
A CGI Program

Our next example is a complete CGI program that plays tic-tac-toe. It was contributed by a student who took the Red Flags class a long time ago--in fact, she was present the very first time I ever gave the class. The code was not too bad to begin with, but a lot of it was omittable.

File-scope `my` variables

When I start to review a program, I usually start at the top. In this case, the first thing I saw was line 13:

```
13        my ($rounds, $round_temp, $squares, $page, $x, $y, $z,
        $cell, $player_move, @available_choices, $computer_move,
        @choices, $round, $winner, $player_move_pretty,
        $computer_move_pretty);
```

File-scope `my` variables are a red flag. When you declare a variable at file scope, it's available everywhere in the entire file. That means that it's effectively global. Such variables have all the usual disadvantages of **global variables**. The question you should ask yourself in each case is "does this variable really need to be available throughout the entire program?" Often, the answer is 'no'.

Sometimes the answer *is* 'yes'. In this case, it is perfectly legitimate to have `$squares` and `$page` be global. `$squares` is the variable that represents the current state of the tic-tac-toe board. Since the whole purpose of the program is to play tic-tac-toe, much of the program needs access to this state, and making `$squares` a global variable is a reasonable way to grant that access. Similarly, `$page` is legitimately global; it is the contents of the CGI data submitted by the web user, and is widely used throughout the

program.

The other variables, however, are less defensible. `@choices` and `$cell` are not used *anywhere*, so they don't belong in the program at all, much less at file scope. Most of the rest are used somewhere, but are really small-scope variables that have been erroneously declared at file scope. A good rule of thumb to follow here is **the shorter the name, the smaller the scope**. A variable named `$x` has no business living at file scope; either it should have a small scope, or we should give it a more appropriate and descriptive name. From an inspection of the places in the program where `$x` is used, we wil see that its name is perfectly reasonable, and that the right response is to fix the scope.

Here's a typical chunk of code that uses `$x`:

```
214        foreach $x(0..2) {
215           if ($squares->[0][$x] eq "?") {
216              print_cell($squares, $page, $x, "0");
217           }else {
218              print ("<td>" . $squares->[0][$x] . "</td>\n");
219           }
220        }
```

To understand this, we first need to understand the contents of `$squares`. It is simply a two-dimensional array listing the `X`'es and `O`'s on the board; if a board square is empty, it contains a `?` at that position instead. The code here is responsible for displaying the top row of the board. It lets `$x`, the column number, run from 0 to 2, and then tests to see if the board square at row 0, column `$x` is empty, and displays the appropriate symbol. `$x` is a perfectly reasonable name for this trivial loop variable; what is unreasonable is to make this trivial loop variable global to the whole program.

`$x` is used in other places in the program, but they are all unrelated to this code. This could cause a bug in the future; the program might set `$x` in once place and then use the value accidentally elsewhere. Better is simply to confine the scope of `$x` to the place where it is used, a simple change:

```
        foreach my $x(0..2) {
           if ($squares->[0][$x] eq "?") {
              print_cell($squares, $page, $x, "0");
           }else {
              print ("<td>" . $squares->[0][$x] . "</td>\n");
           }
     }
```

We should make similar changes to the other places where `$x` is used, and eliminate it from the file-scope `my` declaration.

Repeated Code

The next piece of code that jumped out at me did so because it was so visually distinctive:

```
75        $rounds = {
76                  round1 => {
77                             player => $page->param('round1_x'),
78                             computer => $page->param('round1_o')
79                            },
80                  round2 => {
```

```
 81                                player => $page->param('round2_x'),
 82                                computer => $page->param('round2_o')
 83                              },
       ...
```

We should remember from the previous chapter the `world's largest red flag`: **repeated code is a mistake**. Eliminating the repetition here turns 16 lines of code into 4:

```
for my $rn (1..5) {
  $rounds->{"round$rn"} =
    { player   => $page->param("round${rn}_x"),
      computer => $page->param("round${rn}_o"),
    }
}
```

The structure here is just as easy to see; maybe easier, because you can see instantly that each of the five hash values is constructed the same way, without having to compare manually.

The `$rounds` structure, by the way, records the history of the game up to the present. It has at most five elements because a tic-tac-toe game lasts at most five moves. Each element contains one move for X and one for Y.

Try it Both Ways

My first cut at this looked different. Here it is:

```
$rounds = { map {("round$_" =>
                    { player   => $page->param("round${_}_x"),
                      computer => $page->param("round${_}_o"),
                    }
                  )} (1..5) };
```

I tried it both ways, and decided that the `for` loop was much clearer. The `map` had too many confusing parentheses at the end, and also the puzzling `${_}_` thing in the middle. Please don't be afraid to **try it both ways**, because you may not guess the best way the first time, and it may not be clear what way is best until you see it.

As an exercise in trying it both ways, please consider which of these you think is better:

```
$page->param("round${rn}_x")

$page->param("round$rn" . "_x")
```

("Who cares?" is a legitimate answer here; another is "rename the CGI widgets appropriately and then use `$page->param("round_x_$rn")` instead.")

More Repeated Code

The next block in the program is similar to the previous one:

```
 98        $rounds = {
 99                    round1 => {
100                                player => '?',
```

```
101                                 computer => '?'
102                               },
103                     round2 => {
104                                 player => '?',
105                                 computer => '?'
106                               },
        ...
```

Since it's so similar, it's tempting to make the same kind of replacement, and use this instead:

```
for my $rn (1..5) {
  $rounds->{"round$rn"} =
    { player   => '?',
      computer => '?',
    }
}
```

That's good (16 lines become 4) but in this case we can do better. The question marks are never used anywhere; no part of the program ever examines them! So it's simpler to just leave `$rounds` entirely undefined. We can delete lines 97-119 entirely, so instead of 16 lines becoming 4, we now have 16 lines becoming *zero*.

I know lots of people come into my class skeptical about what I will do to a program to make it smaller; they worry about obfuscation and compression. But I don't think anyone will complain that it is too hard to understand that concise new code I used to replace these 16 lines. This is conciseness at its best!

Lines 74--120 have turned into this block:

```
if ($round > 0) {
    for my $rn (1..5) {
      $rounds->{"round$rn"} =
        { player   => $page->param("round${rn}_x"),
          computer => $page->param("round${rn}_o"),
        }
    }
}
```

We can do better than this. Consider what happens partway through the game, say in round 3. At that point, the `$rounds->{"round4"}` and `$rounds->{"round5"}` parts of the structure will contain hashes that are full of undefined values. Similarly, in round 1, most of the values are full of undefined values. What would happen if we got rid of the `if` test and let the loop execute even in round 0, which is the initial state of the game before anyone has placed any X'es or O's? Clearly, `$rounds` would be filled with undefined values. But as the code stands, it doesn't initialize `$rounds` at all, which means that any code that looks into `$rounds` sees that it is full of undefined values anyway. So the test is not doing anything, and the program will work just the same if we get rid of it:

```
    for my $rn (1..5) {
      $rounds->{"round$rn"} =
        { player   => $page->param("round${rn}_x"),
          computer => $page->param("round${rn}_o"),
        }
    }
```

The good advice here is **avoid special cases**. Often you have a test to check for some sort of exceptional condition, as we had the test here making sure that rounds was not 0. When you have such a test, ask

yourself what would happen if you didn’t bother with the test, and just went ahead anyway. More often than not the answer is "there would be a disastrous failure." But sometimes, as in this case, ther answer is "nothing at all," and then you can eliminate the special case.

The code of lines 74-120 has shrunk drastically, from 33 lines to 4.

More Repeated Data Structures

Since we’re in the neighborhood, let’s look at lines 44-48:

```
44      if ($round > 0) {
45        $squares = [
46              [
47               $page->param('[0][0]'),
48               $page->param('[0][1]'),
49               $page->param('[0][2]')
50              ],
             ...
```

Once again, we can replace the repeated code with a loop, in this case a nested loop:

```
            if ($round > 0) {
              for my $x (0..2) {
                for my $y (0..2) {
                  $squares->[$x][$y] = $page->param("[$x][$y]");
                }
              }
            } else ...
```

10 lines have become 3.

Avoid Special Cases

Now let’s look at the `else` half of that `if-else`:

```
62      }else {
63         $squares = [
64                     ['?', '?', '?'],
65                     ['?', '?', '?'],
77                     ['?', '?', '?']
67                    ];
68      }
```

Do we really need this? Following the Good Advice of the previous section, let’s ask what would happen if we were to eliminate the `if-else` here. Then the `for` loops would execute even in round 0, and `$squares` would be filled with undefined values instead of with question marks. Would that be a problem? Yes, as it turns out. The program no longer displays checkboxes in the empty squares, so it gives the web user no way to indicate where they want to move.

Now we have to go on a bug hunt. Where did the checkboxes go? Here’s what went wrong:

```
214     foreach $x(0..2) {
215        if ($squares->[0][$x] eq "?") {
216           print_cell($squares, $page, $x, "0");
```

```
217         }else {
218            print ("<td>" . $squares->[0][$x] . "</td>\n");
219         }
220     }
```

`print_cell()` is the subroutine that generates the checkboxes. Since the elements of `$squares` no longer contain question marks, the test on line 215 always fails, and no checkboxes are emitted.

We can fix this by rewriting the test to check for empty strings instead of question marks:

```
foreach $x(0..2) {
  if ($squares->[0][$x] eq "") {
      print_cell($squares, $page, $x, "0");
   }else {
      print ("<td>" . $squares->[0][$x] . "</td>\n");
   }
}
```

Now the code is expecting elements of `$squares` to contain either `X`, `O`, or a possibly-undefined empty string, and initializing it to contain all-`undef` is just the right thing to do.

The Good Advice here is a little more subtle than some we have seen before. It's **let `undef` be your 'special' value**. If you need a special value to indicate a special condition, like a square that has not been moved in, or a number that was not supplied, or a value that is unavailable because an error occurred, consider using `undef`.

Note that the advice says *let*, not *make*. Sometimes `undef` won't want to be the special value, and you mustn't try to force it. Here it worked all right.

More Repeated Code

In letting `undef` be the special value, we had to change `"?"` to `""` three times, once in each of the three loops like this one:

```
foreach $x(0..2) {
   if ($squares->[0][$x] eq "") {
      print_cell($squares, $page, $x, "0");
   } else {
      print ("<td>" . $squares->[0][$x] . "</td>\n");
   }
}
```

Instead of having one loop for the first row, one for the second row, and one for the third row, we'll use an outer loop to repeat the inner one three times:

```
for my $row (0..2) {
  for my $column (0..2) {
    if ($squares->[$row][$column] eq "") {
      print_cell($squares, $page, $column, $row);
    } else {
      print ("<td>" . $squares->[$row][$column] . "</td>\n");
    }
  }
  print ("</tr><tr>\n") unless $row == 2;
```

```
}
```

14 lines of code become 5.

Special Cases

This code still has a special case, namely the line that is executed `unless $row == 2`. This part of the program is printing a table; the `</tr>` tag ends a row of the table, and then the `<tr>` tag starts the next row. If the row is the third (`$row == 2`) then this line is suppressed, so no new row is started. The closing `\</tr>` tag for row 2 itself is printed by code outside the loop.

A very subtle and powerful idea in programming is that the structure of the code should mirror the structure of the output it is designed to produce. Here, though, there's no part of the code that corresponds to the printing of an entire row. The `if-else` block in the middle is fully and entirely responsible for printing a single cell in the row, but there's no analogous part of the program for an entire row. The contents of the `for my $row` loop doesn't print an entire row, since it doesn't print the `<tr>` tag that starts the row; instead, it prints part of a row, and then a bit of the next row. We can fix this like this:

```
for my $row (0..2) {
  print "<tr>";
  for my $column (0..2) {
    if ($squares->[$row][$column] eq "") {
      print_cell($squares, $page, $column, $row);
    } else {
      print ("<td>" . $squares->[$row][$column] . "</td>\n");
    }
  }
  print "</tr>\n";
}
```

Now the `for my $row` loop prints one row on each pass; to get it to print three rows, we run the loop three times. The need for the special case is eliminated. A good rule of thumb is that well-designed code can often handle special and non-special cases the same way, as here, where there's no difference between the first row of the table, the last row of the table, or the middle rows of the table.

Sidebar: Code and Data Structure Should be in Harmony

I didn't make this idea up; I got it from a guy named Michael A. Jackson. In 1975 he wrote a really brilliant book about this, called Principles of Program Design. His idea, in a nutshell, is that the control structure of a program should mirror the structure of the input it receives, and should also mirror the structure of the output it produces. If the input and output have different structures, you get a possibly tricky problem, and you need to write your program in an intermediate structure that can harmonize the input and the output structures.

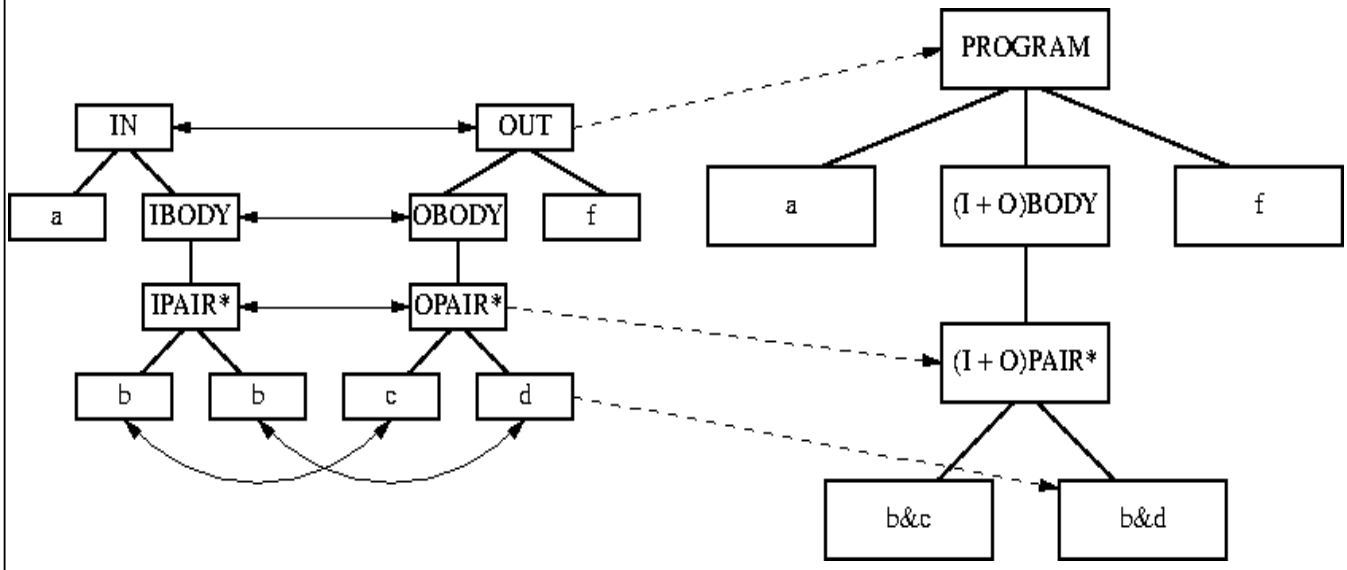

figure 2.1: `jsp`

Jackson's *JSP* method shows how to determine the correspondences between the input and output structures and how to derive the appropriate program structure from this correspondence; that's what the book is about. It's a great, great book. So why doesn't anyone read it? Because all the examples are in COBOL. It was 1975. Oh, well.

Here's a small example of a common idiom that might benefit from being harmonized with its output structure. W e want to produce a string like this:

```
['key1','key2','key3','key4','key5','key6']
```

The common idiom for this is:

```
print CLIENT "['" . join("','" => @keys) . "']", EOL;
```

The "harmonize code and data structure" advice suggests that we might do better to write the code like this:

```
print CLIENT "[", join("," => map("'$_'", @keys)), "]", EOL;
```

It's a little longer, but it might be clearer; the original code contains a trick, and the replacement code is entirely trickless. What do you think?

`print_cell()` is an Unnecessary Subroutine

The subroutine `print_cell()` is called from only one place. This is not itself a red flag; such a subroutine is often valuable for dividing the program into manageable chunks. But in this case, there is another issue to consider: `print_cell()` is not doing very much:

```
295     sub print_cell {
296        my $squares = $_[0];  ##import semi-existant array of squares
```

```
297         my $page = $_[1];     ##import html stuffs
298         my $x = $_[2];        ##import the number that the foreach is o
299         my $row = $_[3];      ##import the row we are on
300         my $cell = "[$row][$x]";  ##get square coordinates for use in n
301
302         print ("<td>");
303         print ("<input type='checkbox' name='choice' value='$cell'>\n")
304         print ("</td>");
305     }
```

Almost all of this code is structural; only the three `print` lines are actually doing anything interesting.

The purpose of `print_cell()`, as its name suggests, is to print a cell in a table. It prints the checkbox that represents an empty square in the tic-tac-toe board. This code is doing something analogous; it prints the HTML for a *non*-empty square in the tic-tac-toe board:

```
 236           print ("<td>" . $squares->[2][$x] . "</td>\n");
```

If printing nonempty squares can be inline, why can't printing empty squares also be inline? It would look like this:

```
for my $row (0..2) {
  print "<tr>";
  foreach my $col (0..2) {
    if ($squares->[$row][$col] eq "") {
      print ("<td><input type='checkbox' name='choice' value='[$row][$col]
    }else {
      print ("<td>" . $squares->[$row][$col] . "</td>\n");
    }
  }
  print "</tr>\n";
}
```

having moved `print_cell()`'s one line of interesting code up to the place where it is actually used, we can eliminate the function, and its six lines of structural code, from the program. The resule: nine lines become zero.

Now we might notice a small amount of repeated code in the block we're working on: both `print`s are responsible for printing out the `<td>` tags. We might try factoring this out:

```
for my $row (0..2) {
  print "<tr>";
  foreach my $col (0..2) {
    print "<td>"
    print $squares->[$row][$col]
       || "<input type='checkbox' name='choice' value='[$row][$col]'>";
    print "</td>"
  }
  print "</tr>\n";
}
```

Notice that this handy use of `||` is enabled by our choice of `undef` to represent an empty box.

Question marks in `print_final_table()`

The program has one more bug as a result of our changing the question marks to empty strings. Formerly, the `print_final_table()` function, which is responsible for displaying the final state of the tic-tac-toe board when the game is over, would display question marks in the empty squares; it no longer does. We'll fix this in a moment; first let's eliminate some repeated code from `print_final-table()`:

```
414     foreach $x(0..2) {              ##print first row
415        print ("<td align=center>" . $squares->[0][$x] . "</td>\n");
416     }
417     print ("</tr><tr valign=middle>\n");
  (Repeat twice more.)
```

Instead, we will use a [[for]] loop:

```
for my $row (0..2) {
  for my $col (0..2) {
    print ("<td align=center>" . $squares->[$row][$col] . "</td>\n");
  }
  print ("</tr><tr valign=middle>\n") unless $row == 2;
}
```

Eight lines become four. Now let's fix that missing-question-mark bug:

```
for my $row (0..2) {
  for my $col (0..2) {
     print ("<td align=center>"
           . ($squares->[$row][$col] || "?")
           . "</td>\n");
  }
  print ("</tr><tr valign=middle>\n") unless $row == 2;
}
```

There's a very interesting lesson lurking here, one that's somewhat more fundamental than some of the advice we've seen. It's that you should **try not to confuse internal and external representations of data**. What does this mean?

Data that is nice and convenient for people to use and understand is not necessarily in the format that is easiest for the computer to deal with, and vice versa. Since the two formats serve different purposes, it's often a mistake to try to use the same representation for both purposes. For internal use, inside the program, you want to choose a representation that is convenient for the computer. Don't represent the data internally in a pretty, people-friendly version, just because it needs to be displayed in that format. Instead, convert the internal representation to the friendly representation at the last moment, just before it is output.

In this case, the lesson is: don't represent an empty box as a question mark just because you want it to print that way. One reason why you shouldn't it because it can make it hard to change the output. For example, suppose we decided that we wanted the empty squares to display blank in the final display, instead of as question marks. We would want to change the question marks to `" "`'s.

In the original version of the program, a lot of unrelated code would have to change to support this:

```
63     $squares = [
64                 ['?', '?', '?'],
       ...
```

```
214     foreach $x(0..2) {
215        if ($squares->[0][$x] eq "?") {
        ...
```

None of this has any obvious connection with the output formatting.

In the modified version, only one line of code must change; it's the line of code in `print_final_table()` that actually emits the question marks:

```
for my $row (0..2) {
  for my $col (0..2) {
     print ("<td align=center>"
           . ($squares->[$row][$col] || " ")
           . "</td>\n");
  }
  print ("</tr><tr valign=middle>\n") unless $row == 2;
}
```

`print_final_table()`

At this point I got to wondering why there was a separate `print_final_table` function at all. A lot of its code appears to be repeated in the main program, where it is responsible for printing non-final tables. It would be nice to merge all this code into a single function. We will do this, and call the merged version `print_table()`.

The two code sections we are merging are not identical. For example, the final table has question marks in place of checkboxes in the empty squares. So we'll give `print_table()` an extra parameter to tell it whether we want it to produce final or non-final tables. First we'll fix this:

```
395     my $squares = $_[0];
396     my $page = $_[1];
397     my $winner = $_[2];
398     my $round = $_[3];
399     my $rounds = $_[4];
```

Five lines become one:

```
my ($squares, $page, $winner, $round, $rounds, $final) = @_;
```

We've added the extra parameter at the end here.

The next thing I did was to extract the two sections of code into separate files and use the `diff` tool on them; this prints a display of all the differences between the two sections. Code sections that are the same can be merged. If there's code in one version but not in the other, it can be sequestered into an `if` block that is predicated on the value of the `$final` argument. For example, this:

```
440        open (MAIL, "| /usr/sbin/sendmail -t");
441        print MAIL "To: author\@example.com\n";
        ...
444        close MAIL;
```

becomes this:

```
    if ($final) {
```

```
        open (MAIL, "| /usr/sbin/sendmail -t");
        print MAIL "To: author\@example.com\n";
    ...
        close MAIL;
}
```

This was a fairly substantial change, so I wanted to be sure I didn't break the program. I saved a complete set of program outputs before I started making the changes, and then I compared them afterward to make sure there were no output errors.

Sidebar: On Tools
When you program, you're using a computer, and it's important to take advantage of its strengths, which are automation and attention to detail. use tools like `diff` to quickly and thoroughly locate all the differences between two similar pieces of code. If you're building web applications, use the `GET` tool that comes with Perl's `LWP` module to download and save program outputs in files; then use `diff` to compare the "before" and "after" versions of the outputs. Automate testing. The Good Advice to remember here is to **use the computer to automate programming tasks**.

Calls to `print_final_table(...)` now change into calls to `print_table(..., "final")`. The `"final"` argument is just a flag, so any true value will do; I've chosen to use the string `"final"` here instead of the obvious `1`. I'm not certain this is a good idea. On the one hand, one might argue that this is "self-documenting"; on the other hand, perhaps it's excessively cute. (What do you think?)

Then the 14 lines of code in the main program at lines 189--251 that prints the non-final tables becomes simply:

```
print_table($squares, $page, $winner, $round, $rounds);
```

The gross earnings is the elimination of the 14 lines of code; the expenses were adding four more lines to `print_table()` and the one line in the main program to call it.

If it seems a little bizarre that lines 189--251 contained only 14 lines of code, remember that that is after I removed dumplicate code; originally there were 22 lines there. Even so, the code-to-space ratio is about 3/1; perhaps that is excessive.

In the course of merging the two parts of the program the print tables, I also detected and fixed a bug. `print_final_table()` had:

```
404         print $page->startform(action=>'tic_tac.cgi',
405                                method=>'POST'
406                               );
```

This code generates the following HTML output:

```
<form method="action" action="tic_tac.cgi" enctype="method" POST>
```

This is totally garbled. It was undetected because it appeared only on the final page, which has no `submit` button because the game is over. Less code overall means less unexercised code, which means fewer secret, latent bugs.

The gross length of the program is already down by 25%.

`evaluate_board()`

Glancing over the program, the `evaluate_board()` function looks like a nice, fat target. It is the function that is responsible for deciding if anyone has actually won the game. It has two big blocks like this one:

```
360         foreach $x (0..2) {
361             if (
362                 ($squares->[$x][0] eq $squares->[$x][1]) and
363                 ($squares->[$x][1] eq $squares->[$x][2])
364                ) {
365                 $winner = $squares->[$x][0];
366                 return $winner;
367             }
368         }
          ...
```

This checks to see if anyone has gotten three in a row horizontally; the other block checks the same thing for the verticals. There is also a special case to handle the diagonals. Looking at this, I scratches my head and said "There are only 8 ways to win in tic-tac-toe! How can an analysis of only 8 cases require 21 lines of code?"

Often, complicated logic can be replaced with a table of some sort. There are two ways to do this. One is to have the program can generate the table itself at run time. The code to generate the table may be complicated, but it is unlikely to be more complicated than the code it is replacing. But often, if the table is small, you can generate the table in advance and hardwire it into the program. In this case the table will only contain 8 items, so that is what we will do:

```
my @table = (
  [ 0,0 , 0,1 , 0,2 ],
  [ 1,0 , 1,1 , 1,2 ],
  [ 2,0 , 2,1 , 2,2 ],
  [ 0,0 , 1,0 , 2,0 ],
  [ 0,1 , 1,1 , 2,1 ],
  [ 0,2 , 1,2 , 2,2 ],
  [ 0,0 , 1,1 , 2,2 ],
  [ 0,2 , 1,1 , 2,0 ],
);

for my $win (@table) {
  my ($x1, $y1, $x2, $y2, $x3, $y3) = @$win;
  if ($board->[$x1][$y1] eq $board->[$x2][$y2]
  &&  $board->[$x1][$y1] eq $board->[$x3][$y3]) {
    return $board->[$x1][$y1];
  }
}
```

The result is that the 21 lines of code have been replaced by five lines, plus the table. (I’m not sure whether the table should count as zero lines, two lines, nine lines, or something else. Regardless, it is much simpler.)

Incidentally, there is a bug in this version and in the original program. The `return` line should actually read:

```
return $board->[$x1][$y1] if $board->[$x1][$y1];
```

Without this change, the function incorrectly evaluates this position:

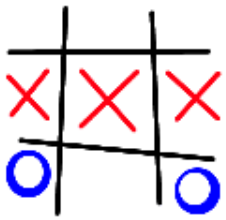

figure 2.2: `ttt-bug-sm`

Sidebar: Let’s not be doctrinaire

figure 2.3: `littleredbk-med`

Let’s take time to remember something very important: a red flag is a warning that something *might* be wrong, that you should stop and *ask* yourself if you have an opportunity to improve the code. The answer is not always "yes"; not every red flag requires that something be fixed. The Good Advice is only advice; it is not commandments, and it is not always going to be correct.

In the rewritten version of `evaluate_board()`, I have:

```
my ($x1, $y1,
    $x2, $y2,
    $x3, $y3) = @$win;
```

I said back in the introduction that similarly-named scalar variables are a red flag, and I cited `$x1`, `$x2`, `$x3` specifically as examples. I said that such variables should almost always be replaced by an array. Well? Have I ignored my own advice here?

No, I did not ignore it. But I decided not to take it. I decided to **try it both ways**, and I found that the array was a big loser, because it led to this unreadable mess:

```
if ($board->[$x[0]][$y[0]] eq $board->[$x[1]][$y[1]]
&&  $board->[$x[0]][$y[0]] eq $board->[$x[2]][$y[2]]) {
  return $board->[$x[0]][$y[0]];
```

Families of scalar variables should almost always be replaced by an array, but *almost* always is not the same as *always* always. Please try to remember this. If you remember only one piece of Good Advice from this book, please remember that **this advice is not a substitute for independent thought**.

`make_move_pretty()` and uniform data representations

`make_move_pretty()` has two branches:

```
451         if ($move =~ /:/) {
452            $move =~ s/^(\d):(\d)/$1$2/;
453         }
454         else {
455            $move =~ s/^\[(\d)\]\[(\d)\]/$1$2/;
456         }
```

It's trying to take apiece of data that represents a move and put it into a canonical form. It needs two branches because there are two unrelated presentations of moves. A move into the upper right square might be represented as either of:

```
0:2
[0][2]
```

Actually the form returned by `make_move_pretty()`, `02` is a third representation. Why not simply use `02` everywhere? For example, this code:

```
287             $z = "[$x][$y]";
288             if ($move eq $z) {
```

becomes this:

```
                if ($move eq "$x$y") {
```

If we did this, we could eliminate five lines of code from `make_move_pretty()`. The good advice here is **don't multiply representations unnecessarily**.

`get_computer_choice()`

This subroutine is the kernel of the sophisticated aritifical intelligence that allows this program to play tic-tac-toe. The algorithm it uses to generate its move is: select an empty square at random.

```
195     sub get_computer_choice {
196        my @available_choices = @_;
197        my $length = @available_choices;
198        my $number = int(rand() * ($length - 1));
199        my $choice = $available_choices[$number];
200        return $choice;  ## this will be a coordinate, in the form of $x$
201     }
```

Here we have many examples of **variable use immediately follows assignment**. `@available_choices` is just a copy of the arguments, and there's no reason to copy them from the array they're in already into a new array, so:

```
sub get_computer_choice {
   my $length = @_;
   my $number = int(rand() * ($length - 1));
   my $choice = $_[$number];
   return $choice;  ## this will be a coordinate, in the form of $x$
}
```

Now we may as well eliminate `$length`:

```
sub get_computer_choice {
   my $number = int(rand() * (@_ - 1));
```

```
    return $_[$number];
}
```

The presence of `@_ - 1` is an oddity; it almost always indicates an off-by-one error, since it asks for a number that is one *less* than the number of array elements. In this program, it *is* an error; it should have been:

```
sub get_computer_choice {
    my $number = int(rand(@_));
    return $_[$number];
}
```

At this point `$number` is another example of **variable use immediately follows assignment**, so we try eliminating it in the same way:

```
sub get_computer_choice {
    $_[int rand @_];
}
```

Now we have a one-line function, so we have to wonder whether it is necessary at all; inlining it into the one place from which it is called produces:

```
$computer_move = $available_choices[rand @available_choices];
```

and another six lines of code are gone.

The program at this point is displayed in `ttt2|appendix`. It's already substantially smaller than it was: 231 lines of code have become 133.

There are two major actions we can take to improve the program from here. One is to use template files instead of a lot of HTML-generating code; this will also move all the HTML out of the program, making it easier to follow the logic. For this, see Appendix ???. The other is to use *higher-order functions* to encapsulate common patterns of code. For example, there are several places in the program that perform some action for each square of the board; a higher-order function could be built that would get an action as its argument, and perform that action for each square of the board, allowing the repeated constrol structures to be eliminated. For examples of this, see Appendix ???. With these changes, we can eliminated another 11 lines of code.
