gnome.pages.gitlab.gnome.org

GtkSource: Language Definition v2.0 Tutorial

15-19 minutes

Guide to the GtkSourceView language definition file format

A language definition for the C language

The version 2 here refers to the language definition file format, not to the version of GtkSourceView. This tutorial is suitable for GtkSourceView 2, 3 and 4.

To describe the syntax of a language GtkSourceView uses an XML format which defines nested context to be highlighted. Each context roughly corresponds to a portion of the syntax which has to be highlighted (e.g. keywords, strings, comments), and can contain nested contexts (e.g. escaped characters.)

In this tutorial we will analyze a simple example to highlight a subset of C, based on the full C language definition.

Like every well formed XML document, the language description starts with a XML declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
```

After the usual preamble, the main tag is the `<language>` element:

```
<language id="c" name="C" version="2.0" _section="Source">
```

The attribute `id` is used in external references and defines a standard way to refer to this language definition, while the `name` attribute is the name presented to the user.

The attribute `section` (it is translatable using gettext prepending a _), tells the category where this language should be grouped when it is presented to the user. Currently available categories in GtkSourceView are “Source”, “Script”, “Markup”, “Scientific” and “Other”.

The attribute `version` specifies the version of the xml syntax used in your language definition file, so it should always be `2.0`.

The `<language>` element contains three sections: `<metadata>`, `<styles>` and `<definitions>`

The `<metadata>` element is optional and provides a collection of properties which specify arbitrary information about the language definition file itself. It is particularly important to specify the conventional `mimetypes` and `globs` properties that GtkSourceView uses to automatically detect which syntax highlighting to use for a given file. They respectively contain a semi-colon separated list of mimetypes and filename extensions.

```
<metadata>
  <property name="mimetypes">text/x-c;text/x-csrc</property>
  <property name="globs">*.c</property>
</metadata>
```

This element contains every association between the styles used in the description and the defaults stored internally in GtkSourceView. For each style there is a `<style>` element:

```
<style id="comment" name="Comment" map-to="def:comment"/>
```

This defines a `comment` style, which inherits the font properties from the defaults style `def:comment`. The `name` attribute is the name to show to the user (that string could for example be used by a GUI tool to edit or create style schemes).

For each style used in the language definition there is a corresponding `<style>` element; every style can be used in different contexts, so they will share the same appearance.

```
<style id="string" name="String" map-to="def:string"/>
<style id="escaped-character" name="Escaped Character" map-
to="def:special-char"/>
<style id="preprocessor" name="Preprocessor" map-
to="def:preprocessor"/>
<style id="included-file" name="Included File" map-
to="def:string"/>
<style id="char" name="Character" map-to="def:character"/>
<style id="keyword" name="Keyword" map-to="def:keyword"/>
<style id="type" name="Data Type" map-to="def:type"/>
```

Following the `<styles>` element there is the `<definitions>` element, which contains the description proper of the syntax:

Here we should define a main context, the one we enter at the beginning of the file: to do so we use the `<context>` tag, with an `id` equal to the `id` of the `<language>` element:

The element `<include>` contains the list of sub-contexts for the current context: as we are in the main context we should put here the top level contexts for the C language:

The first context defined is the one for single-line C style comments: they start with a double slash `//` and end at the end of the line:

```
<context id="comment" style-ref="comment">
  <start>\/\/</start>
  <end>$</end>
</context>
```

The `<start>` element contains the regular expression telling the highlighting engine to enter in the defined context, until the terminating regular expression contained in the `<end>` element is found.

Those regular expressions are PCRE regular expressions in the form `/regex/options` (see the documentation of PCRE for details.) If there are no options to be specified and you don't need

to match the spaces at the start and at the end of the regular expression, you can omit the slashes, putting here only `regex`.

The possible options are:

- `i`: case insensitive;
- `x`: extended (spaces are ignored and it is possible to put comments starting with # and ending at the end of the line);
- `s`: the metacharacter `.` matches the `\n`.

You can set the default options using the `<default-regex-options` tag before the `<definitions>` element. To disable a group of options, instead, you have to precede them with a hyphen (`-`). [FIXME: add an example]

In GtkSourceView are available also some extensions to the standard perl style regular expressions:

- `\%[` and `\%]` are custom word boundaries, which can be redefined with the `<keyword-char-class>` tag (in contrast with `\b`);
- `\%{id}` will include the regular expression defined in the `<define-regex>` tag with the same id, useful if you have common portions of regular expressions used in different contexts;
- `\%{subpattern@start}` can be used only inside the `<end>` tag and will be substituted with the string matched in the corresponding sub-pattern (can be a number or a name if named sub-patterns are used) in the preceding `<start>` element. For an example see the implementation of here-documents in the `sh.lang` language description distributed with GtkSourceView.

The next context is for C-style strings. They start and end with a double quote but they can contain escaped double quotes, so we should make sure we don't end the string prematurely:

```
<context id="string" end-at-line-end="true" style-ref="string">
```

The `end-at-line-end` attribute tells the engine that the current context should be forced to terminate at the end of the line, even if the ending regular expression is not found, and that an error should be displayed.

```
<start>"</start>
<end>"</end>
<include>
```

To implement the escape handling we include a `escape` context:

```
  <context id="escape" style-ref="escaped-character">
    <match>\\.</match>
  </context>
```

This is a simple context matching a single regular expression, contained in the `<match>` element. This context will extend its parent, causing the ending regular expression of the `"string"` context to not match the escaped double quote.

Multiline C-style comment can span over multiple lines and cannot be escaped, but to make

things more interesting we want to highlight every internet address contained:

```
<context id="comment-multiline" style-ref="comment">
  <start>\/\*</start>
  <end>\*\/</end>
  <include>
    <context id="net-address" style-ref="net-address" extend-
parent="false">
```

In this case, the child should be terminated if the end of the parent is found, so we use `false` in the `extend-parent` attribute.

```
      <match>http:\/\/[^\s]*</match>
    </context>
  </include>
</context>
```

For instance in the following comment the string `http://www.gnome.org*/` matches the `net-address` context but it contains the end of the parent context (`*/`.) As `extend-parent` is false, only `http://www.gnome.org` is highlighted as an address and `*/` is correctly recognized as the end of the comment.

```
/* This is a comment http://www.gnome.org */
```

Character constants in C are delimited by single quotes (`'`) and can contain escaped characters:

```
<context id="char" end-at-line-end="true" style-ref="string">
  <start>'</start>
  <end>'</end>
  <include>
    <context ref="escape"/>
```

The `ref` attribute is used when we want to reuse a previously defined context. Here we reuse the `escape` context defined in the `string` context, without repeating its definition.

Using `ref` it is also possible to refer to contexts defined in other languages, preceding the id of the context with the id of the containing language, separating them with a colon:

```
<context ref="def:decimal"/>
<context ref="def:float"/>
```

The definitions for decimal and float constants are in a external file, with id `def`, which is not associated with any language but contains reusable contexts which every language definition can import.

The `def` language file contains an `in-comment` context that can contain addresses and tags such as FIXME and TODO, so we can write a new version of our `comment-multiline` context that uses the definitions from `def.lang`.

```
<context id="comment-multiline" style-ref="comment">
  <start>\/\*</start>
  <end>\*\/</end>
```

```
  <include>
    <context ref="def:in-comment"/>
```

Keywords can be grouped in a context using a list of <keyword> elements:

```
<context id="keywords" style-ref="keyword">
  <keyword>if</keyword>
  <keyword>else</keyword>
  <keyword>for</keyword>
  <keyword>while</keyword>
  <keyword>return</keyword>
  <keyword>break</keyword>
  <keyword>switch</keyword>
  <keyword>case</keyword>
  <keyword>default</keyword>
  <keyword>do</keyword>
  <keyword>continue</keyword>
  <keyword>goto</keyword>
  <keyword>sizeof</keyword>
</context>
```

Keywords with different meaning can be grouped in different context, making possible to highlight them differently:

```
<context id="types" style-ref="type">
  <keyword>char</keyword>
  <keyword>const</keyword>
  <keyword>double</keyword>
  <keyword>enum</keyword>
  <keyword>float</keyword>
  <keyword>int</keyword>
  <keyword>long</keyword>
  <keyword>short</keyword>
  <keyword>signed</keyword>
  <keyword>static</keyword>
  <keyword>struct</keyword>
  <keyword>typedef</keyword>
  <keyword>union</keyword>
  <keyword>unsigned</keyword>
  <keyword>void</keyword>
</context>
```

You can also set a prefix (or a suffix) common to every keyword using the <prefix> and <suffix> tags:

```
<context id="preprocessor" style-ref="preprocessor">
  <prefix>^#</prefix>
```

If not specified, <prefix> and <suffix> are set to, respectively, \%[and \%].

```
  <keyword>define</keyword>
  <keyword>undef</keyword>
```

Keep in mind that every keyword is a regular expression:

```
  <keyword>if(n?def)?</keyword>
  <keyword>else</keyword>
  <keyword>elif</keyword>
  <keyword>endif</keyword>
</context>
```

In C, there is a common practice to use `#if 0` to express multi-line nesting comments. To make things easier to the user, we want to highlight these pseudo-comments as comments:

```
<context id="if0-comment" style-ref="comment">
  <start>^#if 0\b</start>
  <end>^#(endif|else|elif)\b</end>
  <include>
```

As `#if 0` comments are nesting, we should consider that inside a comment we can find other `#if`s with the corresponding `#endif`s, avoiding the termination of the comment on the wrong `#endif`. To do so we use a nested context, that will extend the parent on every nested `#if`/`#endif`:

```
  <context id="if-in-if0">
    <start>^#if(n?def)?\b</start>
    <end>^#endif\b</end>
    <include>
```

Nested contexts can be recursive:

```
      <context ref="if-in-if0"/>
    </include>
  </context>
  </include>
</context>
```

Because contexts defined before have higher priority, `if0-comment` will never be matched. To make things work we should move it before the `preprocessor` context, thus giving `if0-comment` a higher priority.

For the `#include` preprocessor directive it could be useful to highlight differently the included file:

```
<context id="include" style-ref="preprocessor">
  <match>^#include (".*"|&amp;lt;.*&amp;gt;)</match>
  <include>
```

To do this we use grouping sub-patterns in the regular expression, associating them with a context with the `sub-pattern` attribute:

```
    <context id="included-file" sub-pattern="1"
```

```
            style-ref="included-file"/>
```

In the `sub-pattern` attribute we could use:

- 0: the whole regular expression;
- 1: the first sub-pattern (a sub-espression enclosed in parenthesis);
- 2: the second;
- …
- `name`: a named sub-pattern with name `name` (see the PCRE documentation.)

We could also use a `where` attribute with value `start` or `end` to specify the regular expression the context is referring, when we have both the `<start>` and `<end>` element.

Having defined a good subset of the C syntax we close every tag still open:

```
</include>
</context>
</definitions>
</language>
```

The full language definition

This is the full language definition for the subset of C taken in consideration for this tutorial:

```
<?xml version="1.0" encoding="UTF-8"?>
<language id="c" name="C" version="2.0" _section="Source">
  <metadata>
    <property name="mimetypes">text/x-c;text/x-csrc</property>
    <property name="globs">*.c</property>
  </metadata>
  <styles>
    <style id="comment" name="Comment" map-to="def:comment"/>
    <style id="string" name="String" map-to="def:string"/>
    <style id="escaped-character" name="Escaped Character" map-
to="def:special-char"/>
    <style id="preprocessor" name="Preprocessor" map-
to="def:preprocessor"/>
    <style id="included-file" name="Included File" map-
to="def:string"/>
    <style id="char" name="Character" map-to="def:character"/>
    <style id="keyword" name="Keyword" map-to="def:keyword"/>
    <style id="type" name="Data Type" map-to="def:type"/>
  </styles>
  <definitions>
    <context id="c">
      <include>
```

```
        <context id="comment" style-ref="comment">
          <start>\/\/</start>
          <end>$</end>
        </context>

        <context id="string" end-at-line-end="true" style-
ref="string">
          <start>"</start>
          <end>"</end>
          <include>
            <context id="escape" style-ref="escaped-character">
              <match>\\.</match>
            </context>
          </include>
        </context>

        <context id="comment-multiline" style-ref="comment">
          <start>\/\*</start>
          <end>\*\/</end>
          <include>
            <context ref="def:in-comment"/>
          </include>
        </context>

        <context id="char" end-at-line-end="true" style-
ref="string">
          <start>'</start>
          <end>'</end>
          <include>
            <context ref="escape"/>
          </include>
        </context>

        <context ref="def:decimal"/>
        <context ref="def:float"/>

        <context id="keywords" style-ref="keyword">
          <keyword>if</keyword>
          <keyword>else</keyword>
          <keyword>for</keyword>
          <keyword>while</keyword>
          <keyword>return</keyword>
          <keyword>break</keyword>
          <keyword>switch</keyword>
          <keyword>case</keyword>
          <keyword>default</keyword>
```

```
    <keyword>do</keyword>
    <keyword>continue</keyword>
    <keyword>goto</keyword>
    <keyword>sizeof</keyword>
  </context>

  <context id="types" style-ref="type">
    <keyword>char</keyword>
    <keyword>const</keyword>
    <keyword>double</keyword>
    <keyword>enum</keyword>
    <keyword>float</keyword>
    <keyword>int</keyword>
    <keyword>long</keyword>
    <keyword>short</keyword>
    <keyword>signed</keyword>
    <keyword>static</keyword>
    <keyword>struct</keyword>
    <keyword>typedef</keyword>
    <keyword>union</keyword>
    <keyword>unsigned</keyword>
    <keyword>void</keyword>
  </context>

  <context id="if0-comment" style-ref="comment">
    <start>^#if 0\b</start>
    <end>^#(endif|else|elif)\b</end>
    <include>
      <context id="if-in-if0">
        <start>^#if(n?def)?\b</start>
        <end>^#endif\b</end>
        <include>
          <context ref="if-in-if0"/>
        </include>
      </context>
    </include>
  </context>

  <context id="preprocessor" style-ref="preprocessor">
    <prefix>^#</prefix>
    <keyword>define</keyword>
    <keyword>undef</keyword>
    <keyword>if(n?def)?</keyword>
    <keyword>else</keyword>
    <keyword>elif</keyword>
    <keyword>endif</keyword>
```

```
        </context>

        <context id="include" style-ref="preprocessor">
          <match>^#include (".*"|&amp;lt;.*&amp;gt;)</match>
          <include>
            <context id="included-file"
                     sub-pattern="1"
                     style-ref="included-file"/>
          </include>
        </context>

      </include>
    </context>
  </definitions>
</language>
```