

This file is copyright © 2006 Mark Jason Dominus.
Unauthorized distribution in any medium is
absolutely forbidden.

1. Tests
 2. The Preprocessor
 3. Special Cases
 4. Flag Variables
 5. Punctuation
 6. The Condition that Ate Michigan
 7. Extraction
-

Bio-informatics

The author of this program posted it to Usenet [1] in August 2001 with the subject `can this perl script be more elegant/shorter ?`. The original code is Program ???, and a typical input to the program is in Program ???. It jumped out at me because it's a superb example of **families of variable names**. Glancing over the program, the first thing you see is lines 26--49:

```
26     SWITCH: {
27         $chain_number==1 && do {
28             @chain1 = split (//,$chain_input);
29             $scan_chain_length1 = @chain1; };
30         $chain_number==2 && do {
31             @chain2 = split (//,$chain_input);
32             $scan_chain_length2 = @chain2; };
33         $chain_number==3 && do {
34             ...
35             ...
45         $chain_number==7 && do {
46             @chain7 = split (//,$chain_input);
47             $scan_chain_length7 = @chain7;
48             &printout;
49         };
```

As usual, we will replace the family with an array. This time it's a family of arrays, so we will try replacing arrays `@chain1` through `@chain7` with `@chain` a single array of arrays, and see what happens. Similarly, `$scan_chain_length1` through `$scan_chain_length7` become a single array, `@scan_chain_length`:

```
$chain[$chain_number] = [split //, $chain_input];
$scan_chain_length[$chain_number] = @{$chain[$chain_number]};
&printout if $chain_number == 7;
```

23 lines have become 3. One nice thing about this transformation is that it's completely mechanical: we can do it without understanding anything at all about what the code is really doing.

Closer examination of the code later reveals that we can trim this a little more, because of the seven

`$scan_chain_lengthI<n>` variables, six were never used; the only one that is mentioned again is `$scan_chain_length3`. So we can get rid of the entire `@scan_chain_length` array. Later on, when `$scan_chain_length3` is used, we'll just write `@{$chain[3]}` explicitly. The drawback of this is that the expression is slightly more complicated and there's a little more punctuation, but the benefit is one fewer variable for the maintenance programmer to remember the purpose of. `@{$chain[3]}` speaks for itself. With this change, the 23 lines have become 2:

```
$chain[$chain_number] = [split //, $chain_input];
&printout if $chain_number == 7;
```

At this point, the main loop of the program has shrunk enough so that we can see the whole thing at once, which is almost always a good thing:

```
while (<INFILE2>) {
  chomp;
  $ct_scanout = 1 if (/apply\s*"grp[0-9]_unload"/);
  $chain_test=1 if (/CHAIN_TEST/);

  if (( /\t*chain\s+"chain([0-9])\"/) && ($chain_test)) {
    $chain_number = $1;
    &cleanup;
    $chain_input = (split /=/, $_)[1];
    $chain_input =~ tr/"//d;
    $chain[$chain_number] = [split //, $chain_input];
    &printout if $chain_number == 7;
  }
}
#end of While statement
```

One immediate benefit we reap from this contraction of the main loop is that we can get rid of the **#end of while statement comment**. Such comments are red flags; they suggest immediately that your blocks are too long or too complicated or both. [2]

Sidebar about indenting editors?

The next most prominent part of the program is now:

```
56             $chain1[$i] =~ s/0/L/g; $chain1[$i] =~ s/1/H/g;
57             $chain2[$i] =~ s/0/L/g; $chain2[$i] =~ s/1/H/g;
58             $chain3[$i] =~ s/0/L/g; $chain3[$i] =~ s/1/H/g;
59             $chain4[$i] =~ s/0/L/g; $chain4[$i] =~ s/1/H/g;
60             $chain5[$i] =~ s/0/L/g; $chain5[$i] =~ s/1/H/g;
61             $chain6[$i] =~ s/0/L/g; $chain6[$i] =~ s/1/H/g;
62             $chain7[$i] =~ s/0/L/g; $chain7[$i] =~ s/1/H/g;
63             print OUTFILE2 "\n(ct_so
64             $chain1[$i]$chain2[$i]$chain3[$i]$chain4[$i]$chain5[$i]$chain6[$i]$c
65             )";
```

Since we've replaced the family of seven arrays with a single array, we can replace the repeated code with a single `for` loop. We can also clean up the long `print` expression on line 64:

```
for (@chain) {
  $_->[$i] =~ s/0/L/g;
  $_->[$i] =~ s/1/H/g;
}
my @chars = map $_->[$i], @chain;
print OUTFILE2 "\n(ct_so\n", join("", @chars), "\n");
```

15 lines have become 5.

In the past I've suggested replacing `\n` sequences with literal newlines for clarity; here I did the opposite. But try it both ways and see which one you prefer.

We're not finished cleaning up the mess of chains. Lines 71-77 have another batch:

```
71             $chain1[$i] =~ s/X/0/g; $chain2[$i] =~ s/X/0/g;
72             $chain3[$i] =~ s/X/0/g; $chain4[$i] =~ s/X/0/g;
73             $chain5[$i] =~ s/X/0/g; $chain6[$i] =~ s/X/0/g;
74             $chain7[$i] =~ s/X/0/g;
75             print OUTFILE2 "\n(ct_si $ct_si{tdi}
76     $chain1[$i]$chain2[$i]$chain3[$i]$chain4[$i]$chain5[$i]$chain6[$i]$c
77             )";
```

8 lines become 4:

```
for (@chain) { $_->[$i] =~ s/X/0/g }
my @chars = map $_->[$i], @chain;
print OUTFILE2 "\n(ct_si $ct_si{tdi}\n", join("", @chars), "\n )";
```

This version of the program is Program ???.

Tests

In case you hadn't noticed by now, this book is about maintenance programming. The most most important rule for maintenance programmers is:

First, do no harm.

That's all well and good, but how do you know if you're doing harm or not? There is only one answer: you must have a test suite.

Sidebar: Testing

I joined the cult of the automated test while I was writing the very complicated `Tie::File` module. I had little confidence that I would be able to make it work properly, but the tests were a smashing success. I had fewer bugs, and the ones I did have did not recur. But even more important, I felt happy with the code and confident in its quality.

I have more fun programming now. It used to be that whenever I'd make a change I'd worry that maybe I had broken something. I didn't notice how uneasy and unhappy this made me feel. With the automatic test suite, I feel a lot better all the time.

If testing makes programmers happy, why don't more of them do it? I think it's because a lot of the most visible proponents of automated testing are a bunch of freaks. They make testing into a lot of work. They lay a big guilt trip on you if you're not doing it the "right" way, or if you haven't written "enough" tests. I wish these people would go away.



figure 9.1: Stewart

I believe, very strongly:

Writing tests should be quick and easy

If it's not quick and easy, you are not doing it correctly

You don't have to write a thousand tests. The benefit of tests scales very nicely. A thousand tests is good. A hundred tests is also good. So is ten tests. Even one test is better than no tests at all. They say that a journey of a thousand miles begins with a single step, so don't start out saying "Oh, I have to write a test for every single feature of my program." If you do that, you'll be discouraged, and you won't want to write any at all. Start by saying "I'll write one test." Then write it. Then if you like it, maybe write another one.

You don't have to write all the tests before you write any code. You don't have to write any. If you feel like writing some before you start, go ahead. If not, that's fine too.

A good time to write a test is when you find or fix a bug. You can add a test for that bug. Then you can be sure that that bug will never come back.

Another good time is when you add a new feature. You can add a test or two for the feature; nothing fancy.

If you do this, soon you *will* have a thousand tests. You'll be surprised at how quickly they can pile up. (Your boss might be surprised too.) But if they don't pile up, don't stress out about it. Soon you'll see how much benefit you can get from just a few simple tests; then you will *want* to write more.

This program is a fine demonstration that test suites do not have to be complicated or sophisticated, and that testing does not have to be onerous or time-consuming. Here's what I used to test this program.

First, I made a record of what the output was supposed to look like:

```
% perl extract1.pl
% mv OUT.out SAMPLE-OUTPUT
```

The program likes to write its output to `OUT.out`, so I moved it out of the way. Then I wrote the following test script, which I called `test.pl`:

```
2 #!/usr/bin/perl
```

```
my $prog = shift || 'extract3.pl';
chdir("/home/mjd/TPC/2002/Flags/extract") or die $!;
system("perl $prog");
system("cmp -s SAMPLE-OUTPUT OUT.out");
print $? == 0 ? "ok\n" : "not ok\n";
```

After each change to the program, I ran `test.pl`. If it said `ok`, I went on to the next change. If it said `not ok`, I would examine the `OUT.out` file to see what was wrong, perhaps using `diff` to compare it with the baseline `SAMPLE-OUTPUT` file.

That's all; just five lines of code. That is enough.

Testing, like so many things, obeys the 90-10 rule of effort. You can put a huge amount of trouble and time into writing exhaustive tests. Or you can put in a little bit of trouble and get a tremendous benefit cheap. Don't let the testing fanatics scare you away from the cheap, easy gains you can get from simple test programs like this one.

The Preprocessor

Our next red flag is **making two passes** over the input. [3] Like other red flags, it is not Wrong Wrong Wrong. It's just a sign that you should stop and reflect a little, and ask yourself "Do I really need two passes here? Or could I do both transformations in a single pass? What would happen if I did?"



figure 9.2: thinker

Often the result is simpler and smaller code, and usually more efficient code as well.

The example program has an entire preprocessing step:

```
3     $input1 = 'chaintest.scan';
4     $output= "OUT";
5     $output2= "OUT.out";
6
7     open (INFILE,$input1)||die "cannot open $input1";
8     open(OUTFILE,">$output")||die "canoot\n";
9     open(OUTFILE2,">$output2")||die "canoot\n";
10
11     #####
12     [C[&preprocess;]C]
13     open (INFILE2,$output)||die "cannot open $input1";
```

The original input file is copied through `&preprocess` into the output file `OUT`, which is then opened for input. The main program then transforms `OUT` to `OUT.out`, the final output. What is this preprocessing step doing? Let's look at `&preprocess`:

```
sub preprocess{
    while (<INFILE>){
        chomp;
```

```

        s/$/;/g if (/apply/);
        print OUTFILE ("$_\n");
        last if ( /^SCAN_CELLS/);
    }
    close OUTFILE;
}

```

This subroutine does exactly two things: It appends a semicolon to the end of every line that matches `apply`, and it stops copying when it sees the label `SCAN_CELLS`. When I saw this, I said "Oy gevalt! For this, we needed a subroutine and an auxiliary file?" The main routine is already passing over the input and making lots more transformations than that; if we merge these two little bits of code into the main loop, they will hardly make it more complicated than it is. Let's see what it looks like.

Well, actually, it turned out to be a little trickier than I thought. The addition of semicolons to the `apply` lines looked innocent enough, but then later on, we have:

```

14     $/=";";
15     $ct_scanout = 0;
16     while (<INFILE2>) {
29     ... }

```

Those semicolons were treated as record separators! And when I examined the sample input, it turned out that the `apply` lines *already* had semicolons on them! Here's a typical example:

```

apply "grp1_load" 0 =;

```

So I was really puzzled. What's going on here? Why is line 90 (in the preprocessor) adding semicolons to lines that have them already?

```

90     s/$/;/g if (/apply/);

```

This is not just a rhetorical question. It's a maintenance programmer's constant dilemma: "what the heck is that doing there?"

There are at least three possibilities:

1. The addition of semicolons might be entirely superfluous, even incorrect, and so the right thing to do is to get rid of it entirely.
2. It might be there because the original author knows that not all the inputs will always have the `apply` lines properly terminated with semicolons, and the sample input is unusually well-formed. In this case the right thing to do is to leave it in.
3. It might be there even though the original author expects all the `apply` lines to be properly terminated, on the theory that you never know what might happen. In this case the right thing might be to go either way, depending on whether the original programmer was being prudent or paranoid.

When you don't know the answer to a question like this, and the original author or some other authority is unavailable, there's often a middle way that's safe *and* clean. In this case, the middle way is to assume that the semicolons will always be present, but to have the program raise a fatal error if they're missing. Then if our assumption about the semicolons turns out to be wrong, the user will find out about it:

```
die "Missing ';' in 'apply' line" if /apply[^\;]*\n/;
```

The rest of the preprocessor just chops off the input file when it sees `SCAN_CELLS`; we can emulate it by inserting

```
last if /^SCAN_CELLS/;
```

into the main loop. Having done this, we can eliminate the `preprocess()` function, the `INFILE2` and `OUTFILE2` filehandles, and the `$input2` and `$output2` variables, for a net gain of 10 lines gone. While we're in the neighborhood of these, let's fix the error messages of lines 7--9:

```
7   open (INFILE,$input1)||die "cannot open $input1";
8   open(OUTFILE,">$output")||die "cannot\n";
9   open(OUTFILE2,">$output2")||die "cannot\n";
```

Better is

```
open (INFILE, "$input") || die "cannot open $input: $!";
open (OUTFILE, ">$output") || die "cannot open $output: $!";
```

because **error messages should describe the cause of the problem**. [4]

Special Cases

The next thing that worries me is the special case in line 27:

```
27   &printout if $chain_number == 7;
```

The program reads and processes seven chains, and after the seventh, it emits its output. But are there always exactly seven chains? Will there always be exactly seven chains? If some day there are more, the program will still emit its output after the seventh, ignoring the following ones. If there are fewer, it won't output at all.

Numbers like 7 in your program are red flags, you should try to avoid **arbitrary numeric constants** in your code.

Sidebar: Numeric Constants

Sometimes arbitrary numeric constants are warranted. For example, sometimes they're truly immutable constants imposed on us from the outside world, as in:

```
my $ltime = localtime($d * 86400); # Number of seconds in a day
```

Here the 86400 is certainly never going to change. And sometimes they're entirely arbitrary, as in

```
print STDERR "Analyzed $i words.\n"
if $DEBUG && $i++ % 1000 == 0;
```

in which case the value doesn't matter. But even when legitimate, they should almost always be accompanied by some sort of comment.

Sometimes even innocuous-looking constants can be eliminated. For example, the 11 here is not too bad:

```
# 12 months in a year
%m2n = map {$mon[$_] => $_} (0..11);
```

but perhaps it would be better to write this instead:

```
%m2n = map {$mon[$_] => $_} (0..$#mon);
```

not, of course, because the number of months in a year might change, but to save the need to explain the 11. What do you think?

In this program, the special case for chain 7 is unnecessary, because the `apply` sections that contain the chains are terminated by special `end;` markers! I wrote some code to take advantage of this, which is in Program ???, but then I decided it was probably too much effort to expend in the pursuit of something that might be entirely unnecessary---**sufficient unto the day is the evil thereof**, after all.

Still, however, we can take the middle way, and put in some checks to make sure that the input contains exactly seven chains:

```
ADD DEMO CODE HERE
```

Flag Variables

The main loop now has this form:

```
while (<INFILE>) {
    ...
    $chain_test=1 if (/CHAIN_TEST/);
    if (/apply/ && ($chain_test)) {
        ...
    }
}
```

This looks at first like an instance of **variable use immediately follows assignment**, but it's not. That is only a red flag when the variable is used only once after it is assigned. Here, however, `$chain_test` is checked repeatedly after it is set, because it is inside a `while` loop.

It's still valuable to avoid such flag variables when possible, however, because the state of the flag is just one more thing that the maintainer must remember while debugging the program. Often, there is a

way to write the code that is just as simple with no flag variable, as there is here. Since the purpose of the flag is to skip *all* the data up until the CHAIN_TEST marker is seen, we can get a simpler program by expressing that more straightforwardly. We'll have a second loop that scans the data until the CHAIN_TEST marker is seen, and start the main loop only afterward:

```
while (<INFILE>) { last if /CHAIN_TEST/ }

while (<INFILE>) {
    ...
    if (/apply/) {
        ...
    }
}
```

Unfortunately, this isn't *quite* correct. The input contains CHAIN_TEST, which is a start code, and SCAN_CELLS, which is an end code. But what if SCAN_CELLS *preceded* CHAIN_TEST? The original program would have preprocessed away the start code, and then produced no output at all. But in our revised program, the first loop gobbles up SCAN_CELLS, ignoring it; it was supposed to exit. To fix this, we will need to change the upper loop to:

```
while (<INFILE>) {
    exit if /^SCAN_CELLS/;
    last if /CHAIN_TEST/;
}
```

Now the SCAN_CELLS test appears twice, violating the Cardinal Rule of Computer Programming, which is **do not repeat code**. Was eliminating the flag worth the code? We should **try it both ways** and see:

The new version:

```
while (<INFILE>) {
    exit if ( /^SCAN_CELLS/);
    last if /CHAIN_TEST/;
}
while (<INFILE>) {
    chomp;
    last if ( /^SCAN_CELLS/);
    if (/apply/) {
```

The old version:

```
while (<INFILE>) {
    chomp;
    last if ( /^SCAN_CELLS/);
    $chain_test=1 if (/CHAIN_TEST/);

    if (/apply/ && ($chain_test)) {
```

Which do you think is better?

I eventually decided that I liked it better the way it was, so we'll leave the \$chain_test flag in. But let's rename it to \$seen_CHAIN_TEST, which better describes its function.

Punctuation

We now have:

```
if (/apply/ && ($seen_CHAIN_TEST)) {
```

and the original code was:

```
21 if (( /\t*chain\s+\\"chain([0-9])\\"/) && ($chain_test)){
```

Let's **avoid excess punctuation**. At the very least, we can get rid of the superstitious parentheses. (See Section ???.) Also, the backslashes before the " marks are unnecessary; a quick run of `perl -nle 'print if /"\".*"/'` will confirm this.

Similarly, these:

```
last if ( /^SCAN_CELLS/);
$seen_CHAIN_TEST=1 if (/CHAIN_TEST/);
$ct_scanout = 1 if (/apply\s*\\"grp[0-9]_unload\\"/);
```

should become:

```
last if /^SCAN_CELLS/;
$seen_CHAIN_TEST=1 if /CHAIN_TEST/;
$ct_scanout=1 if /apply\s*"grp[0-9]_unload"/;
```

The Condition that Ate Michigan

Here's what the main loop looks like now:

```
while (<INFILE>) {
  chomp;
  last if /^SCAN_CELLS/;
  $seen_CHAIN_TEST=1 if /CHAIN_TEST/;

  if (/apply/ && $seen_CHAIN_TEST) {
    $ct_scanout = 1 if /apply\s*"grp[0-9]_unload"/;

    next unless /\t*chain\s+"chain([0-9])"/;
    $chain_number = $1;
    &cleanup;
    $chain_input = (split /=/, $_)[1];
    $chain_input =~ tr/"//d;
    $chain[$chain_number] = [split //, $chain_input];
    &printout if $chain_number == 7;
  }
}
```

That's not too bad, but the `if` condition has eaten up most of the rest of the code. It's not exactly Michigan; this is only a mild example of the Condition that ate Michigan. [5] The usual prophylaxis is to use `next`:

```
while (<INFILE>) {
  chomp;
  last if /^SCAN_CELLS/;
  $seen_CHAIN_TEST=1 if /CHAIN_TEST/;
```

```
next unless /apply/ && $seen_CHAIN_TEST;

$ct_scanout = 1 if /apply\s*"grp[0-9]_unload"/;

...
```

Extraction

Now let's tackle the rest of that block:

```
next unless /\t*chain\s+"chain([0-9])"/;
$chain_number = $1;
&cleanup;
$chain_input = (split /=/, $_)[1];
$chain_input =~ tr/"//d;
$chain[$chain_number] = [split //, $chain_input];
```

After returning from `cleanup()`, the data looks like this:

```
chain"chain1"="00110....11001"
```

where I have omitted about 500 0's and 1's in the middle there. Then the main-loop code does some extra work to extract the `00110....11001` part. Why doesn't `cleanup()` finish cleaning up?

The Good Advice here is to try to `cluster functionality`^[G]. This means that related things should be kept together. If you're going to move the cleanup code into a subroutine named `cleanup()`, you should move *all* of it.

Every module should do one thing well.

-- Brian Kernighan

[1] <21724be2.0108301341.149624c1@posting.google.com>

[2] Or sometimes that they are poorly indented.

[3] Making seven passes is also a red flag!

[4] In my classes I also make a joke about the word `cannot` in the messages, showing a picture of Donald E. Knuth.

[5] Perhaps we might say that it is only snacking on Delaware today.