This manual covers TextMate 1.5.1 — Go to TextMate 2 manual

12 Language Grammars

Language grammars are used to assign names to document elements such as keywords, comments, strings or similar. The purpose of this is to allow styling (syntax highlighting) and to make the text editor "smart" about which context the caret is in. For example you may want a key stroke or tab trigger to act differently depending on the context, or you may want to disable spell check as you type those portions of your text document which are not prose (e.g. HTML tags).

The language grammar is used only to parse the document and assign names to subsets of this document. Then scope selectors can be used for styling, preferences and deciding how keys and tab triggers should expand.

For a more thorough introduction to this concept see the introduction to scopes blog post.

12.1 Example Grammar

You can create a new language grammar by opening the bundle editor (Window → Show Bundle Editor) and select "New Language" from the add button in the lower left corner.

This will give you a starting grammar which will look like the one below, so let us start by explaining that.

```
 1  {  scopeName = 'source.untitled';
 2     fileTypes = ( );
 3     foldingStartMarker = '\{\s*$';
 4     foldingStopMarker = '^\s*\}';
 5     patterns = (
 6        {  name = 'keyword.control.untitled';
 7           match = '\b(if|while|for|return)\b';
 8        },
 9        {  name = 'string.quoted.double.untitled';
10           begin = '"';
11           end = '"';
12           patterns = (
13              {  name = 'constant.character.escape.untitled';
14                 match = '\\.';
15              }
16           );
17        },
18     );
19  }
```

The format is the property list format and at the root level there are five key/value pairs:

- `scopeName` (line 1) — this should be a unique name for the grammar, following the convention of being a dot-separated name where each new (left-most) part specializes the name. Normally it would be a two-part name where the first is either `text` or `source` and the second is the name of the language or document type. But if you are specializing an existing type, you probably want to derive the name from the type you are specializing. For example Markdown is `text.html.markdown` and Ruby on Rails (`rhtml` files) is `text.html.rails`. The advantage of deriving it from (in this case) `text.html` is that everything which works in the `text.html` scope will also work in the `text.html.«something»` scope (but with a lower precedence than something specifically targeting `text.html.«something»`).

- `fileTypes` (line 2) — this is an array of file type extensions that the grammar should (by default) be used with. This is referenced when TextMate does not know what grammar to use for a file the user opens. If however the user selects a grammar from the language pop-up in the status bar, TextMate will remember that choice.

- `foldingStartMarker` / `foldingStopMarker` (line 3-4) — these are regular expressions that lines (in the document) are matched against. If a line matches one of the patterns (but not both), it becomes a folding marker (see the foldings section for more info).

- `patterns` (line 5-18) — this is an array with the actual rules used to parse the document. In this example there are two rules (line 6-8 and 9-17). Rules will be explained in the next section.

There are two additional (root level) keys which are not used in the example:

- `firstLineMatch` — a regular expression which is matched against the first line of the document (when it is first loaded). If it matches, the grammar is used for the document (unless there is a user override). Example: `^#!/.*\bruby\b`.

- `repository` — a dictionary (i.e. key/value pairs) of rules which can be included from other places in the grammar. The key is the name of the rule and the value is the actual rule. Further explanation (and example) follow with the description of the `include` rule key.

12.2 Language Rules

A language rule is responsible for matching a portion of the document. Generally a rule will specify a name which gets assigned to the part of the document which is matched by that rule.

There are two ways a rule can match the document. It can either provide a single regular expression, or two. As with the `match` key in the first rule above (lines 6-8), everything which matches that regular expression will then get the name specified by that rule. For example the first rule above assigns the name `keyword.control.untitled` to the following keywords: `if`, `while`, `for` and `return`. We can then use a scope selector of `keyword.control` to have our theme style these keywords.

The other type of match is the one used by the second rule (lines 9-17). Here two regular expressions are given using the `begin` and `end` keys. The name of the rule will be assigned from where the begin pattern matches to where the end pattern matches (including both matches). If there is no match for the end pattern, the end of the document is used.

In this latter form, the rule can have sub-rules which are matched against the part between the begin and end matches. In our example here we match strings that start and end with a quote character and escape characters are marked up as `constant.character.escape.untitled` inside the matched strings (line 13-15).

*Note that the regular expressions are matched against only a **single line of the document** at a time. That means it is **not possible to use a pattern that matches multiple lines***. The reason for this is technical: being able to restart the parser at an arbitrary line and having to re-parse only the minimal number of lines affected by an edit. In most situations it is possible to use the begin/end model to overcome this limitation.

12.3 Rule Keys

What follows is a list of all keys which can be used in a rule.

- `name` — the name which gets assigned to the portion matched. This is used for styling and scope-specific settings and actions, which means it should generally be derived from one of the standard names (see naming conventions later).

- `match` — a regular expression which is used to identify the portion of text to which the name should be assigned. Example: `'\b(true|false)\b'`.

- `begin`, `end` — these keys allow matches which span several lines and must both be mutually exclusive with the `match` key. Each is a regular expression pattern. `begin` is the pattern that starts the block and `end` is the pattern which ends the block. Captures from the `begin` pattern can be referenced in the `end` pattern by using normal regular expression back-references. This is often used with here-docs, for example:

  ```
  {   name = 'string.unquoted.here-doc';
      begin = '<<(\w+)';  // match here-doc token
      end = '^\1$';       // match end of here-doc
  }
  ```

  A `begin`/`end` rule can have nested patterns using the `patterns` key. For example we can do:

  ```
  {  begin = '<%'; end = '%>'; patterns = (
        { match = '\b(def|end)\b'; … },
        …
     );
  };
  ```

  The above will match `def` and `end` keywords inside a `<% … %>` block (though for embedded languages see info about the `include` key later).

- `contentName` — this key is similar to the `name` key but only assigns the name to the text **between** what is matched by the `begin`/`end` patterns. For example to get the text between `#if 0` and `#endif` marked up as a comment, we would do:

  ```
  {  begin = '#if 0(\s.*)?$'; end = '#endif';
     contentName = 'comment.block.preprocessor';
  };
  ```

- `captures`, `beginCaptures`, `endCaptures` — these keys allow you to assign attributes to the captures of the `match`, `begin`, or `end` patterns. Using the `captures` key for a `begin`/`end` rule is short-hand for giving both `beginCaptures` and `endCaptures` with same values.

The value of these keys is a dictionary with the key being the capture number and the value being a dictionary of attributes to assign to the captured text. Currently `name` is the only attribute supported. Here is an example:

```
{  match = '(@selector\()(.*?)(\))';
   captures = {
      1 = { name = 'storage.type.objc'; };
      3 = { name = 'storage.type.objc'; };
   };
};
```

In that example we match text like `@selector(windowWillClose:)` but the `storage.type.objc` name will only be assigned to `@selector(` and `)`.

- `include` — this allows you to reference a different language, recursively reference the grammar itself or a rule declared in this file's repository.

  1. To reference another language, use the scope name of that language:

     ```
     {  begin = '<\?(php|=)?'; end = '\?>'; patterns = (
           { include = "source.php"; }
        );
     }
     ```

  2. To reference the grammar itself, use `$self`:

     ```
     {  begin = '\('; end = '\)'; patterns = (
           { include = "$self"; }
        );
     }
     ```

  3. To reference a rule from the current grammars repository, prefix the name with a pound sign (`#`):

     ```
     patterns = (
        {  begin = '"'; end = '"'; patterns = (
              { include = "#escaped-char"; },
              { include = "#variable"; }
           );
        },
        …
     ); // end of patterns
     repository = {
        escaped-char = { match = '\\.'; };
        variable =     { match = '\$[a-zA-Z0-9_]+'; };
     };
     ```

     This can also be used to match recursive constructs like balanced characters:

     ```
     patterns = (
        {  name = 'string.unquoted.qq.perl';
           begin = 'qq\('; end = '\)'; patterns = (
              { include = '#qq_string_content'; },
           );
        },
        …
     ); // end of patterns
     repository = {
        qq_string_content = {
           begin = '\('; end = '\)'; patterns = (
              { include = '#qq_string_content'; },
           );
        };
     };
     ```

     This will correctly match a string like: `qq( this (is (the) entire) string)`.

12.4 Naming Conventions

TextMate is free-form in the sense that you can assign basically any name you wish to any part of the document that you can markup with the grammar system and then use that name in scope selectors.

There are however conventions so that one theme can target as many languages as possible, without having dozens of rules specific to each language and also so that functionality (mainly preferences) can be re-used across languages, e.g. you probably do not want an apostrophe to be auto-paired when inserted in strings and comments, regardless of the language you are in, so it makes sense to only set this up once.

Before going through the conventions, here are a few things to keep in mind:

1. A minimal theme will only assign styles to 10 of the 11 root groups below (`meta` does not get a visual style), so you should "spread out" your naming i.e. instead of putting everything below `keyword` (as your formal language definition may insist) you should think "would I want these two elements styled differently?" and if so, they should probably be put into different root groups.

2. Even though you should "spread out" your names, when you have found the group in which you want to place your element (e.g. `storage`) you should re-use the existing names used below that group (for `storage` that is `modifier` or `type`) rather than make up a new sub-type. You should however append as much information to the sub-type you choose. For example if you are matching the `static` storage modifier, then instead of just naming it `storage.modifier` use `storage.modifier.static.«language»`. A scope selector of just `storage.modifier` will match both, but having the extra information in the name means it is possible to specifically target it disregarding the other storage modifiers.

3. Put the language name last in the name. This may seem redundant, since you can generally use a scope selector of: `source.«language» storage.modifier`, but when embedding languages, this is not always possible.

And now the 11 root groups which are currently in use with some explanation about their intended purpose. This is presented as a hierarchical list but the actual scope name is obtained by joining the name from each level with a dot. For example `double-slash` is `comment.line.double-slash`.

- `comment` — for comments.
  - `line` — line comments, we specialize further so that the type of comment start character(s) can be extracted from the scope.
    - `double-slash` — `// comment`
    - `double-dash` — `-- comment`
    - `number-sign` — `# comment`
    - `percentage` — `% comment`
    - *character* — other types of line comments.
  - `block` — multi-line comments like `/* … */` and `<!-- … -->`.
    - `documentation` — embedded documentation.
- `constant` — various forms of constants.
  - `numeric` — those which represent numbers, e.g. `42`, `1.3f`, `0x4AB1U`.
  - `character` — those which represent characters, e.g. `&lt;`, `\e`, `\031`.
    - `escape` — escape sequences like `\e` would be `constant.character.escape`.
  - `language` — constants (generally) provided by the language which are "special" like `true`, `false`, `nil`, `YES`, `NO`, etc.
  - `other` — other constants, e.g. colors in CSS.
- `entity` — an entity refers to a larger part of the document, for example a chapter, class, function, or tag. We do not scope the entire entity as `entity.*` (we use `meta.*` for that). But we do use `entity.*` for the "placeholders" in the larger entity, e.g. if the entity is a chapter, we would use `entity.name.section` for the chapter title.
  - `name` — we are naming the larger entity.
    - `function` — the name of a function.
    - `type` — the name of a type declaration or class.
    - `tag` — a tag name.
    - `section` — the name is the name of a section/heading.
  - `other` — other entities.
    - `inherited-class` — the superclass/baseclass name.
    - `attribute-name` — the name of an attribute (mainly in tags).
- `invalid` — stuff which is "invalid".
  - `illegal` — illegal, e.g. an ampersand or lower-than character in HTML (which is not part of an entity/tag).
  - `deprecated` — for deprecated stuff e.g. using an API function which is deprecated or using styling with strict HTML.
- `keyword` — keywords (when these do not fall into the other groups).
  - `control` — mainly related to flow control like `continue`, `while`, `return`, etc.
  - `operator` — operators can either be textual (e.g. `or`) or be characters.
  - `other` — other keywords.
- `markup` — this is for markup languages and generally applies to larger subsets of the text.
  - `underline` — underlined text.
    - `link` — this is for links, as a convenience this is derived from `markup.underline` so that if there is no theme rule which specifically targets `markup.underline.link` then it will inherit the underline style.
  - `bold` — bold text (text which is strong and similar should preferably be derived from this name).

    - `heading` — a section header. Optionally provide the heading level as the next element, for example `markup.heading.2.html` for `<h2>…</h2>` in HTML.
    - `italic` — italic text (text which is emphasized and similar should preferably be derived from this name).
    - `list` — list items.
        - `numbered` — numbered list items.
        - `unnumbered` — unnumbered list items.
    - `quote` — quoted (sometimes block quoted) text.
    - `raw` — text which is verbatim, e.g. code listings. Normally spell checking is disabled for `markup.raw`.
    - `other` — other markup constructs.
- `meta` — the meta scope is generally used to markup larger parts of the document. For example the entire line which declares a function would be `meta.function` and the subsets would be `storage.type`, `entity.name.function`, `variable.parameter` etc. and only the latter would be styled. Sometimes the meta part of the scope will be used only to limit the more general element that is styled, most of the time meta scopes are however used in scope selectors for activation of bundle items. For example in Objective-C there is a meta scope for the interface declaration of a class and the implementation, allowing the same tab-triggers to expand differently, depending on context.

- `storage` — things relating to "storage".
    - `type` — the type of something, `class`, `function`, `int`, `var`, etc.
    - `modifier` — a storage modifier like `static`, `final`, `abstract`, etc.
- `string` — strings.
    - `quoted` — quoted strings.
        - `single` — single quoted strings: `'foo'`.
        - `double` — double quoted strings: `"foo"`.
        - `triple` — triple quoted strings: `"""Python"""`.
        - `other` — other types of quoting: `$'shell'`, `%s{...}`.
    - `unquoted` — for things like here-docs and here-strings.
    - `interpolated` — strings which are "evaluated": `` `date` ``, `$(pwd)`.
    - `regexp` — regular expressions: `/(\w+)/`.
    - `other` — other types of strings (should rarely be used).
- `support` — things provided by a framework or library should be below `support`.
    - `function` — functions provided by the framework/library. For example `NSLog` in Objective-C is `support.function`.
    - `class` — when the framework/library provides classes.
    - `type` — types provided by the framework/library, this is probably only used for languages derived from C, which has `typedef` (and `struct`). Most other languages would introduce new types as classes.
    - `constant` — constants (magic values) provided by the framework/library.
    - `variable` — variables provided by the framework/library. For example `NSApp` in AppKit.
    - `other` — the above should be exhaustive, but for everything else use `support.other`.
- `variable` — variables. Not all languages allow easy identification (and thus markup) of these.
    - `parameter` — when the variable is declared as the parameter.
    - `language` — reserved language variables like `this`, `super`, `self`, etc.
    - `other` — other variables, like `$some_variables`.

Drag Commands

Scope Selectors