NAME

TAP::Parser::Grammar - A grammar for the Test Anything Protocol.

VERSION

Version 3.26

SYNOPSIS

```
use TAP::Parser::Grammar;
my $grammar = $self->make_grammar({
  iterator => $tap_parser_iterator,
  parser   => $tap_parser,
  version  => 12,
});

my $result = $grammar->tokenize;
```

DESCRIPTION

`TAP::Parser::Grammar` tokenizes lines from a *TAP::Parser::Iterator* and constructs *TAP::Parser::Result* subclasses to represent the tokens.

Do not attempt to use this class directly. It won't make sense. It's mainly here to ensure that we will be able to have pluggable grammars when TAP is expanded at some future date (plus, this stuff was really cluttering the parser).

METHODS

Class Methods

new

```
my $grammar = TAP::Parser::Grammar->new({
    iterator => $iterator,
    parser   => $parser,
    version  => $version,
});
```

Returns *TAP::Parser* grammar object that will parse the TAP stream from the specified iterator. Both `iterator` and `parser` are required arguments. If `version` is not set it defaults to `12` (see *set_version* for more details).

Instance Methods

set_version

```
$grammar->set_version(13);
```

Tell the grammar which TAP syntax version to support. The lowest supported version is 12. Although 'TAP version' isn't valid version 12 syntax it is accepted so that higher version numbers may be parsed.

tokenize

```
my $token = $grammar->tokenize;
```

This method will return a *TAP::Parser::Result* object representing the current line of TAP.

token_types

```
my @types = $grammar->token_types;
```

Returns the different types of tokens which this grammar can parse.

syntax_for

```
my $syntax = $grammar->syntax_for($token_type);
```

Returns a pre-compiled regular expression which will match a chunk of TAP corresponding to the token type. For example (not that you should really pay attention to this, `$grammar->syntax_for('comment')` will return `qr/^#(.*)/`.

handler_for

```
my $handler = $grammar->handler_for($token_type);
```

Returns a code reference which, when passed an appropriate line of TAP, returns the lexed token corresponding to that line. As a result, the basic TAP parsing loop looks similar to the following:

```
my @tokens;
my $grammar = TAP::Grammar->new;
LINE: while ( defined( my $line = $parser->_next_chunk_of_tap ) ) {
    for my $type ( $grammar->token_types ) {
        my $syntax  = $grammar->syntax_for($type);
        if ( $line =~ $syntax ) {
            my $handler = $grammar->handler_for($type);
            push @tokens => $grammar->$handler($line);
            next LINE;
        }
    }
    push @tokens => $grammar->_make_unknown_token($line);
}
```

TAP GRAMMAR

NOTE: This grammar is slightly out of date. There's still some discussion about it and a new one will be provided when we have things better defined.

The *TAP::Parser* does not use a formal grammar because TAP is essentially a stream-based protocol. In fact, it's quite legal to have an infinite stream. For the same reason that we don't apply regexes to streams, we're not using a formal grammar here. Instead, we parse the TAP in lines.

For purposes for forward compatibility, any result which does not match the following grammar is currently referred to as *TAP::Parser::Result::Unknown*. It is *not* a parse error.

A formal grammar would look similar to the following:

```
(*
    For the time being, I'm cheating on the EBNF by allowing
    certain terms to be defined by POSIX character classes by
    using the following syntax:

      digit ::= [:digit:]

    As far as I am aware, that's not valid EBNF.  Sue me.  I
    didn't know how to write "char" otherwise (Unicode issues).
    Suggestions welcome.
*)

tap            ::= version? { comment | unknown } leading_plan lines
```

```
                    |
                    lines trailing_plan {comment}

version         ::= 'TAP version ' positiveInteger {positiveInteger} "\n"

leading_plan    ::= plan skip_directive? "\n"

trailing_plan   ::= plan "\n"

plan            ::= '1..' nonNegativeInteger

lines           ::= line {line}

line            ::= (comment | test | unknown | bailout ) "\n"

test            ::= status positiveInteger? description? directive?

status          ::= 'not '? 'ok '

description     ::= (character - (digit | '#')) {character - '#'}

directive       ::= todo_directive | skip_directive

todo_directive  ::= hash_mark 'TODO' ' ' {character}

skip_directive  ::= hash_mark 'SKIP' ' ' {character}

comment         ::= hash_mark {character}

hash_mark       ::= '#' {' '}

bailout         ::= 'Bail out!' {character}

unknown         ::= { (character - "\n") }

(* POSIX character classes and other terminals *)

digit              ::= [:digit:]
character          ::= ([:print:] - "\n")
positiveInteger    ::= ( digit - '0' ) {digit}
nonNegativeInteger ::= digit {digit}
```

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

If you *really* want to subclass *TAP::Parser*'s grammar the best thing to do is read through the code. There's no easy way of summarizing it here.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator*, *TAP::Parser::Result*,