

```

my @ftree = map {
    # add to parent's Child list ( $found );
    s/\s+$/;
    my $depth = if ( $node->{Parent} ) {
        # Start at one (though rather than 0
        $parent_stack[ $depth ] = $node;
        while ( $found->{Parent} ) {
            # create node structure

```

# Proving Tests

by Andy Armstrong  
andy@hexten.net



**T**est::Harness is responsible for running test scripts, analysing their output and reporting success or failure. When I type `make test` (or `./Build test`) for a module, Test::Harness is usually used to run the tests (not all modules use Test::Harness but the majority do).

To start exploring some of the features of Test::Harness I need to switch from `make test` to the `prove` command (which ships with Test::Harness). For the following examples I'll also need a recent version of Test::Harness installed; 3.14 is current as I write this.

For the examples, I'm going to assume that I'm working with a "normal" Perl module distribution. Specifically I'll assume that typing `make` or `./Build` causes the built, ready-to-install module code to be available below `./lib/lib` and `./lib/arch` and that there's a directory called `t` that contains my tests. Test::Harness isn't hardwired to that configuration but it saves me from explaining which files live where for each example.

Back to `prove`; like `make test` it runs a test suite—but it provides far more control over which tests are executed, in what order and how their results are reported. Typically `make test` runs all the test scripts below the `t` directory. To do the same thing with `prove` I type:

```
prove -rb t
```

The switches here are `-r` to recurse into any directories below `t` and `-b` which adds `./lib/lib` and `./lib/arch` to Perl's include path so that the tests can find the code they will be testing. If I'm testing a module of which an earlier version is already installed I need to be careful about the include path to make sure I'm not running my tests against the installed version rather than the new one that I'm working on.

Unlike `make test`, typing `prove` doesn't automatically rebuild my module. If I forget to `make` before `prove` I will be testing against older versions of those files—which inevitably leads to confusion. I either get into the habit of typing

```
make && prove -rb t
```

or – if I have no XS code that needs to be built I use the modules below `lib` instead

```
prove -Ilib -r t
```

So far I've shown nothing that make `test` doesn't do. I'll fix that.

## ■ Saved State-----

If I have failing tests in a test suite that consists of more than a handful of scripts and takes more than a few seconds to run, it rapidly becomes tedious to run the whole test suite repeatedly as I track down problems.

I can tell `prove` just to run the tests that are failing by specifying only the test files that I want to run:

```
prove -bt/this_fails.t t/so_does_this.t
```

That speeds things up but I have to make a note of which tests are failing and make sure that I run those tests. Instead I can use `prove`'s `--state` switch and have it keep track of failing tests for me. First I do a complete run of the test suite and tell `prove` to save the results:

```
prove -rb --state=save t
```

That stores a machine-readable summary of the test run in a file called `.prove` in the current directory. If I have failures I can then run just the failing scripts by telling `prove` to only run those that had failures:

```
prove -b --state=failed
```

I can also tell `prove` to save the results again so that it updates its idea of which tests failed:

```
prove -b --state=failed,save
```

As soon as one of my failing tests passes, `prove` removes it from the list of failed tests. Eventually I fix them all and `prove` can find no failing tests to run:

```
Files=0, Tests=0,  0 wallclock secs ( 0.00
usr + 0.00 sys = 0.00 CPU)
Result: NOTESTS
```

**Find the *lib* directory and include it in @INC:**

```
perl -Mlib program.pl
```