tangentsky Documentation

Release 0.6.9.dev0

Josh Bialkowski

Feb 06, 2020

Contents

The `cmake-format` project provides Quality Assurance (QA) tools for `cmake`:

- `cmake-annotate` can generate pretty HTML from your listfiles
- `cmake-format` can format your listfiles nicely so that they don't look like crap.
- `cmake-lint` can check your listfiles for problems
- `ctest-to` can parse a ctest output tree and translate it into a more structured format (either JSON or XML).

CHAPTER 1

Installation

All of the tools are included as part of the `cmake-format` python distribution package.

1.1 Install with pip

The easiest way to install `cmake-format` is from pypi.org using pip. For example:

```
pip install cmake_format
```

If you're on a linux-type system (such as ubuntu) the above command might not work if it would install into a system-wide location. If that's what you really want you might need to use `sudo`, e.g.:

```
sudo pip install cmake_format
```

In general though I wouldn't really recommend doing that though since things can get pretty messy between your system python distributions and your `pip` managed directories. Alternatively you can install it for your user with:

```
pip install --user cmake_format
```

which I would probably recommend for most users.

Note: If you wish to use a configuration file in YAML format you'll want to install with the optional `YAML` feature, e.g.:

```
pip install cmake_format[YAML]
```

1.2 Install from source

You can also install from source with pip. You can download a release package from github or pypi and then install it directly with pip. For example:

```
pip install v0.6.9.tar.gz
```

Note that the release packages on github are automatically generated from git tags which are the same commit used to generate the corresponding version package on `pypi.org`. So whether you install a particular version from github or pypi shouldn't matter.

Pip can also install directly from github. For example:

```
pip install git+https://github.com/cheshirekow/cmake_format.git
```

If you wish to test a pre-release or dev package from a branch called `foobar` you can install it with:

```
pip install "git+https://github.com/cheshirekow/cmake_format.git@foobar"
```

1.3 IDE Integrations

For the formatter specifically:

- There is an official vscode extension
- Someone also created a sublime plugin

Note that for both plugins `cmake-format` itself must be installed separately.

1.4 Pre-commit

If you are a user of the pre-commit project you can easily add the formatter, `cmake-format`, to your hooks with the following addition to your `.pre-commit-hooks.yaml` file.

```
repos:
  - repo: https://github.com/cheshirekow/cmake-format-precommit
    rev: v0.6.9
    hooks:
    - id: cmake-format
```

CHAPTER 2

Configuration

The different tools utilize a common parsing core and so share a common configuration system and in fact can share the same configuration file(s).

The tools accept configuration files in yaml, json, or python format.

Note: In order to read your configuration files in YAML format, the tools require the `pyyaml` python package to be installed. This dependency is not enforced by default during installation since this is an optional feature. It can be enabled by installing with a command like `pip install cmake_format[YAML]`. Or you can install `pyyaml` manually.

You may specify a path to one or more configuration files with the `--config-file` (`--config-files`) command line option. Otherwise, the tools will search the ancestry of each source file looking for a configuration file to use. If no configuration file is found it will use sensible defaults.

Automatically detected configuration files may have any name that matches `\.?cmake-format(.yaml|.json|.py)`.

If you'd like to create a new configuration file, `cmake-format` can help by dumping out the default configuration in your preferred format. You can run `cmake-format --dump-config [python|json|yaml]` to print the default configuration to `stdout` and use that as a starting point.

Warning: `cmake-format` will detect and load an automatic configuration file, even when executing `--dump-config`. This is so that you can debug the *active* configuration from a certain location in your tree. If you have a corrupt / un-parsable configuration file in an automatic location `--dump-config` may fail. You can always `cd` to a clean location (`/` or `/tmp`) and execute `--dump-config` to get a default configuration.

Here is an example python-style configuration file with the default options and help-text. Some detailed explanation and examples can be found at *Various configuration options and parameters*.

```
# ---------------------------------
# Options affecting listfile parsing
```

(continues on next page)

(continued from previous page)

```
# ----------------------------------
with section("parse"):

  # Specify structure for custom cmake functions
  additional_commands = { 'format_and_lint': { 'kwargs': { 'CC': '*',
                                       'CCDEPENDS': '*',
                                       'CMAKE': '*',
                                       'EXCLUDE': '*',
                                       'JS': '*',
                                       'PY': '*'}},
    'pkg_find': {'kwargs': {'PKG': '*'}}}

  # Specify variable tags.
  vartags = []

  # Specify property tags.
  proptags = []

# -----------------------------
# Options affecting formatting.
# -----------------------------
with section("format"):

  # How wide to allow formatted cmake files
  line_width = 80

  # How many spaces to tab for indent
  tab_size = 2

  # If an argument group contains more than this many sub-groups (parg or kwarg
  # groups) then force it to a vertical layout.
  max_subgroups_hwrap = 2

  # If a positional argument group contains more than this many arguments, then
  # force it to a vertical layout.
  max_pargs_hwrap = 6

  # If a cmdline positional group consumes more than this many lines without
  # nesting, then invalidate the layout (and nest)
  max_rows_cmdline = 2

  # If true, separate flow control names from their parentheses with a space
  separate_ctrl_name_with_space = False

  # If true, separate function names from parentheses with a space
  separate_fn_name_with_space = False

  # If a statement is wrapped to more than one line, than dangle the closing
  # parenthesis on its own line.
  dangle_parens = False

  # If the trailing parenthesis must be 'dangled' on its on line, then align it
  # to this reference: `prefix`: the start of the statement,  `prefix-indent`:
  # the start of the statement, plus one indentation  level, `child`: align to
  # the column of the arguments
  dangle_align = 'prefix'
```

(continues on next page)

(continued from previous page)

```
  # If the statement spelling length (including space and parenthesis) is
  # smaller than this amount, then force reject nested layouts.
  min_prefix_chars = 4

  # If the statement spelling length (including space and parenthesis) is larger
  # than the tab width by more than this amount, then force reject un-nested
  # layouts.
  max_prefix_chars = 10

  # If a candidate layout is wrapped horizontally but it exceeds this many
  # lines, then reject the layout.
  max_lines_hwrap = 2

  # What style line endings to use in the output.
  line_ending = 'unix'

  # Format command names consistently as 'lower' or 'upper' case
  command_case = 'canonical'

  # Format keywords consistently as 'lower' or 'upper' case
  keyword_case = 'unchanged'

  # A list of command names which should always be wrapped
  always_wrap = []

  # If true, the argument lists which are known to be sortable will be sorted
  # lexicographicall
  enable_sort = True

  # If true, the parsers may infer whether or not an argument list is sortable
  # (without annotation).
  autosort = False

  # By default, if cmake-format cannot successfully fit everything into the
  # desired linewidth it will apply the last, most agressive attempt that it
  # made. If this flag is True, however, cmake-format will print error, exit
  # with non-zero status code, and write-out nothing
  require_valid_layout = False

  # A dictionary mapping layout nodes to a list of wrap decisions. See the
  # documentation for more information.
  layout_passes = {}

# ------------------------------------------------
# Options affecting comment reflow and formatting.
# ------------------------------------------------
with section("markup"):

  # What character to use for bulleted lists
  bullet_char = '*'

  # What character to use as punctuation after numerals in an enumerated list
  enum_char = '.'

  # If comment markup is enabled, don't reflow the first comment block in each
  # listfile. Use this to preserve formatting of your copyright/license
  # statements.
```

(continues on next page)

(continued from previous page)

```
  first_comment_is_literal = False

  # If comment markup is enabled, don't reflow any comment block which matches
  # this (regex) pattern. Default is `None` (disabled).
  literal_comment_pattern = None

  # Regular expression to match preformat fences in comments default=
  # ``r'^\s*([`~]{3}[`~]*)(.*)$'``
  fence_pattern = '^\\s*([`~]{3}[`~]*)(.*)$'

  # Regular expression to match rulers in comments default=
  # ``r'^\s*[^\w\s]{3}.*[^\w\s]{3}$'``
  ruler_pattern = '^\\s*[^\\w\\s]{3}.*[^\\w\\s]{3}$'

  # If a comment line matches starts with this pattern then it is explicitly a
  # trailing comment for the preceeding argument. Default is '#<'
  explicit_trailing_pattern = '#<'

  # If a comment line starts with at least this many consecutive hash
  # characters, then don't lstrip() them off. This allows for lazy hash rulers
  # where the first hash char is not separated by space
  hashruler_min_length = 10

  # If true, then insert a space between the first hash char and remaining hash
  # chars in a hash ruler, and normalize its length to fill the column
  canonicalize_hashrulers = True

  # enable comment markup parsing and reflow
  enable_markup = True

# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # a list of lint codes to disable
  disabled_codes = []

  # regular expression pattern describing valid function names
  function_pattern = '[0-9a-z_]+'

  # regular expression pattern describing valid macro names
  macro_pattern = '[0-9A-Z_]+'

  # regular expression pattern describing valid names for variables with global
  # scope
  global_var_pattern = '[0-9A-Z][0-9A-Z_]+'

  # regular expression pattern describing valid names for variables with global
  # scope (but internal semantic)
  internal_var_pattern = '_[0-9A-Z][0-9A-Z_]+'

  # regular expression pattern describing valid names for variables with local
  # scope
  local_var_pattern = '[0-9a-z_]+'

  # regular expression pattern describing valid names for privatedirectory
```

(continues on next page)

(continued from previous page)

```
  # variables
  private_var_pattern = '_[0-9a-z_]+'

  # regular expression pattern describing valid names for publicdirectory
  # variables
  public_var_pattern = '[0-9A-Z][0-9A-Z_]+'

  # regular expression pattern describing valid names for keywords used in
  # functions or macros
  keyword_pattern = '[0-9A-Z_]+'

  # In the heuristic for C0201, how many conditionals to match within a loop in
  # before considering the loop a parser.
  max_conditionals_custom_parser = 2

  # Require at least this many newlines between statements
  min_statement_spacing = 1

  # Require no more than this many newlines between statements
  max_statement_spacing = 1
  max_returns = 6
  max_branches = 12
  max_arguments = 5
  max_localvars = 15
  max_statements = 50

# --------------------------------
# Options affecting file encoding
# --------------------------------
with section("encode"):

  # If true, emit the unicode byte-order mark (BOM) at the start of the file
  emit_byteorder_mark = False

  # Specify the encoding of the input file. Defaults to utf-8
  input_encoding = 'utf-8'

  # Specify the encoding of the output file. Defaults to utf-8. Note that cmake
  # only claims to support utf-8 so be careful when using anything else
  output_encoding = 'utf-8'

# ----------------------------------
# Miscellaneous configurations options.
# ----------------------------------
with section("misc"):

  # A dictionary containing any per-command configuration overrides. Currently
  # only `command_case` is supported.
  per_command = {}
```

CHAPTER 3

cmake-annotate

The `cmake-annotate` frontend program which can create semantic HTML documents from parsed listfiles. This enables, in particular, semantic highlighting for your code documentation.

```
usage:
cmake-annotate [-h]
               [--format {page,stub}]
               [-o OUTFILE_PATH]
               [-c CONFIG_FILE]
               infilepath [infilepath ...]

Parse cmake listfiles and re-emit them with semantic annotations in HTML.

Some options regarding parsing are configurable by providing a configuration
file. The configuration file format is the same as that used by cmake-format,
and the same file can be used for both programs.

cmake-format can spit out the default configuration for you as starting point
for customization. Run with `--dump-config [yaml|json|python]`.

positional arguments:
  infilepaths

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -f {page,stub,iframe}, --format {page,stub,iframe}
                        whether to output a standalone `page` complete with
                        <html></html> tags, or just the annotated content
  -o OUTFILE_PATH, --outfile-path OUTFILE_PATH
                        Where to write the formatted file. Default is stdout.
  -c CONFIG_FILE, --config-file CONFIG_FILE
                        path to configuration file
```

`--format stub` will output just the marked-up listfile content. The markup is done as `<span>` elements with different css classes for each parse-tree node or lexer token. The content is not encapsulated in any root element (such

as a). --format page will embed that content into a full page with a root <html> and an embedded stylesheet.

The example listfile in the README, for example, can be rendered as:

CHAPTER 4

cmake-format

`cmake-format` can format your listfiles nicely so that they don't look like crap.

4.1 Features

4.1.1 Markup

`cmake-format` is for the exceptionally lazy. It will even format your comments for you. It will reflow your comment text to within the configured line width. It also understands a very limited markup format for a couple of common bits.

rulers: A ruler is a line which starts with and ends with three or more non-alphanum or space characters:

```
# ---- This is a Ruler ----
# cmake-format will know to keep the ruler separated from the
# paragraphs around it. So it wont try to reflow this text as
# a single paragraph.
# ---- This is also a Ruler ---
```

list: A list is started on the first encountered list item, which starts with a bullet character (`*`) followed by a space followed by some text. Subsequent lines will be included in the list item until the next list item is encountered (the bullet must be at the same indentation level). The list must be surrounded by a pair of empty lines. Nested lists will be formatted in nested text:

```
# here are some lists:
#
# * item 1
# * item 2
#
#   * subitem 1
#   * subitem 2
#
# * second list item 1
# * second list item 2
```

enumerations: An enumeration is similar to a list but the bullet character is some integers followed by a period. New enumeration items are detected as long as either the first digit or the punctuation lines up in the same column as the previous item. `cmake-format` will renumber your items and align their labels for you:

```
# This is an enumeration
#
#   1. item
#   2. item
#   3. item
```

fences: If you have any text which you do not want to be formatted you can guard it with a pair of fences. Fences are three or more tilde characters:

```
# ~~~
# This comment is fenced
#   and will not be formatted
# ~~~
```

Note that comment fences guard reflow of *comment text*, and not cmake code. If you wish to prevent formatting of cmake, code, see below. In addition to fenced-literals, there are three other ways to preserve comment text from markup and/or reflow processing:

- The `--first-comment-is-literal` configuration option will exactly preserve the first comment in the file. This is intended to preserve copyright or other formatted header comments.
- The `--literal-comment-pattern` configuration option allows for a more generic way to identify comments which should be preserved literally. This configuration takes a regular expression pattern.
- The `--enable-markup` configuration option globally enables comment markup processing. It defaults to true so set it to false if you wish to globally disable comment markup processing. Note that trailing whitespace is still chomped from comments.

4.1.2 Disable Formatting Locally

You can locally disable and enable code formatting by using the special comments `# cmake-format: off` and `# cmake-format: on`.

4.1.3 Sort Argument Lists

Starting with version *0.5.0*, `cmake-format` can sort your argument lists for you. If the configuration includes `autosort=True` (the default), it will replace:

```
add_library(foobar STATIC EXCLUDE_FROM_ALL
            sourcefile_06.cc
            sourcefile_03.cc
            sourcefile_02.cc
            sourcefile_04.cc
            sourcefile_07.cc
            sourcefile_01.cc
            sourcefile_05.cc)
```

with:

```
add_library(foobar STATIC EXCLUDE_FROM_ALL
            sourcefile_01.cc
```

(continues on next page)

(continued from previous page)

```
            sourcefile_02.cc
            sourcefile_03.cc
            sourcefile_04.cc
            sourcefile_05.cc
            sourcefile_06.cc
            sourcefile_07.cc)
```

This is implemented for any argument lists which the parser knows are inherently sortable. This includes the following cmake commands:

- `add_library`
- `add_executable`

For most other cmake commands, you can use an annotation comment to hint to `cmake-format` that the argument list is sortable. For instance:

```
set(SOURCES
    # cmake-format: sortable
    bar.cc
    baz.cc
    foo.cc)
```

Annotations can be given in a line-comment or a bracket comment. There is a long-form and a short-form for each. The acceptable formats are:

Line Comment	long	`# cmake-format: <tag>`
Line Comment	short	`# cmf: <tag>`
Bracket Comment	long	`#[[cmake-format: <tag>]]`
Bracket Comment	short	`#[[cmf: <tag>]]`

In order to annotate a positional argument list as sortable, the acceptable tags are: `sortable` or `sort`. For the commands listed above where the positinal argument lists are inherently sortable, you can locally disable sorting by annotating them with `unsortable` or `unsort`. For example:

```
add_library(foobar STATIC
            # cmake-format: unsort
            sourcefile_03.cc
            sourcefile_01.cc
            sourcefile_02.cc)
```

Note that this is only needed if your configuration has enabled `autosort`, and you can globally disable sorting by making setting this configuration to `False`.

4.1.4 Custom Commands

Due to the fact that cmake is a macro language, *cmake-format* is, by necessity, a *semantic* source code formatter. In general it tries to make smart formatting decisions based on the meaning of arguments in an otherwise unstructured list of arguments in a cmake statement. *cmake-format* can intelligently format your custom commands, but you will need to tell it how to interpret your arguments.

Currently, you can do this by adding your command specifications to the *additional_commands* configuration variables, e.g.:

```
# Additional FLAGS and KWARGS for custom commands
additional_commands = {
  "foo": {
    "pargs": 2,
    "flags": ["BAR", "BAZ"],
    "kwargs": {
      "HEADERS": '*',
      "SOURCES": '*',
      "DEPENDS": '*',
    }
  }
}
```

The format is a nested dictionary mapping statement names (dictionary keys) to argument specifications. For the example specification above, the custom command would look something like this:

```
foo(hello world
    HEADERS a.h b.h c.h d.h
    SOURCES a.cc b.cc c.cc d.cc
    DEPENDS flub buzz bizz
    BAR BAZ)
```

4.2 Usage

Basic usage for `cmake-format` is:

```
usage:
cmake-format [-h]
             [--dump-config {yaml,json,python} | -i | -o OUTFILE_PATH]
             [-c CONFIG_FILE]
             infilepath [infilepath ...]

Parse cmake listfiles and format them nicely.

Formatting is configurable by providing a configuration file. The configuration
file can be in json, yaml, or python format. If no configuration file is
specified on the command line, cmake-format will attempt to find a suitable
configuration for each ``inputpath`` by checking recursively checking it's
parent directory up to the root of the filesystem. It will return the first
file it finds with a filename that matches '\.?cmake-format(.yaml|.json|.py)'.

cmake-format can spit out the default configuration for you as starting point
for customization. Run with `--dump-config [yaml|json|python]`.

positional arguments:
  infilepaths

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -l {error,warning,info,debug}, --log-level {error,warning,info,debug}
  --dump-config [{yaml,json,python}]
                        If specified, print the default configuration to
                        stdout and exit
  --dump {lex,parse,layout,markup}
```

(continues on next page)

(continued from previous page)

```
  --no-help             When used with --dump-config, will omit helptext
                        comments in the output
  --no-default          When used with --dump-config, will omit any unmodified
                        configuration value.
  -i, --in-place
  --check               Exit with status code 0 if formatting would not change
                        file contents, or status code 1 if it would
  -o OUTFILE_PATH, --outfile-path OUTFILE_PATH
                        Where to write the formatted file. Default is stdout.
  -c CONFIG_FILES [CONFIG_FILES ...], --config-files CONFIG_FILES [CONFIG_FILES ...]
                        path to configuration file(s)
```

Nearly all *Configuration* options are also available as command line options:

```
Options affecting listfile parsing:
  --vartags [VARTAGS [VARTAGS ...]]
                        Specify variable tags.
  --proptags [PROPTAGS [PROPTAGS ...]]
                        Specify property tags.

Options affecting formatting.:
  --line-width LINE_WIDTH
                        How wide to allow formatted cmake files
  --tab-size TAB_SIZE   How many spaces to tab for indent
  --max-subgroups-hwrap MAX_SUBGROUPS_HWRAP
                        If an argument group contains more than this many sub-
                        groups (parg or kwarg groups) then force it to a
                        vertical layout.
  --max-pargs-hwrap MAX_PARGS_HWRAP
                        If a positional argument group contains more than this
                        many arguments, then force it to a vertical layout.
  --max-rows-cmdline MAX_ROWS_CMDLINE
                        If a cmdline positional group consumes more than this
                        many lines without nesting, then invalidate the layout
                        (and nest)
  --separate-ctrl-name-with-space [SEPARATE_CTRL_NAME_WITH_SPACE]
                        If true, separate flow control names from their
                        parentheses with a space
  --separate-fn-name-with-space [SEPARATE_FN_NAME_WITH_SPACE]
                        If true, separate function names from parentheses with
                        a space
  --dangle-parens [DANGLE_PARENS]
                        If a statement is wrapped to more than one line, than
                        dangle the closing parenthesis on its own line.
  --dangle-align {prefix,prefix-indent,child,off}
                        If the trailing parenthesis must be 'dangled' on its
                        on line, then align it to this reference: `prefix`:
                        the start of the statement, `prefix-indent`: the start
                        of the statement, plus one indentation level, `child`:
                        align to the column of the arguments
  --min-prefix-chars MIN_PREFIX_CHARS
                        If the statement spelling length (including space and
                        parenthesis) is smaller than this amount, then force
                        reject nested layouts.
  --max-prefix-chars MAX_PREFIX_CHARS
                        If the statement spelling length (including space and
```

(continues on next page)

(continued from previous page)

```
                        parenthesis) is larger than the tab width by more than
                        this amount, then force reject un-nested layouts.
  --max-lines-hwrap MAX_LINES_HWRAP
                        If a candidate layout is wrapped horizontally but it
                        exceeds this many lines, then reject the layout.
  --line-ending {windows,unix,auto}
                        What style line endings to use in the output.
  --command-case {lower,upper,canonical,unchanged}
                        Format command names consistently as 'lower' or
                        'upper' case
  --keyword-case {lower,upper,unchanged}
                        Format keywords consistently as 'lower' or 'upper'
                        case
  --always-wrap [ALWAYS_WRAP [ALWAYS_WRAP ...]]
                        A list of command names which should always be wrapped
  --enable-sort [ENABLE_SORT]
                        If true, the argument lists which are known to be
                        sortable will be sorted lexicographicall
  --autosort [AUTOSORT]
                        If true, the parsers may infer whether or not an
                        argument list is sortable (without annotation).
  --require-valid-layout [REQUIRE_VALID_LAYOUT]
                        By default, if cmake-format cannot successfully fit
                        everything into the desired linewidth it will apply
                        the last, most agressive attempt that it made. If this
                        flag is True, however, cmake-format will print error,
                        exit with non-zero status code, and write-out nothing

Options affecting comment reflow and formatting.:
  --bullet-char BULLET_CHAR
                        What character to use for bulleted lists
  --enum-char ENUM_CHAR
                        What character to use as punctuation after numerals in
                        an enumerated list
  --first-comment-is-literal [FIRST_COMMENT_IS_LITERAL]
                        If comment markup is enabled, don't reflow the first
                        comment block in each listfile. Use this to preserve
                        formatting of your copyright/license statements.
  --literal-comment-pattern LITERAL_COMMENT_PATTERN
                        If comment markup is enabled, don't reflow any comment
                        block which matches this (regex) pattern. Default is
                        `None` (disabled).
  --fence-pattern FENCE_PATTERN
                        Regular expression to match preformat fences in
                        comments default= ``r'^\s*([`~]{3}[`~]*)(.*)$'``
  --ruler-pattern RULER_PATTERN
                        Regular expression to match rulers in comments
                        default= ``r'^\s*[^\w\s]{3}.*[^\w\s]{3}$'``
  --explicit-trailing-pattern EXPLICIT_TRAILING_PATTERN
                        If a comment line matches starts with this pattern
                        then it is explicitly a trailing comment for the
                        preceeding argument. Default is '#<'
  --hashruler-min-length HASHRULER_MIN_LENGTH
                        If a comment line starts with at least this many
                        consecutive hash characters, then don't lstrip() them
                        off. This allows for lazy hash rulers where the first
                        hash char is not separated by space
```

(continues on next page)

(continued from previous page)

```
  --canonicalize-hashrulers [CANONICALIZE_HASHRULERS]
                        If true, then insert a space between the first hash
                        char and remaining hash chars in a hash ruler, and
                        normalize its length to fill the column
  --enable-markup [ENABLE_MARKUP]
                        enable comment markup parsing and reflow

Options affecting the linter:
  --disabled-codes [DISABLED_CODES [DISABLED_CODES ...]]
                        a list of lint codes to disable
  --function-pattern FUNCTION_PATTERN
                        regular expression pattern describing valid function
                        names
  --macro-pattern MACRO_PATTERN
                        regular expression pattern describing valid macro
                        names
  --global-var-pattern GLOBAL_VAR_PATTERN
                        regular expression pattern describing valid names for
                        variables with global scope
  --internal-var-pattern INTERNAL_VAR_PATTERN
                        regular expression pattern describing valid names for
                        variables with global scope (but internal semantic)
  --local-var-pattern LOCAL_VAR_PATTERN
                        regular expression pattern describing valid names for
                        variables with local scope
  --private-var-pattern PRIVATE_VAR_PATTERN
                        regular expression pattern describing valid names for
                        privatedirectory variables
  --public-var-pattern PUBLIC_VAR_PATTERN
                        regular expression pattern describing valid names for
                        publicdirectory variables
  --keyword-pattern KEYWORD_PATTERN
                        regular expression pattern describing valid names for
                        keywords used in functions or macros
  --max-conditionals-custom-parser MAX_CONDITIONALS_CUSTOM_PARSER
                        In the heuristic for C0201, how many conditionals to
                        match within a loop in before considering the loop a
                        parser.
  --min-statement-spacing MIN_STATEMENT_SPACING
                        Require at least this many newlines between statements
  --max-statement-spacing MAX_STATEMENT_SPACING
                        Require no more than this many newlines between
                        statements
  --max-returns MAX_RETURNS
  --max-branches MAX_BRANCHES
  --max-arguments MAX_ARGUMENTS
  --max-localvars MAX_LOCALVARS
  --max-statements MAX_STATEMENTS

Options affecting file encoding:
  --emit-byteorder-mark [EMIT_BYTEORDER_MARK]
                        If true, emit the unicode byte-order mark (BOM) at the
                        start of the file
  --input-encoding INPUT_ENCODING
                        Specify the encoding of the input file. Defaults to
                        utf-8
  --output-encoding OUTPUT_ENCODING
```

(continues on next page)

(continued from previous page)

```
                        Specify the encoding of the output file. Defaults to
                        utf-8. Note that cmake only claims to support utf-8 so
                        be careful when using anything else
```

4.3 Example

Will turn this:

```
# The following multiple newlines should be collapsed into a single newline




cmake_minimum_required(VERSION 2.8.11)
project(cmake_format_test)

# This multiline-comment should be reflowed
# into a single comment
# on one line

# This comment should remain right before the command call.
# Furthermore, the command call should be formatted
# to a single line.
add_subdirectories(foo bar baz
  foo2 bar2 baz2)

# This very long command should be wrapped
set(HEADERS very_long_header_name_a.h very_long_header_name_b.h very_long_header_name_
↪c.h)

# This command should be split into one line per entry because it has a long argument␣
↪list.
set(SOURCES source_a.cc source_b.cc source_d.cc source_e.cc source_f.cc source_g.cc␣
↪source_h.cc)

# The string in this command should not be split
set_target_properties(foo bar baz PROPERTIES COMPILE_FLAGS "-std=c++11 -Wall -Wextra")

# This command has a very long argument and can't be aligned with the command
# end, so it should be moved to a new line with block indent + 1.
some_long_command_name("Some very long argument that really needs to be on the next␣
↪line.")

# This situation is similar but the argument to a KWARG needs to be on a
# newline instead.
set(CMAKE_CXX_FLAGS "-std=c++11 -Wall -Wno-sign-compare -Wno-unused-parameter -xx")

set(HEADERS header_a.h header_b.h # This comment should
                                  # be preserved, moreover it should be split
                                  # across two lines.
    header_c.h header_d.h)


# This part of the comment should
```

(continues on next page)

(continued from previous page)

```
# be formatted
# but...
# cmake-format: off
# This bunny should remain untouched:
# .
# | |
#  ()
#  c( uu}
# cmake-format: on
#          while this part should
#          be formatted again

# This is a paragraph
#
# This is a second paragraph
#
# This is a third paragraph

# This is a comment
# that should be joined but
# TODO(josh): This todo should not be joined with the previous line.
# NOTE(josh): Also this should not be joined with the todo.

if(foo)
if(sbar)
# This comment is in-scope.
add_library(foo_bar_baz foo.cc bar.cc # this is a comment for arg2
                                      # this is more comment for arg2, it should be␣
↪joined with the first.
    baz.cc) # This comment is part of add_library

other_command(some_long_argument some_long_argument) # this comment is very long and␣
↪gets split across some lines

other_command(some_long_argument some_long_argument some_long_argument) # this␣
↪comment is even longer and wouldn't make sense to pack at the end of the command so␣
↪it gets it's own lines
endif()
endif()


# This very long command should be broken up along keyword arguments
foo(nonkwarg_a nonkwarg_b HEADERS a.h b.h c.h d.h e.h f.h SOURCES a.cc b.cc d.cc␣
↪DEPENDS foo bar baz)

# This command uses a string with escaped quote chars
foo(some_arg some_arg "This is a \"string\" within a string")

# This command uses an empty string
foo(some_arg some_arg "")

# This command uses a multiline string
foo(some_arg some_arg "
    This string is on multiple lines
")

# No, I really want this to look ugly
```

(continues on next page)

(continued from previous page)

```
# cmake-format: off
add_library(a b.cc
  c.cc         d.cc
           e.cc)
# cmake-format: on
```

into this:

```
# The following multiple newlines should be collapsed into a single newline

cmake_minimum_required(VERSION 2.8.11)
project(cmake_format_test)

# This multiline-comment should be reflowed into a single comment on one line

# This comment should remain right before the command call. Furthermore, the
# command call should be formatted to a single line.
add_subdirectories(foo bar baz foo2 bar2 baz2)

# This very long command should be wrapped
set(HEADERS very_long_header_name_a.h very_long_header_name_b.h
            very_long_header_name_c.h)

# This command should be split into one line per entry because it has a long
# argument list.
set(SOURCES
    source_a.cc
    source_b.cc
    source_d.cc
    source_e.cc
    source_f.cc
    source_g.cc
    source_h.cc)

# The string in this command should not be split
set_target_properties(foo bar baz PROPERTIES COMPILE_FLAGS
                                             "-std=c++11 -Wall -Wextra")

# This command has a very long argument and can't be aligned with the command
# end, so it should be moved to a new line with block indent + 1.
some_long_command_name(
  "Some very long argument that really needs to be on the next line.")

# This situation is similar but the argument to a KWARG needs to be on a newline
# instead.
set(CMAKE_CXX_FLAGS
    "-std=c++11 -Wall -Wno-sign-compare -Wno-unused-parameter -xx")

set(HEADERS
    header_a.h header_b.h # This comment should be preserved, moreover it should
                          # be split across two lines.
    header_c.h header_d.h)

# This part of the comment should be formatted but...
# cmake-format: off
# This bunny should remain untouched:
# .
```

(continues on next page)

(continued from previous page)

```
# | |
#  ()
#  c( uu}
# cmake-format: on
# while this part should be formatted again

# This is a paragraph
#
# This is a second paragraph
#
# This is a third paragraph

# This is a comment that should be joined but
# TODO(josh): This todo should not be joined with the previous line.
# NOTE(josh): Also this should not be joined with the todo.

if(foo)
  if(sbar)
    # This comment is in-scope.
    add_library(
      foo_bar_baz
      foo.cc bar.cc # this is a comment for arg2 this is more comment for arg2,
                    # it should be joined with the first.
      baz.cc) # This comment is part of add_library

    other_command(
      some_long_argument some_long_argument) # this comment is very long and
                                             # gets split across some lines

    other_command(
      some_long_argument some_long_argument some_long_argument) # this comment
                                                                # is even longer
                                                                # and wouldn't
                                                                # make sense to
                                                                # pack at the
                                                                # end of the
                                                                # command so it
                                                                # gets it's own
                                                                # lines
  endif()
endif()

# This very long command should be broken up along keyword arguments
foo(nonkwarg_a nonkwarg_b
    HEADERS a.h b.h c.h d.h e.h f.h
    SOURCES a.cc b.cc d.cc
    DEPENDS foo
    bar baz)

# This command uses a string with escaped quote chars
foo(some_arg some_arg "This is a \"string\" within a string")

# This command uses an empty string
foo(some_arg some_arg "")

# This command uses a multiline string
foo(some_arg some_arg "
```

(continues on next page)

(continued from previous page)

```
    This string is on multiple lines
")

# No, I really want this to look ugly
# cmake-format: off
add_library(a b.cc
  c.cc          d.cc
            e.cc)
# cmake-format: on
```

4.4 Formatting Algorithm

The formatter works by attempting to select an appropriate `position` and `wrap` (collectively referred to as a "layout") for each node in the layout tree. Positions are represented by `(row, col)` pairs and the wrap dictates how childen of that node are positioned.

4.4.1 Wrapping

`cmake-format` implements three styles of wrapping. The default wrapping for all nodes is horizontal wrapping. If horizontal wrapping fails to emit an admissible layout, then a node will advance to either vertical wrapping or nested wrapping (which one depends on the type of node).

Horizontal Wrapping

Horizontal wrapping is like "word wrap". Each child is assigned a position immediately following it's predecessor, so long as that child fits in the remaining space up to the column limit. Otherwise the child is moved to the next line:

```
|                          |<- col-limit
|        |
|     |
|           |
```

Note that a line comment can force an early newline:

```
|                          |<- col-limit
|   #           |
|                  |
|     |
|           |
```

Note that wrapping happens at the depth of the layout tree, so if we have multiple groups of multiple arguments each, then each group will be placed as if it were a single unit:

```
|                                  |<- col-limit
| ( ) ( )         |
| (   ) |
```

Groups may be parenthetical groups (as above) or keyword groups:

```
|                                  |<- col-limit
|              |
|      |
```

or any other grouping assigned by the parser.

In the event that a subgroup cannot be packed within a single line of full column width, it will be wrapped internally, and the next group placed on the next line:

```
|                                  |<- col-limit
|                |
|        |
|                            |
|                   |
```

In particular the following is never a valid packing (where the two groups are siblings) in the layout tree:

```
|                                  |<- col-limit
|                     |
|                   |
|                        |
```

Vertical Wrapping

Vertical wrapping assigns each child to the next row:

Again, note that this happens at the depth of the layout tree. In particular children may be wrapped horizontally within the subtrees:

```
|            |<- col-limit
|                 |
|        |
|                |
|              |
|                    |
```

Nesting

Nesting places children in a column which is one `tab_width` to the right of the parent node's position, and one line below. For example:

```
|                          |<- col-limit
|                     |
|              |
|         |
|            |
```

In a more deeply nested layout tree, we might see the following:

```
|                              |<- col-limit
|                         |
|                         |
```

(continues on next page)

(continued from previous page)

```
|                  |
|              |
|                 |
|                           |
|                       |
|                         |
|                       |
```

Depending on how `cmake-format` is configured, elements at different depths may be nested differently. For example:

```
|                              |<- col-limit
|                          |
|            |
|             |
|                 |
|                   |
|                |
```

Note that the only nodes that can nest are `STATEMENT` and `KWARGGROUP` nodes. These nodes necessarily only have one child, an `ARGGROUP` node. Therefore there really isn't a notion of "wrapping" for these nodes.

4.4.2 Formatting algorithm

For top-level nodes in the layout tree (i.e. `COMMENT`, `STATEMENT`, `BODY`, `FLOW_CONTROL`, etc...) the positioning is straight forward and these nodes are laid out in a single pass. Each child is positioned on the first line after the output cursor of it's predecessor, and at a column `config.format.tab_size` to the right of it's parent.

`STATEMENTS` however, are laid out over several passes until the text for that subtree is accepted. Each pass is governed by a specification mapping pass number to a wrap decision (i.e. a boolean indicating whether or not to wrap vertical or nest children)

Layout Passes

The current algorithm works in a kind of top-down refinement. When a node is laid out by calling it's `reflow()` method, it is informed of its parent's current pass number (`passno`). It then iterates through its own `passno` from zero up to it's parent's `passno` and terminates at the first admissible layout. Note that within the layout of the node itself, it's current `passno` can only affect its `wrap` decision. However, because each of its children will advance through their own passes, the overall layout of a subtree between two different passes may change, even if the node at the subtree root didn't change it's `wrap` decision between those passes.

This approach seems to work well even for *deeply nested* or *complex* statements.

Newline decision

When a node is in horizontal layout mode (`wrap=False`), there are a couple of reasons why the algorithm might choose to insert a newline between two of it's children.

1. If a token would overflow the column limit, insert a newline (e.g. the usual notion of wrapping)
2. If the token is the last token before a closing parenthesis, and the token plus the parenthesis would overflow the column limit, then insert a newline.

3. If a token is preceeded by a line comment, then the token cannot be placed on the same line as the comment (or it will become part of the comment) so a newline is inserted between them.
4. If a token is a line comment which is not associated with an argument (e.g. it is a "free" comment at the current scope) then it will not be placed on the same line as a preceeding argument token. If it was, then subsequent parses would associate this comment with that argument. In such a case, a newline is inserted between the preceeding argument and the line comment.
5. If the node is an interior node, and one of it's children is internally wrapped (i.e. consumes more than two lines) then it will not be placed on the same line as another node. In such a case a newlines is inserted.
6. If the node is an interior node and a child fails to find an admissible layout at the current cursor, a newline is inserted and a new layout attempt is made for the child.

Admissible layouts

There are a couple of reasons why a layout may be deemed inadmissible:

1. If the bounding box of a node overflows the column limit
2. If a node is horizontally wrapped at the current `passno` but consumes more than `max_lines_hwrap` lines
3. If the node is horizontally wrapped at the current `passno` but the node path is marked as `always_wrap`

Comments

A (multi-line) comment on the last row does not contribute to the height for the purposes of this thresholding, but one on any other line does. Another way to say this is that comments are excluded from the size computation, but their influence on other argument is not:

```
# This content is 3 lines tall
foobarbaz_hello(
   argument_one argument_two # this comment is two lines long and it
                             # forces the next argument onto line three
   argument_three argument_four)

# This is only 2 lines tall
foobarbaz_hello(
   argument_one argument_two argument_three
   argument_four # this comment is two lines long and wraps but it
   # has no contribution to the size of the content.
```

Dealing with comments during horizontal wrapping can be a little tricky. They definitely induce a newline at their termination, but they may also predicate a newline in front of the commented argument. See the examples in *Case Studies/Comments*. We don't necessarily need to deal with this right now. The user can always force the issue by adding some comment strings that force a comment width, like this:

```
set(HEADERS header_a.h header_b.h header_c.h
    header_d.h # This comment is pretty long and if it's argument is close
               # to the edge of the column then the comment gets wrapped
               # very poorly ------------------------
    header_e.h header_f.h)
```

The string of dashes `------------------------` is long enough that the minimum width of the comment block is given by:

```
# This comment is pretty
# long and if it's
# argument is close to the
# edge of the column then
# the comment gets wrapped
# very poorly
# ------------------------
```

Which would preclude it from being crammed into the right-most slot.

4.5 Case Studies

This is a collection of interesting cases that are illustrative of different concepts for formatting options.

4.5.1 Positional Arguments

Lots of short args, looks good all on one line:

```
add_subdirectories(foo bar baz foo2 bar2 baz2)
```

Also doesn't look too bad when wrapped horizontally:

```
add_subdirectories(
    foo bar baz foo2 bar2 baz2 foo3 bar3 baz3 foo4 bar4 baz4 foo5 bar5 baz5
    foo6 bar6 baz6 foo7 bar7 baz7 foo8 bar8 baz8 foo9 bar9 baz9)
```

Though probably matches expectations better if it is wrapped vertically, even if it does look like shit:

```
add_subdirectories(
    foo
    bar
    baz
    foo2
    bar2
    baz2
    foo3
    bar3
    baz3
    foo4
    bar4
    baz4
    foo5
    bar5
    baz5
    foo6
    bar6
    baz6
    foo7
    bar7
    baz7
    foo8
    bar8
    baz8
    foo9
```

(continues on next page)

(continued from previous page)

```
    bar9
    baz9)
```

Just a couple of long args, looks bad wrapped horizontally:

```
set(HEADERS very_long_header_name_a.h very_long_header_name_b.h
    very_long_header_name_c.h)
```

and looks better wrapped vertically, horizontally nested:

```
set(HEADERS
    very_long_header_name_a.h
    very_long_header_name_b.h
    very_long_header_name_c.h)
```

also looks pretty good packed after the first argument:

```
set(HEADERS very_long_header_name_a.h
            very_long_header_name_b.h
            very_long_header_name_c.h)
```

or possibly nested:

```
   set(HEADERS
         very_long_header_name_a.h
         very_long_header_name_b.h
         very_long_header_name_c.h)

   set(
     HEADERS
       very_long_header_name_a.h
       very_long_header_name_b.h
       very_long_header_name_c.h)

but this starts to look a little inconsistent when other arguments are
used::

   set(
     HEADERS PARENT_SCOPE
       very_long_header_name_a.h
       very_long_header_name_b.h
       very_long_header_name_c.h)
```

Lots of medium-length args, looks good vertical, horizontally nested:

```
set(SOURCES
    source_a.cc
    source_b.cc
    source_d.cc
    source_e.cc
    source_f.cc
    source_g.cc)
```

Interestingly, if the PARGGROUP list is one level deeper, the distinction between what looks good is a little blurrier. With lots of short names, I think it looks good both ways:

```
# This very long command should be broken up along keyword arguments
foo(nonkwarg_a nonkwarg_b
    HEADERS a.h b.h c.h d.h e.h f.h
    SOURCES a.cc b.cc d.cc
    DEPENDS foo
    bar baz)
```

versus:

```
# This very long command should be broken up along keyword arguments
foo(nonkwarg_a nonkwarg_b
    HEADERS a.h
            b.h
            c.h
            d.h
            e.h
            f.h
    SOURCES a.cc b.cc d.cc
    DEPENDS foo
    bar baz)
```

though it does seems like the same rules can be applied here if we include some configuration for both number of arguments and length of arguments.

4.5.2 Keyword Arguments

When chidren include both positionals and keyword arguments, it looks good with each group wrapped vertically, while grand children are wrapped horizontally:

```
set_target_properties(
  foo bar baz
  PROPERTIES COMPILE_FLAGS "-std=c++11 -Wall -Wextra")
```

I think it makes sense even with just a single positional, even if it is a little sparsish:

```
set_target_properties(
  foo
  PROPERTIES COMPILE_FLAGS "-std=c++11 -Wall -Wextra -Wfoobarbazoption")
```

Another option is to put the "short" positional arguments on the first line, but while this might look good in some cases I think the cost of inconstency is high. This also introduces some readability issues in that the positional arguments are easy to overlook in this layout:

```
set_target_properties(foo
  PROPERTIES COMPILE_FLAGS "-std=c++11 -Wall -Wextra -Wfoobarbazoption")
```

Though in the case that it could all be on one line so we should do that:

```
set_target_properties(
    foo PROPERTIES COMPILE_FLAGS "-std=c++11 -Wall -Wextra")
```

And `set_target_properties` is actually kind of special because `PROPERITES` doesn't act much like a keyword. It's more of a separator after which we parse thins in pairs. This might look nicer, but I'm not sure we can really make it fit with other rules:

```
set_target_properties(
  foo PROPERTIES
  COMPILE_FLAGS "-std=c++11 -Wall -Wextra -Wfoobarbazoption"
  LINKER_FLAGS "-fpic -someotheroption")
```

Though with custom parse logic we might be able to do so. The custom parser would include `PROPERTIES` in the positional arguments and would label the first half of each pair as a `KEYWORD` node.

4.5.3 set()

`set()` is somewhat of a special case due to things like this:

```
set(args
    OUTPUT fizz.txt
    COMMAND foo --bar --baz
    DEPENDS foo.txt bar.txt baz.txt
    COMMENT "This is my rule"
    BYPRODUCTS buzz.txt)
add_custom_command(${arg})
```

I suppose it's not the worse case that we just horizontally wrap it by default in which case the user can enforce wrapping with line comments. It would be nice if they could somehow annotate it though, like with a comment `# cmf: as=add_custom_command`. That sounds complicated though. One really fancy solution would be to scan for potential kwargs, then try to match against a known command based on the registry.

Note that `set()` isn't the only command like this. There are likely to be other commands, specifically wrapper commands, that might take an unstructured argument list which becomes structured under the hood.

4.5.4 Comments

Argument comments can get a little tricky, because this looks bad:

```
set(HEADERS header_a.h header_b.h header_c.h header_d.h # This comment is
                                                       # pretty long and
                                                       # if it's argument
                                                       # is close to the
                                                       # edge of the column
                                                       # then the comment
                                                       # gets wrapped very
                                                       # poorly
    header_e.h header_f.h)
```

and this looks good:

```
set(HEADERS
    header_a.h
    header_b.h
    header_c.h
    header_d.h # This comment is pretty long and if it's argument is close
               # to the edge of the column then the comment gets wrapped
               # very poorly
    header_e.h
    header_f.h)
```

but this also looks acceptable and I could imagine some organization choosing to go this route with their style configuration:

```
set(HEADERS header_a.h header_b.h header_c.h
    header_d.h # This comment is pretty long and if it's argument is close
               # to the edge of the column then the comment gets wrapped
               # very poorly
    header_e.h header_f.h)
```

So I'm not sure that the presence of a line comment should necessarily predicate a vertical wrapping. Rather, I think the choice of wrapping strategy should be independant of the presence of a comment. In the case of horizontal wrapping though, we need some kind of threshold or score to determine when a comment has gotten "too smooshed" and the whole thing should move to the next line. In the example above:

```
# option A:
set(HEADERS header_a.h header_b.h header_c.h header_d.h # This comment is
                                                       # pretty long and
                                                       # if it's argument
                                                       # is close to the
                                                       # edge of the column
                                                       # then the comment
                                                       # gets wrapped very
                                                       # poorly
    header_e.h header_f.h)

# option B:
set(HEADERS header_a.h header_b.h header_c.h
    header_d.h # This comment is pretty long and if it's argument is close
               # to the edge of the column then the comment gets wrapped
               # very poorly
    header_e.h header_f.h)
```

"Option A" lays out the comment on eight lines while "option B" lays out the comment in three lines. I'm not sure what the threshold should be for choosing one over the other. Should it be based on how many lines the comment is, or how much whitespace we introduce due to it? In "Option A" we introduce seven lines of whitespace between consecutive rows of arguments whereas in "Option B" we only add two. Should it be based on aspect ratio?

And, honestly, "Option A" isn't all that bad. I'm not sure it would cross everyones threshold for inducing a wrap.

For this particular example I think the best looking layout is the vertical wrapping, but we don't want the presence of a line comment to automatically indluce vertical wrapping. For instance in this example, we definitely want to keep horizontal wrapping, we just want the line comment to induce an early wrap:

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/foobar_doc.stamp
  COMMAND sphinx-build -M html #
          ${CMAKE_CURRENT_SOURCE_DIR} #
          ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND touch ${CMAKE_CURRENT_BINARY_DIR}/foobar_doc.stamp
  DEPENDS ${foobar_docs}
  WORKING_DIRECTORY ${CMAKE_SOURCE_DIR})
```

One comprimise solution is to change the behavior of the line comment depending on the nature of the PARGGROUP. The parser can tag each PARGGROUP with it's `default_wrap` (either "horizontal" or "vertical"). Then, when a wrap is required the default wrap can be used. A wrap might be required due to:

- arguments overflow the column width
- exceed threshold in number or size of arguments

- presence of a line comment

This comprimise is the reason the previous version of `cmake-format` had a distinct `HPACK` wrapping algorithm. It allowed us a configuration where all wrapping would be vertical wrapping.

A second comprimise solution, which is compatible with the previous solution, is to make the wrapping tunable by an annotation comment. For instance:

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/foobar_doc.stamp
  COMMAND sphinx-build -M html # cmf:hwrap
          ${CMAKE_CURRENT_SOURCE_DIR} #
          ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND touch ${CMAKE_CURRENT_BINARY_DIR}/foobar_doc.stamp
  DEPENDS ${foobar_docs}
  WORKING_DIRECTORY ${CMAKE_SOURCE_DIR})
```

where the `hwrap` annotation would change the default behavior of the line comment from inducing vertical wrapping to inducing a newline within vertical wrapping. If the annotation syntax requires too many characters, we could use something like double-hash `##`, hash-h `#h`, or hash-v (``#v`) for this purpose. This could be a sandard "microtag" format including the ability to set the list sortable. For example: `#v,s` would be "vertical, sortable"

Another interesting case is if we have an argument comment on a keyword argument, or a prefix group. For example:

```
set(foobarbaz # comment about foobarbaz
    value_one value_two value_three value_four value_five value_six
    value_seven value_eight)
```

Should that be formatted as above, or as:

```
set(foobarbaz # comment about foobarbaz
              value_one value_two value_three value_four value_five
              value_six value_seven value_eight)
```

If we're already formatting set as:

```
set(foobarbaz value_one value_two value_three value_four value_five
              value_six value_seven value_eight)
```

4.5.5 Nesting

When logic get's nested, the need to nest after long command names becomes more apparent:

```
if(foo)
  if(sbar)
    # This comment is in-scope.
    add_library(
      foo_bar_baz
      foo.cc
      bar.cc # this is a comment for arg2 this is more comment for
             # arg2, it should be joined with the first.
      baz.cc) # This comment is part of add_library

    other_command(
        some_long_argument some_long_argument) # this comment is very
                                               # long and gets split
                                               # across some lines
```

(continues on next page)

(continued from previous page)

```
    other_command(some_long_argument some_long_argument some_long_argument)
    # this comment is even longer and wouldn't make sense to pack at the
    # end of the command so it gets it's own lines
    endif()
  endif()
```

Another good example is `add_custom_comand()`:

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/foobar_doc.stamp
  COMMAND sphinx-build -M html ${CMAKE_CURRENT_SOURCE_DIR}
          ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND touch ${CMAKE_CURRENT_BINARY_DIR}/foobar_doc.stamp
  DEPENDS ${foobar_docs}
  WORKING_DIRECTORY ${CMAKE_SOURCE_DIR})
```

But note the tricky bit here. I think we definitely want the COMMAND ARGGOUP (which is a single PARGGROUP) to be horizontally wrapped.

There are also some commands with second (or more) levels of keyword arguments, and it's not clear if the nesting rules are best applied top-down:

```
install(
  TARGETS foo bar baz
  ARCHIVE DESTINATION <dir>
          PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
          CONFIGURATIONS Debug Release
          COMPONENT foo-component
          OPTIONAL EXCLUDE_FROM_ALL NAMELINK_SKIP
  LIBRARY DESTINATION <dir>
          PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
          CONFIGURATIONS Debug Release
          COMPONENT foo-component
          OPTIONAL EXCLUDE_FROM_ALL NAMELINK_SKIP
  RUNTIME DESTINATION <dir>
          PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
          CONFIGURATIONS Debug Release
          COMPONENT foo-component
          OPTIONAL EXCLUDE_FROM_ALL NAMELINK_SKIP)
```

Or bottom-up:

```
install(
  TARGETS foo bar baz
  ARCHIVE
    DESTINATION <dir>
    PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
    CONFIGURATIONS Debug Release
    COMPONENT foo-component
    OPTIONAL EXCLUDE_FROM_ALL NAMELINK_SKIP
  LIBRARY
    DESTINATION <dir>
    PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
    CONFIGURATIONS Debug Release
    COMPONENT foo-component
    OPTIONAL EXCLUDE_FROM_ALL NAMELINK_SKIP
```

(continues on next page)

(continued from previous page)

```
  RUNTIME
    DESTINATION <dir>
    PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE
    CONFIGURATIONS Debug Release
    COMPONENT foo-component
    OPTIONAL EXCLUDE_FROM_ALL NAMELINK_SKIP)
```

4.5.6 Conditionals

Treating boolean operators as keyword arguments works pretty well, so long as we treat parenthetical groups as a single unit:

```
set(matchme "_DATA_\\|_CMAKE_\\|INTRA_PRED\\|_COMPILED\\|_HOSTING\\|_PERF_\\|CODER_")
if(("${var}" MATCHES "_TEST_" AND NOT "${var}" MATCHES "${matchme}")
   OR (CONFIG_AV1_ENCODER
       AND CONFIG_ENCODE_PERF_TESTS
       AND "${var}" MATCHES "_ENCODE_PERF_TEST_")
   OR (CONFIG_AV1_DECODER
       AND CONFIG_DECODE_PERF_TESTS
       AND "${var}" MATCHES "_DECODE_PERF_TEST_")
   OR (CONFIG_AV1_ENCODER AND "${var}" MATCHES "_TEST_ENCODER_")
   OR (CONFIG_AV1_DECODER AND "${var}" MATCHES "_TEST_DECODER_"))
  list(APPEND aom_test_source_vars ${var})
endif()
```

I don't think there's any reason to add structure for the internal operators like `MATCHES`. In particular children of a boolean operator can be simple positional argument groups (horizontally-wrapped). We can tag the internal operator as a keyword but we don't need to create a KWARGGROUP for it.

4.5.7 Internally Wrapped Positionals

The third kwarg (AND) in this statement looks bad because it is Internally wrapped. The second option looks better:

```
set(matchme "_DATA_\|_CMAKE_\|INTRA_PRED\|_COMPILED\|_HOSTING\|_PERF_\|CODER_")
if(("${var}" MATCHES "_TEST_" AND NOT "${var}" MATCHES "${matchme}")
   OR (CONFIG_AV1_ENCODER AND CONFIG_ENCODE_PERF_TESTS AND "${var}" MATCHES
                                                          "_ENCODE_PERF_TEST_"
      ))
  list(APPEND aom_test_source_vars ${var})
endif()

set(matchme "_DATA_\|_CMAKE_\|INTRA_PRED\|_COMPILED\|_HOSTING\|_PERF_\|CODER_")
if(("${var}" MATCHES "_TEST_" AND NOT "${var}" MATCHES "${matchme}")
   OR (CONFIG_AV1_ENCODER
       AND CONFIG_ENCODE_PERF_TESTS
       AND "${var}" MATCHES "_ENCODE_PERF_TEST_"))
  list(APPEND aom_test_source_vars ${var})
endif()
```

However, this short `set()` statement looks better if we don't push the internally wrapped argument to the next line:

```
set(sources # cmake-format: sortable
            bar.cc baz.cc foo.cc)
```

Perhaps the difference is that in the latter case it's going to consume two lines anyway... whereas in the former case it would only consume one line.

4.5.8 Columnized arguments

Some very long statements with a large number of keywords might look nice and organized if we columize the child argument groups. For example:

```
ExternalProject_Add(
    FOO
    PREFIX           ${FOO_PREFIX}
    TMP_DIR          ${TMP_DIR}
    STAMP_DIR        ${FOO_PREFIX}/stamp
    # Download
    DOWNLOAD_DIR     ${DOWNLOAD_DIR}
    DOWNLOAD_NAME    ${FOO_ARCHIVE_FILE_NAME}
    URL              ${STORAGE_URL}/${FOO_ARCHIVE_FILE_NAME}
    URL_MD5          ${FOO_MD5}
    # Patch
    PATCH_COMMAND    ${PATCH_COMMAND} ${PROJECT_SOURCE_DIR}/patch.diff
    # Configure
    SOURCE_DIR       ${SRC_DIR}
    CMAKE_ARGS       ${CMAKE_OPTS}
    # Build
    BUILD_IN_SOURCE  1
    BUILD_BYPRODUCTS ${CUR_COMPONENT_ARTIFACTS}
    # Logging
    LOG_CONFIGURE    1
    LOG_BUILD        1
    LOG_INSTALL      1
)
```

Note what `clang-format` does for these cases. If two consecutive keywords are more than `n` characters different in length, then break columns, which might come out something like this:

```
ExternalProject_Add(
    FOO
    PREFIX    ${FOO_PREFIX}
    TMP_DIR   ${TMP_DIR}
    STAMP_DIR ${FOO_PREFIX}/stamp
    # Download
    DOWNLOAD_DIR  ${DOWNLOAD_DIR}
    DOWNLOAD_NAME ${FOO_ARCHIVE_FILE_NAME}
    URL     ${STORAGE_URL}/${FOO_ARCHIVE_FILE_NAME}
    URL_MD5 ${FOO_MD5}
    # Patch
    PATCH_COMMAND ${PATCH_COMMAND} ${PROJECT_SOURCE_DIR}/patch.diff
    # Configure
    SOURCE_DIR ${SRC_DIR}
    CMAKE_ARGS ${CMAKE_OPTS}
    # Build
    BUILD_IN_SOURCE  1
    BUILD_BYPRODUCTS ${CUR_COMPONENT_ARTIFACTS}
    # Logging
    LOG_CONFIGURE 1
    LOG_BUILD     1
```

(continues on next page)

(continued from previous page)

```
    LOG_INSTALL    1
)
```

As an experimental feature, we could require a tag `# cmf: columnize` to enable this formatting.

4.5.9 Algorithm Ideas and Notes

Layout Passes

Up through version 0.5.2 each node would lay itself out using pass numbers `[0, <parent-passno>]`. This worked pretty well, but actually I would like the nesting to be a little more depth dependant. For example I would like depth 0 (statement) to nest rather early, while I would like higher depths (i.e. KWARGS) to nest later, but go vertical earlier.

One alternative is to have a global `passno` and apply different rules at each pass until things fit, but the probem with this option is that two subtrees might require fastly different passes. We don't want to vertically wrap one all kwargs just because one needs to.

CHAPTER 5

cmake-lint

The `cmake-lint` program will check your listfiles for style violations, common mistakes, and anti-patterns.

5.1 Usage

Basic usage for `cmake-lint` is:

```
usage:
cmake-lint [-h]
           [--dump-config {yaml,json,python} | -o OUTFILE_PATH]
           [-c CONFIG_FILE]
           infilepath [infilepath ...]

Check cmake listfile for lint

positional arguments:
  infilepaths

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -l {error,warning,info,debug}, --log-level {error,warning,info,debug}
  --dump-config [{yaml,json,python}]
                        If specified, print the default configuration to
                        stdout and exit
  -o OUTFILE_PATH, --outfile-path OUTFILE_PATH
                        Write errors to this file. Default is stdout.
  -c CONFIG_FILES [CONFIG_FILES ...], --config-files CONFIG_FILES [CONFIG_FILES ...]
                        path to configuration file(s)
```

Nearly all *Configuration* options are also available as command line options:

```
Options affecting listfile parsing:
  --vartags [VARTAGS [VARTAGS ...]]
                        Specify variable tags.
  --proptags [PROPTAGS [PROPTAGS ...]]
                        Specify property tags.

Options affecting formatting.:
  --line-width LINE_WIDTH
                        How wide to allow formatted cmake files
  --tab-size TAB_SIZE   How many spaces to tab for indent
  --max-subgroups-hwrap MAX_SUBGROUPS_HWRAP
                        If an argument group contains more than this many sub-
                        groups (parg or kwarg groups) then force it to a
                        vertical layout.
  --max-pargs-hwrap MAX_PARGS_HWRAP
                        If a positional argument group contains more than this
                        many arguments, then force it to a vertical layout.
  --max-rows-cmdline MAX_ROWS_CMDLINE
                        If a cmdline positional group consumes more than this
                        many lines without nesting, then invalidate the layout
                        (and nest)
  --separate-ctrl-name-with-space [SEPARATE_CTRL_NAME_WITH_SPACE]
                        If true, separate flow control names from their
                        parentheses with a space
  --separate-fn-name-with-space [SEPARATE_FN_NAME_WITH_SPACE]
                        If true, separate function names from parentheses with
                        a space
  --dangle-parens [DANGLE_PARENS]
                        If a statement is wrapped to more than one line, than
                        dangle the closing parenthesis on its own line.
  --dangle-align {prefix,prefix-indent,child,off}
                        If the trailing parenthesis must be 'dangled' on its
                        on line, then align it to this reference: `prefix`:
                        the start of the statement, `prefix-indent`: the start
                        of the statement, plus one indentation level, `child`:
                        align to the column of the arguments
  --min-prefix-chars MIN_PREFIX_CHARS
                        If the statement spelling length (including space and
                        parenthesis) is smaller than this amount, then force
                        reject nested layouts.
  --max-prefix-chars MAX_PREFIX_CHARS
                        If the statement spelling length (including space and
                        parenthesis) is larger than the tab width by more than
                        this amount, then force reject un-nested layouts.
  --max-lines-hwrap MAX_LINES_HWRAP
                        If a candidate layout is wrapped horizontally but it
                        exceeds this many lines, then reject the layout.
  --line-ending {windows,unix,auto}
                        What style line endings to use in the output.
  --command-case {lower,upper,canonical,unchanged}
                        Format command names consistently as 'lower' or
                        'upper' case
  --keyword-case {lower,upper,unchanged}
                        Format keywords consistently as 'lower' or 'upper'
                        case
  --always-wrap [ALWAYS_WRAP [ALWAYS_WRAP ...]]
                        A list of command names which should always be wrapped
```

(continues on next page)

(continued from previous page)

```
  --enable-sort [ENABLE_SORT]
                        If true, the argument lists which are known to be
                        sortable will be sorted lexicographicall
  --autosort [AUTOSORT]
                        If true, the parsers may infer whether or not an
                        argument list is sortable (without annotation).
  --require-valid-layout [REQUIRE_VALID_LAYOUT]
                        By default, if cmake-format cannot successfully fit
                        everything into the desired linewidth it will apply
                        the last, most agressive attempt that it made. If this
                        flag is True, however, cmake-format will print error,
                        exit with non-zero status code, and write-out nothing

Options affecting comment reflow and formatting.:
  --bullet-char BULLET_CHAR
                        What character to use for bulleted lists
  --enum-char ENUM_CHAR
                        What character to use as punctuation after numerals in
                        an enumerated list
  --first-comment-is-literal [FIRST_COMMENT_IS_LITERAL]
                        If comment markup is enabled, don't reflow the first
                        comment block in each listfile. Use this to preserve
                        formatting of your copyright/license statements.
  --literal-comment-pattern LITERAL_COMMENT_PATTERN
                        If comment markup is enabled, don't reflow any comment
                        block which matches this (regex) pattern. Default is
                        `None` (disabled).
  --fence-pattern FENCE_PATTERN
                        Regular expression to match preformat fences in
                        comments default= ``r'^\s*([`~]{3}[`~]*)(.*)$'``
  --ruler-pattern RULER_PATTERN
                        Regular expression to match rulers in comments
                        default= ``r'^\s*[^\w\s]{3}.*[^\w\s]{3}$'``
  --explicit-trailing-pattern EXPLICIT_TRAILING_PATTERN
                        If a comment line matches starts with this pattern
                        then it is explicitly a trailing comment for the
                        preceeding argument. Default is '#<'
  --hashruler-min-length HASHRULER_MIN_LENGTH
                        If a comment line starts with at least this many
                        consecutive hash characters, then don't lstrip() them
                        off. This allows for lazy hash rulers where the first
                        hash char is not separated by space
  --canonicalize-hashrulers [CANONICALIZE_HASHRULERS]
                        If true, then insert a space between the first hash
                        char and remaining hash chars in a hash ruler, and
                        normalize its length to fill the column
  --enable-markup [ENABLE_MARKUP]
                        enable comment markup parsing and reflow

Options affecting the linter:
  --disabled-codes [DISABLED_CODES [DISABLED_CODES ...]]
                        a list of lint codes to disable
  --function-pattern FUNCTION_PATTERN
                        regular expression pattern describing valid function
                        names
  --macro-pattern MACRO_PATTERN
                        regular expression pattern describing valid macro
```

(continues on next page)

(continued from previous page)

```
                        names
  --global-var-pattern GLOBAL_VAR_PATTERN
                        regular expression pattern describing valid names for
                        variables with global scope
  --internal-var-pattern INTERNAL_VAR_PATTERN
                        regular expression pattern describing valid names for
                        variables with global scope (but internal semantic)
  --local-var-pattern LOCAL_VAR_PATTERN
                        regular expression pattern describing valid names for
                        variables with local scope
  --private-var-pattern PRIVATE_VAR_PATTERN
                        regular expression pattern describing valid names for
                        privatedirectory variables
  --public-var-pattern PUBLIC_VAR_PATTERN
                        regular expression pattern describing valid names for
                        publicdirectory variables
  --keyword-pattern KEYWORD_PATTERN
                        regular expression pattern describing valid names for
                        keywords used in functions or macros
  --max-conditionals-custom-parser MAX_CONDITIONALS_CUSTOM_PARSER
                        In the heuristic for C0201, how many conditionals to
                        match within a loop in before considering the loop a
                        parser.
  --min-statement-spacing MIN_STATEMENT_SPACING
                        Require at least this many newlines between statements
  --max-statement-spacing MAX_STATEMENT_SPACING
                        Require no more than this many newlines between
                        statements
  --max-returns MAX_RETURNS
  --max-branches MAX_BRANCHES
  --max-arguments MAX_ARGUMENTS
  --max-localvars MAX_LOCALVARS
  --max-statements MAX_STATEMENTS

Options affecting file encoding:
  --emit-byteorder-mark [EMIT_BYTEORDER_MARK]
                        If true, emit the unicode byte-order mark (BOM) at the
                        start of the file
  --input-encoding INPUT_ENCODING
                        Specify the encoding of the input file. Defaults to
                        utf-8
  --output-encoding OUTPUT_ENCODING
                        Specify the encoding of the output file. Defaults to
                        utf-8. Note that cmake only claims to support utf-8 so
                        be careful when using anything else
```

5.2 Example

Given the following linty file:

```
# The line is too long and exceeds the default 80 character column limit enforced␣
↪cmake-lint

# This line has trailing whitespace
```

(continues on next page)

(continued from previous page)

```
# This line has the wrong line endings


function(BAD_FUNCTION_NAME badArgName good_arg_name)
  # Function names should be lower case
endfunction()

#
macro(bad_macro_name badArgName good_arg_name)
  # Macro names should be upper case
endmacro()

if(FOOBAR)
  foreach(loopvar a b c d)
    # cmake-lint: disable=C0103
    foreach(LOOPVAR2 a b c d)
      # pass
    endforeach()
  endforeach()
  foreach(LOOPVAR3 a b c d)
    # pass
  endforeach()

endif()

set(VARNAME varvalue CACHE STRING)

cmake_minimum_required_version(VERSION 2.8.11 VERSION 3.16)

add_custom_command()

set(_form TARGET PRE_BUILD)
add_custom_command(
  ${form}
  COMMAND echo "hello"
  COMMENT "echo hello")

add_custom_command(
  TARGRET PRE_BUILD
  COMMAND echo "hello"
  COMMENT "echo hello")

add_custom_command(OUTPUT foo)

add_custom_target(foo ALL)

file()

set(_form TOUCH)
file(${form} foo.py)

file(TOUCHE foo.py)

break()

continue()
```

(continues on next page)

(continued from previous page)

```
# foo: docstring
function(foo)
  foreach(arg ${ARGN})
    if(arg STREQUAL "FOO")
      # parse FOO
    elseif(arg STREQUAL "BAR")
      # parse BAR
    elseif(arg MATCHES "BAZ.*")
      # parse BAZ
    endif()
  endforeach()
endfunction()

set(_foo "hello") set(_bar "goodbye")

set(_foo "hello")


set(_bar "goodbye")

# foo: docstring
function(foo "arg-name")
  # pass
endfunction()

# foo: docstring
function(foo arg arg)
  # pass
endfunction()

# foo: docstring
function(foo arg ARG)
  # pass
endfunction()

return()
message("This code is unreachable")

# cmake-lint: disable=E0109,C0321
# foo:docstring, too many arguments
function(foo, a0, a1, a2, a3, a4, a5)

  # Too many branches, too many returns
  if(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
```

(continues on next page)

(continued from previous page)

```
  elseif(blah) return()
  elseif(blah) return()
  elseif(blah) return()
  else() return()
  endif()

  # too many statements
  message(foo) message(foo) message(foo) message(foo) message(foo) message(foo)
  message(foo) message(foo) message(foo) message(foo) message(foo) message(foo)
  message(foo) message(foo) message(foo) message(foo) message(foo) message(foo)
  message(foo) message(foo) message(foo) message(foo) message(foo) message(foo)
  message(foo) message(foo) message(foo) message(foo) message(foo) message(foo)
endfunction()

# cmake-lint: disable=C0111
# cache (global) variables should be upper snake
set(MyGlobalVar CACHE STRING "my var")
# internal variables are treated as private and should be upper snake with an
# underscore prefix
set(MY_INTERNAL_VAR CACHE INTERNAL "my var")
# directory-scope variables should be upper-snake (public) or lower-snake with
# underscore prefix
set(_INVALID_PRIVATE_NAME "foo")
set(invalid_public_name "foo")
function(foo)
  set(INVALID_LOCAL_NAME "foo")
endfunction()

set(CMAKE_Cxx_STANDARD "11")

list(APPEND CMAKE_Cxx_STANDARD "11")

message("Using C++ standard ${CMAKE_Cxx_STANDARD}")

# This file is missing a final newline
```

The output is:

```
cmake_lint/test/expect_lint.cmake
=================================
cmake_lint/test/expect_lint.cmake:00: [C0301] Line too long (92/80)
cmake_lint/test/expect_lint.cmake:02: [C0303] Trailing whitespace
cmake_lint/test/expect_lint.cmake:04: [C0327] Wrong line ending (windows)
cmake_lint/test/expect_lint.cmake:08,00: [C0111] Missing docstring on function or␣
↪macro declaration
cmake_lint/test/expect_lint.cmake:08,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:08,09: [C0103] Invalid function name "BAD_FUNCTION_
↪NAME"
cmake_lint/test/expect_lint.cmake:13,00: [C0112] Empty docstring on function or macro␣
↪declaration
cmake_lint/test/expect_lint.cmake:13,06: [C0103] Invalid function name "bad_macro_name
↪"
cmake_lint/test/expect_lint.cmake:17,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:24,10: [C0103] Invalid loopvar name "LOOPVAR3"
cmake_lint/test/expect_lint.cmake:30,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:30,27: [E1120] Missing required positional argument
cmake_lint/test/expect_lint.cmake:30,33: [E1120] Missing required positional argument
```

(continues on next page)

(continued from previous page)

```
cmake_lint/test/expect_lint.cmake:32,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:32,46: [E1122] Duplicate keyword argument VERSION
cmake_lint/test/expect_lint.cmake:34,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:34,19: [E1120] Missing required positional argument
cmake_lint/test/expect_lint.cmake:36,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:38,02: [C0114] Form descriminator hidden behind␣
↪variable dereference
cmake_lint/test/expect_lint.cmake:42,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:43,02: [E1126] Invalid form descriminator
cmake_lint/test/expect_lint.cmake:47,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:47,19: [C0113] Missing COMMENT in statement which␣
↪allows it
cmake_lint/test/expect_lint.cmake:47,19: [E1125] Missing required keyword argument␣
↪COMMAND
cmake_lint/test/expect_lint.cmake:49,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:49,18: [C0113] Missing COMMAND in statement which␣
↪allows it
cmake_lint/test/expect_lint.cmake:49,18: [C0113] Missing COMMENT in statement which␣
↪allows it
cmake_lint/test/expect_lint.cmake:51,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:51,05: [E1120] Missing required positional argument
cmake_lint/test/expect_lint.cmake:53,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:54,05: [C0114] Form descriminator hidden behind␣
↪variable dereference
cmake_lint/test/expect_lint.cmake:56,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:56,05: [E1126] Invalid form descriminator
cmake_lint/test/expect_lint.cmake:58,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:58,00: [E0103] break outside of loop
cmake_lint/test/expect_lint.cmake:58,00: [W0101] Unreachable code
cmake_lint/test/expect_lint.cmake:60,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:60,00: [E0103] continue outside of loop
cmake_lint/test/expect_lint.cmake:60,00: [W0101] Unreachable code
cmake_lint/test/expect_lint.cmake:64,02: [C0201] Consider replacing custom parser␣
↪logic with cmake_parse_arguments
cmake_lint/test/expect_lint.cmake:75,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:75,18: [C0321] Multiple statements on a single line
cmake_lint/test/expect_lint.cmake:77,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:80,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:83,13: [E0109] Invalid argument name "arg-name" in␣
↪function/macro definition
cmake_lint/test/expect_lint.cmake:88,17: [E0108] Duplicate argument name arg in␣
↪function/macro definition
cmake_lint/test/expect_lint.cmake:93,17: [C0202] Argument name ARG differs from␣
↪existing argument only in case
cmake_lint/test/expect_lint.cmake:97,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:97,00: [W0101] Unreachable code
cmake_lint/test/expect_lint.cmake:102,00: [R0911] Too many return statements 17/6
cmake_lint/test/expect_lint.cmake:102,00: [R0912] Too many branches 17/12
cmake_lint/test/expect_lint.cmake:102,00: [R0913] Too many named arguments 6/5
cmake_lint/test/expect_lint.cmake:102,00: [R0915] Too many statements 65/50
cmake_lint/test/expect_lint.cmake:134,04: [C0103] Invalid CACHE variable name
↪"MyGlobalVar"
cmake_lint/test/expect_lint.cmake:137,04: [C0103] Invalid INTERNAL variable name "MY_
↪INTERNAL_VAR"
cmake_lint/test/expect_lint.cmake:140,04: [C0103] Invalid directory variable name "_
↪INVALID_PRIVATE_NAME"
cmake_lint/test/expect_lint.cmake:141,04: [C0103] Invalid directory variable name
↪"invalid_public_name"
```

(continues on next page)

(continued from previous page)

```
cmake_lint/test/expect_lint.cmake:143,06: [C0103] Invalid local variable name
↪"INVALID_LOCAL_NAME"
cmake_lint/test/expect_lint.cmake:146,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:146,04: [W0105] Assignment to variable 'CMAKE_Cxx_
↪STANDARD' which matches a built-in except for case
cmake_lint/test/expect_lint.cmake:148,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:148,12: [W0105] Assignment to variable 'CMAKE_Cxx_
↪STANDARD' which matches a built-in except for case
cmake_lint/test/expect_lint.cmake:150,00: [C0305] too many newlines between statements
cmake_lint/test/expect_lint.cmake:150,08: [W0105] Assignment to variable 'CMAKE_Cxx_
↪STANDARD' which matches a built-in except for case
cmake_lint/test/expect_lint.cmake:152: [C0304] Final newline missing

Summary
=======
files scanned: 1
found lint:
  Convention: 43
       Error: 12
    Refactor: 4
     Warning: 6
```

5.3 Linter Checks List

Below is a running list of planned and implemented lint checks.

5.3.1 Implemented

Here is a list of implemented lint checks. More detailed information about each check is available in *Lint Code Reference*.

C0102	Black listed name "{:s}"
C0103	Invalid {:s} name "{:s}"
C0111	Missing docstring on function or macro declaration
C0112	Empty docstring on function or macro declaration
C0113	Missing {:s} in statement which allows it
C0114	Form descriminator hidden behind variable dereference
C0201	Consider replacing custom parser logic with cmake_parse_arguments
C0202	Argument name {:s} differs from existing argument only in case
C0301	Line too long ({:d}/{:d})
C0303	Trailing whitespace
C0304	Final newline missing
C0305	{:s} newlines between statements
C0321	Multiple statements on a single line
C0327	Wrong line ending ({:s})
E0011	Unrecognized file option {:s}
E0012	Bad option value {:s}
E0103	{:s} outside of loop
E0108	Duplicate argument name {:s} in function/macro definition

Continued on next page

Table 5.1 – continued from previous page

E0109	Invalid argument name {:s} in function/macro definition
E1120	Missing required positional argument
E1121	Too many positional arguments
E1122	Duplicate keyword argument {:s}
E1125	Missing required keyword argument {:s}
E1126	Invalid form descriminator
R0911	Too many return statements {:d}/{:d}
R0912	Too many branches {:d}/{:d}
R0913	Too many named arguments {:d}/{:d}
R0914	Too many local variables {:d}/{:d}
R0915	Too many statements {:d}/{:d}
W0101	Unreachable code
W0104	Use of deprecated command {:s}
W0105	{:s} variable '{:s}' which matches a built-in except for case

5.3.2 Planned

Here are some planned lint checks based on the kinds of things that `pylint` looks for. If there's a particular check you'd like to see added, please open an issue on github.

C0203	uncanonical spelling of {:s}
C0204	uncanonical keyword order
C0205	use of bad-idea command {:s}
C0302	Too many lines in listfile
C0303	Wrong hanging indentation
C0304	raw assignment to property
E0001	(syntax error raised for a listfile; message varies)
E0011	Unrecognized file option %r
E0102	%s already defined line %s
E1123	Passing unexpected keyword argument %r in function call
F0001	(error prevented analysis; message varies)
F0002	%s: %s (message varies)
F0010	error while code parsing: %s
I0010	Unable to consider inline option %r
I0011	Locally disabling %s
I0012	Locally enabling %s
I0013	Ignoring entire file
I0020	Suppressed %s (from line %d)
I0021	Useless suppression of %s
R0201	Macro could be a function
R0401	Cyclic import (%s)
R0801	Similar lines in %s files
W0102	Using local variable %r before assignment
W0103	Undefined local variable %r
W0104	Use of deprecated command
W0105	Use of deprecated kwarg
W0106	Use of deprecated command form {:s}
W0212	Access to a private variable %s of a find module
W0402	Uses of a deprecated find module %r

Continued on next page

Table 5.2 – continued from previous page

W0622	Redefining built-in %r
W1401	Anomalous backslash in string: '%s'.

5.4 Lint Code Reference

5.4.1 C0102

message

```
Black listed name "{:s}"
```

description

Used when the name is listed in the "bad-names" black list.

This message belongs to the basic checker.

explanation

cmake-lint can be customized to help enforce coding guidelines that discourage or forbid use of certain names for variables, functions, etc.. These names are specified with the bad-names option. This message is raised whenever a name is in the list of names defined with the bad-names option.

5.4.2 C0103

message

```
Invalid {:s} name "{:s}"
```

description

Used when a name doesn't doesn't fit the naming convention associated to its type (function, macro, variable, . . .).

This message belongs to the basic checker.

explanation

The naming convention is defined with a regular expression, and the naming convention is satisfied if the name matches the regular expression.

5.4.3 C0111

message

```
Missing docstring on function or macro declaration
```

description

Used when a function or macro is defined without a documentation comment immediately preceeding it.

This message belongs to the basic checker.

explanation

So, you've written a some fancy function that makes it "easier" to declare build steps. Congratulations. You probably shouldn't have, but thats OK. Now that you did, how should people use it? What arguments does it take? What are the semantics of those arguements? You should include documentation in a comment block prior to the function declaration with this information.

5.4.4 C0112

message

```
Empty docstring on function or macro declaration
```

description

Used when a function or macro is preceeded by an empty comment string, rather that one with useful documentation.

This message belongs to the basic checker.

explanation

Ok so you saw C0111 and figured you'd be clever right? Sorry, no dice. Please include some useful documentation so that code readers know what your function/macro does and how to use it.

5.4.5 C0113

message

```
Missing {:s} in statement which allows it
```

5.4.6 C0114

message

```
Form descriminator hidden behind variable dereference
```

description

Used when a keyword used to descriminate betwen different forms of a command is hidden behind a variable dereference.

This message is implemented by individual command checkers.

explanation

Some cmake commands have very different behavior depending on the presence of a particular keyword (see e.g. the *file* command). And because cmake is a macro languge that keyword can actually be held inside a variable. Thus the keyword might not actually be visible to cmake-lint (or humans). In general there is no reason to do this and it really hurts readability since different descriminator keywords yield essentially different commands.

5.4.7 C0201

message

```
Consider replacing custom parser logic with cmake_parse_arguments
```

description

Used when custom parse logic is detected.

5.4.8 C0202

message

```
Argument name {:s} differs from existing argument only in case
```

5.4.9 C0301

message

```
Line too long ({:d}/{:d})
```

description

Used when a line is longer than the limit specified in the line-length option.

explanation

It is a good idea to keep each line within a maximum length to keep it from wrapping past the edge of an editing window. This improves readability and tempers other developers' irritability!

The default value of the line-length option is 80, the customary width of a terminal window.

Note that the line length and the limit are counted in characters, not in Bytes needed to represent these characters.

5.4.10 C0303

message

```
Trailing whitespace
```

description

Used when a line has one or more whitespace characters directly before the line end character(s).

This message belongs to the basic checker.

explanation

Such trailing whitespace is visually indistinguishable and some editors will trim them.

5.4.11 C0304

message

```
Final newline missing
```

description

Used when a listfile has no line end character(s) on its last line.

This message belongs to the basic checker.

explanation

While cmake itself does not require line end character(s) on the last line, is simply good practice to have it.

5.4.12 C0305

message

```
{:s} newlines between statements
```

5.4.13 C0321

message

```
Multiple statements on a single line
```

5.4.14 C0327

message

```
Wrong line ending ({:s})
```

description

Used when a line ends with the wrong line ending character. e.g. A line ends with “rn” when configured for “n”.

This message belongs to the basic checker.

explanation

While cmake itself does not enforce a particular line ending, it is good practice for a project to be consist with their line endings.

5.4.15 E0011

message

```
Unrecognized file option {:s}
```

description

Used when an unrecognized pragma is encountered.

explanation

cmake-lint allows for some inline comments to supress warnings (among other things). This lint is emitted if a bad option key is provided in such a pragma

5.4.16 E0012

message

```
Bad option value {:s}
```

description

Used when a cmake-lint pragma is encountered which attempts to alter some option in an invalid way.

This message belongs to the basic checker.

explanation

cmake-lint allows for some inline comments to supress warnings (among other things). This lint is emitted if a bad option is provided to one of these pragmas.

5.4.17 E0103

message

```
{:s} outside of loop
```

description

Used when a break() or continue() statement is used outside a loop.

This message belongs to the basic checker.

5.4.18 E0108

message

```
Duplicate argument name {:s} in function/macro definition
```

5.4.19 E0109

message

```
Invalid argument name {:s} in function/macro definition
```

5.4.20 E1120

message

```
Missing required positional argument
```

description

Used when a positional argument group expecting an exact number of arguments is closed (by a parenthesis) before that number of arguments is found.

This message belongs to the basic checker.

5.4.21 E1121

message

```
Too many positional arguments
```

description

Used when a positional argument is found when no argument group is expected.

This message is implemented by individual command checkers

5.4.22 E1122

message

```
Duplicate keyword argument {:s}
```

description

Used when a keyword shows up more than once within an argument group. In general, only COMMAND is allowed more than once.

5.4.23 E1125

message

```
Missing required keyword argument {:s}
```

5.4.24 E1126

message

```
Invalid form descriminator
```

description

Used when a keyword used to descriminate between different command forms is ommitted.

5.4.25 R0911

message

```
Too many return statements {:d}/{:d}
```

5.4.26 R0912

message

```
Too many branches {:d}/{:d}
```

5.4.27 R0913

message

```
Too many named arguments {:d}/{:d}
```

5.4.28 R0914

message

```
Too many local variables {:d}/{:d}
```

5.4.29 R0915

message

```
Too many statements {:d}/{:d}
```

5.4.30 W0101

message

```
Unreachable code
```

5.4.31 W0104

message

```
Use of deprecated command {:s}
```

5.4.32 W0105

message

```
{:s} variable '{:s}' which matches a built-in except for case
```

description

This warning means that you are using a variable such as, for example, *cmake_cxx_standard* which matches a builtin variable (*CMAKE_CXX_STANDARD*) except for the case. If this was intentional, then it's bad practice as it causes confusion (there are two variables in the namespace with identical name except for case), though it was probably not intentional and you probably aren't assigning to the correct variable.

This warning may be emitted for assignment (e.g. *set()* or *list()*) as well as for variable expansion in an argument (e.g. *"${CMAKE_Cxx_STANDARD}"*).

CHAPTER 6

ctest-to

The `ctest-to` program can parse listfiles from a ctest output tree and generate a more structured representation of the test spec.

6.1 Usage

```
usage: ctest-to [-h] [--log-level {debug,info,warning,error}] [--json | --xml]
                [directory]

Parse ctest testfiles and re-emit the test specification in a more structured
format.

positional arguments:
  directory

optional arguments:
  -h, --help            show this help message and exit
  --log-level {debug,info,warning,error}
  --json
  --xml
```

6.2 Example

Here are some examples generated by the ctest file for this repository:

```
[
  {
    "name": "cmake_format-TestAddCustomCommand",
    "argv": [
      "python",
```

(continues on next page)

(continued from previous page)

```
      "-Bm",
      "cmake_format.command_tests",
      "TestAddCustomCommand"
    ],
    "cwd": "/code/cmake_format/.build/nd.x86/cmake_format/command_tests",
    "props": {
      "working_directory": "/code/cmake_format"
    }
  },
  {
    "name": "cmake_format-TestAddCustomCommand_py3",
    "argv": [
      "python3",
      "-Bm",
      "cmake_format.command_tests",
      "TestAddCustomCommand"
    ],
    "cwd": "/code/cmake_format/.build/nd.x86/cmake_format/command_tests",
    "props": {
      "working_directory": "/code/cmake_format"
    }
  },
...
]
```

```
<ctest>
  <test cwd="/code/cmake_format/.build/nd.x86/cmake_format/command_tests" name="cmake_
↪format-TestAddCustomCommand" working_directory="/code/cmake_format">
    <argv>
      <arg value="python"/>
      <arg value="-Bm"/>
      <arg value="cmake_format.command_tests"/>
      <arg value="TestAddCustomCommand"/>
    </argv>
  </test>
  <test cwd="/code/cmake_format/.build/nd.x86/cmake_format/command_tests" name="cmake_
↪format-TestAddCustomCommand_py3" working_directory="/code/cmake_format">
    <argv>
      <arg value="python3"/>
      <arg value="-Bm"/>
      <arg value="cmake_format.command_tests"/>
      <arg value="TestAddCustomCommand"/>
    </argv>
  </test>
...
</ctest>
```

CHAPTER 7

Various configuration options and parameters

7.1 Options affecting listfile parsing

7.1.1 additional_commands

Specify structure for custom cmake functions

default value:

```
{ 'foo': { 'flags': ['BAR', 'BAZ'],
           'kwargs': {'DEPENDS': '*', 'HEADERS': '*', 'SOURCES': '*'}}}
```

detailed description:

Use this variable to specify how to parse custom cmake functions. See *Implementing Custom Parsers*.

config-file entry:

```
# ----------------------------------
# Options affecting listfile parsing
# ----------------------------------
with section("parse"):

  # Specify structure for custom cmake functions
  additional_commands = { 'foo': { 'flags': ['BAR', 'BAZ'],
             'kwargs': {'DEPENDS': '*', 'HEADERS': '*', 'SOURCES': '*'}}}
```

7.1.2 vartags

Specify variable tags.

default value:

```
[]
```

detailed description:

Specify a mapping of variable patterns (python regular expression) to a list of tags. Any time a a variable matching this pattern is encountered the tags can be used to affect the parsing/formatting. For example:

```
vartags = [
  (".*_COMMAND", ["cmdline"])
]
```

Specifies that any variable ending in _COMMAND be tagged as `cmdline`. This will affect the formatting by preventing the arguments from being vertically wrapped.

Note: this particular rule is builtin so you do not need to include this in your configuration. Use the configuration variable to add new rules.

command-line option:

```
--vartags [VARTAGS [VARTAGS ...]]
                      Specify variable tags.
```

config-file entry:

```
# ----------------------------------
# Options affecting listfile parsing
# ----------------------------------
with section("parse"):

  # Specify variable tags.
  vartags = []
```

7.1.3 proptags

Specify property tags.

default value:

```
[]
```

detailed description:

Specify a mapping of property patterns (python regular expression) to a list of tags. Any time a a property matching this pattern is encountered the tags can be used to affect the parsing/formatting. For example:

```
proptags = [
  (".*_DIRECTORIES", ["file-list"])
]
```

Specifies that any property ending in `_DIRECTORIES` be tagged as `file-list`. In the future this may affect formatting by allowing arguments to be sorted (but currently has no effect).

Note: this particular rule is builtin so you do not need to include this in your configuration. Use the configuration variable to add new rules.

command-line option:

```
--proptags [PROPTAGS [PROPTAGS ...]]
                      Specify property tags.
```

config-file entry:

```
# ----------------------------------
# Options affecting listfile parsing
# ----------------------------------
with section("parse"):

  # Specify property tags.
  proptags = []
```

7.2 Options affecting formatting.

7.2.1 line_width

How wide to allow formatted cmake files

default value:

```
80
```

detailed description:

`line_width` specifies the number of columns that `cmake-format` should fit commands into. This is the number of columns at which arguments will be wrapped.

```
# line_width = 80 (default)
add_library(libname STATIC sourcefile_one.cc sourcefile_two.cc
                           sourcefile_three.cc sourcefile_four.cc)

# line_width = 100
add_library(libname STATIC sourcefile_one.cc sourcefile_two.cc sourcefile_three.cc
                           sourcefile_four.cc)
```

command-line option:

```
--line-width LINE_WIDTH
                      How wide to allow formatted cmake files
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # How wide to allow formatted cmake files
  line_width = 80
```

7.2.2 tab_size

How many spaces to tab for indent

default value:

```
2
```

detailed description:

`tab_size` indicates how many spaces should be used to indent nested "scopes". For example:

```
# tab_size = 2 (default)
if(this_condition_is_true)
  message("Hello World")
endif()

# tab_size = 4
if(this_condition_is_true)
    message("Hello World")
endif()
```

command-line option:

```
--tab-size TAB_SIZE  How many spaces to tab for indent
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # How many spaces to tab for indent
  tab_size = 2
```

7.2.3 max_subgroups_hwrap

If an argument group contains more than this many sub-groups (parg or kwarg groups) then force it to a vertical layout.

default value:

```
2
```

detailed description:

A "subgroup" in this context is either a positional or keyword argument group within the current depth of the statement parse tree. If the number of "subgroups" at this depth is greater than `max_subgroups_hwrap` then hwrap-formatting is inadmissable and a vertical layout will be selected.

The default value for this parameter is *2*.

Consider the following two examples:

```
# This statement has two argument groups, so hwrap is admissible
add_custom_target(target1 ALL COMMAND echo "hello world")

# This statement has three argument groups, so the statement will format
# vertically
add_custom_target(
   target2 ALL
   COMMAND echo "hello world"
   COMMAND echo "hello again")
```

In the first statement, there are two argument groups. We can see them with `--dump parse`

```
└─ BODY: 1:0
  └─ STATEMENT: 1:0
       ├─ FUNNAME: 1:0
       ├─ LPAREN: 1:17
       ├─ ARGGROUP: 1:18
       │    ├─ PARGGROUP: 1:18  <-- group 1
       │    │    ├─ ARGUMENT: 1:18
```

(continues on next page)

(continued from previous page)

```
        │   │   └─ FLAG: 1:26
        │   └─ KWARGGROUP: 1:30  <-- group 2
        │       ├─ KEYWORD: 1:30
        │       └─ ARGGROUP: 1:38
        │           └─ PARGGROUP: 1:38
        │               ├─ ARGUMENT: 1:38
        │               └─ ARGUMENT: 1:43
        └─ RPAREN: 1:56
```

The second statement has three argument groups:

```
└─ BODY: 1:0
    └─ STATEMENT: 1:0
        ├─ FUNNAME: 1:0
        ├─ LPAREN: 1:17
        ├─ ARGGROUP: 2:5
        │   ├─ PARGGROUP: 2:5  <-- group 1
        │   │   ├─ ARGUMENT: 2:5
        │   │   └─ FLAG: 2:13
        │   ├─ KWARGGROUP: 3:5  <-- group 2
        │   │   ├─ KEYWORD: 3:5
        │   │   └─ ARGGROUP: 3:13
        │   │       └─ PARGGROUP: 3:13
        │   │           ├─ ARGUMENT: 3:13
        │   │           ├─ ARGUMENT: 3:18
        │   └─ KWARGGROUP: 4:5  <-- group 3
        │       ├─ KEYWORD: 4:5
        │       └─ ARGGROUP: 4:13
        │           └─ PARGGROUP: 4:13
        │               ├─ ARGUMENT: 4:13
        │               └─ ARGUMENT: 4:18
        └─ RPAREN: 4:31
```

command-line option:

```
--max-subgroups-hwrap MAX_SUBGROUPS_HWRAP
                      If an argument group contains more than this many sub-
                      groups (parg or kwarg groups) then force it to a
                      vertical layout.
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If an argument group contains more than this many sub-groups (parg or kwarg
  # groups) then force it to a vertical layout.
  max_subgroups_hwrap = 2
```

7.2.4 max_pargs_hwrap

If a positional argument group contains more than this many arguments, then force it to a vertical layout.

default value:

```
6
```

detailed description:

This configuration parameter is relavent only to positional argument groups. A positional argument group is a list of "plain" arguments. If the number of arguments in the group is greater than this number, then then hwrap-formatting is inadmissable and a vertical layout will be selected.

The default value for this parameter is 6

Consider the following two examples:

```
# This statement has six arguments in the second group and so hwrap is
# admissible
set(sources filename_one.cc filename_two.cc filename_three.cc
            filename_four.cc filename_five.cc filename_six.cc)

# This statement has seven arguments in the second group and so hwrap is
# inadmissible
set(sources
    filename_one.cc
    filename_two.cc
    filename_three.cc
    filename_four.cc
    filename_five.cc
    filename_six.cc
    filename_seven.cc)
```

command-line option:

```
--max-pargs-hwrap MAX_PARGS_HWRAP
                      If a positional argument group contains more than this
                      many arguments, then force it to a vertical layout.
```

config-file entry:

```
# -----------------------------
# Options affecting formatting.
# -----------------------------
with section("format"):

  # If a positional argument group contains more than this many arguments, then
  # force it to a vertical layout.
  max_pargs_hwrap = 6
```

7.2.5 max_rows_cmdline

If a cmdline positional group consumes more than this many lines without nesting, then invalidate the layout (and nest)

default value:

```
2
```

detailed description:

`max_pargs_hwrap` does not apply to positional argument groups for shell commands. These are never columnized and always hwrapped. However, if the wrapped format exceeds this many lines, then the group will also be nested.

command-line option:

```
--max-rows-cmdline MAX_ROWS_CMDLINE
                      If a cmdline positional group consumes more than this
                      many lines without nesting, then invalidate the layout
                      (and nest)
```

config-file entry:

```
# ------------------------------
# Options affecting formatting.
# ------------------------------
with section("format"):

  # If a cmdline positional group consumes more than this many lines without
  # nesting, then invalidate the layout (and nest)
  max_rows_cmdline = 2
```

7.2.6 separate_ctrl_name_with_space

If true, separate flow control names from their parentheses with a space

default value:

```
False
```

command-line option:

```
--separate-ctrl-name-with-space [SEPARATE_CTRL_NAME_WITH_SPACE]
                      If true, separate flow control names from their
                      parentheses with a space
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If true, separate flow control names from their parentheses with a space
  separate_ctrl_name_with_space = False
```

7.2.7 separate_fn_name_with_space

If true, separate function names from parentheses with a space

default value:

```
False
```

command-line option:

```
--separate-fn-name-with-space [SEPARATE_FN_NAME_WITH_SPACE]
                      If true, separate function names from parentheses with
                      a space
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If true, separate function names from parentheses with a space
  separate_fn_name_with_space = False
```

7.2.8 dangle_parens

If a statement is wrapped to more than one line, than dangle the closing parenthesis on its own line.

default value:

```
False
```

detailed description:

If a statement is wrapped to more than one line, than dangle the closing parenthesis on its own line. For example:

```
# dangle_parens = False (default)
set(sources filename_one.cc filename_two.cc filename_three.cc
            filename_four.cc filename_five.cc filename_six.cc)

# dangle_parens = True
set(sources filename_one.cc filename_two.cc filename_three.cc
          filename_four.cc filename_five.cc filename_six.cc
)  # <-- this is a dangling parenthesis
```

The default is `false`.

command-line option:

```
--dangle-parens [DANGLE_PARENS]
                      If a statement is wrapped to more than one line, than
                      dangle the closing parenthesis on its own line.
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If a statement is wrapped to more than one line, than dangle the closing
  # parenthesis on its own line.
  dangle_parens = False
```

7.2.9 dangle_align

If the trailing parenthesis must be 'dangled' on its on line, then align it to this reference: *prefix*: the start of the statement, *prefix-indent*: the start of the statement, plus one indentation level, *child*: align to the column of the arguments

default value:

```
'prefix'
```

detailed description:

If the trailing parenthesis must be 'dangled' on it's on line, then align it to this reference. Options are:

- `prefix`: the start of the statement,
- `prefix-indent`: the start of the statement, plus one indentation level
- `child`: align to the column of the arguments

For example:

```
# dangle_align = "prefix"
set(sources filename_one.cc filename_two.cc filename_three.cc
          filename_four.cc filename_five.cc filename_six.cc
)  # <-- aligned to the statement

# dangle_align = "prefix-indent"
set(sources filename_one.cc filename_two.cc filename_three.cc
          filename_four.cc filename_five.cc filename_six.cc
  )  # <-- plus one indentation level

# dangle_align = "child"
set(sources filename_one.cc filename_two.cc filename_three.cc
          filename_four.cc filename_five.cc filename_six.cc
    )  # <-- aligned to "sources"
```

command-line option:

```
--dangle-align {prefix,prefix-indent,child,off}
                      If the trailing parenthesis must be 'dangled' on its
                      on line, then align it to this reference: `prefix`:
                      the start of the statement, `prefix-indent`: the start
                      of the statement, plus one indentation level, `child`:
                      align to the column of the arguments
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If the trailing parenthesis must be 'dangled' on its on line, then align it
  # to this reference: `prefix`: the start of the statement,  `prefix-indent`:
  # the start of the statement, plus one indentation  level, `child`: align to
  # the column of the arguments
  dangle_align = 'prefix'
```

7.2.10 min_prefix_chars

If the statement spelling length (including space and parenthesis) is smaller than this amount, then force reject nested layouts.

default value:

```
4
```

detailed description:

This value only comes into play when considering whether or not to nest arguments below their parent. If the number of characters in the parent is less than this value, we will not nest. In the example below, we'll set `line_width=40`

for illustration:

```
# min_prefix_chars = 4 (default)
message(
  "With the default value, this "
  "string is allowed to nest beneath "
  "the statement")

# min_prefix_chars = 8
message("With the default value, this "
        "string is allowed to nest beneath "
        "the statement")
```

command-line option:

```
--min-prefix-chars MIN_PREFIX_CHARS
                      If the statement spelling length (including space and
                      parenthesis) is smaller than this amount, then force
                      reject nested layouts.
```

config-file entry:

```
# ------------------------------
# Options affecting formatting.
# ------------------------------
with section("format"):

  # If the statement spelling length (including space and parenthesis) is
  # smaller than this amount, then force reject nested layouts.
  min_prefix_chars = 4
```

7.2.11 max_prefix_chars

If the statement spelling length (including space and parenthesis) is larger than the tab width by more than this amount, then force reject un-nested layouts.

default value:

```
10
```

command-line option:

```
--max-prefix-chars MAX_PREFIX_CHARS
                      If the statement spelling length (including space and
                      parenthesis) is larger than the tab width by more than
                      this amount, then force reject un-nested layouts.
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If the statement spelling length (including space and parenthesis) is larger
  # than the tab width by more than this amount, then force reject un-nested
  # layouts.
  max_prefix_chars = 10
```

7.2.12 max_lines_hwrap

If a candidate layout is wrapped horizontally but it exceeds this many lines, then reject the layout.

default value:

```
2
```

detailed description:

Usually the layout algorithm will prefer to do a simple "word-wrap" of positional arguments, if it can. However if such a simple word-wrap would exceed this many lines, then that layout is rejected, and further passes are tried. The default value is `max_lines_hwrap=2` so, for example:

```
message("This message can easily be wrapped" "to two lines so there is no"
        "problem with using" "horizontal wrapping")
message(
  "However this message cannot be wrapped to two lines because the "
  "arguments are too long. It would require at least three lines."
  "As a result, a simple word-wrap is rejected"
  "And each argument"
  "gets its own line")
```

command-line option:

```
--max-lines-hwrap MAX_LINES_HWRAP
                      If a candidate layout is wrapped horizontally but it
                      exceeds this many lines, then reject the layout.
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If a candidate layout is wrapped horizontally but it exceeds this many
```

(continues on next page)

(continued from previous page)

```
  # lines, then reject the layout.
  max_lines_hwrap = 2
```

7.2.13 line_ending

What style line endings to use in the output.

default value:

```
'unix'
```

detailed description:

This is a string indicating which style of line ending `cmake-format` should use when writing out the formatted file. If `line_ending="unix"` (default) then the output will contain a single newline character (`\n`) at the end of each line. If `line_ending="windows"` then the output will contain a carriage-return and newline pair (`\r\n`). If `line_ending="auto"` then `cmake-format` will observe the first line-ending of the input file and will use style that all lines in the output.

command-line option:

```
--line-ending {windows,unix,auto}
                      What style line endings to use in the output.
```

config-file entry:

```
# ------------------------------
# Options affecting formatting.
# ------------------------------
with section("format"):

  # What style line endings to use in the output.
  line_ending = 'unix'
```

7.2.14 command_case

Format command names consistently as ‘lower’ or ‘upper’ case

default value:

```
'canonical'
```

detailed description:

`cmake` ignores case in command names. Very old projects tend to use uppercase for command names, while modern projects tend to use lowercase. There are three options for this variable:

- `upper`: format commands as uppercase
- `lower`: format commands as lowercase
- `canonical`: format standard commands as they are formatted in the `cmake` documentation.

`canonical` is generally the same as `lower` except that some third-party find modules that have moved into the distribution (e.g. `ExternalProject_Add`).

command-line option:

```
--command-case {lower,upper,canonical,unchanged}
                      Format command names consistently as 'lower' or
                      'upper' case
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # Format command names consistently as 'lower' or 'upper' case
  command_case = 'canonical'
```

7.2.15 keyword_case

Format keywords consistently as ‘lower’ or ‘upper’ case

default value:

```
'unchanged'
```

detailed description:

`cmake` ignores the case of sentinal words (keywords) in argument lists. Generally projects tend to prefer uppercase (`keyword_case="upper"`) which is the default. Alternatively, this may also be set to `lower` to format keywords as lowercase.

command-line option:

```
--keyword-case {lower,upper,unchanged}
                      Format keywords consistently as 'lower' or 'upper'
                      case
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # Format keywords consistently as 'lower' or 'upper' case
  keyword_case = 'unchanged'
```

7.2.16 always_wrap

A list of command names which should always be wrapped

default value:

```
[]
```

command-line option:

```
--always-wrap [ALWAYS_WRAP [ALWAYS_WRAP ...]]
                      A list of command names which should always be wrapped
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # A list of command names which should always be wrapped
  always_wrap = []
```

7.2.17 enable_sort

If true, the argument lists which are known to be sortable will be sorted lexicographicall

default value:

```
True
```

command-line option:

```
--enable-sort [ENABLE_SORT]
                      If true, the argument lists which are known to be
                      sortable will be sorted lexicographicall
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If true, the argument lists which are known to be sortable will be sorted
  # lexicographicall
  enable_sort = True
```

7.2.18 autosort

If true, the parsers may infer whether or not an argument list is sortable (without annotation).

default value:

```
False
```

command-line option:

```
--autosort [AUTOSORT]
                      If true, the parsers may infer whether or not an
                      argument list is sortable (without annotation).
```

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # If true, the parsers may infer whether or not an argument list is sortable
  # (without annotation).
  autosort = False
```

7.2.19 require_valid_layout

By default, if cmake-format cannot successfully fit everything into the desired linewidth it will apply the last, most agressive attempt that it made. If this flag is True, however, cmake-format will print error, exit with non-zero status code, and write-out nothing

default value:

```
False
```

detailed description:

By default, if cmake-format cannot successfully fit everything into the desired linewidth it will apply the last, most agressive attempt that it made. If this flag is True, however, cmake-format will print error, exit with non- zero status code, and write-out nothing

command-line option:

```
--require-valid-layout [REQUIRE_VALID_LAYOUT]
                       By default, if cmake-format cannot successfully fit
                       everything into the desired linewidth it will apply
                       the last, most agressive attempt that it made. If this
                       flag is True, however, cmake-format will print error,
                       exit with non-zero status code, and write-out nothing
```

config-file entry:

```
# ------------------------------
# Options affecting formatting.
# ------------------------------
with section("format"):

  # By default, if cmake-format cannot successfully fit everything into the
  # desired linewidth it will apply the last, most agressive attempt that it
  # made. If this flag is True, however, cmake-format will print error, exit
  # with non-zero status code, and write-out nothing
  require_valid_layout = False
```

7.2.20 layout_passes

A dictionary mapping layout nodes to a list of wrap decisions. See the documentation for more information.

default value:

```
{}
```

detailed description:

See the *Formatting Algorithm* section for more information on how *cmake-format* uses multiple passes to converge on the final layout of the listfile source code. This option can be used to override the default behavior. The format of this option is a dictionary, where the keys are the names of the different layout node classes:

- StatementNode
- ArgGroupNode
- KWargGroupNode
- PargGroupNode
- ParenGroupNode

The dictionary values are a list of pairs (2-tuples) in the form of `(passno, wrap-decision)`. Where `passno` is the pass number at which the wrap-decision becomes active, and `wrap-decision` is a boolean `(true/false)`. For each layout pass, the decision of whether or not the node should wrap (either nested, or vertical) is looked-up from this map.

config-file entry:

```
# ----------------------------
# Options affecting formatting.
# ----------------------------
with section("format"):

  # A dictionary mapping layout nodes to a list of wrap decisions. See the
  # documentation for more information.
  layout_passes = {}
```

7.3 Options affecting comment reflow and formatting.

7.3.1 bullet_char

What character to use for bulleted lists

default value:

```
'*'
```

command-line option:

```
--bullet-char BULLET_CHAR
                      What character to use for bulleted lists
```

config-file entry:

```
# ------------------------------------------------
# Options affecting comment reflow and formatting.
# ------------------------------------------------
with section("markup"):

  # What character to use for bulleted lists
  bullet_char = '*'
```

7.3.2 enum_char

What character to use as punctuation after numerals in an enumerated list

default value:

```
'.'
```

command-line option:

```
--enum-char ENUM_CHAR
                      What character to use as punctuation after numerals in
                      an enumerated list
```

config-file entry:

```
# --------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------
with section("markup"):

  # What character to use as punctuation after numerals in an enumerated list
  enum_char = '.'
```

7.3.3 first_comment_is_literal

If comment markup is enabled, don't reflow the first comment block in each listfile. Use this to preserve formatting of your copyright/license statements.

default value:

```
False
```

command-line option:

```
--first-comment-is-literal [FIRST_COMMENT_IS_LITERAL]
                      If comment markup is enabled, don't reflow the first
                      comment block in each listfile. Use this to preserve
                      formatting of your copyright/license statements.
```

config-file entry:

```
# --------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------
with section("markup"):

  # If comment markup is enabled, don't reflow the first comment block in each
  # listfile. Use this to preserve formatting of your copyright/license
  # statements.
  first_comment_is_literal = False
```

7.3.4 literal_comment_pattern

If comment markup is enabled, don't reflow any comment block which matches this (regex) pattern. Default is *None* (disabled).

default value:

```
None
```

command-line option:

```
--literal-comment-pattern LITERAL_COMMENT_PATTERN
                      If comment markup is enabled, don't reflow any comment
                      block which matches this (regex) pattern. Default is
                      `None` (disabled).
```

config-file entry:

```
# ------------------------------------------------
# Options affecting comment reflow and formatting.
# ------------------------------------------------
with section("markup"):

  # If comment markup is enabled, don't reflow any comment block which matches
  # this (regex) pattern. Default is `None` (disabled).
  literal_comment_pattern = None
```

7.3.5 fence_pattern

Regular expression to match preformat fences in comments default= `r'^\s*([`~]{3}[`~]*)(.*)$'`

default value:

```
'^\\s*([`~]{3}[`~]*)(.*)$'
```

command-line option:

```
--fence-pattern FENCE_PATTERN
                      Regular expression to match preformat fences in
                      comments default= ``r'^\s*([`~]{3}[`~]*)(.*)$'``
```

config-file entry:

```
# --------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------
with section("markup"):

  # Regular expression to match preformat fences in comments default=
  # ``r'^\s*([`~]{3}[`~]*)(.*)$'``
  fence_pattern = '^\\s*([`~]{3}[`~]*)(.*)$'
```

7.3.6 ruler_pattern

Regular expression to match rulers in comments default= `r'^\s*[^\w\s]{3}.*[^\w\s]{3}$'`

default value:

```
'^\\s*[^\\w\\s]{3}.*[^\\w\\s]{3}$'
```

command-line option:

```
--ruler-pattern RULER_PATTERN
                      Regular expression to match rulers in comments
                      default= ``r'^\s*[^\w\s]{3}.*[^\w\s]{3}$'``
```

config-file entry:

```
# --------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------
with section("markup"):

  # Regular expression to match rulers in comments default=
  # ``r'^\s*[^\w\s]{3}.*[^\w\s]{3}$'``
  ruler_pattern = '^\\s*[^\\w\\s]{3}.*[^\\w\\s]{3}$'
```

7.3.7 explicit_trailing_pattern

If a comment line matches starts with this pattern then it is explicitly a trailing comment for the preceeding argument. Default is '#<'

default value:

```
'#<'
```

command-line option:

```
--explicit-trailing-pattern EXPLICIT_TRAILING_PATTERN
                        If a comment line matches starts with this pattern
                        then it is explicitly a trailing comment for the
                        preceeding argument. Default is '#<'
```

config-file entry:

```
# --------------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------------
with section("markup"):

  # If a comment line matches starts with this pattern then it is explicitly a
  # trailing comment for the preceeding argument. Default is '#<'
  explicit_trailing_pattern = '#<'
```

7.3.8 hashruler_min_length

If a comment line starts with at least this many consecutive hash characters, then don't lstrip() them off. This allows for lazy hash rulers where the first hash char is not separated by space

default value:

```
10
```

command-line option:

```
--hashruler-min-length HASHRULER_MIN_LENGTH
                        If a comment line starts with at least this many
                        consecutive hash characters, then don't lstrip() them
                        off. This allows for lazy hash rulers where the first
                        hash char is not separated by space
```

config-file entry:

```
# --------------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------------
with section("markup"):

  # If a comment line starts with at least this many consecutive hash
  # characters, then don't lstrip() them off. This allows for lazy hash rulers
  # where the first hash char is not separated by space
  hashruler_min_length = 10
```

7.3.9 canonicalize_hashrulers

If true, then insert a space between the first hash char and remaining hash chars in a hash ruler, and normalize its length to fill the column

default value:

```
True
```

command-line option:

```
--canonicalize-hashrulers [CANONICALIZE_HASHRULERS]
                      If true, then insert a space between the first hash
                      char and remaining hash chars in a hash ruler, and
                      normalize its length to fill the column
```

config-file entry:

```
# --------------------------------------------
# Options affecting comment reflow and formatting.
# --------------------------------------------
with section("markup"):

  # If true, then insert a space between the first hash char and remaining hash
  # chars in a hash ruler, and normalize its length to fill the column
  canonicalize_hashrulers = True
```

7.3.10 enable_markup

enable comment markup parsing and reflow

default value:

```
True
```

command-line option:

```
--enable-markup [ENABLE_MARKUP]
                      enable comment markup parsing and reflow
```

config-file entry:

```
# ------------------------------------------------
# Options affecting comment reflow and formatting.
# ------------------------------------------------
with section("markup"):

  # enable comment markup parsing and reflow
  enable_markup = True
```

7.4 Options affecting the linter

7.4.1 disabled_codes

a list of lint codes to disable

default value:

```
[]
```

command-line option:

```
--disabled-codes [DISABLED_CODES [DISABLED_CODES ...]]
                      a list of lint codes to disable
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # a list of lint codes to disable
  disabled_codes = []
```

7.4.2 function_pattern

regular expression pattern describing valid function names

default value:

```
'[0-9a-z_]+'
```

command-line option:

```
--function-pattern FUNCTION_PATTERN
                      regular expression pattern describing valid function
                      names
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # regular expression pattern describing valid function names
  function_pattern = '[0-9a-z_]+'
```

7.4.3 macro_pattern

regular expression pattern describing valid macro names

default value:

```
'[0-9A-Z_]+'
```

command-line option:

```
--macro-pattern MACRO_PATTERN
                      regular expression pattern describing valid macro
                      names
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # regular expression pattern describing valid macro names
  macro_pattern = '[0-9A-Z_]+'
```

7.4.4 global_var_pattern

regular expression pattern describing valid names for variables with global scope

default value:

```
'[0-9A-Z][0-9A-Z_]+'
```

command-line option:

```
--global-var-pattern GLOBAL_VAR_PATTERN
                     regular expression pattern describing valid names for
                     variables with global scope
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # regular expression pattern describing valid names for variables with global
  # scope
  global_var_pattern = '[0-9A-Z][0-9A-Z_]+'
```

7.4.5 internal_var_pattern

regular expression pattern describing valid names for variables with global scope (but internal semantic)

default value:

```
'_[0-9A-Z][0-9A-Z_]+'
```

command-line option:

```
--internal-var-pattern INTERNAL_VAR_PATTERN
                     regular expression pattern describing valid names for
                     variables with global scope (but internal semantic)
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # regular expression pattern describing valid names for variables with global
  # scope (but internal semantic)
  internal_var_pattern = '_[0-9A-Z][0-9A-Z_]+'
```

7.4.6 local_var_pattern

regular expression pattern describing valid names for variables with local scope

default value:

```
'[0-9a-z_]+'
```

command-line option:

```
--local-var-pattern LOCAL_VAR_PATTERN
                      regular expression pattern describing valid names for
                      variables with local scope
```

config-file entry:

```
# --------------------------
# Options affecting the linter
# --------------------------
with section("lint"):

  # regular expression pattern describing valid names for variables with local
  # scope
  local_var_pattern = '[0-9a-z_]+'
```

7.4.7 private_var_pattern

regular expression pattern describing valid names for privatedirectory variables

default value:

```
'_[0-9a-z_]+'
```

command-line option:

```
--private-var-pattern PRIVATE_VAR_PATTERN
                      regular expression pattern describing valid names for
                      privatedirectory variables
```

config-file entry:

```
# --------------------------
# Options affecting the linter
# --------------------------
with section("lint"):

  # regular expression pattern describing valid names for privatedirectory
  # variables
  private_var_pattern = '_[0-9a-z_]+'
```

7.4.8 public_var_pattern

regular expression pattern describing valid names for publicdirectory variables

default value:

```
'[0-9A-Z][0-9A-Z_]+'
```

command-line option:

```
--public-var-pattern PUBLIC_VAR_PATTERN
                      regular expression pattern describing valid names for
                      publicdirectory variables
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # regular expression pattern describing valid names for publicdirectory
  # variables
  public_var_pattern = '[0-9A-Z][0-9A-Z_]+'
```

7.4.9 keyword_pattern

regular expression pattern describing valid names for keywords used in functions or macros

default value:

```
'[0-9A-Z_]+'
```

command-line option:

```
--keyword-pattern KEYWORD_PATTERN
                      regular expression pattern describing valid names for
                      keywords used in functions or macros
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):
```

(continues on next page)

(continued from previous page)

```
  # regular expression pattern describing valid names for keywords used in
  # functions or macros
  keyword_pattern = '[0-9A-Z_]+'
```

7.4.10 max_conditionals_custom_parser

In the heuristic for C0201, how many conditionals to match within a loop in before considering the loop a parser.

default value:

```
2
```

command-line option:

```
--max-conditionals-custom-parser MAX_CONDITIONALS_CUSTOM_PARSER
                      In the heuristic for C0201, how many conditionals to
                      match within a loop in before considering the loop a
                      parser.
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # In the heuristic for C0201, how many conditionals to match within a loop in
  # before considering the loop a parser.
  max_conditionals_custom_parser = 2
```

7.4.11 min_statement_spacing

Require at least this many newlines between statements

default value:

```
1
```

command-line option:

```
--min-statement-spacing MIN_STATEMENT_SPACING
                      Require at least this many newlines between statements
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # Require at least this many newlines between statements
  min_statement_spacing = 1
```

7.4.12 max_statement_spacing

Require no more than this many newlines between statements

default value:

```
1
```

command-line option:

```
--max-statement-spacing MAX_STATEMENT_SPACING
                        Require no more than this many newlines between
                        statements
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):

  # Require no more than this many newlines between statements
  max_statement_spacing = 1
```

7.4.13 max_returns

default value:

```
6
```

command-line option:

```
--max-returns MAX_RETURNS
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):
  max_returns = 6
```

7.4.14 max_branches

default value:

```
12
```

command-line option:

```
--max-branches MAX_BRANCHES
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):
  max_branches = 12
```

7.4.15 max_arguments

default value:

```
5
```

command-line option:

```
--max-arguments MAX_ARGUMENTS
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):
  max_arguments = 5
```

7.4.16 max_localvars

default value:

```
15
```

command-line option:

```
--max-localvars MAX_LOCALVARS
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):
  max_localvars = 15
```

7.4.17 max_statements

default value:

```
50
```

command-line option:

```
--max-statements MAX_STATEMENTS
```

config-file entry:

```
# ----------------------------
# Options affecting the linter
# ----------------------------
with section("lint"):
  max_statements = 50
```

7.5 Options affecting file encoding

7.5.1 emit_byteorder_mark

If true, emit the unicode byte-order mark (BOM) at the start of the file

default value:

```
False
```

detailed description:

If `true` (the default is `false`) then output the unicode byte-order at the start of the document.

command-line option:

```
--emit-byteorder-mark [EMIT_BYTEORDER_MARK]
                      If true, emit the unicode byte-order mark (BOM) at the
                      start of the file
```

config-file entry:

```
# ------------------------------
# Options affecting file encoding
# ------------------------------
with section("encode"):

  # If true, emit the unicode byte-order mark (BOM) at the start of the file
  emit_byteorder_mark = False
```

7.5.2 input_encoding

Specify the encoding of the input file. Defaults to utf-8

default value:

```
'utf-8'
```

detailed description:

Specify the input encoding of the file. The format of this string is anything understood by the `encoding=` keyword of the python `open()` function. The default is `utf-8`.

command-line option:

```
--input-encoding INPUT_ENCODING
                      Specify the encoding of the input file. Defaults to
                      utf-8
```

config-file entry:

```
# ------------------------------
# Options affecting file encoding
# ------------------------------
with section("encode"):

  # Specify the encoding of the input file. Defaults to utf-8
  input_encoding = 'utf-8'
```

7.5.3 output_encoding

Specify the encoding of the output file. Defaults to utf-8. Note that cmake only claims to support utf-8 so be careful when using anything else

default value:

```
'utf-8'
```

detailed description:

Specify the output encoding of the file. The format of this string is anything understood by the `encoding=` keyword of the python `open()` function. The default is `utf-8`.

command-line option:

```
--output-encoding OUTPUT_ENCODING
                      Specify the encoding of the output file. Defaults to
                      utf-8. Note that cmake only claims to support utf-8 so
                      be careful when using anything else
```

config-file entry:

```
# ------------------------------
# Options affecting file encoding
# ------------------------------
with section("encode"):

  # Specify the encoding of the output file. Defaults to utf-8. Note that cmake
  # only claims to support utf-8 so be careful when using anything else
  output_encoding = 'utf-8'
```

7.6 Miscellaneous configurations options.

7.6.1 per_command

A dictionary containing any per-command configuration overrides. Currently only *command_case* is supported.

default value:

```
{}
```

config-file entry:

```
# ------------------------------------
# Miscellaneous configurations options.
# ------------------------------------
with section("misc"):

  # A dictionary containing any per-command configuration overrides. Currently
  # only `command_case` is supported.
  per_command = {}
```

CHAPTER 8

Implementing Custom Parsers

8.1 Using the simple specification

You can tell the parser how to interpret your custom commands by specifying the format of their call signature using the `addtitional_commands` configuration variable. The format of the variable is a dictionary (a mapping in JSON/YAML) where the keys should be the name of the command and the values satisfy the following (recursive) schema:

```
POSITIONAL_SPEC = {
  # Number of positional arguments. Can be an integer indicating an exact
  # number of arguments, or a string satisfying the regex:
  # "(\d*)\+?|\*|\?". The semantics of the string:
  #
  #   * "?": zero or one
  #   * "*": zero or more
  #   * "+": one or more
  #   * (\d+): exactly `n` positional arguments (e.g. "3")
  #   * (\d+)\+ : as many positional arguments as available, but at least
  #     `n` (e.g. "3+")
  #
  "nargs": <string,int>,

  # Stores a list of keywords that are treated as positional arguments and
  # included in the positional group, but annotated as keywords for special
  # processing (e.g. case canonicalization).
  "flags": [<string>, ...],

  # Stores a list of tags to apply to this positional argument group. See
  # below for more information on tags.
  "tags": [<string>]

  # If true, then this specification will be repeated indefinitately
  "legacy": <bool>,
}
```

(continues on next page)

(continued from previous page)

```
KEYWORD_SPEC = {
  "pargs": [POSITIONAL_SPEC, ...],
  "kwargs": {<string>: KEYWORD_SPEC}
}
```

As a shorthand:

- If a keyword specification contains no nested *kwargs* and only one positional argument group, you may omit the *KEYWORD_SPEC* overhead and write the *POSITIONAL_SPEC* directly (including shorthands below)
- There is only one positional group, you may omit the list
- The a positional contains now flags, and is not tagged, you may omit the outer dictionary and write only the *nargs* value.

8.1.1 Tags

Positional argument groups support the following tags:

- *cmdline*: The positional arguments of this group are command line arguments in a shell command. In particular this means that they will not be wrapped vertically
- *file-list*: The positional arguments of this group are names/paths of files. In the future this tag will enable certain filters for formatting or linting, such as automatic sorting.

8.2 Generating simple specifications

There is an experimental utility called `cmake-genparsers` which can generate simple specifications for cmake functions and macros that use `cmake_parse_arguments`. The tool has very limited ability to process cmake arguments. There is a frontend included in the python distribution. Usage is

```
usage:
cmake-genparsers [-h] [-o OUTFILE_PATH] infilepath [infilepath ...]

Parse cmake listfiles, find function and macro declarations, and generate
parsers for them.

positional arguments:
  infilepaths

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -l {error,warning,info,debug}, --log-level {error,warning,info,debug}
  -o OUTFILE_PATH, --outfile-path OUTFILE_PATH
                        Write results to this file. Default is stdout.
  -f {json,yaml,python}, --output-format {json,yaml,python}
```

For example:

```
:~$ cmake-genparsers /usr/share/cmake-3.10/Modules/*.cmake
{ 'add_command': {'pargs': {'nargs': 1}},
  'add_compiler_export_flags': {'pargs': {'nargs': 0}},
```

(continues on next page)

(continued from previous page)

```
  'add_feature_info': {'pargs': {'nargs': 3}},
  'add_file_dependencies': {'pargs': {'nargs': 1}},
  'add_flex_bison_dependency': {'pargs': {'nargs': 2}},
  'add_jar': { 'kwargs': { 'ENTRY_POINT': 1,
                           'INCLUDE_JARS': '+',
                           'MANIFEST': 1,
                           'OUTPUT_DIR': 1,
                           'OUTPUT_NAME': 1,
...
```

or

```
:~$ cmake-genparsers -f yaml /usr/share/cmake-3.10/Modules/*.cmake
add_file_dependencies:
  pargs:
    nargs: 1
android_add_test_data:
  pargs:
    nargs: 1+
    flags: []
  kwargs: {}
get_bundle_main_executable:
  pargs:
...
```

This tool is still in the early stages of development so don't be surprised if it chokes on some of your input files, or if it does not propery generate specifications for your commands.

CHAPTER 9

CMake Parse Tree

This document is intended to describe the high level organization of how cmake listfiles are parsed and organized into an abstract syntax tree.

Digestion and formatting of a listfile is done in four phases:

- tokenization
- parsing
- layout tree construction
- layout / reflow

9.1 Tokenizer

Listfiles are first digested into a sequence of tokens. The tokenizer is implemented in *lexer.py* an defines the following types of tokens:

Token Type	Description	Example
QUOTED_LITERAL	A single or double quoted string, from the first quote to the first subsequent un-escaped quote	`"foo" 'bar'`
BRACKET_ARGUMENT	A bracket-quoted argument of a cmake-statement	`[=[hello foo]=]`
NUMBER	Unquoted numeric literal	`1234`
LEFT_PAREN	A left parenthesis	`(`
RIGHT_PAREN	A right parenthesis	`)`
WORD	An unquoted literal string which matches lexical rules such that it could be a cmake entity name, such as the name of a function or variable	`foo foo_bar`
DEREF	A variable dereference expression, from the dollar sign up to the outer most right curly brace	`${foo} ${foo_${bar}}`
NEWLINE	A single carriage return, newline or (carriage-return, newline) pair	
WHITESPACE	A continuous sequence of space, tab or other ascii whitespace	
BRACKET_COMMENT	A bracket-quoted comment string	`#[=[hello]=]`
COMMENT	A single line starting with a hash	`# hello world`
UNQUOTED_LITERAL	A sequence of non-whitespace characters used as a cmake argument but not satisfying the requirements of a cmake name	`--verbose`
FORMAT_OFF	A special comment disabling cmake-format temporarily	`cmake-format: off`
FORMAT_OFF	A special comment re-enabling	`cmake-format: on`

Each token covers a continuous sequence of characters of the input file. Futhermore, the sequence of tokens digest from the file covers the entire range of infile offsets. The `Token` object stores information about the input file byte offset, line number, and column number of it's start location. Note that for `utf-8` input where a character may be composed of more than one byte, the `(row, col)` location is the location of the character while the `offset` is the index of the first byte of the character.

You can inspect the tokenization of a listfile by executing `cmake-format` with `--dump lex`. For example:

```
Token(type=NEWLINE, content='\n', line=1, col=0)
Token(type=WORD, content='cmake_minimum_required', line=2, col=0)
Token(type=LEFT_PAREN, content='(', line=2, col=22)
Token(type=WORD, content='VERSION', line=2, col=23)
Token(type=WHITESPACE, content=' ', line=2, col=30)
Token(type=UNQUOTED_LITERAL, content='3.5', line=2, col=31)
Token(type=RIGHT_PAREN, content=')', line=2, col=34)
Token(type=NEWLINE, content='\n', line=2, col=35)
Token(type=WORD, content='project', line=3, col=0)
Token(type=LEFT_PAREN, content='(', line=3, col=7)
Token(type=WORD, content='demo', line=3, col=8)
Token(type=RIGHT_PAREN, content=')', line=3, col=12)
Token(type=NEWLINE, content='\n', line=3, col=13)
Token(type=WORD, content='if', line=4, col=0)
Token(type=LEFT_PAREN, content='(', line=4, col=2)
Token(type=WORD, content='FOO', line=4, col=3)
Token(type=WHITESPACE, content=' ', line=4, col=6)
Token(type=WORD, content='AND', line=4, col=7)
Token(type=WHITESPACE, content=' ', line=4, col=10)
Token(type=LEFT_PAREN, content='(', line=4, col=11)
Token(type=WORD, content='BAR', line=4, col=12)
```

(continues on next page)

(continued from previous page)

```
Token(type=WHITESPACE, content=' ', line=4, col=15)
Token(type=WORD, content='OR', line=4, col=16)
Token(type=WHITESPACE, content=' ', line=4, col=18)
Token(type=WORD, content='BAZ', line=4, col=19)
Token(type=RIGHT_PAREN, content=')', line=4, col=22)
Token(type=RIGHT_PAREN, content=')', line=4, col=23)
Token(type=NEWLINE, content='\n', line=4, col=24)
Token(type=WHITESPACE, content='  ', line=5, col=0)
Token(type=WORD, content='add_library', line=5, col=2)
Token(type=LEFT_PAREN, content='(', line=5, col=13)
Token(type=WORD, content='hello', line=5, col=14)
Token(type=WHITESPACE, content=' ', line=5, col=19)
Token(type=UNQUOTED_LITERAL, content='hello.cc', line=5, col=20)
Token(type=RIGHT_PAREN, content=')', line=5, col=28)
Token(type=NEWLINE, content='\n', line=5, col=29)
Token(type=WORD, content='endif', line=6, col=0)
Token(type=LEFT_PAREN, content='(', line=6, col=5)
Token(type=RIGHT_PAREN, content=')', line=6, col=6)
Token(type=NEWLINE, content='\n', line=6, col=7)
```

9.2 Parser: Syntax Tree

`cmake-format` parses the token stream in a single pass. The state machine of the parser is maintained by the program stack (i.e. the parse functions are called recursively) and each node type in the tree has it's own parse function.

There are fourteen types of nodes in the parse tree. They are described below along with the list of possible child node types.

9.2.1 Node Types

Node Type	Description	Allowed Children
BODY	A generic section of a cmake document. This node type is found at the root of the parse tree and within conditional/flow control statements	COMMENT STATEMENT WHITESPACE
WHITESPACE	A consecutive sequence of whitespace tokens between any two other types of nodes.	(none)
COMMENT	A sequence of one or more comment lines. The node consistes of all consecutive comment lines unbroken by additional newlines or a single BRACKET_COMMENT token.	(token)
STATEMENT	A cmake statement (i.e. function call)	ARGGROUP COMMENT FUNNAME
FLOW_CONTROL	Two or more cmake statements and their nested bodies representing a flow control construct (i.e. `if` or `foreach`).	STATEMENT BODY
ARGGROUP	A top-level collection of one or more positional, kwarg, or flag groups	PARGGROUP KWARGGROUP PARENGROUP FLAGGROUP COMMENT
PARGGROUP	A grouping of one or more positional arguments.	ARGUMENT COMMENT
FLAGGROUP	A grouping of one or more positional arguments, each of which is a flag	FLAG COMMENT
KWARGGROUP	A KEYWORD group, starting with the keyword and ending with the last argument associated with that keyword	KEYWORD ARGGROUP
PARENGROUP	A parenthetical group, starting with a left parenthesis and ending with the matching right parenthesis	ARGGROUP
FUNNAME	Consists of a single token containing the name of the function/command in a statement with that keyword	(token)
ARGUMENT	Consists of a single token, containing the literal argument of a statement, and optionally a comment associated with it	(token) COMMENT
KEYWORD	Consists of a single token, containing the literal keyword of a keyword group, and optionally a comment associated with it	(token) COMMENT
FLAG	Consists of a single token, containing the literal keyword of a statment flag, and optionally a comment associated with it	(token) COMMENT
ONOFFSWITCH	Consists of a single token, containing the sentinal comment line `# cmake-format: on` or `# cmake-format: off`.	(token)

You can inspect the parse tree of a listfile by `cmake-format` with `--dump parse`. For example:

```
└─ BODY: 1:0
   ├─ WHITESPACE: 1:0
   │  └─ Token(type=NEWLINE, content='\n', line=1, col=0)
   ├─ STATEMENT: 2:0
   │  ├─ FUNNAME: 2:0
   │  │  └─ Token(type=WORD, content='cmake_minimum_required', line=2, col=0)
   │  ├─ LPAREN: 2:22
   │  │  └─ Token(type=LEFT_PAREN, content='(', line=2, col=22)
   │  ├─ ARGGROUP: 2:23
   │  │  └─ KWARGGROUP: 2:23
   │  │     ├─ KEYWORD: 2:23
   │  │     │  └─ Token(type=WORD, content='VERSION', line=2, col=23)
   │  │     ├─ Token(type=WHITESPACE, content=' ', line=2, col=30)
   │  │     └─ ARGGROUP: 2:31
```

(continues on next page)

(continued from previous page)

```
    │   │           └─ PARGGROUP: 2:31
    │   │               └─ ARGUMENT: 2:31
    │   │                   └─ Token(type=UNQUOTED_LITERAL, content='3.5', line=2,␣
↪col=31)
    │   └─ RPAREN: 2:34
    │       └─ Token(type=RIGHT_PAREN, content=')', line=2, col=34)
    ├─ WHITESPACE: 2:35
    │   └─ Token(type=NEWLINE, content='\n', line=2, col=35)
    ├─ STATEMENT: 3:0
    │   ├─ FUNNAME: 3:0
    │   │   └─ Token(type=WORD, content='project', line=3, col=0)
    │   ├─ LPAREN: 3:7
    │   │   └─ Token(type=LEFT_PAREN, content='(', line=3, col=7)
    │   ├─ ARGGROUP: 3:8
    │   │   └─ PARGGROUP: 3:8
    │   │       └─ ARGUMENT: 3:8
    │   │           └─ Token(type=WORD, content='demo', line=3, col=8)
    │   └─ RPAREN: 3:12
    │       └─ Token(type=RIGHT_PAREN, content=')', line=3, col=12)
    ├─ WHITESPACE: 3:13
    │   └─ Token(type=NEWLINE, content='\n', line=3, col=13)
    ├─ FLOW_CONTROL: 4:0
    │   ├─ STATEMENT: 4:0
    │   │   ├─ FUNNAME: 4:0
    │   │   │   └─ Token(type=WORD, content='if', line=4, col=0)
    │   │   ├─ LPAREN: 4:2
    │   │   │   └─ Token(type=LEFT_PAREN, content='(', line=4, col=2)
    │   │   ├─ ARGGROUP: 4:3
    │   │   │   ├─ PARGGROUP: 4:3
    │   │   │   │   ├─ ARGUMENT: 4:3
    │   │   │   │   │   └─ Token(type=WORD, content='FOO', line=4, col=3)
    │   │   │   │   └─ Token(type=WHITESPACE, content=' ', line=4, col=6)
    │   │   │   └─ KWARGGROUP: 4:7
    │   │   │       ├─ KEYWORD: 4:7
    │   │   │       │   └─ Token(type=WORD, content='AND', line=4, col=7)
    │   │   │       ├─ Token(type=WHITESPACE, content=' ', line=4, col=10)
    │   │   │       └─ ARGGROUP: 4:11
    │   │   │           └─ PARENGROUP: 4:11
    │   │   │               ├─ LPAREN: 4:11
    │   │   │               │   └─ Token(type=LEFT_PAREN, content='(', line=4, col=11)
    │   │   │               ├─ ARGGROUP: 4:12
    │   │   │               │   ├─ PARGGROUP: 4:12
    │   │   │               │   │   ├─ ARGUMENT: 4:12
    │   │   │               │   │   │   └─ Token(type=WORD, content='BAR', line=4,␣
↪col=12)
    │   │   │               │   │   └─ Token(type=WHITESPACE, content=' ', line=4,␣
↪col=15)
    │   │   │               │   └─ KWARGGROUP: 4:16
    │   │   │               │       ├─ KEYWORD: 4:16
    │   │   │               │       │   └─ Token(type=WORD, content='OR', line=4,␣
↪col=16)
    │   │   │               │       ├─ Token(type=WHITESPACE, content=' ', line=4,␣
↪col=18)
    │   │   │               │       └─ ARGGROUP: 4:19
    │   │   │               │           └─ PARGGROUP: 4:19
    │   │   │               │               └─ ARGUMENT: 4:19
    │   │   │               │                   └─ Token(type=WORD, content='BAZ',␣
↪line=4, col=19)
```

(continues on next page)

(continued from previous page)

```
    │   │   │   │               └─ RPAREN: 4:22
    │   │   │   │                   └─ Token(type=RIGHT_PAREN, content=')', line=4,␣
↪col=22)
    │   │   │   └─ RPAREN: 4:23
    │   │   │       └─ Token(type=RIGHT_PAREN, content=')', line=4, col=23)
    │   │   ├─ BODY: 4:24
    │   │   │   ├─ WHITESPACE: 4:24
    │   │   │   │   ├─ Token(type=NEWLINE, content='\n', line=4, col=24)
    │   │   │   │   └─ Token(type=WHITESPACE, content='  ', line=5, col=0)
    │   │   │   ├─ STATEMENT: 5:2
    │   │   │   │   ├─ FUNNAME: 5:2
    │   │   │   │   │   └─ Token(type=WORD, content='add_library', line=5, col=2)
    │   │   │   │   ├─ LPAREN: 5:13
    │   │   │   │   │   └─ Token(type=LEFT_PAREN, content='(', line=5, col=13)
    │   │   │   │   ├─ ARGGROUP: 5:14
    │   │   │   │   │   ├─ PARGGROUP: 5:14
    │   │   │   │   │   │   ├─ ARGUMENT: 5:14
    │   │   │   │   │   │   │   └─ Token(type=WORD, content='hello', line=5, col=14)
    │   │   │   │   │   │   └─ Token(type=WHITESPACE, content=' ', line=5, col=19)
    │   │   │   │   │   └─ PARGGROUP: 5:20
    │   │   │   │   │       └─ ARGUMENT: 5:20
    │   │   │   │   │           └─ Token(type=UNQUOTED_LITERAL, content='hello.cc',␣
↪line=5, col=20)
    │   │   │   │   └─ RPAREN: 5:28
    │   │   │   │       └─ Token(type=RIGHT_PAREN, content=')', line=5, col=28)
    │   │   │   └─ WHITESPACE: 5:29
    │   │   │       └─ Token(type=NEWLINE, content='\n', line=5, col=29)
    │   │   └─ STATEMENT: 6:0
    │   │       ├─ FUNNAME: 6:0
    │   │       │   └─ Token(type=WORD, content='endif', line=6, col=0)
    │   │       ├─ LPAREN: 6:5
    │   │       │   └─ Token(type=LEFT_PAREN, content='(', line=6, col=5)
    │   │       ├─ ARGGROUP: 0:0
    │   │       └─ RPAREN: 6:6
    │   │           └─ Token(type=RIGHT_PAREN, content=')', line=6, col=6)
    │   └─ WHITESPACE: 6:7
    │       └─ Token(type=NEWLINE, content='\n', line=6, col=7)
```

9.3 Formatter: Layout Tree

As of version `0.4.0`, `cmake-format` will create a tree structure parallel to the parse tree and called the "layout tree". Each node in the layout tree points to at most one node in the parse tree. The structure of the layout tree is essentially the same as the parse tree with the following exceptions:

1. The primary argument group of a statement is expanded, so that the possible children of a `STATEMENT` layout node are: `ARGGROUP`, `ARGUMENT`, `COMMENT`, `FLAG`, `FUNNAME`, `KWARGROUP`.
2. `WHITESPACE` nodes containing less than two newlines are dropped, and not represented in the layout tree.

You can inspect the layout tree of a listfile by `cmake-format` with `--dump layout`. For example:

```
└─ BODY,(passno=0,wrap=F,ok=T) pos:(0,0) colextent:35
    ├─ STATEMENT,(passno=0,wrap=F,ok=T) pos:(0,0) colextent:35
    │   ├─ FUNNAME,(passno=0,wrap=F,ok=T) pos:(0,0) colextent:22
    │   ├─ LPAREN,(passno=0,wrap=F,ok=T) pos:(0,22) colextent:23
```

(continues on next page)

(continued from previous page)

```
    │   ├─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(0,23) colextent:34
    │   │   └─ KWARGGROUP,(passno=0,wrap=F,ok=T) pos:(0,23) colextent:34
    │   │       ├─ KEYWORD,(passno=0,wrap=F,ok=T) pos:(0,23) colextent:30
    │   │       └─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(0,31) colextent:34
    │   │           └─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(0,31) colextent:34
    │   │               └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(0,31) colextent:34
    │   └─ RPAREN,(passno=0,wrap=F,ok=T) pos:(0,34) colextent:35
    ├─ STATEMENT,(passno=0,wrap=F,ok=T) pos:(1,0) colextent:13
    │   ├─ FUNNAME,(passno=0,wrap=F,ok=T) pos:(1,0) colextent:7
    │   ├─ LPAREN,(passno=0,wrap=F,ok=T) pos:(1,7) colextent:8
    │   ├─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(1,8) colextent:12
    │   │   └─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(1,8) colextent:12
    │   │       └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(1,8) colextent:12
    │   └─ RPAREN,(passno=0,wrap=F,ok=T) pos:(1,12) colextent:13
    └─ FLOW_CONTROL,(passno=0,wrap=F,ok=T) pos:(2,0) colextent:29
        ├─ STATEMENT,(passno=0,wrap=F,ok=T) pos:(2,0) colextent:24
        │   ├─ FUNNAME,(passno=0,wrap=F,ok=T) pos:(2,0) colextent:2
        │   ├─ LPAREN,(passno=0,wrap=F,ok=T) pos:(2,2) colextent:3
        │   ├─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,3) colextent:23
        │   │   ├─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,3) colextent:6
        │   │   │   └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(2,3) colextent:6
        │   │   └─ KWARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,7) colextent:23
        │   │       ├─ KEYWORD,(passno=0,wrap=F,ok=T) pos:(2,7) colextent:10
        │   │       └─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,11) colextent:23
        │   │           └─ PARENGROUP,(passno=0,wrap=F,ok=T) pos:(2,11) colextent:23
        │   │               ├─ LPAREN,(passno=0,wrap=F,ok=T) pos:(2,11) colextent:12
        │   │               ├─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,12) colextent:22
        │   │               │   ├─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,12)␣
↪colextent:15
        │   │               │   │   └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(2,12)␣
↪colextent:15
        │   │               │   └─ KWARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,16)␣
↪colextent:22
        │   │               │       ├─ KEYWORD,(passno=0,wrap=F,ok=T) pos:(2,16)␣
↪colextent:18
        │   │               │       └─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,19)␣
↪colextent:22
        │   │               │           └─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(2,
↪19) colextent:22
        │   │               │               └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(2,
↪19) colextent:22
        │   │               └─ RPAREN,(passno=0,wrap=F,ok=T) pos:(2,22) colextent:23
        │   └─ RPAREN,(passno=0,wrap=F,ok=T) pos:(2,23) colextent:24
        ├─ BODY,(passno=0,wrap=F,ok=T) pos:(3,2) colextent:29
        │   └─ STATEMENT,(passno=0,wrap=F,ok=T) pos:(3,2) colextent:29
        │       ├─ FUNNAME,(passno=0,wrap=F,ok=T) pos:(3,2) colextent:13
        │       ├─ LPAREN,(passno=0,wrap=F,ok=T) pos:(3,13) colextent:14
        │       ├─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(3,14) colextent:28
        │       │   ├─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(3,14) colextent:19
        │       │   │   └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(3,14) colextent:19
        │       │   └─ PARGGROUP,(passno=0,wrap=F,ok=T) pos:(3,20) colextent:28
        │       │       └─ ARGUMENT,(passno=0,wrap=F,ok=T) pos:(3,20) colextent:28
        │       └─ RPAREN,(passno=0,wrap=F,ok=T) pos:(3,28) colextent:29
        └─ STATEMENT,(passno=0,wrap=F,ok=T) pos:(4,0) colextent:7
            ├─ FUNNAME,(passno=0,wrap=F,ok=T) pos:(4,0) colextent:5
            ├─ LPAREN,(passno=0,wrap=F,ok=T) pos:(4,5) colextent:6
            ├─ ARGGROUP,(passno=0,wrap=F,ok=T) pos:(4,6) colextent:6
```

(continues on next page)

(continued from previous page)

```
            └─ RPAREN,(passno=0,wrap=F,ok=T) pos:(4,6) colextent:7
```

9.4 Example file

The example file used to create the tree dumps above is::

```
cmake_minimum_required(VERSION 3.5)
project(demo)
if(FOO AND (BAR OR BAZ))
  add_library(hello hello.cc)
endif()
```

CHAPTER 10

Parser Algorithm

TODO(josh): add notes regarding the standard parse function and what it does with regard to positionals and kwargs. Then add notes about the new style parser which uses nested function calls. Note that the motivation for this is commands that take multiple forms. We need computational logic to look at the first argument to determine how to parse the rest of it.

CHAPTER 11

Automatic Parsers

cmake provides help commands which can print out the usage information for all of the builtin statements that it supports. You can get a list of commands with

```
cmake --help-command-list
```

And you can get the help text for a command with (for example):

```
cmake --help-command add_custom_command
```

In general (but not always) the usage string is given an a restructured-text block that looks like this:

```
::

 add_custom_command(TARGET <target>
                    PRE_BUILD | PRE_LINK | POST_BUILD
                    COMMAND command1 [ARGS] [args1...]
                    [COMMAND command2 [ARGS] [args2...] ...]
                    [BYPRODUCTS [files...]]
                    [WORKING_DIRECTORY dir]
                    [COMMENT comment]
                    [VERBATIM] [USES_TERMINAL])
```

The syntax of these usage strings isn't 100% consistent but if we could generate a parser that even understands *most* of these strings then that would greatly reduce the maintenance load.

11.1 Expect Objects

The output of the specification parser is an "Expect Tree". A tree of objects representing what is expected from a statement. If a sequence of tokens satisfies an expected subtree then the a corresponding parse tree is generated. If a mandatory expected subtree is not satisfied then an error is generated. If an optional expected subtree is not satisfied then the next sibling is tried.

11.2 Case studies

11.2.1 Inconsistent usage of angle brackets

Sometimes mandatory arguments are shown in angle brackets, sometimes not. I can't really figure a pattern for when they are used and when they are not.

11.2.2 Ellipses

Whether or not there is a space between an elipsis and the preceeding token seems to imply something about what is repeated.

```
add_custom_command(TARGET <target>
                   PRE_BUILD | PRE_LINK | POST_BUILD
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [BYPRODUCTS [files...]]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment]
                   [VERBATIM] [USES_TERMINAL])
```

Note that the elipsis for "COMMAND" is inside the bracket above, but is outside the bracket here:

```
add_dependencies(<target> [<target-dependency>]...)
```

11.2.3 Choices

Pipe character is used to separate choices.

```
configure_file(<input> <output>
               [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
               [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

Sometimes it's a mandatory choice

```
ctest_test([BUILD <build-dir>] [APPEND]
           [START <start-number>]
           [END <end-number>]
           [STRIDE <stride-number>]
           [EXCLUDE <exclude-regex>]
           [INCLUDE <include-regex>]
           [EXCLUDE_LABEL <label-exclude-regex>]
           [INCLUDE_LABEL <label-include-regex>]
           [EXCLUDE_FIXTURE <regex>]
           [EXCLUDE_FIXTURE_SETUP <regex>]
           [EXCLUDE_FIXTURE_CLEANUP <regex>]
           [PARALLEL_LEVEL <level>]
           [TEST_LOAD <threshold>]
           [SCHEDULE_RANDOM <ON|OFF>]
           [STOP_TIME <time-of-day>]
           [RETURN_VALUE <result-var>]
           [CAPTURE_CMAKE_ERROR <result-var>]
           [QUIET]
           )
```

Sometimes the choice is among literals, in which case there are no surrounding brackets.

```
file(GLOB <variable>
     [LIST_DIRECTORIES true|false] [RELATIVE <path>]
     [<globbing-expressions>...])
```

11.2.4 Manditory Sequence

In this case the literal pattern is listed inside the mandatory group pattern (angle brackets).

```
file(GENERATE OUTPUT output-file
     <INPUT input-file|CONTENT content>
     [CONDITION expression])
```

This one is pretty complex, and also demonstrates the nested bracket usage. I think the indication here is that "you must have one of these choices.

```
get_property(<variable>
             <GLOBAL             |
              DIRECTORY [dir]    |
              TARGET    <target> |
              SOURCE    <source> |
              INSTALL   <file>   |
              TEST      <test>   |
              CACHE     <entry>  |
              VARIABLE>
             PROPERTY <name>
             [SET | DEFINED | BRIEF_DOCS | FULL_DOCS])
```

11.2.5 Nested Optionals

```
install(TARGETS targets... [EXPORT <export-name>]
        [[ARCHIVE|LIBRARY|RUNTIME|OBJECTS|FRAMEWORK|BUNDLE|
          PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
         [DESTINATION <dir>]
         [PERMISSIONS permissions...]
         [CONFIGURATIONS [Debug|Release|...]]
         [COMPONENT <component>]
         [OPTIONAL] [EXCLUDE_FROM_ALL]
         [NAMELINK_ONLY|NAMELINK_SKIP]
        ] [...]
        [INCLUDES DESTINATION [<dir> ...]]
        )
```

11.2.6 Multiple Forms

```
string(SUBSTRING <string> <begin> <length> <output variable>)
string(STRIP <string> <output variable>)
string(GENEX_STRIP <input string> <output variable>)
string(COMPARE LESS <string1> <string2> <output variable>)
```

11.3 Conclusion

After implementing a prototype parser and testing it on some of the above cases it is clear that the help text is not very consistent and is likely to be very challenging to get an implementation that works reliably and knows when it fails. For example:

```
add_custom_command(TARGET <target>
                   PRE_BUILD | PRE_LINK | POST_BUILD
                   COMMAND command1 [ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [BYPRODUCTS [files...]]
                   [WORKING_DIRECTORY dir]
                   [COMMENT comment]
                   [VERBATIM] [USES_TERMINAL])
```

In this form of the command, the *PRE_BUILD PRE_LINK* or *POST_BUILD* argument is required. Normally it seems like they would put this in angle brackets as *<PRE_BUILD|PRE_LINK|POST_BUILD>* but they do not. So it's ambiguous where the pipes are splitting and what the groups are.

CHAPTER 12

Contributing to cmake-format

12.1 General Rules

1. Please use `pylint` and `flake8` to check your code for lint. There are config files in the repo.
2. There is a python `unittest` suite in `cmake_formattests.py`. Run with `python -Bm cmake_format.tests` (ensure modified code is on the python path). Alternatively you can use `cmake` (`make test`, `ninja test`, etc) or bazel (`bazel test ...`).
3. There's an `autopep8` config file in the repo as well. Feel free to use that to format the code. Note that `autopep8` and `pylint` disagree in a few places so using `autopep8` may require some manual edits afterward.

12.2 Build Systems

You don't need to "build" `cmake-format`, but there are `cmake` and `bazel` build rules that you can use if you find it convenient. For the most part your contribution is in good shape if the `test` and `lint` (cmake) build targets pass or, equivalently, if `bazel test ...` passes. If you add new files to the repository, please update the corresponding `CMakeLists.txt` and `BUILD` files to keep the lint and test configurations up to date.

12.3 Sidecar Tests

Whether or not to write your tests in cmake or in python will depend largely on the test. If you simply want to assert that a particular snippet of code is formatted in a specific way, writing the test in cmake is most natural. If any logic is needed, then writing in the body of a python function makes the most sense.

Test written in cmake are stored in `.cmake` sidecar files associated with python `unittest` cases (see `command_tests/*`).

The format of these tests are very simple. The test begins with a line comment in the form of:

```
# test: <test-name>
```

And ends with the start of the next test, the end of the file, or a line comment in the form of:

```
# end-test
```

If non-default configuration options are required, or if you would like to assert a lex, parse, or layout result, then you can specify that in an optional bracket comment immediately following the `# test:` sentinel. The bracket comment must begin and end on it's own line, such as:

```
#[=[
   line_width = 10
]=]
```

The content of the comment is parsed as python and variables are interpreted as configuration variables, except for the following:

- expect_lex - a list of lexer tokens indicated the expected lex result
- expect_parse - a tree of parse node names indicated the expected parse tree structure
- expect_layout - a tree of layout node names and geometries indicated the expected layout result.

A test called `test_<test-name>` will be generated at runtime. The test will involve `cmake-format` formatting the code snippet using the default configuration, and asserting that the output matches the input.

12.4 Debugging

12.4.1 Formatting

For the most part, cmake-format ignores white-space when it parses the list file into the parse tree so most of the formatting tests can be restricted to "idempotency" tests. In other words, you write down the listfile code in the format you expect and if `cmake-format` doesn't change it, the test passes.

12.4.2 Lexing, Parsing, Layout

If you suspect a problem with the lexing, parsing, or layout algorithms, try to create a minimial working example of the problem, and use the `--dump [lex|parse|layout]` command line option to inspect the output of the different phases.

Once you've identified the problem, the best thing to do is create a test asserting an expected output at the problematic phase. You can start by asserting an empty output. For example, something like:

```
# test: debug_my_case
#[=[
  expect_lex = []
]=]
if(something)
  your_cmake_code_here(foo bar baz)
else()
```

The test will fail, but the error message will print out the actual result. In the example above:

```
Second list contains 19 additional elements.
First extra element 0:
TokenType.WORD

- []
+ [TokenType.WORD,
+  TokenType.LEFT_PAREN,
+  TokenType.WORD,
+  TokenType.RIGHT_PAREN,
+  TokenType.NEWLINE,
+  TokenType.WHITESPACE,
+  TokenType.WORD,
+  TokenType.LEFT_PAREN,
+  TokenType.WORD,
+  TokenType.WHITESPACE,
+  TokenType.WORD,
+  TokenType.WHITESPACE,
+  TokenType.WORD,
+  TokenType.RIGHT_PAREN,
+  TokenType.NEWLINE,
+  TokenType.WORD,
+  TokenType.LEFT_PAREN,
+  TokenType.RIGHT_PAREN,
+  TokenType.NEWLINE]
```

You can copy-paste the actual output as the expected output as a starting point for your test. You can modify the "expected" specification to match what the output *should* be. Then, as you iterate, you can use the test to know when you've fixed the problem.

12.5 Pull Requests

Feel free to make a pull request on github, though please take note of the following rules and guidelines. These rules are enforced through the travis CI builds so if travis passes your submission is probably in good shape.

12.5.1 Squash your feature

Please squash your changes when issuing a pull request so that the request is for a single commit. This helps us move the patch from the public github mirror into the upstream respository. When updating your request, please squash additional commits (you will likely need to force-push to your feature branch).

12.5.2 Rebase before submit

Please rebase your patch on the current HEAD before submitting the pull request. This helps us to keep a tidy history. When we merge your commit we don't want to create graph connections across long regions of the git history. Travis will fail any pull request which is more than 10 commits behind the HEAD.

12.5.3 Sign your commit

When making a pull request, please sign the commit (use `git commit -S`).

12.5.4 Sign the copyright assignment

Please sign the copyright assignment agreement (details below) using the same PGP key you use to sign the commit, and please ensure that the key is available on the popular keyservers. Travis will fetch it from https://keyserver.ubuntu.com

12.6 Copyright Assignment

To sign the copyright assignment agreement the quick way, run:

```
python -Bm cmake_format.contrib.sign_ca
```

from the root of the repository.

For the long way, please follow this process:

1. Copy the file `cmake_format/contrib/individual_ca.txt` to some working directory as, e.g., `cmake-format-ca.txt.in`.
2. Replace the template strings at the bottom with your actual name and email address.
3. Sign the document with e.g.:

   ```
   gpg --output cmake-format-ca.txt --clearsign cmake-format-ca.txt.in``
   ```

   Please be sure to use the same pgp key that you'll be using to sign your commits.
4. Copy the asci-armored signature packet at the bottom of the signed document and paste it into `cmake_format/contrib/signature_db.json`. Include this change in your first pull request.

12.7 Un-Assigned contributions

In general, copyright for contributions should be assigned to the project. This should keep everything on the level should we find the need to offer `cmake-format` though additional licenses in the future.

If you'd like to make a significant contribution to `cmake-format` but don't agree to the terms of the copyright assignment please contact us to set up an alternate agreement. Otherwise, please consider filing a feature-request for changes you would like to see implemented.

CHAPTER 13

Release Notes

Details of changes can be found in the changelog, but this file will contain some high level notes and highlights from each release.

13.1 v0.6 series

13.1.1 v0.6.9

The parser now performs token look-ahead when parsing a comment within a statement. This allows it to determine whether a comment belongs to the current semantic node or one higher up in the tree (previously it would assign all comments to the most recent semantic node). This should prevent some unusual indentation of comments within deep statements.

Some `cmake-lint` crashes have been fixed and the test coverage has increased significantly. There are still some outstanding issues but it should crash less frequently and with more helpful information.

Detailed documentation for configuration options is now generated automatically, including default value, command line syntax, and example configuration file entry.

13.1.2 v0.6.8

There is now an embeded a database of known variables and properties. `cmake-lint` uses this database to implement checks on assignment/use of variables that are likely to be typos of a builtin variable name. There are also two new configuration options `vartags` and `proptags` that can be used to affect how the parser and formatter treat certain variables and properties.

Line comments within a statement are now consumed the same as line comments at block-scope. This means that your multiline mid-statement comments will be reflowed whereas they would previously have been left alone.

The CI Build has gotten a little more complicated. Generated documentation sources are no longer committed to the repository. Instead, they are pushed to a separate staging repository from which the read-the-docs pages are built.

13.1.3 v0.6.7

With this release, the specification format for custom commands has been updated to enable a number of new features. Positional argument groups now support "tags" which can be used to influence the formatting for some special cases. The format now also supports multiple positional argument groups. Lastly, there is a new **experimental** tool `cmake-genparsers` which can automatically generate parser specifications from your custom commands that use the standard `cmake_parse_arguments`.

There is a new configuration option `max_rows_cmdline` which applies only to shell commands and determines when a shell command should nest under it's keyword argument.

13.1.4 v0.6.6

The configuration datastructures have been overhauled and configuration options are now separated into different groupings based on which part of the processing pipeline they are relevent to. Legacy configuration files (without sections) are still supported, though they may be deprecated in the future. *cmake-format* can update your configuration file for you with the following command:

There is a new configuration option *explicit_trailing_pattern* which can be used to define a particular character sequence used by comments that are explicitly matched as trailing comments of an argument or statement. See the docs for more information.

Configuration files can now include additional configuration files. This might help keep configurations organized if you are maintaining a database of custom command definitions.

13.1.5 v0.6.5

This is largely a maintenance release, implementing explicit parse logic for all cmake commands that don't already have parsers. One additional configuration option is added allowing `cmake-lint` to globally ignore specific lint ids.

13.1.6 v0.6.4

This release includes implementations for many more lint checkers. Under the hood there was a pretty significant refactor of the parser, though none of the parse logic has changed. The refactor was primarily to split up the very large *parser* module, and to make it easier to access qualifiers of the parse tree during lint checks.

You can see a running list of all the implemented checkers at the docs.

13.1.7 v0.6.3

This release finally includes some progress on a long-standing goal: a cmake-linter built on the same foundation as the formatter. As of this release The `cmake-format` python package now comes with both the `cmake-format` and `cmake-lint` programs. The linter is still in a relatively early state and lacks many features, but should continue to grow with the formatter in future releases.

Along with the new linter, this release also includes some reorganization of the documentation in order to more clearly separate information about the different programs that are distributed from this repository.

13.1.8 v0.6.2

This is a maintenance release. Some additional command parsers have moved out of the standard parse model improving the parse of these commands. This release also includes some groundwork scripts to parse the usage strings in the cmake help text. Additionally:

- `--in-place` will preserve file mode
- The new `--check` command line option will not format the file, but exit with non-zero status if any changes would be made
- The new `--require-valid-layout` option will exit with non-zero status if an admissible layout is not found.

13.1.9 v0.6.1

This is primarily a documentation update. Some of the testing infrastructure has changed but no user-facing code has been modified.

13.1.10 v0.6.0

This release includes a significant refactor of the formatting logic. Details of the new algorithm are described in the documentation. As a result of the algorithm changes, some config options have changed too. The following config options are removed:

- `max_subargs_per_line` (see `max_pargs_hwrap`)
- `nest_threshold` (see `min_prefix_chars`)
- `algorithm_order` (see `layout_passes`)

And the following config options have been added:

- `max_subgroups_hwrap`
- `max_pargs_hwrap`
- `dangle_align`
- `min_prefix_chars`
- `max_prefix_chars`
- `max_lines_hwrap`
- `layout_passes`
- `enable_sort`

Also as a result of the algorithm changes, the default layout has changed. By default, `cmake-format` will now prefer to nest long lists rather than aligning them to the opening parenthesis of a statement. Also, due to the new configuration options, the output of `cmake-format` is likely to be different with your current configs.

Additionally, `cmake-format` will now tend to prefer a normal "horizontal" wrap for relatively long lists of positional arguments (e.g. source files in `add_library`) whereas it would previously prefer a vertical layout (one-entry per line). This is a consequence of an ambiguity between which positional arguments should be vertical versus which should be wrapped. Two planned features (layout tags and positional semantics) should help to provide enough control to get the layout you want in these lists.

I acknowledge that it is not ideal for formatting to change between releases but this is an unfortunate inevitability at this stage of development. The changes in this release elminate a number of inconsistencies and also adds the groundwork for future planned features and options. Hopefully we are getting close to a stable state and a 1.0 release.

13.2 v0.5 series

13.2.1 v0.5.5

This is a maintenance release fixing a few minor bugs and enhancements. One new feature is that the `--config` command line option now accepts a list of config files, which should allow for including multiple databases of command specifications —— v0.5.4 ——

This is a maintenance release fixing a couple of bugs and adding some missing documentation. One notable feature added is that, during in-place formatting, if the file content is unchanged `cmake-format` will no-longer write the file.

13.2.2 v0.5.3

This hotfix release fixes a bug that would crash cmake-format if no configuration file was present. It also includes some small under-the-hood changes in preparation for an overhaul of the formatting logic.

13.2.3 v0.5.2

This release fixes a few bugs and does some internal prep work for upcoming format algorithm changes. The documentation on the format algorithm is a little ahead of the code state in this release. Also, the documentation theme has changed to something based on read-the-docs (I hope you like it).

- Add missing forms of `add_library()` and `add_executable()`
- `--autosort` now defaults to `False` (it can be somewhat suprising) and it doesn't always get it right.
- Configuration options in `--help` and in the example configurations from `--dump-config` are now split into hopefully meaningful sections.
- `cmake-format` no longer tries to infer "keywords" or "flags" from `COMMAND` strings. This matching wasn't good enough as there is way too much variance in how programs design their command line options.

13.2.4 v0.5.1

The 0.5.0 release involved some pretty big changes to the parsing engine and introduced a new format algorithm. These two things combined unfortunately lead to a lot of new bugs. The full battery of pre-release tests wasn't run and so a lot of those issues popped up after release. Hopefully most of those are squashed in this release.

- Fixed lots of bugs introduced in 0.5.0
- `cmake-format` has a channel on discord now. Come chat about it at https://discord.gg/NgjwyPy

13.2.5 v0.5.0

- Overhauled the parser logic enabling arbitrary implementations of statement parsers. The generic statement parser is now implemented by the `standard_parse` function (or the `StandardParser` functor, which is used to load legacy `additional_commands`).

- New custom parser logic for deep cmake statements such as:
  - `install`
  - `file`
  - `ExternalProject_XXX`
  - `FetchContent_XXX`
- `cmake-format` can now sort your argument lists for you (such as lists of files). This enabled with the `autosort` config option. Some argument lists are inherently sortable (e.g. the list of sources supplied to `add_library` or `add_executable`). Other commands (e.g. `set()` which cannot be inferred sortable can be explicitly tagged using a comment at the beginning of the list. See the README for more information.
- A consequence of the above is that the parse tree for `set()` has changed, and so it's default formatting in many cases has also changed. You can restore the old behavior by adding the following to your config:

```
additional_commands = {
  "set": {
    "flags": ["FORCE", "PARENT_SCOPE"],
    "kwargs": {
      "CACHE": "*"
    }
  }
}
```

- The default command case has changed from `lower` to `canonical` (which is a new option). In most cases this is the same as `lower` but for some standard, non-builtin commands the canonical spelling is CamelCase (i.e. `ExternalProject_Add`).
- There is a new `cmake-annotate` program distributed with the package. It can generate semantic HTML renderings of your listfiles (see the documentation for details).

13.3 v0.4 series

13.3.1 v0.4.5

- Add travis CI configuration for public github repo

13.3.2 v0.4.4

- Add the ability to dump out markup parse lists for debugging.
- Add the ability to dump out a semantic HTML markup of a listfile, allowing for easy server-side semantic highlighting of documentation pages. See *cmake-annotate*.

13.3.3 v0.4.2

- Added the brand new `Visual Studio Code` extension, which can be found in the `vscode` marketplace! You can now use `cmake-format` to "Format Document" in *vscode*.
- Some new configuration options to allow user-specified literal fences and rulers in comment markup.
- New configuration options to preserve literal comment blocks at the start of your listfiles (intended for copyright statements), as well as to disable comment reflow alltogether.

- Fixed some bugs and improved some error messages

Enjoy!

CHAPTER 14

Changelog

14.1 v0.6 series

14.1.1 v0.6.9

- fix RTD push on Travis if the pseudo-release is already up-to-date
- added pattern for at-words to the lexer and new token type
- fix custom parsers didn't set lint spec for PositionalGroupNode
- don't pedantically error out on empty JSON config file
- parser can comprehend comments belonging to higher than the current depth in the semantic tree.
- cleanup README, move some stuff to documentation pointers
- generate the config options details page rather than hand writing it
- Closes: #109: Formatting of files containing @VARIBLE@ fails
- Closes: #122: parse_positionals consumes non-related comments after code
- Closes: #168: cmake-lint crashes on *add_library* function
- Closes: 031922e, 5bcc447, c07a668, efba824, 7eb2cb6

14.1.2 v0.6.8

- Reduce packaging depependency version numbers
- Add build rules to generate variable and property pattern lists
- Implement lint checks on assignment/use of variables that are "close" to builtins except for case.
- Move first_token from configuration object into format context
- Add line, col info to lex error message

- Fix wrong root parser for FetchContent_MakeAvailable
- Fix missing support for string integer npargs
- Fix missing spec for derived classes of PositionalGroupNode
- Fix on/off switch doesn't work inside a statement
- Fix extraneous whitespace inserted before line comment in some statements
- Add more helpful error message on failed configfile parse
- Move documentation build to build time and push documentation artifacts to an artifact repository
- Closes #162: cmake-lint crashes when evaluating *math*
- Closes #163: cmake-lint crashes when using *VERBATIM* in *add_custom_target*
- Closes #164: Internal error FetchContent_MakeAvailable
- Closes: 000bf9a, 6e4ef70, 85a3985, 9a3afa6, c297b3d, cf4570e

14.1.3 v0.6.7

- Add missing dependency on six
- Update pylint, flake8 used in CI
- Remove spurious config warning for some unfiltered command line options
- Add tags field to positional argument groups. Assign "file-list" tag to file lists from add_library and add_executable
- Remove "sortable" flag from root TreeNode class
- Custom commands can specify if a positional group is a command line
- Custom commands can specify multiple positional groups
- `max_pargs_hwrap` does not apply to `cmdline` groups
- Remove stale members from `TreeNode`
- Format extension.ts with two spaces
- Create a tool to generate parsers from `cmake_parse_args`
- Closes #139: Disable wrap for custom functions
- Closes #159: Missing dependency on six
- Closes: 6ef7d0d, 9669d02, cc60267, cf7ac49, cfa3c02, eefbde3, e75513a,
- Closes: f704714

14.1.4 v0.6.6

- Fix greedy match for bracket comments
- Implement some more readable error messages
- Add source support for sidecar tests
- Overhaul the configuration data structures, dividing configuration up among different classes.
- Remove configuration fields from config object __init__

- Add dump-config options to exclude helptext or defaults
- Implement explicit trailing comments
- Implement "include" from config files
- Move logging init into main() functions
- Closes #156: Linter exception when parsing certain multiline comments
- **Closes: 19baaf5, 200a6ed, 3435d8a, 4e6ca84, 6397d42, 9fbebee, b7fb891,** f097478

14.1.5 v0.6.5

- Fix bullet formatting in README
- Capture some input exceptions and print a more friendly error message
- Fix partialmethod docstrings in command tests
- Add a more detailed configuration description and samples
- Get rid of "extra" dictionary hack to get valid reflow bit out of `process_file` in `__main__.py`.
- Implement disabled lint codes config option
- Add preamble and summary to cmake-lint output
- Add test to ensure all cmake commands are in the database
- Implement all commands available in cmake 3.10.2
- Closes #154: cmake-lint: Human readable errors
- Closes: 06918d6, 0b6db3a, 26582bc, 7039c5c, c8d18f1, ea5583e, eb4fb01,
- Closes: 5abae5c

14.1.6 v0.6.4

- Split `parser.py` module into `parse/` sub package.
- Implement derived classes of *TreeNode* for for many parsers
- Refactor parse functions into class members of the appropriate *TreeNode* derived class
- Add lint checkers for: C0103, C0113, C0202, C0305, E0103, E0104, E0108, E0109, E1120, E1122, W0101
- Fix wrong token pattern for unquoted literal was including literal tabs and newlines (not literal "t" and "n").
- Closes #153: Spaces added to generator expressions

14.1.7 v0.6.3

- Add `ctest-to` program
- Add `cmake-lint` program
- Separate documentation by program
- Add some more detailed configuration documentation
- Make some of the config logic generic and push into a base class

- Some groundwork for cleaning up the config into different sections
- Fix externalproject_add_stepdependencies
- Closes #152: AssertionError on externalproject_add_stepdependencies

14.1.8 v0.6.2

- some initial work on cmake helptext/usage parser
- fix set_target_properties
- fix TOUCH_NOCREATE
- copymode during –in-place
- add –check command
- supress spurious warnings in tests
- create add_custom_target parser
- update add_custom_command parser with different forms
- fix target form of install command
- implement require-valid-layout and add tests
- sidecar tests don't need a companion pyfile
- fix some typos in documentation
- Closes #133: Better handling of un-wrappable, too-long lines
- Closes #134: Wrong formatting of install(TARGETS)
- Closes #140: add USES_TERMINAL to kwargs
- Closes #142: Add a –check option that doesn't write the files
- Closes #143: Broken file attributes after formatting
- Closes #144: Wrong warning about "file(TOUCH_NOCREATE . . .)"
- Closes #145: Bad formatting for *set_target_properties*
- Closes #147: foreach format
- Closes #150: Contributing documentation
- Closes #151: README.rst: fix two typos

14.1.9 v0.6.1

- consolidate `--config-file` command line flag variants
- add documentation on integration with `pre-commit`
- add documentation on sidecar tests
- simplify the tag format for sidecar tests
- add support of config options and lex/parse/layout assertions in sidecar tests
- add documentation on debugging with tests
- add tests to validate pull requests

- move most tests into sidecar files

14.1.10 v0.6.0

Significant refactor of the formatting logic.

- Move `format_tests` into `command_tests.misc_tests`
- Prototype sidecar tests for easier readability/maintainability
- ArgGroupNodes gain representation in the layout tree
- Get rid of `WrapAlgo`
- Eliminate vertical/nest as separate decisions. Nesting is just the wrap decision for StatementNode and KwargNode wheras vertical is the wrap decision for PargGroupnode and ArgGroupNode.
- Replace `algorithm_order` with `_layout_passes`
- Get rid of `default_accept_layout` and move logic into a member function
- Move configuration and `node_path` into new `StackContext`
- Stricter valid-child-set for most layout nodes

14.2 v0.5 series

14.2.1 v0.5.5

- Python config files now have `__file__` set in the global namespace
- Add parse support for `BYPRODUCTS` in `add_custom_command`
- Modify vscode extension cwd to better support subtree configuration files
- Fix vscode extension args type configuration
- Support multiple config files
- Closes #121: Support `BYPRODUCTS`
- Closes #123: Allow multiple config files
- Closes #125: Swap ordering of cwd location in vscode extension
- Closes #128: Include LICENSE.txt in sdist and wheel
- Closes #129: cmakeFormat.args in settings.json yields Incorrect type
- Closes #131: cmakeFormat.args is an array of items of type string

14.2.2 v0.5.4

- Don't write un-changed file content when doing in-place formatting
- Fix windows line-endings dropped during read
- Add documentation on how to add custom commands
- Fix yaml-loader returns None instead of empty dictionary for an empty yaml config file.
- Closes #114: Example of adding custom cmake functions/macros

- Closes #117: Fix handling of –dump-config with empty existing yaml config
- Closes #118: Avoid writing outfile unnecessarily
- Closes #119: Fix missing newline argument
- Closes #120: auto-line ending option not working correctly under Windows

14.2.3 v0.5.3

- add some configuration options for next format Refactor
- update documentation source generator scripts and run to get updated dynamic doc texts
- add a couple more case studies
- split reflow methods into smaller methods per case
- fix os.expanduser on None
- Closes #115: crash when no config file

14.2.4 v0.5.2

- add parsers for different forms of `add_library()` and `add_executable()`
- move `add_library`, `add_executable()` and `install()` parsers to their own modules
- don't infer sortability in `add_library` or `add_executable()` if the descriminator token might be a cmake variable hiding the descriminator spelling
- Split configuration options into different groups during dump and –help
- Refactor long `_reflow()` implementations, splitting into methods for the different wrap cases. This is in preparation for the next rev of the format algorithm.
- Add documentation on the format algorithm and some case studies.
- Autosort defaults to `False`
- Changed documentation theme to something based on rtd
- Get rid of `COMMAND` kwarg specialization
- Closes #111: Formatting breaks `add_library`
- Closes #112: expanduser on configfile_path

14.2.5 v0.5.1

- Fix empty kwarg can yield a parg group node with only whitespace children
- Fix `file(READ ...)` and `file(STRINGS ...)` parser kwargs using set syntax instead of dict syntax
- Fix agressive positional parser within conditional parser
- Fix missing endif, endwhile in parsemap
- Split parse functions out into separate modules for better organization
- Add more sanity tests for `file(...)`.
- Remove README from online docs, replace with expanded documentation for each README section

- Restore ability to accept paren-group in arbitrary parg-group
- Fix missing tests on travis
- Fix new tests using unicode literals (affects python2)
- Fix command parser after –
- Closes #104: Extra space for export targets
- Closes #106: Formatting of `file(READ)` fails
- Closes #107: multiline cmake commands
- Closes #108: Formatting of `file(STRING)` fails
- Closes #110: Formatting of Nested Expressions Fails

14.2.6 v0.5.0

- Implement canonical command case
- Canonicalize capitalization of keys in cmdspec
- Add README documentation regarding fences and enable/disable
- Statement parsers are now generic functions. Old standard parser remains for most statements, but some statements now have custom parsers.
- Implement deeper parse logic for `install()` and `file()` commands, improving the formatting of these statements.
- Implement input/output encoding configuration parameters
- Implement hashruler markup logic and preserve hashrulers if markup is disable or if configured to do so.
- Implement autosort and sortable tagging
- Separate cmake-annotate frontend
- Provider a `Loader=` to yaml `load()`
- Fix python3 lint
- Fix bad lexing of make-style variables
- Fix multiple hash chars `lstrip()ed` from comments
- Closes #62: Possible improvement on formatting “file”
- Closes #75: configurable positioning of flags
- Closes #87: Hash-rulers are stripped when markup disabled
- Closes #91: Add missing keyword arguments to project command
- Closes #95: added argument –encoding to allow for non-utf8
- Closes #98: Fix kwargs/flag index for non-lowercase functions
- Closes #100: Extra linebreak inserted when ‘$(‘ encountered
- Closes #101: Provide a Loader to yaml.load
- Closes #102: fences does not work as expected

14.3 v0.4 series

14.3.1 v0.4.5

- Fix testing instructions in README
- Fix dump-config instructions in README
- Remove numpy dependency
- Add travis CI configuration
- Fix some issues with lint under python3
- Closes #40
- Closes #76
- Closes #77
- Closes #80
- Fixes #82: Keyword + long coment + long argument asserts

14.3.2 v0.4.4

- Fix bug where rulers wouldn't break bulleted lists in comment markup
- Add missing flags COMPONENT and CONFIGURATIONS to command spec
- add `--dump markup` to dump the markup parse tree for debugging comment formatting behavior
- fix *invalid NoneType value* for *–literal-comment-pattern*
- shebang is preserved if present (without additional options)
- fix trailing comment of kwarg group consumes rparen
- add test to verify correct consumption of args matching outer kwargs
- add new quoted assignment pattern to lexer for cases like quoted compile definitions
- add *–dump html-stub* and *–dump html-page* listfile renderers
- Fixes #56: ignores boolean configuration values
- Closes #66: Positional argument of keyword incorrectly matched as keyword of containing command
- Resolves #73: Control of macro/function renaming
- Fixes #74: shebang in cmake scripts
- Fixes #79: BOM (Byte-order-mark) crashes parser
- Closes #81: Fix comment handling in kwarg group
- Fixes #85: commands: find_package broken
- Fixes #86: Breaking in Quotes

14.3.3 v0.4.3

- dump_config now dumps the active config, including loaded from file or modified by command line
- use cmake macros for cleaner listfiles
- fix argparse defaults override config file settings for boolean args

Closed issues:

- Fixes #70: ignores boolean configuration values

14.3.4 v0.4.2

- Add visual studio code extension
- Add algorithm order config option
- Add user specified fence regex config option
- Add user specified ruler regex config option
- Add config option to disable comment formatting altogether
- Fix get_config bug in `__main__`
- Fix missing elseif command specification
- Fix missing elseif/else paren spacing when specified
- Add enable_markup config option
- Fix kwargstack early breaking in conditionals
- Add some notes for developers.
- Add warning if formatter is inactive at the end of a print
- Add config options to preserve first comment or any matching a regex

Closed issues:

- Fixes #34: if conditions with many elements
- Closes #35: break_before_args
- Implements #42: user specified string for fencing
- Implements #43: allow custom string for rulers
- Fixes #45: config file not loaded properly
- Fixes #51: competing herustics for 2+ argument statements
- Implements #60: option to not reflow initial comment block
- Implements #61: add non-builtin commands
- Fixes #63: elseif like if
- Implements #65: warn if off doesn't have corresponding on
- Closes #67: global option to not format comments
- Fixes #68: seperate-ctrl-name-with-space

14.3.5 v0.4.1

- Add missing numpy dependency to setup.py
- Fix arg comments dont force vpack
- Fix arg comments dont force dangle parenthesis
- Add some missing function specifications

Closed issues:

- Fixes #53: add numpy as required
- Closes #54: more cmake commands
- Fixes #55: function with interior comment
- Fixes #56: function with trailing comment
- Fixes #59: improve export

14.3.6 v0.4.0

- Overhaul parser into a cleaner single-pass implementation that generates a more complete representation of the syntax tree.
- Parser now recognizes arbitrary nested command specifications. Keyword argument groups are formatted like statements.
- Complete rewrite of formatter (see docs for design)
- Support line comments inside statements and argument groups
- Add some additional command specifications
- Add `--dump [lex|parse|layout]` debug commands
- `--dump-config` dumps the active configuration (after loading)
- Add keyword case correction
- Improve layout of complicated boolean expressions

Closed issues:

- Implements #10: treat COMPONENT keyword different
- Implements #37: –dump-config dumps current config
- Implements #39: always wrap for certain functions
- Fixes #46: leading comment in function body
- Fixes #47: function argument incorrectly appended
- Implements #48: improve install `target_*`
- Fixes #49: removes entire while() sections
- Fixes #50: indented comments appended to preceding line

14.4 v0.3 series

14.4.1 v0.3.6

- Implement "auto" line ending option *#27*
- Implement command casing *#29*
- Implement stdin as an input file *#30*

Closed issues:

14.4.2 v0.3.5

- Fix #28: lexing pattern for quoted strings with escaped quotes
- Add lex tests for quoted strings with escaped quotes
- Fix windows format test

Closed issues:

14.4.3 v0.3.4

- Don't use tempfile.NamedTemporaryFile because it has different (and, honestly, buggy behavior) compared to codecs.open() or io.open()
- Use io.open() instead of codecs.open(). I'm not sure why to prefer one over the other but since io.open is more or less required for printing to stdout I'll use io.open for everything
- Lexer consumes windows line endings as line endings
- Add inplace invocation test
- Add line ending configuration parameter
- Add configuration parameter command line documentation
- Add documentation to python config file dump output
- Strip trailing whitespace and normalize line endings in bracket comments

14.4.4 v0.3.3

- Convert all string literals in format.py to unicode literals
- Added python3 tests
- Attempt to deal with python2/python3 string differences by using codecs and io modules where appropriate. I probably got this wrong somewhere.
- Fix missing comma in config file matching

Closed issues:

- Implement #13: option to dangle parenthesis
- Fix #17: trailing comment stripped from commands with no arguments
- Fix #21: corruption upon trailing whitespace

- Fix #23: wrapping long arguments has some weird extra newline or missing indentation space.
- Fix #25: cannot invoke cmake-format with python3

14.4.5 v0.3.2

- Move configuration to it's own module
- Add lexer/parser support for bracket arguments and bracket comments
- Make stable_wrap work for any `prefix` / `subsequent_prefix`.
- Preserve scope-level bracket comments verbatim
- Add markup module with parse/format support for rudimentary markup in comments including nested bulleted and enumerated lists, and fenced blocks.
- Add pyyaml as an extra dependency in pip configuration

Closed issues:

- Fix #16: argparse defaults always override config

14.4.6 v0.3.1

- use exec instead of execfile for python3 compatibility

14.4.7 v0.3.0

- fix #2 : parser matching builtin logical expression names should not be case sensitive
- fix #3 : default code used to read `long_description` can't decode utf8
- implement #7 : add configuration option to separate control statement or function name from parenthesis
- implement #9 : allow configuration options specified from command line
- Add support for python as the configfile format
- Add `--dump-config` option
- Add support for "separator" lines in comments. Any line consisting of only five or more non-alphanum characters will be preserved verbatim.
- Improve python3 support by using `print_function`

Closed issues:

14.5 v0.2 series

14.5.1 v0.2.1

- fix bug in reflow if text goes to exactly the end of the line
- add python module documentation to sphinx autodoc
- make formatting of COMMANDs a bit more compact

14.5.2 v0.2.0

- add unit tests using python unit test framework
- accept configuration as yaml or json
- Implemented custom cmake AST parser, getting rid of dependency on cmlp
- Removed static global command configuration
- If no configuration file specified, search for a file based on the input file path.
- Moved code out of `__main__.py` and into modules
- More documentation and general cleanup
- Add `setup.py`
- Tested on a production codebase with 350+ listfiles and a manual scan of changes looked good, and the build seems to be healthy.

CHAPTER 15

cmake_format package

15.1 Module contents

Parse cmake listfiles and format them nicely

15.2 Submodules

15.3 cmake_format.configuration module

class cmake_format.configuration.**Configuration**(***kwargs*)

Bases: cmake_format.config_util.ConfigObject

Various configuration options and parameters

encode
Implements the descriptor interface for nested configuration objects.

format
Implements the descriptor interface for nested configuration objects.

lint
Implements the descriptor interface for nested configuration objects.

markup
Implements the descriptor interface for nested configuration objects.

misc
Implements the descriptor interface for nested configuration objects.

parse
Implements the descriptor interface for nested configuration objects.

resolve_for_command(*command_name*, *config_key*, *default_value=None*)
Check for a per-command value or override of the given configuration key and return it if it exists. Otherwise return the global configuration value for that key.

class cmake_format.configuration.**EncodingConfig**(***kwargs*)
Bases: cmake_format.config_util.ConfigObject

Options affecting file encoding

emit_byteorder_mark = False

input_encoding = 'utf-8'

output_encoding = 'utf-8'

class cmake_format.configuration.**FormattingConfig**(***kwargs*)
Bases: cmake_format.config_util.ConfigObject

Options affecting formatting.

always_wrap = []

autosort = False

command_case = 'canonical'

dangle_align = 'prefix'

dangle_parens = False

enable_sort = True

keyword_case = 'unchanged'

layout_passes = {}

line_ending = 'unix'

line_width = 80

linewidth

max_lines_hwrap = 2

max_pargs_hwrap = 6

max_prefix_chars = 10

max_rows_cmdline = 2

max_subgroups_hwrap = 2

min_prefix_chars = 4

require_valid_layout = False

separate_ctrl_name_with_space = False

separate_fn_name_with_space = False

set_line_ending(*detected*)

tab_size = 2

class cmake_format.configuration.**LinterConfig**(***kwargs*)
Bases: cmake_format.config_util.ConfigObject

Options affecting the linter

disabled_codes = []

function_pattern = '[0-9a-z_]+'

global_var_pattern = '[0-9A-Z][0-9A-Z_]+'

internal_var_pattern = '_[0-9A-Z][0-9A-Z_]+'

keyword_pattern = '[0-9A-Z_]+'

local_var_pattern = '[0-9a-z_]+'

macro_pattern = '[0-9A-Z_]+'

max_arguments = 5

max_branches = 12

max_conditionals_custom_parser = 2

max_localvars = 15

max_returns = 6

max_statement_spacing = 1

max_statements = 50

min_statement_spacing = 1

private_var_pattern = '_[0-9a-z_]+'

public_var_pattern = '[0-9A-Z][0-9A-Z_]+'

class cmake_format.configuration.**MarkupConfig**(***kwargs*)

Bases: cmake_format.config_util.ConfigObject

Options affecting comment reflow and formatting.

bullet_char = '*'

canonicalize_hashrulers = True

enable_markup = True

enum_char = '.'

explicit_trailing_pattern = '#<'

fence_pattern = '^\\s*([`~]{3}[`~]*)(.*)$'

first_comment_is_literal = False

hashruler_min_length = 10

literal_comment_pattern = None

ruler_pattern = '^\\s*[^\\w\\s]{3}.*[^\\w\\s]{3}$'

class cmake_format.configuration.**MiscConfig**(***kwargs*)

Bases: cmake_format.config_util.ConfigObject

Miscellaneous configurations options.

per_command = {}

class cmake_format.configuration.**ParseConfig**(***kwargs*)

Bases: cmake_format.config_util.ConfigObject

Options affecting listfile parsing

additional_commands = {'foo': {'flags': ['BAR', 'BAZ'], 'kwargs': {'DEPENDS': '*',

```
proptags = []
vartags = []
```

15.4 cmake_format.commands module

Command specifications for cmake built-in commands.

class cmake_format.commands.**CommandSpec**(*name*, *pargs=None*, *flags=None*, *kwargs=None*)
Bases: object

A command specification is primarily a dictionary mapping keyword arguments to command specifications. It also includes a command name and number of positional arguments

add(*name*, *pargs=None*, *flags=None*, *kwargs=None*)

add_conditional(*name*)

is_flag(*key*)

is_kwarg(*key*)

cmake_format.commands.**add_standard_nonbuiltins**(*fn_spec*)
Add commands provided by "standard" listfiles

cmake_format.commands.**get_default_config**()
Return the default per-command configuration database

cmake_format.commands.**get_fn_spec**()
Return a dictionary mapping cmake function names to a dictionary containing kwarg specifications.

cmake_format.commands.**make_conditional_spec**(*name=None*)

cmake_format.commands.**parse_pspec**(*pargs*, *flags*)
Parse a positional argument specification.

15.5 cmake_format.common module

class cmake_format.common.**EnumObject**(*value*)
Bases: object

Simple enumeration base. Design inspired by clang python bindings BaseEnumeration. Subclasses must provide class member _id_map.

as_dict()

classmethod assign_names()

classmethod from_id(*qid*)

classmethod from_name(*name*)

classmethod get(*name*, *default=None*)

name
Get the enumeration name of this value.

classmethod register_value(*value*, *obj*)

exception cmake_format.common.**InternalError**(*msg=None*)
Bases: Exception

Raised when we encounter something we do not expect, indicating a problem with the code itself.

exception cmake_format.common.**UserError**(*msg=None*)
Bases: Exception

Raised when we encounter a fatal problem with usage: e.g. parse error, config error, input error, etc.

cmake_format.common.**stable_wrap**(*wrapper*, *paragraph_text*)
textwrap doesn't appear to be stable. We run it multiple times until it converges

15.6 cmake_format.formatter module

class cmake_format.formatter.**ArgGroupNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

A group of arguments. This is the single child node of either a *StatementNode* or *KwargGroupNode* which then contains any further group nodes.

has_terminal_comment()
An ArgGroup is a container for one or more PARGGROUP, FLAGGROUP, or KWARGGROUP subtrees. Any terminal comment will belong to one of it's children.

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**AssertTypeDescriptor**(*assert_type*, *hidden_name*)
Bases: object

class cmake_format.formatter.**BodyNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

Top-level node for a given "scope" depth. This node is the root of a document, or the root of any nested statement scopes.

class cmake_format.formatter.**CommentNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

A line comment or bracket comment. If parented by a group node then this comment acts as an argument. If parented by a scalar node, then this comment acts like an argument comment.

is_tag()

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**Cursor**(*x*, *y*)
Bases: object

Lightweight class to encode integer positions in a 2d grid.

- x = row
- y = cols

clone()
Return a new *Cursor* object with the same value as this one.

class cmake_format.formatter.**CursorFile**(*config*)
Bases: object

assert_at(*cursor*)

assert_lt(*cursor*)

cursor

forge_cursor(*cursor*)

getvalue()

write(*copy_text*)

write_at(*cursor*, *text*)

class cmake_format.formatter.**FlowControlNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

Top-Level node composed of a flow-control statement and it's associated *BodyNodes*.

class cmake_format.formatter.**KwargGroupNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

A keyword argument group. Contains a keyword, followed by an argument group.

has_terminal_comment()
Return true if this node has a terminal line comment. In particular, this implies that no other node may be packed at the output cursor of this node's layout, and a line-wrap is required.

name
The class name of the derived node type.

class cmake_format.formatter.**LayoutNode**(*pnode*)
Bases: object

An element in the format/layout tree. The structure of the layout tree mirrors that of the parse tree. We could store this info the nodes of the parse tree itself but it's a little cleaner to keep the functionality separate I think.

children
A list of children layout nodes

colextent
The column index of the right-most character in the layout of the subtree rooted at this node. In other words, the width of the bounding box for the subtree rooted at this node.

static create(*pnode*)
Create a new layout node associated with then given parser node.

get_depth()
Compute and return the depth of the subtree rooted at this node. The depth of the tree is the depth of the deepest (leaf) descendant.

has_terminal_comment()
Return true if this node has a terminal line comment. In particular, this implies that no other node may be packed at the output cursor of this node's layout, and a line-wrap is required.

lock(*config*, *stmt_depth=0*)
Lock the tree structure (topology) and prevent further updates. This is mostly for sanity checking. It also computes topological statistics such as *stmt_depth* and *subtree_depth*, and replaces the mutable list of children with an immuatable tuple.

name
The class name of the derived node type.

next_sibling()

node_type
Return the *NodeType* of the corresponding parser node that generated this layout node.

passno
The active pass-number which contributed the current layout of the subtree rooted at this node.

position
A cursor with the (row,col) of the first (i.e. top,left) character in the subtree rooted at this node.

reflow(*stack_context*, *cursor*, *parent_passno=0*)
(re-)compute the layout of this node under the assumption that it should be placed at the given *cursor* on the current *parent_passno*.

reflow_valid
A boolean flag indicating whether or not the current layout is accepted. If False, then further layout passes are required.

rowextent

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**OnOffSwitchNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

Holds a special-case line comment token such as `# cmake-format: off` or `# cmake-format: on`

has_terminal_comment()
Return true if this node has a terminal line comment. In particular, this implies that no other node may be packed at the output cursor of this node's layout, and a line-wrap is required.

name
The class name of the derived node type.

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**ParenGroupNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

A parenthetical group. According to cmake syntax rules, this necessarily implies a boolean logical expression.

has_terminal_comment()
Return true if this node has a terminal line comment. In particular, this implies that no other node may be packed at the output cursor of this node's layout, and a line-wrap is required.

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**ParenNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

Holds parenthesis '(' or ')' for statements or boolean groups.

name
The class name of the derived node type.

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**PargGroupNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

A group of positional arguments.

has_terminal_comment()
Return true if this node has a terminal line comment. In particular, this implies that no other node may be packed at the output cursor of this node's layout, and a line-wrap is required.

lock(*config*, *stmt_depth=0*)
Lock the tree structure (topology) and prevent further updates. This is mostly for sanity checking. It also computes topological statistics such as *stmt_depth* and *subtree_depth*, and replaces the mutable list of children with an immuatable tuple.

class cmake_format.formatter.**ScalarNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

Holds scalar tokens such as statement names, parentheses, or keyword or positional arguments.

has_terminal_comment()
Return true if this node has a terminal line comment. In particular, this implies that no other node may be packed at the output cursor of this node's layout, and a line-wrap is required.

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**StackContext**(*config*, *first_token=None*)
Bases: object

Aggregate information about the current stack. This object is passed down through all of the nested `reflow()` function calls.

push_node(*node*)
Push *node* onto the *node_path* and yield a context manager. Pop *node* off of *node_path* when the context manager *__exit__()s*

class cmake_format.formatter.**StatementNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

Top-level node for a statement.

get_prefix_width(*config*)

name
The class name of the derived node type.

reflow(*stack_context*, *cursor*, *_=0*)
(re-)compute the layout of this node under the assumption that it should be placed at the given *cursor* on the current *parent_passno*.

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**WhitespaceNode**(*pnode*)
Bases: *cmake_format.formatter.LayoutNode*

A series of newlines

write(*config*, *ctx*)
Output text content given the currently configured layout.

class cmake_format.formatter.**WriteContext**(*config*, *infile_content*)
Bases: object

Global state for the writing functions

is_active()

cmake_format.formatter.**clamp**(*value*, *min_value*, *max_value*)
Simple double-ended saturation function.

cmake_format.formatter.**count_arguments**(*children*)
Count the number of positional arguments (excluding line comments and whitespace) within a parg group.

cmake_format.formatter.**count_subgroups**(*children*)
Count the number of positional or kwarg sub groups in an argument group. Ignore comments, and assert that no other types of children are found.

cmake_format.formatter.**create_box_tree**(*pnode*)
Recursively construct a layout tree from the given parse tree

cmake_format.formatter.**dump_tree**(*nodes*, *outfile=None*, *indent=None*)
Print a tree of node objects for debugging purposes

cmake_format.formatter.**dump_tree_for_test**(*nodes*, *outfile=None*, *indent=None*, *increment=None*)
Print a tree of node objects for debugging purposes

cmake_format.formatter.**dump_tree_upto**(*nodes*, *history*, *outfile=None*, *indent=None*)
Print a tree of node objects for debugging purposes

cmake_format.formatter.**filename_node_key**(*layout_node*)
Return the sort key for sortable arguments nodes. This is the case-insensitive spelling of the first token in the node.

cmake_format.formatter.**format_comment_lines**(*node*, *stack_context*, *line_width*)
Reflow comment lines into the given line width, parsing markup as necessary.

cmake_format.formatter.**get_comment_lines**(*config*, *node*)
Given a comment node, iterate through it's tokens and generate a list of textual lines.

cmake_format.formatter.**get_scalar_sequence_len**(*box_children*)

cmake_format.formatter.**is_line_comment**(*node*)
Return true if the node is a pure parser node holding a line comment (i.e. not a bracket comment)

cmake_format.formatter.**layout_tree**(*parsetree_root*, *config*, *linewidth=None*, *first_token=None*)
Top-level function to construct a layout tree from a parse tree, and then iterate through layout passes until the entire tree is satisfactory. Returns the root of the layout tree.

cmake_format.formatter.**need_paren_space**(*spelling*, *config*)
Return whether or not we need a space between the statement name and the starting parenthesis. This aggregates the logic of the two configuration options *separate_ctrl_name_with_space* and *separate_fn_name_with_space*.

cmake_format.formatter.**normalize_line_endings**(*instr*)
Remove trailing whitespace and replace line endings with unix line endings. They will be replaced with config.endl during output

cmake_format.formatter.**sort_arguments**(*children*)

cmake_format.formatter.**test_string**(*nodes*, *indent=None*, *increment=None*)

cmake_format.formatter.**tree_string**(*nodes*, *history=None*)

cmake_format.formatter.**write_tree**(*root_box*, *config*, *infile_content*)
Format the tree for size only, then print all of the boxes to outfile

15.7 cmake_format.lexer module

class cmake_format.lexer.**SourceLocation**
Bases: tuple

Named tuple of (line, col, offset)

col

line

offset

class cmake_format.lexer.**Token**(*tok_type*, *spelling*, *index*, *begin*, *end*)
Bases: object

Lexical unit of a listfile.

content

count_newlines()

get_location()

location()

class cmake_format.lexer.**TokenType**(*value*)
Bases: *cmake_format.common.EnumObject*

ATWORD = TokenType.ATWORD

BRACKET_ARGUMENT = TokenType.BRACKET_ARGUMENT

BRACKET_COMMENT = TokenType.BRACKET_COMMENT

BYTEORDER_MARK = TokenType.BYTEORDER_MARK

COMMENT = TokenType.COMMENT

DEREF = TokenType.DEREF

FORMAT_OFF = TokenType.FORMAT_OFF

FORMAT_ON = TokenType.FORMAT_ON

LEFT_PAREN = TokenType.LEFT_PAREN

NEWLINE = TokenType.NEWLINE

NUMBER = TokenType.NUMBER

QUOTED_LITERAL = TokenType.QUOTED_LITERAL

RIGHT_PAREN = TokenType.RIGHT_PAREN

UNQUOTED_LITERAL = TokenType.UNQUOTED_LITERAL

WHITESPACE = TokenType.WHITESPACE

WORD = TokenType.WORD

cmake_format.lexer.**get_first_non_whitespace_token**(*tokens*)
Return the first token in the list that is not whitespace, or None

cmake_format.lexer.**main**()
Dump tokenized listfile to stdout for debugging.

cmake_format.lexer.**parse_bracket_argument**(*text*)

cmake_format.lexer.**parse_bracket_comment**(*text*)

cmake_format.lexer.**tokenize**(*contents*)
Scan a string and return a list of Token objects representing the contents of the cmake listfile.

15.8 cmake_format.markup module

Functions for parsing comments in markup

class cmake_format.markup.**CommentItem**(*kind*)
Bases: object

class cmake_format.markup.**CommentType**(*value*)
Bases: *cmake_format.common.EnumObject*

BULLET_LIST = CommentType.BULLET_LIST

ENUM_LIST = CommentType.ENUM_LIST

FENCE = CommentType.FENCE

NOTE = CommentType.NOTE

PARAGRAPH = CommentType.PARAGRAPH

RULER = CommentType.RULER

SEPARATOR = CommentType.SEPARATOR

VERBATIM = CommentType.VERBATIM

cmake_format.markup.**format_item**(*config*, *line_width*, *item*)
Return lines of formatted text based on the typeof markup

cmake_format.markup.**format_items**(*config*, *line_width*, *items*)
Return lines of formatted text for the sequence of items within a comment block

cmake_format.markup.**is_hashruler**(*item*)
Return true if the markup item is a hash ruler, i.e.:

```
##############################
# Like this ^^^ or this vvv
##############################
```

cmake_format.markup.**parse**(*lines*, *config=None*)
Parse comment lines. Returns objects of different formatable entities

15.9 cmake_format.parse_funs module

Statement parser functions

cmake_format.parse_funs.**get_legacy_parse**(*cmdspec*)
Construct a parse tree from a legacy command specification

cmake_format.parse_funs.**get_parse_db**()
Returns a dictionary mapping statement name to parse functor for that statement.

cmake_format.parse_funs.**split_legacy_spec**(*cmdspec*)
Split a legacy specification object into pargs, kwargs, and flags

15.10 cmake_format.parse module

class cmake_format.parse.**MockEverything**
Bases: object

Dummy object which implements any interface by mocking all functions with an empty implementation that returns None

class cmake_format.parse.**ParseContext**(*parse_db=None*, *lint_ctx=None*, *config=None*)
Bases: object

Global context passed through every function in the parse stack.

pusharg(*node*)

cmake_format.parse.**parse**(*tokens*, *ctx=None*)
digest tokens, then layout the digested blocks.

15.11 cmake_format.render module

Parse cmake listfiles and emit an html file containing semantic and syntactic annotations.

cmake_format.render.**dump_html**(*node*, *outfile*)
Write to *outfile* an html annoted version of the listfile which has been parsed into the parse tree rooted at *node*

cmake_format.render.**get_html**(*node*, *fullpage=False*)
Return a string containing html markup of the annoted listfile which has been parsed into the parse tree rooted at *node*.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

K

L

M