NAME

prove - Run tests through a TAP harness.

USAGE

```
prove [options] [files or directories]
```

OPTIONS

Boolean options:

```
-v,  --verbose         Print all test lines.
-l,  --lib             Add 'lib' to the path for your tests (-Ilib).
-b,  --blib            Add 'blib/lib' and 'blib/arch' to the path for
                       your tests
-s,  --shuffle         Run the tests in random order.
-c,  --color           Colored test output (default).
     --nocolor         Do not color test output.
     --count           Show the X/Y test count when not verbose
                       (default)
     --nocount         Disable the X/Y test count.
-D   --dry             Dry run. Show test that would have run.
-f,  --failures        Show failed tests.
-o,  --comments        Show comments.
     --ignore-exit     Ignore exit status from test scripts.
-m,  --merge           Merge test scripts' STDERR with their STDOUT.
-r,  --recurse         Recursively descend into directories.
     --reverse         Run the tests in reverse order.
-q,  --quiet           Suppress some test output while running tests.
-Q,  --QUIET           Only print summary results.
-p,  --parse           Show full list of TAP parse errors, if any.
     --directives      Only show results with TODO or SKIP directives.
     --timer           Print elapsed time after each test.
     --trap            Trap Ctrl-C and print summary on interrupt.
     --normalize       Normalize TAP output in verbose output
-T                     Enable tainting checks.
-t                     Enable tainting warnings.
-W                     Enable fatal warnings.
-w                     Enable warnings.
-h,  --help            Display this help
-?,                    Display this help
-H,  --man             Longer manpage for prove
     --norc            Don't process default .proverc
```

Options that take arguments:

```
-I                     Library paths to include.
-P                     Load plugin (searches App::Prove::Plugin::*.)
-M                     Load a module.
-e,  --exec            Interpreter to run the tests ('' for compiled
                       tests.)
     --ext             Set the extension for tests (default '.t')
     --harness         Define test harness to use.  See TAP::Harness.
     --formatter       Result formatter to use. See FORMATTERS.
     --source          Load and/or configure a SourceHandler. See
                       SOURCE HANDLERS.
-a,  --archive out.tgz Store the resulting TAP in an archive file.
```

```
    -j,  --jobs N          Run N test jobs in parallel (try 9.)
         --state=opts      Control prove's persistent state.
         --rc=rcfile       Process options from rcfile
         --rules           Rules for parallel vs sequential processing.
```

NOTES

.proverc

If *~/.proverc* or *./.proverc* exist they will be read and any options they contain processed before the command line options. Options in *.proverc* are specified in the same way as command line options:

```
# .proverc
--state=hot,fast,save
-j9
```

Additional option files may be specified with the `--rc` option. Default option file processing is disabled by the `--norc` option.

Under Windows and VMS the option file is named *_proverc* rather than *.proverc* and is sought only in the current directory.

Reading from STDIN

If you have a list of tests (or URLs, or anything else you want to test) in a file, you can add them to your tests by using a '-':

```
prove - < my_list_of_things_to_test.txt
```

See the `README` in the `examples` directory of this distribution.

Default Test Directory

If no files or directories are supplied, `prove` looks for all files matching the pattern `t/*.t`.

Colored Test Output

Colored test output using *TAP::Formatter::Color* is the default, but if output is not to a terminal, color is disabled. You can override this by adding the `--color` switch.

Color support requires *Term::ANSIColor* on Unix-like platforms and *Win32::Console* on windows. If the necessary module is not installed colored output will not be available.

Exit Code

If the tests fail `prove` will exit with non-zero status.

Arguments to Tests

It is possible to supply arguments to tests. To do so separate them from prove's own arguments with the arisdottle, '::'. For example

```
prove -v t/mytest.t :: --url http://example.com
```

would run *t/mytest.t* with the options '--url http://example.com'. When running multiple tests they will each receive the same arguments.

--exec

Normally you can just pass a list of Perl tests and the harness will know how to execute them. However, if your tests are not written in Perl or if you want all tests invoked exactly the same way, use the `-e`, or `--exec` switch:

```
prove --exec '/usr/bin/ruby -w' t/
```

```
prove --exec '/usr/bin/perl -Tw -mstrict -Ilib' t/
prove --exec '/path/to/my/customer/exec'
```

--merge

If you need to make sure your diagnostics are displayed in the correct order relative to test results you can use the `--merge` option to merge the test scripts' STDERR into their STDOUT.

This guarantees that STDOUT (where the test results appear) and STDERR (where the diagnostics appear) will stay in sync. The harness will display any diagnostics your tests emit on STDERR.

Caveat: this is a bit of a kludge. In particular note that if anything that appears on STDERR looks like a test result the test harness will get confused. Use this option only if you understand the consequences and can live with the risk.

--trap

The `--trap` option will attempt to trap SIGINT (Ctrl-C) during a test run and display the test summary even if the run is interrupted

--state

You can ask `prove` to remember the state of previous test runs and select and/or order the tests to be run based on that saved state.

The `--state` switch requires an argument which must be a comma separated list of one or more of the following options.

`last`

Run the same tests as the last time the state was saved. This makes it possible, for example, to recreate the ordering of a shuffled test.

```
# Run all tests in random order
$ prove -b --state=save --shuffle

# Run them again in the same order
$ prove -b --state=last
```

`failed`

Run only the tests that failed on the last run.

```
# Run all tests
$ prove -b --state=save

# Run failures
$ prove -b --state=failed
```

If you also specify the `save` option newly passing tests will be excluded from subsequent runs.

```
# Repeat until no more failures
$ prove -b --state=failed,save
```

`passed`

Run only the passed tests from last time. Useful to make sure that no new problems have been introduced.

`all`

Run all tests in normal order. Multple options may be specified, so to run all tests with the failures from last time first:

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

`hot`

Run the tests that most recently failed first. The last failure time of each test is stored. The `hot` option causes tests to be run in most-recent- failure order.

```
$ prove -b --state=hot,save
```

Tests that have never failed will not be selected. To run all tests with the most recently failed first use

```
$ prove -b --state=hot,all,save
```

This combination of options may also be specified thus

```
$ prove -b --state=adrian
```

`todo`

Run any tests with todos.

`slow`

Run the tests in slowest to fastest order. This is useful in conjunction with the `-j` parallel testing switch to ensure that your slowest tests start running first.

```
$ prove -b --state=slow -j9
```

`fast`

Run test tests in fastest to slowest order.

`new`

Run the tests in newest to oldest order based on the modification times of the test scripts.

`old`

Run the tests in oldest to newest order.

`fresh`

Run those test scripts that have been modified since the last test run.

`save`

Save the state on exit. The state is stored in a file called *.prove* (*_prove* on Windows and VMS) in the current directory.

The `--state` switch may be used more than once.

```
$ prove -b --state=hot --state=all,save
```

--rules

The `--rules` option is used to control which tests are run sequentially and which are run in parallel, if the `--jobs` option is specified. The option may be specified multiple times, and the order matters.

The most practical use is likely to specify that some tests are not "parallel-ready". Since mentioning a file with --rules doesn't cause it to be selected to run as a test, you can "set and forget" some rules preferences in your .proverc file. Then you'll be able to take maximum advantage of the performance benefits of parallel testing, while some exceptions are still run in parallel.

--rules examples

```
    # All tests are allowed to run in parallel, except those starting with
"p"
    --rules='seq=t/p*.t' --rules='par=**'

    # All tests must run in sequence except those starting with "p", which
should be run parallel
    --rules='par=t/p*.t'
```

--rules resolution

* By default, all tests are eligible to be run in parallel. Specifying any of your own rules removes this one.

* "First match wins". The first rule that matches a test will be the one that applies.

* Any test which does not match a rule will be run in sequence at the end of the run.

* The existence of a rule does not imply selecting a test. You must still specify the tests to run.

* Specifying a rule to allow tests to run in parallel does not make them run in parallel. You still need specify the number of parallel `jobs` in your Harness object.

--rules Glob-style pattern matching

We implement our own glob-style pattern matching for --rules. Here are the supported patterns:

```
** is any number of characters, including /, within a pathname
* is zero or more characters within a filename/directory name
? is exactly one character within a filename/directory name
{foo,bar,baz} is any of foo, bar or baz.
\ is an escape character
```

More advanced specifications for parallel vs sequence run rules

If you need more advanced management of what runs in parallel vs in sequence, see the associated 'rules' documentation in *TAP::Harness* and *TAP::Parser::Scheduler*. If what's possible directly through `prove` is not sufficient, you can write your own harness to access these features directly.

@INC

prove introduces a separation between "options passed to the perl which runs prove" and "options passed to the perl which runs tests"; this distinction is by design. Thus the perl which is running a test starts with the default `@INC`. Additional library directories can be added via the `PERL5LIB` environment variable, via -Ifoo in `PERL5OPT` or via the `-Ilib` option to *prove*.

Taint Mode

Normally when a Perl program is run in taint mode the contents of the `PERL5LIB` environment variable do not appear in `@INC`.

Because `PERL5LIB` is often used during testing to add build directories to `@INC` prove passes the names of any directories found in `PERL5LIB` as -I switches. The net effect of this is that `PERL5LIB` is honoured even when prove is run in taint mode.

FORMATTERS

You can load a custom *TAP::Parser::Formatter*:

```
  prove --formatter MyFormatter
```

SOURCE HANDLERS

You can load custom *TAP::Parser::SourceHandler*s, to change the way the parser interprets particular *sources* of TAP.

```
  prove --source MyHandler --source YetAnother t
```

If you want to provide config to the source you can use:

```
  prove --source MyCustom \
        --source Perl --perl-option 'foo=bar baz' --perl-option avg=0.278 \
        --source File --file-option extensions=.txt --file-option
extensions=.tmp t
        --source pgTAP --pgtap-option pset=format=html --pgtap-option
pset=border=2
```

Each `--$source-option` option must specify a key/value pair separated by an =. If an option can take multiple values, just specify it multiple times, as with the `extensions=` examples above. If the option should be a hash reference, specify the value as a second pair separated by a =, as in the `pset=` examples above (escape = with a backslash).

All `--sources` are combined into a hash, and passed to *"new" in TAP::Harness*'s `sources` parameter.

See *TAP::Parser::IteratorFactory* for more details on how configuration is passed to *SourceHandlers*.

PLUGINS

Plugins can be loaded using the `-P`*`plugin`* syntax, eg:

```
  prove -PMyPlugin
```

This will search for a module named `App::Prove::Plugin::MyPlugin`, or failing that, `MyPlugin`. If the plugin can't be found, `prove` will complain & exit.

You can pass arguments to your plugin by appending `=arg1,arg2,etc` to the plugin name:

```
  prove -PMyPlugin=fou,du,fafa
```

Please check individual plugin documentation for more details.

Available Plugins

For an up-to-date list of plugins available, please check CPAN:

http://search.cpan.org/search?query=App%3A%3AProve+Plugin

Writing Plugins

Please see *"PLUGINS" in App::Prove*.