Kate

Writing a Syntax Highlighting File

Thursday, 24 March 2005 | Christoph Cullmann

Note: Please refer to the Kate Handbook for the most recent version of how to write syntax highlighting files.

Hint: If you want to write a syntax highlighting file, the XML Completion plugin might be of great help.

This section is an overview of the Highlight Definition XML format in KDE4. Based on a small example it will describe the main components and their meaning and usage. The next section will go into detail with the highlight detection rules.

Main sections of Kate Highlight Definition files

The formal definition, aka the DTD is stored in the file `language.dtd` which should be installed on your system in the folder `$KDEDIR/share/apps/katepart/syntax`. If `$KDEDIR` is unset look up the folder by using `kde4-config --prefix`.

An Example

A highlighting file contains a header that sets the XML version and the doctype:

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE language SYSTEM "language.dtd">
```

The root of the definition file is the element **language**. Available attributes are:

Required attributes:

- **name** sets the name of the language. It appears in the menus and dialogs.
- **section** specifies the category.
- **extensions** defines file extensions, like "*.cpp;*.h"

Optional attributes:

- **mimetype** associates files Mime Type based.
- **version** specifies the current version of the definition file.
- **kateversion** specifies the latest supported version of Kate.
- **casesensitive** defines, whether the keywords are case sensitive or not. NOTE: not implemented yet.
- **priority** is necessary if another highlight definition file uses the same extensions. The higher priority will win.
- **author** contains the name of the author and his email-address.
- **license** contains the license, usually LGPL, Artistic, GPL and others. It's important to specify a license, as the kate team needs some legal backing for the distribution of the files.
- **hidden** defines, whether the name should appear in Kate's menus.

So the next line may look like this:

```
<language name="C++" version="1.00" kateversion="2.4" section="Sources" extensions="*.cpp;*.h" >
```

Next comes the **highlighting** element, which contains the optional element **list** and the required elements **contexts** and **itemDatas**.

list elements contain a list of keywords. In this case the keywords are *class* and *const*. You can add as many lists as you need. The **contexts** element contains all contexts. The first context is by default the start of the highlighting. There are two rules in the context *Normal Text*, which match the list of keywords with the name *somename* and a rule that detects a quote and switches the context to *string*. To learn more about rules read the next chapter.
The third part is the **itemDatas** element. It contains all color and font styles needed by the contexts and rules.
In this example, the **itemData** *Normal Text*, *String* and *Keyword* are used.

```
<highlighting> <list name="somename"> <item> class </item> <item> const </item> </list> <contexts> <cc
```

The last part of a highlight definition is the optional **general** section. It may contain information about keywords, code folding, comments and indentation.

The **comment** section defines with what string a single line comment is introduced. You also can define a multiline comments using *multiLine* with the additional attribute *end*. This is used if the user presses the corresponding shortcut for *comment/uncomment*.
The **keywords** section defines whether keyword lists are case sensitive or not. Other attributes will be explained later.

```
<general> <comments> <comment name="singleLine" start="#"/> </comments> <keywords casesensitive="1"/>
```

The Sections in Detail

This part will describe all available attributes for contexts,
itemDatas, keywords, comments, code folding and indentation.

The element **context** belongs into the group **contexts**. A context itself defines context specific rules like what should happen if the highlight system reaches the end of a line. Available attributes are:

- **name** the context name/identifier. Rules will use this name to specify the context to switch to if the rule matches.
- **attribute** the default item data that is used if no rules match in the current context.
- **lineEndContext** defines the context the highlight system switches to if it reaches the end of a line. This may either be a name of another context, **#stay** to not switch the context (i.e. do nothing) or **#pop** which will cause to leave this context. It is possible to use for example **#pop#pop#pop** to pop three times.
- **lineBeginContext** defines the context if a begin of a line is encountered. Default: #stay.
- **fallthrough** defines if the highlight system switches to the context specified in fallthroughContext if no rule matches. Default: *false*.
- **fallthroughContext** specifies the next context if no rule matches.
- **dynamic** if *true*, the context remembers strings/placeholders saved by dynamic rules. This is needed for *HERE* documents for example. Default: *false*.

The element **itemData** is in the group **itemDatas**. It defines the font style and colors. So it is possible to define your own styles and colors, however we recommend to stick to the default styles if possible so that the user will always see the same colors used in different languages. Though, sometimes there is no other way and it is necessary to change color and font attributes. The attributes name and defStyleNum are required, the other optional. Available attributes are:

- **name** sets the name of the itemData. Contexts and rules will use this name in their attribute.
- **attribute** to reference an itemData.
- **defStyleNum** defines which default style to use. Available default styles are explained in detail later.
- **color** defines a color. Valid formats are '#rrggbb' or '#rgb'.
- **selColor** defines the selection color.
- **italic** if *true*, the text will be *italic*.
- **bold** if *true*, the text will be **bold**.
- **underline** if *true*, the text will be underlined.
- **strikeout** if *true*, the text will be ~~stroked out~~.
- **spellChecking** if true, the text will be spell checked, otherwise it will be ignored during spell check.

The element **keywords** in the group **general** defines keyword properties. Available attributes are:

- **casesensitive** may be *true* or *false*. If *true*, all keywords are matched case sensitive. Default: *true*.
- **weakDeliminator** is a list of characters that do *not* act as word delimiters. For example the dot (**.**) is a word delimiter. Assume a keyword in a **list** contains a dot, it will only match if you specify the dot as a weak delimiter.
- **additionalDeliminator** defines additional delimiters.
- **wordWrapDeliminator** defines characters after which a line wrap may occur. Default delimiters and word wrap delimiters are the characters **.():!+,-<=>%&/;?[]^{|}~*, space (' ')** and **tabulator (\t)**.

The element **comment** in the group **comments** defines comment properties which are used for `Tools > Comment` and `Tools > Uncomment`.
Available attributes are:

- **name** is either *singleLine* or *multiLine*.
- If you choose *singleLine* the optinal attribute **position** is available. Default for this attribute is to insert the single line comment string in column 0. If you want it to appear after the whitespaces you have to set it to *afterwhitespace*, like: `position="afterwhitespace"`.
- If you choose *multiLine* the attributes *end* and *region* are required.
- **start** defines the string used to start a comment. In C++ this is `/*`.
- **end** defines the string used to close a comment. In C++ this is `*/`.
- **region** should be the name of the the foldable multiline comment. Assume you have *beginRegion="Comment"* ... *endRegion="Comment"* in your rules, you should use *region="Comment"*. This way uncomment works even if you do not select all the text of the multiline comment. The cursor only must be in the multiline comment.

The element **folding** in the group **general** defines code folding properties. Available attributes are:

- **indentationsensitive** if *true*, the code folding markers will be added indentation based, like in the scripting language Python. Usually you do not need to set it, as it defaults to *false*.

The element **indentation** in the group **general** defines which indenter will be used, however we strongly recommend to omit this element, as the indenter usually will be set by either defining a File Type or by adding a mode line to the text file. If you specify an indenter though, you will force a specific indentation

on the user, which he might not like at all.
Available attributes are:

- **mode** is the name of the indenter. Available indenters right now are: *none*, *normal*, *cstyle*, *haskell*, *lilypond*, *lisp*, *python*, *ruby* and *xml*.

Available Default Styles

Default styles are predefined font and color styles. For convenience Kate provides several default styles, in detail:

- **dsNormal**, used for normal text.
- **dsKeyword**, used for keywords.
- **dsDataType**, used for data types.
- **dsDecVal**, used for decimal values.
- **dsBaseN**, used for values with a base other than 10.
- **dsFloat**, used for float values.
- **dsChar**, used for a character.
- **dsString**, used for strings.
- **dsComment**, used for comments.
- **dsOthers**, used for 'other' things.
- **dsAlert**, used for warning messages.
- **dsFunction**, used for function calls.
- **dsRegionMarker**, used for region markers.
- **dsError**, used for error highlighting and wrong syntax.

Highlight Detection Rules

This section describes the syntax detection rules.

Each rule can match zero or more characters at the beginning of the string they are tested against. If the rule matches, the matching characters are assigned the style or *attribute* defined by the rule, and a rule may ask that the current context is switched.

A rule looks like this:

The *attribute* identifies the style to use for matched characters by name, and the *context* identifies the context to use from here.

The *context* can be identified by:

- An *identifier*, which is the name/identifier of another context.
- An *order* telling the engine to stay in the current context (**#stay**), or to pop back to a previous context used in the string (**#pop**). To go back more steps, the #pop keyword can be repeated: **#pop#pop#pop**

Some rules can have *child rules* which are then evaluated only if the parent rule matched. The entire matched string will be given the attribute defined by the parent rule. A rule with child rules looks like this:

```
<RuleName (attributes)> <ChildRuleName (attributes) /> ... </RuleName>
```

Rule specific attributes vary and are described in the following sections.

Common attributes

All rules have the following attributes in common and are available whenever a **(common attributes)** appears. All following attributes are optional.

- **attribute** maps to a defined *itemData*. Default: the attribute from the *destination context*
- **context** specifies the context to which the highlighting system switches if the rule matches. Default: *#stay*
- **beginRegion** starts a code folding block. Default: unset.
- **endRegion** closes a code folding block. Default: unset.
- **lookAhead**, if *true*, the highlighting system will not process the matches length. Default: *false*.
- **firstNonSpace**, if *true*, the rule only matches, if the string is the first non-whitespace in the line. Default: *false*.
- **column** defines the column. The rule only matches, if the current column matches the given one. Default: unset.

Dynamic rules

Some rules allow the optional attribute **dynamic** of type boolean that defaults to *false*. If dynamic is *true*, a rule can use placeholders representing the text matched by a *regular expression* rule that switched to the current context in its **string** or **char** attributes. In a **string**, the placeholder **%N** (where N is a number) will be replaced with the corresponding capture **N** from the calling regular expression. In a **char** the placeholer must be a number **N** and it will be replaced with the first character of the corresponding capture **N** from the calling regular expression. Whenever a rule allows this attribute it will contain a *(dynamic)*.

- **dynamic** may be either *true* or *false*. Default: *false*.

The Rules in Detail

DetectChar

Detect a single specific character. Commonly used for example to find the ends of quoted strings.

```
<DetectChar char="(character)" (common attributes) (dynamic) />
```

The **char** attribute defines the character to match.

Detect2Chars

Detect two specific characters in a defined order.

```
<Detect2Chars char="(character)" char1="(character)" (common attributes) (dynamic) />
```

The **char** attribute defines the first character to match, **char1** the second.

AnyChar

Detect one character of a set of specified characters.

```
<AnyChar String="(string)" (common attributes) />
```

The **String** attribute defines the set of characters.

StringDetect

Detect an exact string.

```
<StringDetect String="(string)" [insensitive="true|false"] (common attributes) (dynamic) />
```

The **String** attribute defines the string to match. The **insensitive** attribute defaults to *false* and is passed to the string comparison function. If the value is *true* insensitive comparing is used.

WordDetect (KDE >= 4.5, Kate >= 3.5)

Detect an exact string but additionally require word boundaries like a dot (.) or a whitespace on the beginning and the end of the word. You can think of *|b|b* in terms of a regular expression.

```
<WordDetect String="(string)" [insensitive="true|false"] (common attributes) (dynamic) />
```

The **String** attribute defines the string to match. The **insensitive** attribute defaults to *false* and is passed to the string comparison function. If the value is *true* insensitive comparing is used.

RegExpr

Matches against a regular expression.

```
<RegExpr String="(string)" [insensitive="true|false"] [minimal="true|false"] (common attributes) (dynam
```

- The **String** attribute defines the regular expression.
- **insensitive** defaults to *false* and is passed to the regular expression engine.
- **minimal** defaults to *false* and is passed to the regular expression engine.

Because the rules are always matched against the beginning of the current string, a regular expression starting with a caret (**^**) indicates that the rule should only be matched against the start of a line.

keyword

Detect a keyword from a specified list.

```
<keyword String="(list name)" (common attributes) />
```

The **String** attribute identifies the keyword list by name. A list with that name must exist.

Int

Detect an integer number.

```
<Int (common attributes) (dynamic) />
```

This rule has no specific attributes. Child rules are typically used to detect combinations of **L** and **U** after the number, indicating the integer type in program code. Actually all rules are allowed as child rules,

though, the DTD only allowes the child rule **StringDetect**.
The following example matches integer numbers follows by the character 'L'.

```
<Int attribute="Decimal" context="#stay" > <StringDetect attribute="Decimal" context="#stay" String="L"
```

Float

Detect a floating point number.

```
<Float (common attributes) />
```

This rule has no specific attributes. **AnyChar** is allowed as a child rules and typically used to detect combinations, see rule **Int** for reference.

HlCOct

Detect an octal point number representation.

```
<HlCOct (common attributes) />
```

This rule has no specific attributes.

HlCHex

Detect a hexadecimal number representation.

```
<HlCHex (common attributes) />
```

This rule has no specific attributes.

HlCStringChar

Detect an escaped character.

```
<HlCStringChar (common attributes) />
```

This rule has no specific attributes.

It matches literal representations of characters commonly used in program code, for example **\n** (newline) or **\t** (tabulator).

The following characters will match if they follow a backslash (****): **abefnrtv"'?**. Additionally, escaped hexadecimal numbers like for example **\xff** and escaped octal numbers, for example **\033** will match.

HlCChar

Detect an C character.

```
<HlCChar (common attributes) />
```

This rule has no specific attributes.

It matches C characters enclosed in a tick (Example: '**c**'). So in the ticks may be a simple character or an escaped character. See HlCStringChar for matched escaped character sequences.

RangeDetect

Detect a string with defined start and end characters.

```
<RangeDetect char="(character)" char1="(character)" (common attributes) />
```

char defines the character starting the range, **char1** the character ending the range. Usefull to detect for example small quoted strings and the like, but note that since the highlighting engine works on one line at a time, this will not find strings spanning over a line break.

LineContinue

Matches a backslash ('’) at the end of a line.

```
<LineContinue (common attributes) />
```

This rule has no specific attributes.
This rule is useful for switching context at end of line, if the last character is a backslash ('****'). This is needed for example in C/C++ to continue macros or strings.

IncludeRules

Include rules from another context or language/file.

```
<IncludeRules context="contextlink" [includeAttrib="true|false"] />
```

The **context** attribute defines which context to include.
If it a simple string it includes all defined rules into the current context, example:

```
<IncludeRules context="anotherContext" />
```

If the string begins with **##** the highlight system will look for another language definition with the given name, example:

```
<IncludeRules context="##C++" />
```

If **includeAttrib** attribute is *true*, change the destination attribute to the one of the source. This is required to make for example commenting work, if text matched by the included context is a different highlight than the host context.

DetectSpaces

Detect whitespaces.

```
<DetectSpaces (common attributes) />
```

This rule has no specific attributes.
Use this rule if you know that there can several whitespaces ahead, for example in the beginning of indented lines. This rule will skip all whitespace at once, instead of testing multiple rules and skipping one at the time due to no match.

DetectIdentifier

Detect identifier strings (as a regular expression: [a-zA-Z_][a-zA-Z0-9_]*).

```
<DetectIdentifier (common attributes) />
```

This rule has no specific attributes.
Use this rule to skip a string of word characters at once, rather than testing with multiple rules and skipping one at the time due to no match.

Tips & Tricks

Once you have understood how the context switching works it will be easy to write highlight definitions. Though you should carefully check what rule you choose in what situation. Regular expressions are very mighty, but they are slow compared to the other rules. So you may consider the following tips.

- If you only match two characters use **Detect2Chars** instead of **StringDetect**. The same applies to **DetectChar**.
- Regular expressions are easy to use but often there is another much faster way to achieve the same result. Consider you only want to match the character **#**if it is the first character in the line. A regular expression based solution would look like this:

  You can achieve the same much faster in using:

  ```
  <DetectChar attribute="Macro" context="macro" char="#" firstNonSpace="true" />
  ```

  If you want to match the regular expression '**^#**' you can still use **DetectChar** with the attribute **column="0"**. The attribute **column** counts character based, so a tabulator still is only one character.

  - You can switch contexts without processing characters. Assume that you want to switch context when you meet the string **/*, but need to process that string in the next context. The below rule will match, and the lookAhead**attribute will cause the highlighter to keep the matched string for the next context.
  - Use **DetectSpaces** if you know that many whitespaces occur.
  - Use **DetectIdentifier** instead of the regular expression '**[a-zA-Z_]\w***'.
  - Use default styles whenever you can. This way the user will find a familiar environment.
  - Look into other XML-files to see how other people implement tricky rules.
  - You can validate every XML file by using the command `xmllint --dtdvalid language.dtd mySyntax.xml` .
  - If you repeat complex regular expression very often you can use *ENTITIES*. Example:]>

Now you can use *&myref;* instead of the regular expression.