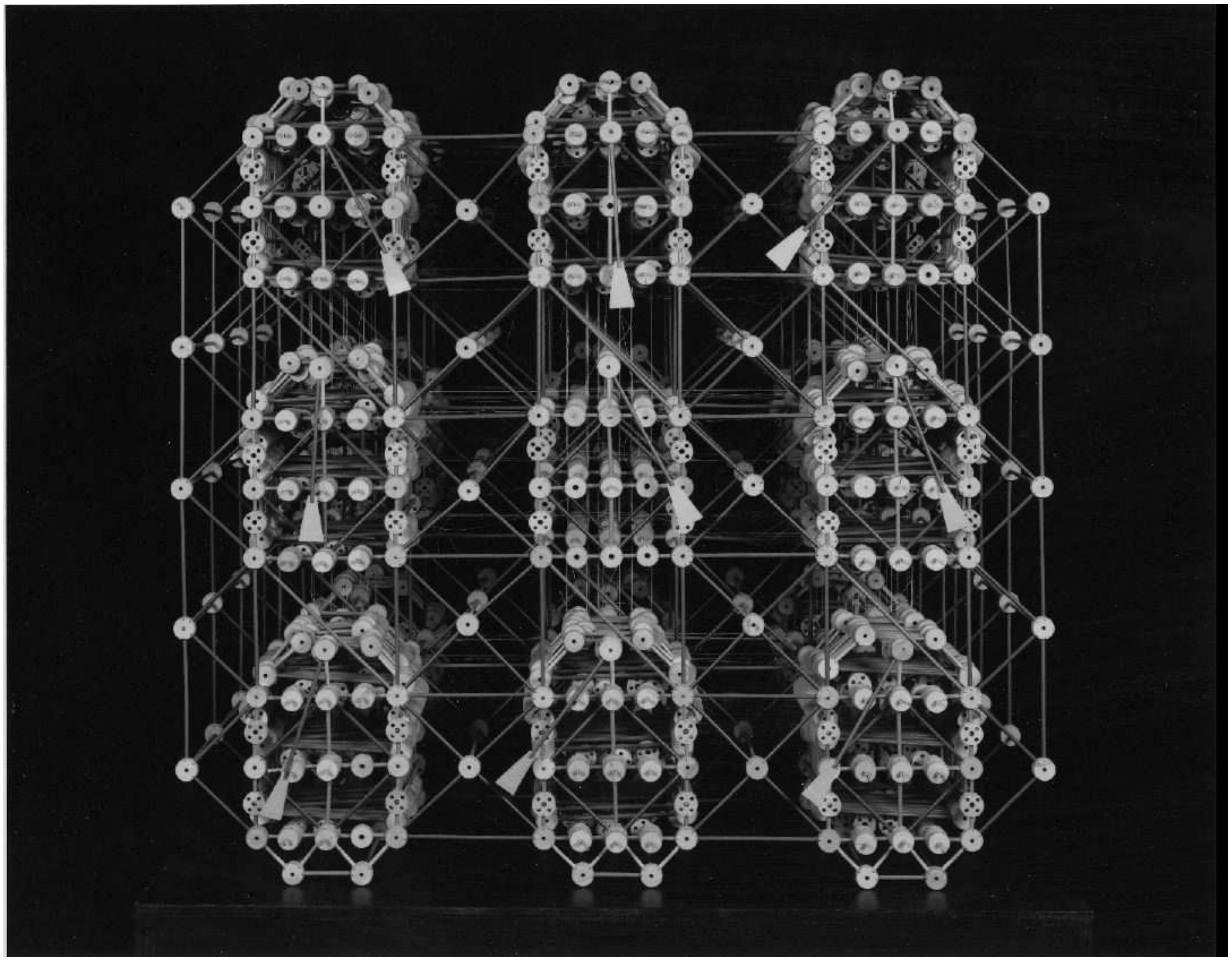


This computer, built of Tinker-Toy parts, plays tic-tac-toe.

10 Example: Tic-Tac-Toe

Now that you've learned about higher-order functions, we're going to look at a large example that uses them extensively. Using the techniques you've learned so far, we're going to write a program that plays perfect tic-tac-toe.

You can load our program into Scheme by typing

```
(load "ttt.scm")
```

(See Appendix A if this doesn't work for you.)

A Warning

Programs don't always come out right the first time. One of our goals in this chapter is to show you how a program is developed, so we're presenting early versions of procedures. These include some mistakes that we made, and also some after-the-fact simplifications to make our explanations easier. If you type in these early versions, they won't work. We will show you how we corrected these "bugs" and also will present a complete, correct version at the end of the chapter.

To indicate the unfinished versions of procedures, we'll use comments like "first version" or "not really part of game."

Technical Terms in Tic-Tac-Toe

We'll number the squares of the board this way:

1	2	3
4	5	6
7	8	9

We'll call a partially filled-in board a "position."

		o
	x	o
x		x

To the computer, the same position will be represented by the word `__o_xox_x`. The nine letters of the word correspond to squares one through nine of the board. (We're thinking ahead to the possibility of using `item` to extract the *n*th square of a given position.)

Thinking about the Program Structure

Our top-level procedure, `ttt`, will return the computer's next move given the current position. It takes two arguments: the current position and whether the computer is playing X or O. If the computer is O and the board looks like the one above, then we'd invoke `ttt` like this:

```
(ttt '__o_xox_x 'o)
```

Here is a sample game:

```
> (ttt '____x____ 'o)                    ; Human goes first in square 5
1                                        ; Computer moves in square 1
> (ttt 'o__xx____ 'o)                    ; Human moves in square 4
6                                        ; Computer blocks in square 6
> (ttt 'o_xxxo___ 'o)                    ; Human moves in square 3
7                                        ; Computer blocks again
> (ttt 'o_xxxoox_ 'o)
2
```

This is not a complete game program! Later, when we talk about input and output, you'll see how to write an interactive program that displays the board pictorially, asks the player where to move, and so on. For now, we'll just write the *strategy* procedure that chooses the next move. As a paying customer, you wouldn't be satisfied with this partial program, but from the programmer's point of view, this is the more interesting part.

Let's plan the computer's strategy in English before we start writing a computer program. How do *you* play tic-tac-toe? You have several strategy rules in your head, some of which are more urgent than others. For example, if you can win on this move, then you just do it without thinking about anything else. But if there isn't anything that immediate, you consider less urgent questions, such as how this move might affect what happens two moves later.

So we'll represent this set of rules by a giant `cond` expression:

```
(define (ttt position me)                    ;; first version
  (cond ((i-can-win?)
         (choose-winning-move))
        ((opponent-can-win?)
         (block-opponent-win))
        ((i-can-win-next-time?)
         (prepare-win))
        (else (whatever))))
```

We're imagining many helper procedures. `I-can-win?` will look at the board and tell if the computer has an immediate winning move. If so, `choose-winning-move` will find that particular move. `Opponent-can-win?` returns true if the human player has an immediate winning move. `Block-opponent-win` will return a move that prevents the computer's opponent from winning, and so on.

We didn't actually start by writing this definition of `ttt`. The particular names of helper procedures are just guesses, because we haven't yet planned the tic-tac-toe strategy in detail. But we did know that this would be the overall structure of our program. This big picture doesn't automatically tell us what to do next; different programmers might fill in the details differently. But it's a framework to keep in mind during the rest of the job.

Our first practical step was to think about the *data structures* in our program. A data structure is a way of organizing several pieces of information into a big chunk. For example, a sentence is a data structure that combines several words in a sequence (that is, in left-to-right order).

In the first, handwavy version of `ttt`, the strategy procedures like `i-can-win?` are called with no arguments, but of course we knew they would need some information about the board position. We began by thinking about how to represent that information within the program.

The First Step: Triples

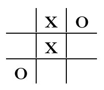
A person looking at a tic-tac-toe board looks at the rows, columns, and diagonals. The question "do I have a winning move?" is equivalent to the question "are there three squares in a line such that two of them are mine and the last one is blank?" In fact, nothing else matters about the game besides these potential winning combinations.

There are eight potential winning combinations: three rows, three columns, and two diagonals. Consider the combination containing the three squares 1, 5, and 9. If it contains both an `x` and an `o` then nobody can win with this combination and there's nothing to think about. But if it contains two `x`s and a free square, we're very interested in the combination. What we want to know in particular is which square is free, since we want to move in that square to win or block.

More generally, the only squares whose *numbers* we care about are the ones we might want to move into, namely, the free ones. So the only interesting information about a square is whether it has an `x` or an `o`, and if not, what its number is.

The information that 1, 5, 9 is a potential winning combination and the information that square 1 contains an `x`, square 5 is empty, and square `9` contains another `x` can be combined into the single word `x5x`. Looking at this word we can see immediately that there are two `x`s in this "triple" and that the free square is square 5. So when we want to know about a three-square combination, we will turn it into a triple of that form.

Here's a sample board position:

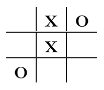


and here is a sentence of all of its triples:

```
(1xo 4x6 o89 14o xx8 o69 1x9 oxo)
```

Take a minute to convince yourself that this sentence really does tell you everything you need to know about the corresponding board position. Once our strategy procedure finds the triples for a board position, it's never going to look at the original position again.

This technique of converting data from one form to another so that it can be manipulated more easily is an important idea in computer science. There are really three representations of the same thing. There's this picture:



as well as the word `_xo_x_o__` and the sentence `(1xo 4x6 o89 14o xx8 o69 1x9 oxo)`. All three of these formats have the same information but are convenient in different ways. The pictorial form is convenient because it makes sense to the person who's playing tic-tac-toe. Unfortunately, you can't type that picture into a computer, so we need a different format, the word `_xo_x_o__`, which contains the *contents* of the nine squares in the picture, but without the lines separating the squares and without the two-dimensional shape.

The third format, the sentence, is quite *inconvenient* for human beings. You'd never want to think about a tic-tac-toe board that way yourself, because the sentence doesn't have the visual simplicity that lets you take in a tic-tac-toe position at a glance. But the sentence of triples is the most convenient representation for our program. `Ttt` will have to answer questions like "can `x` win on the next move?" To do that, it will have to consider an equivalent but more detailed question: "For each of the eight possible winning combinations, can `x` complete that combination on the next move?" It doesn't really matter whether a combination is a row or a column; what does matter is that each of the eight combinations be readily available for inspection by the program. The sentence-of-triples representation obscures part of the available information (which combination is where) to emphasize another part (making the eight combinations explicit, instead of implicit in the nine boxes of the diagram).

The representation of fractions as "mixed numerals," such as $2\frac{1}{3}$, and as "improper fractions," such as $\frac{7}{3}$, is a non-programming example of this idea about multiple representations. A mixed numeral makes it easier for a person to tell how big the number is, but an improper fraction makes arithmetic easier.

Finding the Triples

We said that we would combine the current board position with the numbers of the squares in the eight potential winning combinations in order to compute the things we're calling triples. That was our first task in writing the program.

Our program will start with this sentence of all the winning combinations:

```
(123 456 789 147 258 369 159 357)
```

and a position word such as `_xo_x_o__`; it will return a sentence of triples such as

```
(1xo 4x6 o89 14o xx8 o69 1x9 oxo)
```

All that's necessary is to replace some of the numbers with `x`s and `o`s. This kind of word-by-word translation in a sentence is a good job for `every`.

```
(define (find-triples position)              ;; first version
  (every substitute-triple '(123 456 789 147 258 369 159 357)))
```

We've made up a name `substitute-triple` for a procedure we haven't written yet. This is perfectly OK, as long as we write it before we try to invoke `find-triples`. The `substitute-triple` function will take three digits, such as `258`, and return a triple, such as `2x8`:

```
(define (substitute-triple combination)      ;; first version
  (every substitute-letter combination))
```

This procedure uses `every` to call `substitute-letter` on all three letters.

There's a small problem, though. `Every` always returns a sentence, and we want our triple to be a word. For example, we want to turn the potential winning combination `258` into the word `2x8`, but `every` would return the sentence `(2 x 8)`. So here's our next version of `substitute-triple`:

```
(define (substitute-triple combination)      ;; second version
  (accumulate word (every substitute-letter combination)))
```

`Substitute-letter` knows that letter number 3 of the word that represents the board corresponds to the contents of square 3 of the board. This means that it can just call `item` with the given square number and the board to find out what's in that square. If it's empty, we return the square number itself; otherwise we return the contents of the square.

```
(define (substitute-letter square)           ;; first version
  (if (equal? '_ (item square position))
      square
      (item square position)))
```

Whoops! Do you see the problem?

```
> (substitute-letter 5)
ERROR: Variable POSITION is unbound.
```

Using `Every` with Two-Argument Procedures

Our procedure only takes one argument, `square`, but it needs to know the position so it can find out what's in the given square. So here's the real `substitute-letter`:

```
(define (substitute-letter square position)
  (if (equal? '_ (item square position))
      square
      (item square position)))

> (substitute-letter 5 '_xo_x_o__)
X

> (substitute-letter 8 '_xo_x_o__)
8
```

Now `substitute-letter` can do its job, since it has access to the position. But we'll have to modify `substitute-triple` to invoke `substitute-letter` with two arguments.

This is a little tricky. Let's look again at the way we're using `substitute-letter` inside `substitute-triple`:

```
(define (substitute-triple combination)      ;; second version again
  (accumulate word (every substitute-letter combination)))
```

By giving `substitute-letter` another argument, we have made this formerly correct procedure incorrect. The first argument to `every` must be a function of one argument, not two. This is exactly the kind of situation in which `lambda` can help us: We have a function of two arguments, and we need a function of one argument that does the same thing, but with one of the arguments fixed.

The procedure returned by

```
(lambda (square) (substitute-letter square position))
```

does exactly the right thing; it takes a square as its argument and returns the contents of the position at that square.

Here's the final version of `substitute-triple`:

```
(define (substitute-triple combination position)
  (accumulate word
              (every (lambda (square)
                       (substitute-letter square position))
                     combination)))

> (substitute-triple 456 '_xo_x_o__)
"4X6"

> (substitute-triple 147 '_xo_x_o__)
"14O"

> (substitute-triple 357 '_xo_x_o__)
OXO
```

As you can see, Scheme prints some of these words with double-quote marks. The rule is that a word that isn't a number but begins with a digit must be double-quoted. But in the finished program we're not going to print such words at all; we're just showing you the working of a helper procedure. Similarly, in this chapter we'll show direct invocations of helper procedures in which some of the arguments are strings, but a user of the overall program won't have to use this notation.

We've fixed the `substitute-letter` problem by giving `substitute-triple` an extra argument, so we're going to have to go through the same process with `find-triples`. Here's the right version:

```
(define (find-triples position)
  (every (lambda (comb) (substitute-triple comb position))
         '(123 456 789 147 258 369 159 357)))
```

It's the same trick. `Substitute-triple` is a procedure of two arguments. We use `lambda` to transform it into a procedure of one argument for use with `every`.

We've now finished `find-triples`, one of the most important procedures in the game.

```
> (find-triples '_xo_x_o__)
("1XO" "4X6" O89 "14O" XX8 O69 "1X9" OXO)

> (find-triples 'x_____oxo)
(X23 456 OXO X4O "25X" "36O" X5O "35O")
```

Here again are the jobs of all three procedures we've written so far:

`Substitute-letter`	finds the letter in a single square.
`Substitute-triple`	finds all three letters corresponding to three squares.
`Find-triples`	finds all the letters in all eight winning combinations.

We've done all this because we think that the rest of the program can use the triples we've computed as data. So we'll just compute the triples once for all the other procedures to use:

```
(define (ttt position me)
  (ttt-choose (find-triples position) me))

(define (ttt-choose triples me)              ;; first version
  (cond ((i-can-win? triples me)
         (choose-winning-move triples me))
        ((opponent-can-win? triples me)
         (block-opponent-win triples me))
        ...))
```

Can the Computer Win on This Move?

The obvious next step is to write `i-can-win?`, a procedure that should return `#t` if the computer can win on the current move—that is, if the computer already has two squares of a triple whose third square is empty. The triples `x6x` and `oo7` are examples.

So we need a function that takes a word and a letter as arguments and counts how many times that letter appears in the word. The `appearances` primitive that we used in Chapter 2 (and that you re-implemented in Exercise 9.10) will do the job:

```
> (appearances 'o 'oo7)
2

> (appearances 'x 'oo7)
0
```

The computer "owns" a triple if the computer's letter appears twice and the opponent's letter doesn't appear at all. (The second condition is necessary to exclude cases like `xxo`.)

```
(define (my-pair? triple me)
  (and (= (appearances me triple) 2)
       (= (appearances (opponent me) triple) 0)))
```

Notice that we need a function `opponent` that returns the opposite letter from ours.

```
(define (opponent letter)
  (if (equal? letter 'x) 'o 'x))

> (opponent 'x)
O

> (opponent 'o)
X

> (my-pair? 'oo7 'o)
#T

> (my-pair? 'xo7 'o)
#F

> (my-pair? 'oox 'o)
#F
```

Finally, the computer can win if it owns any of the triples:

```
(define (i-can-win? triples me)                ;; first version
  (not (empty?
        (keep (lambda (triple) (my-pair? triple me))
              triples))))

> (i-can-win? '("1xo" "4x6" o89 "14o" xx8 o69 "1x9" oxo) 'x)
#T

> (i-can-win? '("1xo" "4x6" o89 "14o" xx8 o69 "1x9" oxo) 'o)
#F
```

By now you're accustomed to this trick with `lambda`. `My-pair?` takes a triple and the computer's letter as arguments, but we want a function of one argument for use with `keep`.

If So, in Which Square?

Suppose `i-can-win?` returns `#t`. We then have to find the particular square that will win the game for us. This will involve a repetition of some of the same work we've already done:

```
(define (choose-winning-move triples me)     ;; not really part of game
  (keep number? (first (keep (lambda (triple) (my-pair? triple me))
                             triples))))
```

We again use `keep` to find the triples with two of the computer's letter, but this time we extract the number from the first such winning triple.

We'd like to avoid this inefficiency. As it turns out, generations of Lisp programmers have been in just this bind in the past, and so they've invented a kludge* to get around it.

Remember we told you that everything other than `#f` counts as true? We'll take advantage of that by having a single procedure that returns the number of a winning square if one is available, or `#f` otherwise. In Chapter 6 we called such a procedure a "semipredicate." The kludgy part is that `cond` accepts a clause containing a single expression instead of the usual two expressions; if the expression has any true value, then `cond` returns that value. So we can say

```
(define (ttt-choose triples me)              ;; second version
  (cond ((i-can-win? triples me))
        ((opponent-can-win? triples me))
        ...))
```

where each `cond` clause invokes a semipredicate. We then modify `i-can-win?` to have the desired behavior:

```
(define (i-can-win? triples me)
  (choose-win
   (keep (lambda (triple) (my-pair? triple me))
         triples)))
```

* A kludge is a programming trick that doesn't follow the rules but works anyway somehow. It doesn't rhyme with "sludge"; it's more like "clue" followed by "j" as in "Jim."

```
(define (choose-win winning-triples)
  (if (empty? winning-triples)
      #f
      (keep number? (first winning-triples))))

> (i-can-win? '("1xo" "4x6" o89 "14o" xx8 o69 "1x9" oxo) 'x)
8

> (i-can-win? '("1xo" "4x6" o89 "14o" xx8 o69 "1x9" oxo) 'o)
#F
```

By this point, we're starting to see the structure of the overall program. There will be several procedures, similar to `i-can-win?`, that will try to choose the next move. `I-can-win?` checks to see if the computer can win on this turn, another procedure will check to see if the computer should block the opponent's win next turn, and other procedures will check for other possibilities. Each of these procedures will be what we've been calling "semipredicates." That is to say, each will return the number of the square where the computer should move next, or `#f` if it can't decide. All that's left is to figure out the rest of the computer's strategy and write more procedures like `i-can-win?`.

Second Verse, Same as the First

Now it's time to deal with the second possible strategy case: The computer can't win on this move, but the opponent can win unless we block a triple right now.

(What if the computer and the opponent both have immediate winning triples? In that case, we've already noticed the computer's win, and by winning the game we avoid having to think about blocking the opponent.)

Once again, we have to go through the complicated business of finding triples that have two of the opponent's letter and none of the computer's letter—but it's already done!

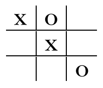
```
(define (opponent-can-win? triples me)
  (i-can-win? triples (opponent me)))

> (opponent-can-win? '("1xo" "4x6" o89 "14o" xx8 o69 "1x9" oxo) 'x)
#F

> (opponent-can-win? '("1xo" "4x6" o89 "14o" xx8 o69 "1x9" oxo) 'o)
8
```

Is that amazing or what?

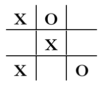
Now the Strategy Gets Complicated

Since our goal here is to teach programming, rather than tic-tac-toe strategy, we're just going to explain the strategy we use and not give the history of how we developed it.

The third step, after we check to see if either player can win on the next move, is to look for a situation in which a move that we make now will give rise to *two* winning triples next time. Here's an example:



Neither `x` nor `o` can win on this move. But if the computer is playing `x`, moving in square 4 or square 7 will produce a situation with two winning triples. For example, here's what happens if we move in square 7:



From this position, `x` can win by moving either in square 3 or in square 4. It's `o`'s turn, but `o` can block only one of these two possibilities. By contrast, if (in the earlier position) `x` moves in square 3 or square 6, that would create a single winning triple for next time, but `o` could block it.

In other words, we want to find *two* triples in which one square is taken by the computer and the other two are free, with one free square shared between the two triples. (In this example, we might find the two triples `x47` and `3x7`; that would lead us to move in square 7, the one that these triples have in common.) We'll call a situation like this a "fork," and we'll call the common square the "pivot." This isn't standard terminology; we just made up these terms to make it easier to talk about the strategy.

In order to write the strategy procedure `i-can-fork?` we assume that we'll need a procedure `pivots` that returns a sentence of all pivots of forks currently available to the computer. In this board, 4 and 7 are the pivots, so the `pivots` procedure would return the sentence `(4 7)`. If we assume `pivots`, then writing `i-can-fork?` is straightforward:

```
(define (i-can-fork? triples me)
  (first-if-any (pivots triples me)))
```

```
(define (first-if-any sent)
  (if (empty? sent)
      #f
      (first sent)))
```

Finding the Pivots

`Pivots` should return a sentence containing the pivot numbers. Here's the plan. We'll start with the triples:

```
(xo3 4x6 78o x47 ox8 36o xxo 3x7)
```

We `keep` the ones that have an `x` and two numbers:

```
(4x6 x47 3x7)
```

We mash these into one huge word:

```
4x6x473x7
```

We sort the digits from this word into nine "buckets," one for each digit:

```
("" "" 3 44 "" 6 77 "" "")
```

We see that there are no ones or twos, one three, two fours, and so on. Now we can easily see that four and seven are the pivot squares.

Let's write the procedures to carry out that plan. `Pivots` has to find all the triples with one computer-owned square and two free squares, and then it has to extract the square numbers that appear in more than one triple.

```
(define (pivots triples me)
  (repeated-numbers (keep (lambda (triple) (my-single? triple me))
                          triples)))

(define (my-single? triple me)
  (and (= (appearances me triple) 1)
       (= (appearances (opponent me) triple) 0)))

> (my-single? "4x6" 'x)
#T
```

```
> (my-single? 'xo3 'x)
#F

> (keep (lambda (triple) (my-single? triple 'x))
        (find-triples 'xo__x___o))
("4X6" X47 "3X7")
```

`My-single?` is just like `my-pair?` except that it looks for one appearance of the letter instead of two.

`Repeated-numbers` takes a sentence of triples as its argument and has to return a sentence of all the numbers that appear in more than one triple.

```
(define (repeated-numbers sent)
  (every first
         (keep (lambda (wd) (>= (count wd) 2))
               (sort-digits (accumulate word sent)))))
```

We're going to read this procedure inside-out, starting with the `accumulate` and working outward.

Why is it okay to `accumulate word` the sentence? Suppose that a number appears in two triples. All we need to know is that number, not the particular triples through which we found it. Therefore, instead of writing a program to look through several triples separately, we can just as well combine the triples into one long word, keep only the digits of that word, and simply look for the ones that appear more than once.

```
> (accumulate word '("4x6" x47 "3x7"))
"4X6X473X7"
```

We now have one long word, and we're looking for repeated digits. Since this is a hard problem, let's start with the subproblem of finding all the copies of a particular digit.

```
(define (extract-digit desired-digit wd)
  (keep (lambda (wd-digit) (equal? wd-digit desired-digit)) wd))

> (extract-digit 7 "4x6x473x7")
77

> (extract-digit 2 "4x6x473x7")
""
```

Now we want a sentence where the first word is all the `1`s, the second word is all the `2`s, etc. We could do it like this:

```
(se (extract-digit 1 "4x6x473x7")
    (extract-digit 2 "4x6x473x7")
    (extract-digit 3 "4x6x473x7")
    ...)
```

but that wouldn't be taking advantage of the power of computers to do that sort of repetitious work for us. Instead, we'll use `every`:

```
(define (sort-digits number-word)
  (every (lambda (digit) (extract-digit digit number-word))
         '(1 2 3 4 5 6 7 8 9)))
```

`Sort-digits` takes a word full of numbers and returns a sentence whose first word is all the ones, second word is all the twos, etc.*

```
> (sort-digits 123456789147258369159357)
(111 22 333 44 5555 66 777 88 999)

> (sort-digits "4x6x473x7")
("" "" 3 44 "" 6 77 "" "")
```

Let's look at `repeated-numbers` again:

```
(define (repeated-numbers sent)
  (every first
         (keep (lambda (wd) (>= (count wd) 2))
               (sort-digits (accumulate word sent)))))

> (repeated-numbers '("4x6" x47 "3x7"))
(4 7)

> (keep (lambda (wd) (>= (count wd) 2))
        '("" "" 3 44 "" 6 77 "" ""))
(44 77)

> (every first '(44 77))
(4 7)
```

* Brian thinks this is a kludge, but Matt thinks it's brilliant and elegant.

This concludes the explanation of `pivots`. Remember that `i-can-fork?` chooses the first pivot, if any, as the computer's move.

Taking the Offensive

Here's the final version of `ttt-choose` with all the clauses shown:

```
(define (ttt-choose triples me)
  (cond ((i-can-win? triples me))
        ((opponent-can-win? triples me))
        ((i-can-fork? triples me))
        ((i-can-advance? triples me))
        (else (best-free-square triples))))
```

You already know about the first three possibilities.

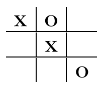
Just as the second possibility was the "mirror image" of the first (blocking an opponent's move of the same sort the computer just attempted), it would make sense for the fourth possibility to be blocking the creation of a fork by the opponent. That would be easy to do:

```
(define (opponent-can-fork? triples me)      ;; not really part of game
  (i-can-fork? triples (opponent me)))
```

Unfortunately, although the programming works, the strategy doesn't. Maybe the opponent has *two* potential forks; we can block only one of them. (Why isn't that a concern when blocking the opponent's wins? It *is* a concern, but if we've allowed the situation to reach the point where there are two ways the opponent can win on the next move, it's too late to do anything about it.)

Instead, our strategy is to go on the offensive. If we can get two in a row on this move, then our opponent will be forced to block on the next move, instead of making a fork. However, we want to make sure that we don't accidentally force the opponent *into* making a fork.

Let's look at this board position again, but from `o`'s point of view:



`x`’s pivots are 4 and 7, as we discussed earlier; `o` couldn’t take both those squares. Instead, look at the triples `369` and `789`, both of which are singles that belong to `o`. So `o` should move in one of the squares 3, 6, 7, or 8, forcing `x` to block instead of setting up the fork. But `o` *shouldn’t* move in square 8, like this:

x	o	
	x	
	o	o

because that would force `x` to block in square 7, setting up a fork!

x	o	
	x	
x	o	o

The structure of the algorithm is much like that of the other strategy possibilities. We use `keep` to find the appropriate triples, take the first such triple if any, and then decide which of the two empty squares in that triple to move into.

```
(define (i-can-advance? triples me)
  (best-move (keep (lambda (triple) (my-single? triple me)) triples)
             triples
             me))

(define (best-move my-triples all-triples me)
  (if (empty? my-triples)
      #f
      (best-square (first my-triples) all-triples me)))
```

`Best-move` does the same job as `first-if-any`, which we saw earlier, except that it also invokes `best-square` on the first triple if there is one.

Since we’ve already chosen the relevant triples before we get to `best-move`, you may be wondering why it needs *all* the triples as an additional argument. The answer is that `best-square` is going to look at the board position from the opponent’s point of view to look for forks.

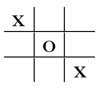
```
(define (best-square my-triple triples me)
  (best-square-helper (pivots triples (opponent me))
                      (keep number? my-triple)))
```

```
(define (best-square-helper opponent-pivots pair)
  (if (member? (first pair) opponent-pivots)
      (first pair)
      (last pair)))
```

We `keep` the two numbers of the triple that we've already selected. We also select the opponent's possible pivots from among all the triples. If one of our two possible moves is a potential pivot for the opponent, that's the one we should move into, to block the fork. Otherwise, we arbitrarily pick the second (`last`) free square.

```
> (best-square "78o" (find-triples 'xo__x___o) 'o)
7

> (best-square "36o" (find-triples 'xo__x___o) 'o)
6

> (best-move '("78o" "36o") (find-triples 'xo__x___o) 'o)
7

> (i-can-advance? (find-triples 'xo__x___o) 'o)
7
```

What if *both* of the candidate squares are pivots for the opponent? In that case, we've picked a bad triple; moving in either square will make us lose. As it turns out, this can occur only in a situation like the following:



If we chose the triple `3o7`, then either move will force the opponent to set up a fork, so that we lose two moves later. Luckily, though, we can instead choose a triple like `2o8`. We can move in either of those squares and the game will end up a tie.

In principle, we should analyze a candidate triple to see if both free squares create forks for the opponent. But since we happen to know that this situation arises only on the diagonals, we can be lazier. We just list the diagonals *last* in the procedure `find-triples`. Since we take the first available triple, this ensures that we won't take a diagonal if there are any other choices.*

* Matt thinks this is a kludge, but Brian thinks it's brilliant and elegant.

Leftovers

If all else fails, we just pick a square. However, some squares are better than others. The center square is part of four triples, the corner squares are each part of three, and the edge squares each a mere two.

So we pick the center if it's free, then a corner, then an edge.

```
(define (best-free-square triples)
  (first-choice (accumulate word triples)
                '(5 1 3 7 9 2 4 6 8)))

(define (first-choice possibilities preferences)
  (first (keep (lambda (square) (member? square possibilities))
               preferences)))

> (first-choice 123456789147258369159357 '(5 1 3 7 9 2 4 6 8))
5

> (first-choice "1xo4x6o8914oxx8o691x9oxo" '(5 1 3 7 9 2 4 6 8))
1

> (best-free-square (find-triples '_________))
5

> (best-free-square (find-triples '____x____))
1
```

Complete Program Listing

```
;;; ttt.scm
;;; Tic-Tac-Toe program

(define (ttt position me)
  (ttt-choose (find-triples position) me))

(define (find-triples position)
  (every (lambda (comb) (substitute-triple comb position))
         '(123 456 789 147 258 369 159 357)))

(define (substitute-triple combination position)
  (accumulate word
              (every (lambda (square)
                       (substitute-letter square position))
                     combination) ))
```

```
(define (substitute-letter square position)
  (if (equal? '_ (item square position))
      square
      (item square position) ))

(define (ttt-choose triples me)
  (cond ((i-can-win? triples me))
        ((opponent-can-win? triples me))
        ((i-can-fork? triples me))
        ((i-can-advance? triples me))
        (else (best-free-square triples)) ))

(define (i-can-win? triples me)
  (choose-win
   (keep (lambda (triple) (my-pair? triple me))
         triples)))

(define (my-pair? triple me)
  (and (= (appearances me triple) 2)
       (= (appearances (opponent me) triple) 0)))

(define (opponent letter)
  (if (equal? letter 'x) 'o 'x))

(define (choose-win winning-triples)
  (if (empty? winning-triples)
      #f
      (keep number? (first winning-triples)) ))

(define (opponent-can-win? triples me)
  (i-can-win? triples (opponent me)) )

(define (i-can-fork? triples me)
  (first-if-any (pivots triples me)) )

(define (first-if-any sent)
  (if (empty? sent)
      #f
      (first sent) ))

(define (pivots triples me)
  (repeated-numbers (keep (lambda (triple) (my-single? triple me))
                          triples)))
```

```
(define (my-single? triple me)
  (and (= (appearances me triple) 1)
       (= (appearances (opponent me) triple) 0)))

(define (repeated-numbers sent)
  (every first
         (keep (lambda (wd) (>= (count wd) 2))
               (sort-digits (accumulate word sent)) )))

(define (sort-digits number-word)
  (every (lambda (digit) (extract-digit digit number-word))
         '(1 2 3 4 5 6 7 8 9) ))

(define (extract-digit desired-digit wd)
  (keep (lambda (wd-digit) (equal? wd-digit desired-digit)) wd))

(define (i-can-advance? triples me)
  (best-move (keep (lambda (triple) (my-single? triple me)) triples)
             triples
             me))

(define (best-move my-triples all-triples me)
  (if (empty? my-triples)
      #f
      (best-square (first my-triples) all-triples me) ))

(define (best-square my-triple triples me)
  (best-square-helper (pivots triples (opponent me))
                      (keep number? my-triple)))

(define (best-square-helper opponent-pivots pair)
  (if (member? (first pair) opponent-pivots)
      (first pair)
      (last pair)))

(define (best-free-square triples)
  (first-choice (accumulate word triples)
                '(5 1 3 7 9 2 4 6 8)))

(define (first-choice possibilities preferences)
  (first (keep (lambda (square) (member? square possibilities))
               preferences)))
```

Exercises

10.1 The `ttt` procedure assumes that nobody has won the game yet. What happens if you invoke `ttt` with a board position in which some player has already won? Try to figure it out by looking through the program before you run it.

A complete tic-tac-toe program would need to stop when one of the two players wins. Write a predicate `already-won?` that takes a board position and a letter (`x` or `o`) as its arguments and returns `#t` if that player has already won.

10.2 The program also doesn't notice when the game has ended in a tie, that is, when all nine squares are already filled. What happens now if you ask it to move in this situation?

Write a procedure `tie-game?` that returns `#t` in this case.

10.3 A real human playing tic-tac-toe would look at a board like this:

o	x	o
o	x	x
x	o	

and notice that it's a tie, rather than moving in square `9`. Modify `tie-game?` from Exercise 10.2 to notice this situation and return `#t`.

(Can you improve the program's ability to recognize ties even further? What about boards with two free squares?)

10.4 Here are some possible changes to the rules of tic-tac-toe:

- What if you could win a game by having three squares forming an L shape in a corner, such as squares 1, 2, and 4?
- What if the diagonals didn't win?
- What if you could win by having *four* squares in a corner, such as 1, 2, 4, and 5?

Answer the following questions for each of these modifications separately: What would happen if we tried to implement the change merely by changing the quoted sentence of potential winning combinations in `find-triples`? Would the program successfully follow the rules as modified?

10.5 Modify `ttt` to play chess.