

NAME

TAP::Parser - Parse TAP output

VERSION

Version 3.36

SYNOPSIS

```
use TAP::Parser;

my $parser = TAP::Parser->new( { source => $source } );

while ( my $result = $parser->next ) {
    print $result->as_string;
}
```

DESCRIPTION

[TAP::Parser](#) is designed to produce a proper parse of TAP output. For an example of how to run tests through this module, see the simple harnesses [examples/](#).

There's a wiki dedicated to the Test Anything Protocol:

[<http://testanything.org>](http://testanything.org)

It includes the [TAP::Parser Cookbook](#):

[<http://testanything.org/testing-with-tap/perl/tap::parser-cookbook.html>](http://testanything.org/testing-with-tap/perl/tap::parser-cookbook.html)

METHODS**Class Methods**

new

```
my $parser = TAP::Parser->new(\%args);
```

Returns a new [TAP::Parser](#) object.

The arguments should be a hashref with *one* of the following keys:

- `source`

CHANGED in 3.18

This is the preferred method of passing input to the constructor.

The `source` is used to create a [TAP::Parser::Source](#) that is passed to the “`iterator_factory_class`” which in turn figures out how to handle the source and creates a `<TAP::Parser::Iterator>` for it. The iterator is used by the parser to read in the TAP stream.

To configure the *IteratorFactory* use the `sources` parameter below.

Note that `source`, `tap` and `exec` are *mutually exclusive*.

- `tap`

CHANGED in 3.18

The value should be the complete TAP output.

The `tap` is used to create a [TAP::Parser::Source](#) that is passed to the “`iterator_factory_class`” which in turn figures out how to handle the source and creates a `<TAP::Parser::Iterator>` for it. The iterator is used by the parser to read in the TAP stream.

To configure the *IteratorFactory* use the `sources` parameter below.

Note that `source`, `tap` and `exec` are *mutually exclusive*.

- `exec`

Must be passed an array reference.

The `exec` array ref is used to create a [TAP::Parser::Source](#) that is passed to the “`iterator_factory_class`” which in turn figures out how to handle the source and creates a `<TAP::Parser::Iterator>` for it. The iterator is used by the parser to read in the TAP stream.

By default the [TAP::Parser::SourceHandler::Executable](#) class will create a [TAP::Parser::Iterator::Process](#) object to handle the source. This passes the array reference strings as command arguments to `IPC::Open3::open3`:

```
exec => [ '/usr/bin/ruby', 't/my_test.rb' ]
```

If any `test_args` are given they will be appended to the end of the command argument list.

To configure the *IteratorFactory* use the `sources` parameter below.

Note that `source`, `tap` and `exec` are *mutually exclusive*.

The following keys are optional.

- `sources`

NEW to 3.18.

If set, `sources` must be a hashref containing the names of the `TAP::Parser::SourceHandlers` to load and/or configure. The values are a hash of configuration that will be accessible to the source handlers via “`config_for`” in `TAP::Parser::Source`.

For example:

```
sources => {
  Perl => { exec => '/path/to/custom/perl' },
  File => { extensions => [ '.tap', '.txt' ] },
  MyCustom => { some => 'config' },
}
```

This will cause [TAP::Parser](#) to pass custom configuration to two of the built-in source handlers - [TAP::Parser::SourceHandler::Perl](#), [TAP::Parser::SourceHandler::File](#) - and attempt to load the `MyCustom` class. See “`load_handlers`” in [TAP::Parser::IteratorFactory](#) for more detail.

The `sources` parameter affects how `source`, `tap` and `exec` parameters are handled.

See [TAP::Parser::IteratorFactory](#), [TAP::Parser::SourceHandler](#) and subclasses for more details.

- `callback`

If present, each callback corresponding to a given result type will be called with the result as the argument if the `run` method is used:

```
my %callbacks = (
  test => \&test_callback,
  plan => \&plan_callback,
  comment => \&comment_callback,
  bailout => \&bailout_callback,
  unknown => \&unknown_callback,
);

my $aggregator = TAP::Parser::Aggregator->new;
for my $file ( @test_files ) {
  my $parser = TAP::Parser->new(
    {
      source => $file,
      callbacks => \%callbacks,
    }
  );
```

```

    $parser->run;
    $aggregator->add( $file, $parser );
}

```

- `switches`

If using a Perl file as a source, optional switches may be passed which will be used when invoking the perl executable.

```

my $parser = TAP::Parser->new( {
    source => $test_file,
    switches => [ '-Ilib' ],
} );

```

- `test_args`

Used in conjunction with the `source` and `exec` option to supply a reference to an @ARGV style array of arguments to pass to the test program.

- `spool`

If passed a filehandle will write a copy of all parsed TAP to that handle.

- `merge`

If false, `STDERR` is not captured (though it is 'relayed' to keep it somewhat synchronized with `STDOUT`.)

If true, `STDERR` and `STDOUT` are the same filehandle. This may cause breakage if `STDERR` contains anything resembling TAP format, but does allow exact synchronization.

Subtleties of this behavior may be platform-dependent and may change in the future.

- `grammar_class`

This option was introduced to let you easily customize which *grammar* class the parser should use. It defaults to `TAP::Parser::Grammar`.

See also "make_grammar".

- `result_factory_class`

This option was introduced to let you easily customize which *result* factory class the parser should use. It defaults to `TAP::Parser::ResultFactory`.

See also "make_result".

- `iterator_factory_class`

CHANGED in 3.18

This option was introduced to let you easily customize which *iterator* factory class the parser should use. It defaults to `TAP::Parser::IteratorFactory`.

Instance Methods

next

```

my $parser = TAP::Parser->new( { source => $file } );
while ( my $result = $parser->next ) {
    print $result->as_string, "\n";
}

```

This method returns the results of the parsing, one result at a time. Note that it is destructive. You can't rewind and examine previous results.

If callbacks are used, they will be issued before this call returns.

Each result returned is a subclass of `TAP::Parser::Result`. See that module and related classes for more

information on how to use them.

run

```
$parser->run;
```

This method merely runs the parser and parses all of the TAP.

make_grammar

Make a new [TAP::Parser::Grammar](#) object and return it. Passes through any arguments given.

The `grammar_class` can be customized, as described in “new”.

make_result

Make a new [TAP::Parser::Result](#) object using the parser’s [TAP::Parser::ResultFactory](#), and return it. Passes through any arguments given.

The `result_factory_class` can be customized, as described in “new”.

make_iterator_factory

NEW to 3.18.

Make a new [TAP::Parser::IteratorFactory](#) object and return it. Passes through any arguments given.

`iterator_factory_class` can be customized, as described in “new”.

INDIVIDUAL RESULTS

If you’ve read this far in the docs, you’ve seen this:

```
while ( my $result = $parser->next ) {
    print $result->as_string;
}
```

Each result returned is a [TAP::Parser::Result](#) subclass, referred to as *result types*.

Result types

Basically, you fetch individual results from the TAP. The six types, with examples of each, are as follows:

- Version


```
TAP version 12
```
- Plan


```
1..42
```
- Pragma


```
pragma +strict
```
- Test


```
ok 3 - We should start with some foobar!
```
- Comment


```
# Hope we don't use up the foobar.
```
- Bailout


```
Bail out! We ran out of foobar!
```
- Unknown


```
... yo, this ain't TAP! ...
```

Each result fetched is a result object of a different type. There are common methods to each result object and different types may have methods unique to their type. Sometimes a type method may be overridden in a subclass, but its use is guaranteed to be identical.

Common type methods*type*

Returns the type of result, such as `comment` or `test`.

as_string

Prints a string representation of the token. This might not be the exact output, however. Tests will have test numbers added if not present, TODO and SKIP directives will be capitalized and, in general, things will be cleaned up. If you need the original text for the token, see the `raw` method.

raw

Returns the original line of text which was parsed.

is_plan

Indicates whether or not this is the test plan line.

is_test

Indicates whether or not this is a test line.

is_comment

Indicates whether or not this is a comment. Comments will generally only appear in the TAP stream if `STDERR` is merged to `STDOUT`. See the `merge` option.

is_bailout

Indicates whether or not this is bailout line.

is_yaml

Indicates whether or not the current item is a YAML block.

is_unknown

Indicates whether or not the current line could be parsed.

is_ok

```
if ( $result->is_ok ) { ... }
```

Reports whether or not a given result has passed. Anything which is **not** a test result returns true. This is merely provided as a convenient shortcut which allows you to do this:

```
my $parser = TAP::Parser->new( { source => $source } );
while ( my $result = $parser->next ) {
    # only print failing results
    print $result->as_string unless $result->is_ok;
}
```

plan methods

```
if ( $result->is_plan ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

plan

```
if ( $result->is_plan ) {
    print $result->plan;
}
```

This is merely a synonym for `as_string`.

directive

```
my $directive = $result->directive;
```

If a SKIP directive is included with the `plan`, this method will return it.

```
1..0 # SKIP: why bother?
```

explanation

```
my $explanation = $result->explanation;
```

If a SKIP directive was included with the plan, this method will return the explanation, if any.

pragma methods

```
if ( $result->is_pragma ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

pragmas

Returns a list of pragmas each of which is a + or - followed by the pragma name.

comment methods

```
if ( $result->is_comment ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

comment

```
if ( $result->is_comment ) {
my $comment = $result->comment;
print "I have something to say: $comment";
}
```

bailout methods

```
if ( $result->is_bailout ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

explanation

```
if ( $result->is_bailout ) {
my $explanation = $result->explanation;
print "We bailed out because ($explanation)";
}
```

If, and only if, a token is a bailout token, you can get an “explanation” via this method. The explanation is the text after the mystical “Bail out!” words which appear in the tap output.

unknown methods

```
if ( $result->is_unknown ) { ... }
```

There are no unique methods for unknown results.

test methods

```
if ( $result->is_test ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

ok

```
my $ok = $result->ok;
```

Returns the literal text of the ok or not ok status.

number

```
my $test_number = $result->number;
```

Returns the number of the test, even if the original TAP output did not supply that number.

description

```
my $description = $result->description;
```

Returns the description of the test, if any. This is the portion after the test number but before the directive.

directive

```
my $directive = $result->directive;
```

Returns either TODO or SKIP if either directive was present for a test line.

explanation

```
my $explanation = $result->explanation;
```

If a test had either a TODO or SKIP directive, this method will return the accompanying explanation, if present.

```
not ok 17 - 'Pigs can fly' # TODO not enough acid
```

For the above line, the explanation is *not enough acid*.

is_ok

```
if ( $result->is_ok ) { ... }
```

Returns a boolean value indicating whether or not the test passed. Remember that for TODO tests, the test always passes.

Note: this was formerly `passed`. The latter method is deprecated and will issue a warning.

is_actual_ok

```
if ( $result->is_actual_ok ) { ... }
```

Returns a boolean value indicating whether or not the test passed, regardless of its TODO status.

Note: this was formerly `actual_passed`. The latter method is deprecated and will issue a warning.

is_unplanned

```
if ( $test->is_unplanned ) { ... }
```

If a test number is greater than the number of planned tests, this method will return true. Unplanned tests will *always* return false for `is_ok`, regardless of whether or not the test `has_todo` (see [TAP::Parser::Result::Test](#) for more information about this).

has_skip

```
if ( $result->has_skip ) { ... }
```

Returns a boolean value indicating whether or not this test had a SKIP directive.

has_todo

```
if ( $result->has_todo ) { ... }
```

Returns a boolean value indicating whether or not this test had a TODO directive.

Note that TODO tests *always* pass. If you need to know whether or not they really passed, check the `is_actual_ok` method.

in_todo

```
if ( $parser->in_todo ) { ... }
```

True while the most recent result was a TODO. Becomes true before the TODO result is returned and stays true until just before the next non- TODO test is returned.

TOTAL RESULTS

After parsing the TAP, there are many methods available to let you dig through the results and determine what is meaningful to you.

Individual Results

These results refer to individual tests which are run.

passed

```
my @passed = $parser->passed; # the test numbers which passed
my $passed = $parser->passed; # the number of tests which passed
```

This method lets you know which (or how many) tests passed. If a test failed but had a TODO directive, it will be counted as a passed test.

failed

```
my @failed = $parser->failed; # the test numbers which failed
my $failed = $parser->failed; # the number of tests which failed
```

This method lets you know which (or how many) tests failed. If a test passed but had a TODO directive, it will **NOT** be counted as a failed test.

actual_passed

```
# the test numbers which actually passed
my @actual_passed = $parser->actual_passed;
```

```
# the number of tests which actually passed
my $actual_passed = $parser->actual_passed;
```

This method lets you know which (or how many) tests actually passed, regardless of whether or not a TODO directive was found.

actual_ok

This method is a synonym for `actual_passed`.

actual_failed

```
# the test numbers which actually failed
my @actual_failed = $parser->actual_failed;
```

```
# the number of tests which actually failed
my $actual_failed = $parser->actual_failed;
```

This method lets you know which (or how many) tests actually failed, regardless of whether or not a TODO directive was found.

todo

```
my @todo = $parser->todo; # the test numbers with todo directives
my $todo = $parser->todo; # the number of tests with todo directives
```

This method lets you know which (or how many) tests had TODO directives.

todo_passed

```
# the test numbers which unexpectedly succeeded
my @todo_passed = $parser->todo_passed;
```

```
# the number of tests which unexpectedly succeeded
my $todo_passed = $parser->todo_passed;
```

This method lets you know which (or how many) tests actually passed but were declared as “TODO” tests.

todo_failed

```
# deprecated in favor of 'todo_passed'. This method was horribly misnamed.
```

This was a badly misnamed method. It indicates which TODO tests unexpectedly succeeded. Will now issue a warning and call `todo_passed`.

skipped

```
my @skipped = $parser->skipped; # the test numbers with SKIP directives
my $skipped = $parser->skipped; # the number of tests with SKIP directives
```

This method lets you know which (or how many) tests had SKIP directives.

Pragmas

pragma

Get or set a pragma. To get the state of a pragma:

```
if ( $p->pragma('strict') ) {
    # be strict
}
```

To set the state of a pragma:

```
$p->pragma('strict', 1); # enable strict mode
```

pragmas

Get a list of all the currently enabled pragmas:

```
my @pragmas_enabled = $p->pragmas;
```

Summary Results

These results are “meta” information about the total results of an individual test program.

plan

```
my $plan = $parser->plan;
```

Returns the test plan, if found.

good_plan

Deprecated. Use *is_good_plan* instead.

is_good_plan

```
if ( $parser->is_good_plan ) { ... }
```

Returns a boolean value indicating whether or not the number of tests planned matches the number of tests run.

Note: this was formerly *good_plan*. The latter method is deprecated and will issue a warning.

And since we’re on that subject ...

tests_planned

```
print $parser->tests_planned;
```

Returns the number of tests planned, according to the plan. For example, a plan of ‘1..17’ will mean that 17 tests were planned.

tests_run

```
print $parser->tests_run;
```

Returns the number of tests which actually were run. Hopefully this will match the number of *\$parser->tests_planned*.

skip_all

Returns a true value (actually the reason for skipping) if all tests were skipped.

start_time

Returns the wall-clock time when the Parser was created.

end_time

Returns the wall-clock time when the end of TAP input was seen.

start_times

Returns the CPU times (like “times” in [perlfunc\(1\)](#)) when the Parser was created.

end_times

Returns the CPU times (like “times” in [perlfunc\(1\)](#)) when the end of TAP input was seen.

has_problems

```
if ( $parser->has_problems ) {
    ...
}
```

This is a ‘catch-all’ method which returns true if any tests have currently failed, any TODO tests unexpectedly succeeded, or any parse errors occurred.

version

```
$parser->version;
```

Once the parser is done, this will return the version number for the parsed TAP. Version numbers were introduced with TAP version 13 so if no version number is found version 12 is assumed.

exit

```
$parser->exit;
```

Once the parser is done, this will return the exit status. If the parser ran an executable, it returns the exit status of the executable.

wait

```
$parser->wait;
```

Once the parser is done, this will return the wait status. If the parser ran an executable, it returns the wait status of the executable. Otherwise, this merely returns the `exit` status.

ignore_exit

```
$parser->ignore_exit(1);
```

Tell the parser to ignore the exit status from the test when determining whether the test passed. Normally tests with non-zero exit status are considered to have failed even if all individual tests passed. In cases where it is not possible to control the exit value of the test script use this option to ignore it.

parse_errors

```
my @errors = $parser->parse_errors; # the parser errors
my $errors = $parser->parse_errors; # the number of parse_errors
```

Fortunately, all TAP output is perfect. In the event that it is not, this method will return parser errors. Note that a junk line which the parser does not recognize is not an error. This allows this parser to handle future versions of TAP. The following are all TAP errors reported by the parser:

- Misplaced plan
 - The plan (for example, ‘1.5’), must only come at the beginning or end of the TAP output.
- No plan
 - Gotta have a plan!
- More than one plan
 - 1..3
 - ok 1 - input file opened
 - not ok 2 - first line of the input valid # todo some data
 - ok 3 read the rest of the file
 - 1..3

Right. Very funny. Don't do that.

- Test numbers out of sequence

```
1..3
ok 1 - input file opened
not ok 2 - first line of the input valid # todo some data
ok 2 read the rest of the file
```

That last test line above should have the number '3' instead of '2'.

Note that it's perfectly acceptable for some lines to have test numbers and others to not have them. However, when a test number is found, it must be in sequence. The following is also an error:

```
1..3
ok 1 - input file opened
not ok - first line of the input valid # todo some data
ok 2 read the rest of the file
```

But this is not:

```
1..3
ok - input file opened
not ok - first line of the input valid # todo some data
ok 3 read the rest of the file
```

get_select_handles

Get an a list of file handles which can be passed to `select` to determine the readiness of this parser.

delete_spool

Delete and return the spool.

```
my $fh = $parser->delete_spool;
```

CALLBACKS

As mentioned earlier, a “callback” key may be added to the `TAP::Parser` constructor. If present, each callback corresponding to a given result type will be called with the result as the argument if the `run` method is used. The callback is expected to be a subroutine reference (or anonymous subroutine) which is invoked with the parser result as its argument.

```
my %callbacks = (
    test => \&test_callback,
    plan => \&plan_callback,
    comment => \&comment_callback,
    bailout => \&bailout_callback,
    unknown => \&unknown_callback,
);

my $aggregator = TAP::Parser::Aggregator->new;
for my $file ( @test_files ) {
    my $parser = TAP::Parser->new(
        {
            source => $file,
            callbacks => \%callbacks,
        }
    );
    $parser->run;
    $aggregator->add( $file, $parser );
}
```

Callbacks may also be added like this:

```
$parser->callback( test => \&test_callback );
$parser->callback( plan => \&plan_callback );
```

The following keys allowed for callbacks. These keys are case-sensitive.

- test
Invoked if `$result->is_test` returns true.
- version
Invoked if `$result->is_version` returns true.
- plan
Invoked if `$result->is_plan` returns true.
- comment
Invoked if `$result->is_comment` returns true.
- bailout
Invoked if `$result->is_unknown` returns true.
- yaml
Invoked if `$result->is_yaml` returns true.
- unknown
Invoked if `$result->is_unknown` returns true.
- ELSE
If a result does not have a callback defined for it, this callback will be invoked. Thus, if all of the previous result types are specified as callbacks, this callback will *never* be invoked.
- ALL
This callback will always be invoked and this will happen for each result after one of the above callbacks is invoked. For example, if [Term::ANSIColor](#) is loaded, you could use the following to color your test output:

```
my %callbacks = (
  test => sub {
    my $test = shift;
    if ( $test->is_ok && not $test->directive ) {
      # normal passing test
      print color 'green';
    }
    elsif ( !$test->is_ok ) { # even if it's TODO
      print color 'white on_red';
    }
    elsif ( $test->has_skip ) {
      print color 'white on_blue';
    }
    elsif ( $test->has_todo ) {
      print color 'white';
    }
  },
  ELSE => sub {
    # plan, comment, and so on (anything which isn't a test line)
    print color 'black on_white';
  }
);
```

```

    },
    ALL => sub {
    # now print them
    print shift->as_string;
    print color 'reset';
    print "\n";
    },
    );

```

- EOF

Invoked when there are no more lines to be parsed. Since there is no accompanying [TAP::Parser::Result](#) object the [TAP::Parser](#) object is passed instead.

TAP GRAMMAR

If you're looking for an EBNF grammar, see [TAP::Parser::Grammar](#).

BACKWARDS COMPATIBILITY

The Perl-QA list attempted to ensure backwards compatibility with [Test::Harness](#). However, there are some minor differences.

Differences

- TODO plans

A little-known feature of [Test::Harness](#) is that it supported TODO lists in the plan:

```

1..2 todo 2
ok 1 - We have liftoff
not ok 2 - Anti-gravity device activated

```

Under [Test::Harness](#), test number 2 would *pass* because it was listed as a TODO test on the plan line. However, we are not aware of anyone actually using this feature and hard-coding test numbers is discouraged because it's very easy to add a test and break the test number sequence. This makes test suites very fragile. Instead, the following should be used:

```

1..2
ok 1 - We have liftoff
not ok 2 - Anti-gravity device activated # TODO

```

- 'Missing' tests

It rarely happens, but sometimes a harness might encounter 'missing tests':

```

ok 1
ok 2
ok 15
ok 16
ok 17

```

[Test::Harness](#) would report tests 3-14 as having failed. For the [TAP::Parser](#) these tests are not considered failed because they've never run. They're reported as parse failures (tests out of sequence).

SUBCLASSING

If you find you need to provide custom functionality (as you would have using [Test::Harness::Straps](#)), you're in luck: [TAP::Parser](#) and friends are designed to be easily plugged-into and/or subclassed.

Before you start, it's important to know a few things:

1. All [TAP::*](#) objects inherit from [TAP::Object](#).
2. Many [TAP::*](#) classes have a *SUBCLASSING* section to guide you.

3. Note that `TAP::Parser` is designed to be the central “maker” - ie: it is responsible for creating most new objects in the `TAP::Parser::*` namespace.
 This makes it possible for you to have a single point of configuring what subclasses should be used, which means that in many cases you’ll find you only need to sub-class one of the parser’s components.
 The exception to this rule are *SourceHandlers & Iterators*, but those are both created with customizable *IteratorFactory*.
4. By subclassing, you may end up overriding undocumented methods. That’s not a bad thing per se, but be forewarned that undocumented methods may change without warning from one release to the next - we cannot guarantee backwards compatibility. If any *documented* method needs changing, it will be deprecated first, and changed in a later release.

Parser Components

Sources

A TAP parser consumes input from a single *raw source* of TAP, which could come from anywhere (a file, an executable, a database, an IO handle, a URI, etc..). The source gets bundled up in a `TAP::Parser::Source` object which gathers some meta data about it. The parser then uses a `TAP::Parser::IteratorFactory` to determine which `TAP::Parser::SourceHandler` to use to turn the raw source into a stream of TAP by way of “Iterators”.

If you simply want `TAP::Parser` to handle a new source of TAP you probably don’t need to subclass `TAP::Parser` itself. Rather, you’ll need to create a new `TAP::Parser::SourceHandler` class, and just plug it into the parser using the *sources* param to “new”. Before you start writing one, read through `TAP::Parser::IteratorFactory` to get a feel for how the system works first.

If you find you really need to use your own iterator factory you can still do so without sub-classing `TAP::Parser` by setting “*iterator_factory_class*”.

If you just need to customize the objects on creation, subclass `TAP::Parser` and override “*make_iterator_factory*”.

Note that *make_source* & *make_perl_source* have been *DEPRECATED* and are now removed.

Iterators

A TAP parser uses *iterators* to loop through the *stream* of TAP read in from the *source* it was given. There are a few types of Iterators available by default, all sub-classes of `TAP::Parser::Iterator`. Choosing which iterator to use is the responsibility of the *iterator factory*, though it simply delegates to the *Source Handler* it uses.

If you’re writing your own `TAP::Parser::SourceHandler`, you may need to create your own iterators too. If so you’ll need to subclass `TAP::Parser::Iterator`.

Note that “*make_iterator*” has been *DEPRECATED* and is now removed.

Results

A TAP parser creates `TAP::Parser::Results` as it iterates through the input *stream*. There are quite a few result types available; choosing which class to use is the responsibility of the *result factory*.

To create your own result types you have two options:

option 1

Subclass `TAP::Parser::Result` and register your new result type/class with the default `TAP::Parser::ResultFactory`.

option 2

Subclass `TAP::Parser::ResultFactory` itself and implement your own `TAP::Parser::Result` creation logic. Then you’ll need to customize the class used by your parser by setting the *result_factory_class* parameter. See “new” for more details.

If you need to customize the objects on creation, subclass [TAP::Parser](#) and override “make_result”.

Grammar

[TAP::Parser::Grammar](#) is the heart of the parser. It tokenizes the TAP input *stream* and produces results. If you need to customize its behaviour you should probably familiarize yourself with the source first. Enough lecturing.

Subclass [TAP::Parser::Grammar](#) and customize your parser by setting the `grammar_class` parameter. See “new” for more details.

If you need to customize the objects on creation, subclass [TAP::Parser](#) and override “make_grammar”

ACKNOWLEDGMENTS

All of the following have helped. Bug reports, patches, (im)moral support, or just words of encouragement have all been forthcoming.

- Michael Schwern
- Andy Lester
- chromatic
- GEOFFR
- Shlomi Fish
- Torsten Schoenfeld
- Jerry Gay
- Aristotle
- Adam Kennedy
- Yves Orton
- Adrian Howard
- Sean & Lil
- Andreas J. Koenig
- Florian Ragwitz
- Corion
- Mark Stosberg
- Matt Kraai
- David Wheeler
- Alex Vandiver
- Cosimo Streppone
- Ville Skyttä

AUTHORS

Curtis “Ovid” Poe <ovid@cpan.org>

Andy Armstong <andy@hexten.net>

Eric Wilhelm @ <ewilhelm at cpan dot org>

Michael Peters <mpeters at plusthree dot com>

Leif Eriksen <leif dot eriksen at bigpond dot com>

Steve Purkis <spurkis@cpan.org>

Nicholas Clark <nick@ccl4.org>

Lee Johnson <notfadeaway at btinternet dot com>

Philippe Bruhat <book@cpan.org>

BUGS

Please report any bugs or feature requests to `bug-test-harness@rt.cpan.org`, or through the web interface at <http://rt.cpan.org/NoAuth/ReportBug.html?Queue=Test-Harness>. We will be notified, and then you'll automatically be notified of progress on your bug as we make changes.

Obviously, bugs which include patches are best. If you prefer, you can patch against bleed by via anonymous checkout of the latest version:

```
git clone git://github.com/Perl-Toolchain-Gang/Test-Harness.git
```

COPYRIGHT & LICENSE

Copyright 2006-2008 Curtis "Ovid" Poe, all rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.