WMLxGettext Documentation

Release 2.0.0

Nobun

Mar 09, 2018

Contents

CHAPTER 1

Documentation for End-Users

This the new release of wmlxgettext, rewritten from scratch in python 3.x. Wmlxgettext is a python3 script that scans a list of .cfg (WML) and .lua files, capturing all translatable string found in the files and creates a pot (.po) file.

From now on, the (old) perl script will be called "wmlxgettext 1.0", while the new python3 script will be called "wmlxgettext 2.x"

1.1 Supported Options for WML/Lua files

When you create an add-on, you may want to see it translated in several languages. You need to create your own .po file. This short page assume you already know how to write a WML file (See also the Official WML Reference).

Here you will find some additional informations, useful for a good usage of wmlxgettext.

Note: Paragraphs from 1.1 to 1.4 will talk about WML code; paragraphs 1.5 and 1.6 will talk about Lua code.

1.1.1 #textdomain <new_current_domain>

You wesnoth add-on, must contain a textdomain. A textdomain is defined in your _main.cfg file:

```
[textdomain]
   name="wesnoth-xyz"
   path="data/add-ons/xyz/translations"
[/textdomain]
```

Note: *xyz* can be everything. We will use *xyz* only as an example.

However, the what it is actually important for wmlxgettext is the following line of text that must appear inside all your add-on files:

```
#textdomain wesnoth-xyz
```

This will change the current_domain. This is useful, since wmlxgettext will capture only the sentences under the right textdomain (only the sentences under the textdomain of your add-on, to avoid to add undesired sentences).

1.1.2 # wmlxgettext: <WML_CODE>

You may need to declare a macro definition that uses unbalanced tags like the ABILITY_FEEDING macro defined on the WML core files:

```
#define ABILITY_FEEDING
# comment omissed
[dummy]
    id=feeding
    name= _ "feeding"
    female_name= _ "female^feeding"
    description=_ " (description omissed)"
[/dummy]
# wmlxgettext: [abilities]
[/abilities]
```

The example shows a macro wich has unbalanced tags, since there is a closing tag `[/abilities]` but not the opening `[abilities]` tag.

Usually, when encountering unbalanced tags, wmlxgettext returns an error, but this time the tags are unbalanced on purpose. This is where the special comment `# wmlxgettext: <WML_CODE>` helps us.

Coming back to the example showed above, the comment `# wmlxgettext: [abilties]` will be ignored by the WML code (so the WML code will be unbalanced, as desired); but wmlxgettext will read `[abilities]` as an actual opened tag, thank of the `# wmlxgettext:` special comment.

In this way, wmlxgettext will open `[abilities]` tag when reading `# wmlxgettext: [abilities]`, that will be sucessfully closed when the tag `[/abilities]` found.

1.1.3 # po: <addedinfo>

Another special comment (*not meaningful on actual WML code, but useful for wmlxgettext purposes*) is `# po:`. Here a fake example:

```
# po: The speaker is still unknown for the player, but he is a male
[message]
   speaker=unknown_speaker
   message= _ "translatable message"
[/message]
```

The special comment `# po:` will add to the translator infos what follows. This is the comments that will be displayed on pot file:

```
[message]: speaker=unknown_speaker
The speaker is still unknown for the player, but he is a male
```

The first line displayed to the translator is automaticly generated by wmlxgettext (standard help message).

The second line displayed to the translator is what you typed after the special comment `# po:`

1.1.4 # po-override: <override-info>

The special comment `# po-override:` is similar to `# po:` special comment already showed before:

```
# po-override: [message]: speaker=FinalBoss
[message]
   speaker=unknown_speaker
   message= _ "translatable message"
[/message]
```

This time, the special comment `# po-override:` will **replace** the default (automaticly generated) message to the translator. This is, infact, the comments that the translator will be see:

```
[message]: speaker=FinalBoss
```

This string will be displayed instead of the default one (in the example, the default overriden message is `[message]: speaker=unknown_speaker`, wich is, infact, not displayed since it is replaced by `# po-override:`).

Note: Unlike `# po:` you can use only **ONE** `# po-override:` for every sentence. However you can use one or more `# po:` special comments together with the `# po-override:` special comment.

1.1.5 Changing Domain in Lua code

Changing the current domain value in Lua uses is very different than the WML counterpart.

- On **WML** you will change the current domain value with the `#textdomain` directive
- On **Lua** code, instead, the same action is performed in a very different way, using this code:

```
local _ = wesnoth.textdomain('wesnoth-xyz')
```

Note: *xyz* can be everything. We will use *xyz* only as an example.

In the example showed above we changed, in lua code, the current domain value to `wesnoth-xyz`.

1.1.6 Special comments on Lua

On lua code you can also use those special comments BEFORE the translatable string that will require an additional or overridden info:

- `-- # po:  <additional info for translator>`
- `-- # po-override:  <info that overrides the default info>`

Note: You can also use `-- po:` and `-- po-override:`, instead of `-- # po:` and `-- # po-override:` Both forms are allowed.

Those special comments works in the same way as the `# po:` and `# po-ovverride:` special comments supported in WML code (see paragraphs 1.3 and 1.4).

Note: The special WML comment `# wmlxgettext:` is instead **unsupported** in lua code.

(*It is needed in WML code to avoid errors when tags are unbalanced on purpose, so it is useless in lua code, wich is a procedural language and not a 'tagged' language*).

1.2 wmlxgettext: how to run

The previous chapter explained how to write a WML and a Lua file in the right way:

- Avoiding unwanted errors using the special comment `# wmlxgettext:` on WML code when you need to use unbalanced tags
- Customizing the message informations displayed to the translator using the special comments `# po:` and `# po-override:`
- And remembering the `#textdomain` directive usage.

So we can assume here that all your .cfg and .lua files (*used by your wesnoth add-on*) are ready to be parsed by wmlxgettext. But how to run wmlxgettext?

wmlxgettext requires to be called using some command line options (unless it will included in the wesnoth GUI tool; in that case, you could use the GUI instead).

Unlike wmlxgettext 1.0 (perl version), this version can be used in **three** possible ways. They will be explained starting from the most suggested one, finishing with the unsuggested one.

The last paragraph, instead, will explain the optional parameters that could be used in any of the three usages explained in the previous paragraphs.

1.2.1 Wmlxgettext with implicit list of files

Note: This is the **only** way that 100% works under windows.

You can avoid to explicitly tells what files must be parsed by wmlxgettext. This is how you can do it on windows:

```
c:\<pythondir>\python wmlxgettext --domain=DOMAIN_NAME --directory=YOUR_ADDON_
↪DIRECTORY --recursive -o ./FILENAME.po
```

On linux/mac, you can simply use:

```
./wmlxgettext --domain=DOMAIN_NAME --directory=YOUR_ADDON_DIRECTORY --recursive -o ./
↪FILENAME.po
```

without explicitly call the python 3.x interpreter.

–domain=DOMAIN_NAME

With the option `--domain`, wmlxgettext will know wich is the `# textdomain` used by your wesnoth add-on. For example, if your _main.cfg will have:

```
[textdomain]
   name="wesnoth-xyz"
   path="data/add-ons/xyz/translations"
[/textdomain]
```

This is what you have to write into the `--domain` parameter:

```
--domain=wesnoth-xyz
```

–directory=ADDON_DIRECTORY

With the option `--directory`, wmlxgettext will know the starting path where all following files/scandirs should be searched. This is a fake example for windows:

```
--directory=c:\games\wesnoth\userdata\data\add-ons\YOUR_ADDON_DIRECTORY
```

–recursive

If `--recursive` option is used, wmlxgettext will scan recursively the directory typed on the `--directory` option and collect all .cfg and .lua files automaticly:

```
./wmlxgettext --domain=domain_name --directory=/home/user/games/wesnoth/userdata/add-
↪ons/Invasion_from_the_Unknown --recursive -o ./file.po
```

In the example showed above, infact, wmlxgettext will watch the directory `/home/user/games/wesnoth/userdata/add-ons/Invasion_from_the_Unknown` and it will collect, recursively, all .cfg / .lua files inside that directories (and subdirectories).

-o [OUTPUT_FILE]

If you use this option, wmlxgettext will actually create a .po file, saving it as `[OUTPUT_FILE]`.

The `-o` options accepts:

- either a file name with absolute path
- or a file name with relative path (for example: `./output.po`)
- or it could setted to "-" (`wmlxgettext -o - ...`) to write output to stdout

Also the parameter `--directory` discussed before can accept both an absolute path or a relative path starting from the current working directory (for exaple: `--directory=.` will assign to the `--directory` option the path of the current working directory).

1.2.2 Wmlxgettext with explicit list of files

Note: This method can work on windows **only if** the list of files is not very long (windows cannot read a very-long command line). Under windows is **highly** suggested to use the method described in the previous paragraph (*Wmlxgettext with implicit list of files*)

Instead of delegating to wmlxgettext the job for you, you can explicitly tells the complete list of files that wmlxgettext must parse:

```
./wmlxgettext --domain=domain_name --directory=/home/user/wesnoth/userdata/add-ons -o␣
↪./file.po Invasion_from_the_Unknown/_main.cfg Invasion_from_the_Unknown/other.cfg [.
↪..]
```

As the example shows, it is **highly suggested** to put the list of files **after** all other options. This is why, in this case, the option `-o ./file.po` is written before the file list starts.

Every file listed in list must be written as a relative path starting from the `--directory` directory.

So, coming back to the example showed above:

- `--directory` is `/home/user/wesnoth/userdata/add-ons`
- file n.1 is `Invasion_from_the_Unknown/_main.cfg`
- file n.2 is `Invasion_from_the_Unknown/other.cfg`.

This means that those two files will be searched and parsed:

- /home/user/wesnoth/userdata/add-ons/Invasion_from_the_Unknown/_main.cfg
- /home/user/wesnoth/userdata/add-ons/Invasion_from_the_Unknown/other.cfg

---

Note: DON'T use the `--recursive` option if you want to explicitly tell the list of the files to parse. If the option `--recursive` is used, the explicit list of file will be ignored.

1.2.3 Wmlxgettext with explicit list of files and output redirection

This is the **unsuggested** way to use wmlxgettext, since output redirection can create issues. When writing to stoud, infact, the console will use its own text codify instead of the UTF-8 codify you could expect.

For this reason, starting from version 2017.06.25.py3 the -o parameter is becomed mandatory, to discourage printing the output to stdout for a casual usage. So the old syntax used by wmlxgettext 1.x (perl version) is not anymore supported.

So you couldn't anymore invoke wmlxgettext like:

```
./wmlxgettext --domain=domain_name --directory=/home/user/wesnoth/userdata/add-ons␣
↪Invasion_from_the_Unknown/_main.cfg Invasion_from_the_Unknown/other.cfg [...] > ./
↪file.po
```

However it is still possible to print the output to stdout instead of to writing an actual file, if you **really** need it:

```
./wmlxgettext -o - --domain=domain_name --directory=/home/user/wesnoth/userdata/add-
↪ons Invasion_from_the_Unknown/_main.cfg Invasion_from_the_Unknown/other.cfg [...] |␣
↪application_accepting_wmlxgettext_stout_as_input_pipe
```

If you use the special value "-" for -o parameter (like showed above), than the output will be printed to stdout as desired.

This way printing to stdout would be possible only if explictly asked and only when it is actually requested on purpose by the user.

It could be, obliovsly, possible to print a file into stdout and redirecting the output to a file, but it is **higly** discouraged. On a standard use case (creating a pot file for a wesnoth addon) you should consider to use method 1 explained two paragraphs ago:

```
./wmlxgettext --domain=DOMAIN_NAME --directory=YOUR_ADDON_DIRECTORY --recursive -o ./
↪FILENAME.po
```

1.2.4 Optional parameters

Wmlxgettext 2.0 supports also other optional parameters, not explained in the previous paragraphs:

- `--warnall`: if used, wmlxgettext will show also optional warnings.
- `--fuzzy`: if used, all sentences stored in the .po file will be marked as fuzzy. (By default, sentences will be **not** marked as fuzzy).
- `--package-version`: With this option, you can immediatly print the package version number into the .po header infos. Usually you will add manually this info, so it is more an "easter egg" than a feature.
- `--no-text-colors`: if you use this flag, you disable colors shown in console when a warning/error message occurs. This option will become useful if wmlxgettext will be added to the python GUI for wesnoth tools (the code needed to 'paint the colors' must be not executed when wmlxgettext is launched from GUI)

Finally there is a last option, that an end-user should **never** use:

- `--initialdomain=INIT_DOMAIN`: It tells the name of the current domain when no `#textdomain` still found in .cfg/.lua file. By default it is `wesnoth` (and don't need to be changed).

1.3 Comparing perl vs python versions

wmlxgettext 2.x (this version), compared with wmlxgettext 1.0 (the old perl script), has its **pros**:

- More flexible command line (the old one is however supported).
- More explicit (and **more understandable**) warning/error messages returned to the end user.
- Optionally, can display a warning message if a WML macro is found into a translatable string (translatable string with WML macro will never be translated)
- Recognizes and captures lua bracketed strings
- Strings captured on a .lua file are reported to their **right** line of code
- Additional comments for translators are added to the **right sentence only**, avoiding to display it where it should not appear.
- Any file reference is written in a single line (like expected in a .po file)
- Can be used **also on windows** (requires python 3.x, however)
- User is not forced to list, one by one, every file that wmlxgettext must parse, but it can use instead the new `--recursive` option; wmlxgettext will search itself all .cfg and .lua files stored there.
- Can be added to the python GUI for (used by all other wesnoth tools)
- The code, even if complex and long, is more modular, and could be fixed/changed/forked in an easier way
- wmlxgettext 2.x sources are **very deeply** documented, in the *Source Code Documentation*.
- Supports another custom directive: `# po-override:` that allows you to override the automated WML/Lua list of informations, replacing it with your own custom message. The `# po:` directive is still available, since it allows you to ADD informations directed to the translator, WITHOUT touching the automated WML/Lua list of informations. `# po:` and `# po-override:` directives can be used together on the same translatable string. However only ONE occurrence of `# po-override:` can be defined for every translatable string.

But it has also its cons:

- code is much more huge (about 1400 lines of code splitted in several files) against the 300-400 lines of code required by the perl version

- execution speed is a bit slower.
- The output created by wmlxgettext 2.x is not exactly the same as the one created by wmlxgettext 1.0 (very small differences, however, nothing really important).

In the following paragraphs of this page we will show deeply some of the differences listed here above (only the most important ones that affect the resulting output .po file) from this rewritten version of wmlxgettext (2.x) vs the old version of wmlxgettext (1.0)

1.3.1 Error/Warning Messages are more understandable now

All type of error messages displayed by wmlxgettext 2.x is clear and intuitive, unlike the error messages displayed by perl wmlxgettext (1.0). We will show here only an example about what happens when unbalanced tags are found by perl wmlxgettext (1.0) and by python wmlxgettext (2.x).

We will use this WML file (ztest.cfg):

```
#textdomain wesnoth-mytest

# WML with unbalanced tag [scenario] ---> parsing this file will return an error
[scenario]
   id=my_scenario
```

As you can see the [scenario] tag is not closed, so both perl wmlxgettext (1.0) and python wmlxgettext (2.x) will return an error. Here is the error message displayed by perl wmlxgettext (1.0):

```
non-empty node stack at end of ztest.cfg at ./wmlxgettext line 203, <FILE> line 5.
WML seems invalid for ztest.cfg, node info extraction forfeited past the error point␣
↪at ./wmlxgettext line 210.
```

Here, instead, is the most user-friendly error message displayed by python wmlxgettext (2.x):

```
error: ztest.cfg:5: End of WML file reached, but some tags were not properly closed.
(nearest unclosed tag is: [scenario])
```

1.3.2 Additional comments for translators are added to the right sentence only

Additional comments for translator are additional informations useful to instructs translators how to translate better a particular sentence. Here we analyze the _main.cfg file of the mainline campaign Liberty (focus your attention at lines from 22 to 27):

```
1  #textdomain wesnoth-l
2  # This version forked from 1.2, 2/10/2007, and prepared for mainline by ESR
3  [textdomain]
4     name="wesnoth-l"
5  [/textdomain]
6
7  # wmlscope: set export=no
8  [campaign]
9    id=Liberty
10   name= _ "Liberty"
11   abbrev= _ "Liberty"
12   rank=110
13   first_scenario=01_The_Raid
14   define=CAMPAIGN_LIBERTY
```

(continues on next page)

(continued from previous page)

```
  icon="units/human-outlaws/fugitive.png~RC(magenta>red)"
  image="data/campaigns/Liberty/images/campaign_image.png"

  {CAMPAIGN_DIFFICULTY EASY   "units/human-peasants/peasant.png~RC(magenta>red)" ( _
↪"Peasant") ( _ "Easy")} {DEFAULT_DIFFICULTY}
  {CAMPAIGN_DIFFICULTY NORMAL "units/human-outlaws/outlaw.png~RC(magenta>red)" ( _
↪"Outlaw") ( _ "Normal")}
  {CAMPAIGN_DIFFICULTY HARD   "units/human-outlaws/fugitive.png~RC(magenta>red)" ( _
↪"Fugitive") ( _ "Difficult")}

  #po: Yes, that is "marchlanders", not "marshlanders".
  #po: "marchlander" is archaic English for an inhabitant of a border region.
  # wmllint: local spelling marchlanders
  description= _ "As the shadow of civil war lengthens across Wesnoth, a band of␣
↪hardy marchlanders revolts against the tyranny of Queen Asheviere. To win their way␣
↪to freedom, they must defeat not just the trained blades of Wesnothian troops but␣
↪darker foes including orcs and undead.

" + _"(Intermediate level, 8 scenarios.)"

  [about]
      title = _ "Campaign Design"
      [entry]
          name = "Scott Klempner"
      [/entry]
  [/about]
  [about]
      title = _ "Prose-doctoring and preparation for mainline"
      [entry]
          name = "Eric S. Raymond (ESR)"
      [/entry]
  [/about]
  [about]
      title = _ "Campaign Maintenance"
      [entry]
          name = "Eric S. Raymond (ESR)"
          comment = "current maintainer"
      [/entry]
      [entry]
          name = "Lari Nieminen (zookeeper)"
          comment = "current maintainer"
      [/entry]
  [/about]
  [about]
      title = _ "Artwork and Graphics Design"
      [entry]
          name = "Brendan Sellner"
      [/entry]
      [entry]
          name = "Kathrin Polikeit (Kitty)"
          comment = "portraits"
      [/entry]
      [entry]
          name = "Shadow"
      [/entry]
      [entry]
          name = "Blarumyrran"
```

(continues on next page)

(continued from previous page)

```
            comment = "story images, Rogue Mage line sprites"
        [/entry]
        [entry]
            name = "Sonny T Yamada (SkyOne)"
            comment = "Sprite animations (defense and attack) of Rogue Mage line"
        [/entry]
    [/about]
[/campaign]

#ifdef CAMPAIGN_LIBERTY

[binary_path]
   path=data/campaigns/Liberty
[/binary_path]

{campaigns/Liberty/utils}
{campaigns/Liberty/scenarios}

[+units]
    {campaigns/Liberty/units}
[/units]

#endif

# wmllint: directory spelling Grey Relana Helicrom Fal Khag
```

As you can see, at line 22 and 23, there are two `#po:` special comments wich add the additional information for translator about the usage of the "marchlanders" word. It is an explaination related to the sentence used in the description of the campaign where the "marchlanders" word is actually used.

So, in this case, you espect that the additional information is added only at the description string:

```
"As the shadow of civil war lengthens across Wesnoth, a band of hardy marchlanders␣
↪revolts against the tyranny of Queen Asheviere. To win their way to freedom, they␣
↪must defeat not just the trained blades of Wesnothian troops but darker foes␣
↪including orcs and undead.
"
```

Whell... perl wmlxgettext (1.0) add too many additional informations, as it showed here:

```
#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:10 _main.cfg:11
msgid "Liberty"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:18
msgid "Easy"
msgstr ""

#. [campaign]: id=Liberty
```

(continues on next page)

(continued from previous page)

```
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:18
msgid "Peasant"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:19
msgid "Normal"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:19
msgid "Outlaw"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:20
msgid "Fugitive"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:20
msgid "Difficult"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:25
msgid ""
"As the shadow of civil war lengthens across Wesnoth, a band of hardy marchlanders␣
↪revolts against the tyranny of Queen Asheviere. To win their way to freedom, they␣
↪must defeat not just the trained blades of Wesnothian troops but darker foes␣
↪including orcs and undead.\n"
"\n"
""
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:27
msgid "(Intermediate level, 8 scenarios.)"
msgstr ""
```

perl wmlxgettext print the additional information not only in the right sentence (where the code is emphatize, that is the only point where the additional information should be added: [line: 52-59]), but also print the additional info where it makes no sense, for example on msgid “Difficult” (where the detail about the usage of “marchlanders” word

is useless).

This functionality, instead, work correctly on python wmlxgettext (2.x):

```
#. [campaign]: id=Liberty
#: _main.cfg:10
#: _main.cfg:11
msgid "Liberty"
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:18
msgid "Peasant"
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:18
msgid "Easy"
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:19
msgid "Outlaw"
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:19
msgid "Normal"
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:20
msgid "Fugitive"
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:20
msgid "Difficult"
msgstr ""

#. [campaign]: id=Liberty
#. Yes, that is "marchlanders", not "marshlanders".
#. "marchlander" is archaic English for an inhabitant of a border region.
#: _main.cfg:25
msgid ""
"As the shadow of civil war lengthens across Wesnoth, a band of hardy marchlanders␣
↪revolts against the tyranny of Queen Asheviere. To win their way to freedom, they␣
↪must defeat not just the trained blades of Wesnothian troops but darker foes␣
↪including orcs and undead.\n"
"\n"
""
msgstr ""

#. [campaign]: id=Liberty
#: _main.cfg:27
msgid "(Intermediate level, 8 scenarios.)"
msgstr ""
```

As you can see, this time the additional information is added **only** when it is expected to be stored (on the sentence

where the “marchlanders” word is used).

CHAPTER 2

Source Code Documentation

This the new release of wmlxgettext, rewritten from scratch in python 3.x. Wmlxgettext is a python3 script that scans a list of .cfg (WML) and .lua files, capturing all translatable string found in the files and creates a pot (.po) file.

From now on, the (old) perl script will be called "wmlxgettext 1.0", while the new python3 script will be called "wmlxgettext 2.x".

Warning: this source documentation is a bit outdated. This documentation is however still valid to understand the wmlxgettext source logic. Please consider to take a look also to source code comments for a more updated source documentation.

2.1 Introduction

This part of the documentation is useful also for end-users not interested to learn how the source code internally work.

2.1.1 Pros and Cons

wmlxgettext 2.x (this version), compared with wmlxgettext 1.0 (the old perl script), has its **pros**:

- More flexible command line (the old one is however supported).
- More explicit (and more understandable) warning/error messages returned to the end user.
- Optionally, can display a warning message if a WML macro is found into a translatable string (translatable string with WML macro will never be translated)
- Recognizes and captures lua bracketed strings
- Strings captured on a .lua file is reported to its **right** line of code
- Any file reference is written in a single line (like expected in a .po file)
- Can be used also on windows (requires python 3.x, however)
- User is not forced to list, one by one, every file that wmlxgettext must parse, but it can use instead the new `--scandirs` option.

- Can be added to the python GUI for (used by all other wesnoth tools)
- The code, even if complex and long, is more modular, and could be fixed/changed/forked in an easier way
- wmlxgettext 2.x sources are **very deeply** documented here.
- Supports another custom directive: `# po-override:` that allows you to override the automated WML/Lua list of informations, replacing it with your own custom message. The `# po:` directive is still available, since it allows you to ADD informations directed to the translator, WITHOUT touching the automated WML/Lua list of informations. `# po:` and `# po-override:` directives can be used together on the same translatable string. However only ONE occurrence of `# po-override:` can be defined for every translatable string.

But it has also its cons:

- code is much more huge (about 1400 lines of code splitted in several files) against the 300-400 lines of code required by the perl version
- execution speed is slower.
- The output created by wmlxgettext 2.x is not exactly the same as the one created by wmlxgettext 1.0 (very small differences, however, nothing really important).

2.1.2 The new command line

wmlxgettxt 2.x could be invoked in the classical way:

```
./wmlxgettext --domain=DOMAIN --directory=DIRECTORY [FILELIST] > file.po
```

this syntax is required by wesnoth in order to build the `pot` target. However this syntax must be considered deprecated for UMC developers.

This other syntax is suggested, instead:

```
./wmlxgettext -o file.po --domain=DOMAIN --directory=DIRECTORY [FILELIST]
```

Or, even better:

```
./wmlxgettext -o file.po --domain=DOMAIN --directory=YOUR_ADDON_DIRECTORY --recursive
```

Using those last two syntaxes, infact, the file `file.po` is directly created by wmlxgettext instead of redirecting the output from `stdout` to the desired file.

If you use the last syntax, wmlxgettext will scan for you (recursively) your addon main directory and automaticly collect all .cfg and .lua files without any need to list them one-by-one.

Moreover, wmlxgettext 2.x, supports more options, that can be listed with:

```
./wmlxgettext --help
```

The most useful added options are:

- `--fuzzy`: allows to create a .po file with all fuzzy strings
- `--warnall`: show optional warnings

2.1.3 Output “lacks”

The .po file created with wmlxgettext 2.x may store the sentences in a different order. This becouse the list of the files could be read in a different order. However, every translatable string related to the same file is stored in the right order.

Lua function information (inside a .lua file or inside a lua code inside a WML file) is more verbose (it is not so good as it may sounds, unluckly). Wmlxgettext 2.x remembers some function names that wmlxgettext 1.0 forgets.

2.2 Error and Warning messages

When a WML or a Lua file contains a problem, wmlxgettext returns a warning (*if the problem is not critical*) or an error (*critical problem*) to the end-user to allow him/her to fix his own wesnoth addon.

When wmlxgettext must return an error, it calls `wmlerr` function; it calls `wmlwarn` function when it should simply display a warning. Both `wmlerr` and `wmlwarn` function are defined in `./pywmlx/wmlerr.py` module.

When importing *pywmlx*, wmlxgettext will import only `wmlerr` and `wmlwarn` functions (and `ansi_setEnabled`), since all other classes/functions in `./pywmlx/wmlerr.py` module are only internally required by `wmlerr` and `wmlwarn` functions to work properly.

2.2.1 About ansi_setEnabled function

When you `import pywmlx`, also `ansi_setEnabled` function is imported from `./pywmlx/wmlerr.py` module.

`ansi_setEnabled` accepts a boolean value (`True` or `False`). By default it is setted to `True`, but it will be `False` if the flag `--no-ansi-colors` was used in the command line:

```
# ./wmlxgettext:100
parser.add_argument(
    '--no-ansi-colors',
    action='store_false',
    default=True,
    dest='ansi_col',
    help=("By default warnings are displayed with colored text. You can " +
          "disable this feature using this flag.\n" +
          "This option doesn't have any effect on windows, since it " +
          "doesn't support ansi colors (on windows colors are ALWAYS" +
          'disabled).')
)
# ...
# ./wmlxgettext:141
pywmlx.ansi_setEnabled(args.ansi_col)
```

When calling this function, wmlxgettext instructs `wmlerr` and `wmlwarn` to use (*or to don't use*) ansi colors when displaying error messages. Ansi colors, however, will be displayed only on Posix OSes (Linux and Mac) and not on Windows, which doesn't support ansi escape codes.

On windows platform, `ansi_setEnable` will be interally ignored, and warning/error messages will be always displayed non-colored.

2.2.2 wmlerr() and wmlwarn() usage

`wmlerr` and `wmlwarn` functions requires the same parameters and they will display the error/warning message in the same way.

`wmlerr` and `wmlwarn` requires those two *string* parameters:

- **finfo**: wich is *filename:X* (where *filename* is the .cfg/.lua file that contains the problem, and *X* is the line number of that file)

- **message**: the message to display

For example, when printing a warning, the warning message will displayed in this way:

```
warning: filename:x: my_message
```

If coloured, "*warning*" will be blue, "*filename:x*" will be yellow, and *warning message* will be white. The same colors will be applied to error message, except the world "*error*" (wich replace the world "*warning*") that will be red.

The last difference:

- `wmlerr` stops wmlxgettext execution;
- `wmlwarn`, instead, does **not** stop wmlxgettext execution.

2.2.3 How wmlerr() and wmlwarn() internally works

`wmlerr` and `wmlwarn` internally behave very differently, since they use python Exceptions / warning system.

`wmlwarn` calls `warnings.warn` function, wich was previously overridden by `my_showwarning` fuction defined in `./pywmlx/wmlerr.py` module (line 67). Override was possible thank of:

```
# ./pywmlx/wmlerr.py: 75
warnings.showwarning = my_showwarning
```

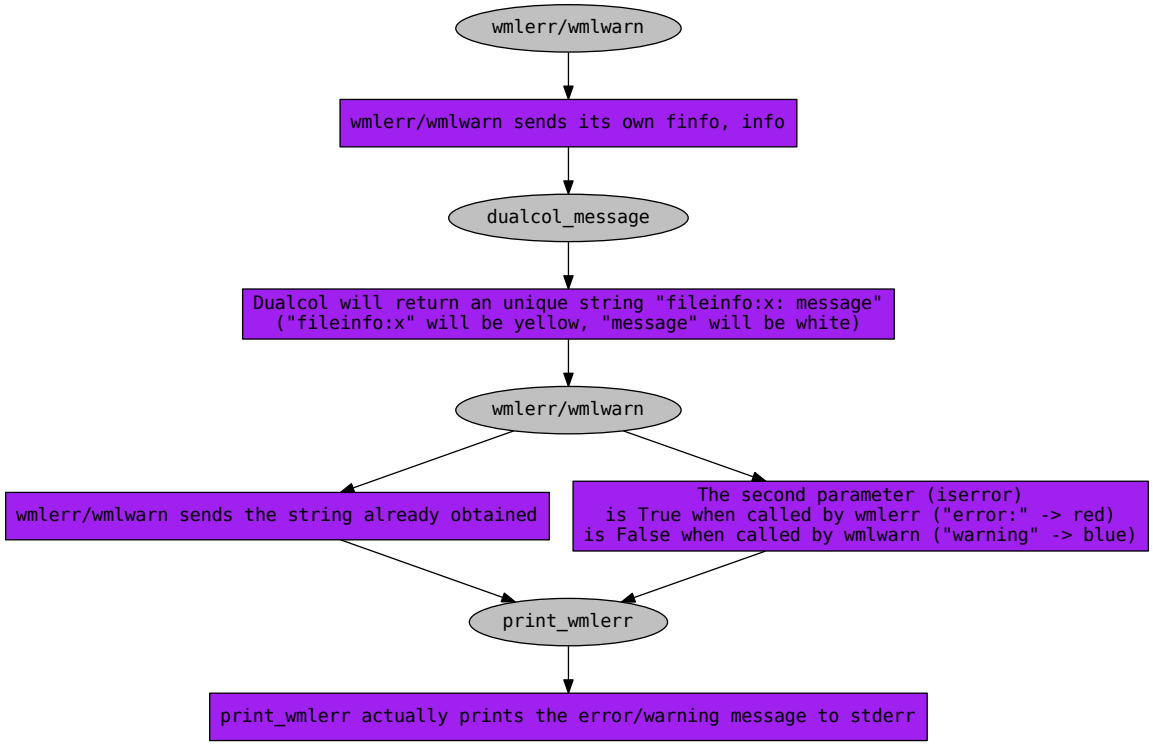
`wmlerr`, instead, raise a python exception (that could be checked on unittest: see *Using unittest with wmlerr()*) and replace its default behaviuour with a custom one.

Python exception system, infact, is very useful while debugging code (it can trace both errors and warnings), but needed the ovverrides already explained. This becouse, by default, python shows line code **of the script** (in this case: line code of *wmlxgettext* itself) when displaying and warning, and adds tracback infos **returned by the script**.

Those kind of infos are completely undersired, since the warnings and errors that *wmlxgettext* should return to the end-user, must say only infos that end-user actually needs (only the errors and messages infos related to WML and Lua files parsed by *wmlxgettext*)

Overrides made by `my_showwarning` (for warns) and by manually raising an exception with a coustom behaviour (in `wmlerr` function, when an error occurs) ensures that errors and messages will show **only** the infos that are actually expected to be announced to the end-user.

Both `wmlerr` and `wmlwarn`, however, internally use `dualcol_message(finfo, message)` and `print_wmlerr(message, iserr)` functions. Those functions will correctly print error/warning message



`dualcol_message` and `print_wmlerr` will **not** add colors:

- if current OS is Windows (even if `ansi_setEnabled` is `True`)
- **or** if `--no-ansi-colors` flag was used in command line (`ansi_setEnabled` is `False`)

2.2.4 Using unittest with wmlerr()

`wmlerr` behave differently if the global variable `is_utest` (global variable of module `./pywmlx/wmlerr.py`) is `False` (default value) or if it is `True` (must be `True` **only** on a unittest session).

During an unittest session, infact, it is required to change that value from `True` to `False`, calling `wmlerr_debug()` function from your unittest module. For this reason, unittest that requires to check `wmlerr` and `wmlwarn` should also explicitly add this import:

```
from pywmlx.wmlerr import wmlerr_debug()
```

since `wmlerr_debug()` is not imported when you simply `import pywmlx`. The function `wmlerr_debug()` must then be called somewhere on your unittest function **before** using `wmlerr()`.

After setting `is_utest` to `False` calling `wmlerr_debug()`, *wmlerr* can raise the exception, maintaining the traceback infos required (on unittest session) to verify that the exception was correctly raised.

2.3 Introducing WML and Lua parser

Wmlxgettext parse .lua and .cfg (WML) files line-by-line through his own Finite State Machine (Deeply explained in the chapter *The State Machine*).

This chapter, instead, will explain, in general, how differently WML and Lua codes are managed, and it will explain also:

- `PoCommentedString` class (module: `./pywmlx/postring.py`)
- `WmlNode` class (module: `./pywmlx/postring.py`)
- `WmlNodeSentence` class (module: `./pywmlx/postring.py`)
- the `./pywmlx/nodemanip.py` module

2.3.1 WML parsing fundamentals

WML (Wesnoth Markup Language) is a "*tagged*" language, like XML. Every .cfg (WML) file contains a list of nested [tags] wich must be properly closed. Here is a (fake) example of a WML file:

```
#textdomain your-textdomain-name
[scenario]
   id=scenario_id
   name= _ "scenario name (translatable)"
   map_data = ...

   [objective]
       description= _ "objective text (translatable)"
   [/objective]

   [event]
     name = "start"

     [message]
        message = _"I am saying something (translatable)"
        speaker = id_of_the_speaker
     [/message]

   [/event]
[/scenario]
```

wmlxgettext must collect all translatable strings, and must keep all important infos contained inside every opened tag. For example, look at the following tag:

```
[message]
   message = _"I am saying something (translatable)"
   speaker = id_of_the_speaker
[/message]
```

wmlxgettext must remember that the translatable string *"I am saying something (translatable)"* appeared at line 15 of your file `some-file.cfg`, inside a `[message]` tag with `speaker=id_of_the_speaker` and store properly those infos into the .po file.

Since the State Machine parser reads any file line-by-line, it is required to store in memory all those infos, on **memory nodes**

2.3.2 WML nodes

Everytime a new open tag is found, a new node is added in memory.

Note: All the three cases showed here are managed in the same way. A new [tag] node is always created:

- [tag] –> A new standard [tag] is opened.
- [+tag] –> A new updating [tag] is opened.
- [-tag] –> Another possible syntax

(+ and - starting sign will be ignored)

When a `[tag]` node has to be added in memory, a new `WmlNode` object is created and added in memory.

A `WmlNode` object will contain those data infos:

- **tagname**: name of tag (it will be saved as *"[tagname]"*)
- **fileref**: filename containing the node (relative path)
- **fileno**: unique **id value** assigned by wmlxgettext for current file
- **sentences**: list of translatable strings found inside the node, stored as WmlNodeSentence objects
- **wmlinfos**: list of wml infos (example: `speaker=id_of_the_speaker`).
- **autowml**: usually `True`. If `False` the `wmlinfos` list will be not used.

This node will be closed when the right *"tag-end-markup"* (`[/tag]`) will be found.

The class `WmlNode` also provides some functions that will be discussed later.

Storing translatable strings

When a translatable string will be found, it will be added in the current node.

Each time a new translatable string found, a new `WmlNodeSentence` object will be added to the current WML node (but only if the current domain is equal to the addon domain).

A `WmlNodeSentence` will have those properties:

- **sentence**: the translatable string (text)
- **ismultiline**: `True` if it is a multi-line string
- **lineno**: line number where the translatable string was located. (if multi-line, the line number where the the string started).
- **lineno_sub**: The name is ambigous. This parameter (integer value) is a progressive value, expecially useful when more than one string was stored in the same line of the same file.
- **overrideinfo**: `None` or *'string'*. If `# po-override: overrideinfo` directive was found, the override-info will be stored here.
- **self.addedinfo**: `None` or *list of strings*. If one or more `# po: addedinfo` directive(s) found, the info will be added in this list.

2.3.3 The postring dictionary

Writing a .po file is the final objective of wmlxgettext. Every translatable string in a .po file must appear **one time only**, and must contain all important useful infos (auto-captured infos and added infos by the UMC developer with `# po: addedinfos` directive in .cfg file source).

Python dictionaries are pair of values (key, value) where *'key'* is **always** unique. Moreover it will allow to quickly search if a translatable string was already stored in memory.

This dictionary is:

```
# ./wmlxgettext:144
sentlist = dict()
```

wich is also known and managed by the state machine parser (wich is discussed in the next chapter).

The dictionary contains all sentences that wmlxgettext will actually write in the .po file

- **key**: the key is a copy of the plain sentence (using only lower letters). Since it is expected that all wesnoth extensions will use english in their .cfg files the string.lower() python function was used here.
- **value**: the value is the sentence, with all additional infos that will be written in the .po file. This value is a `PoCommentedString` object.

So, before actually writing the .po file, wmlxgettext needs to create and update its dictionary of `PoCommentedString` objects.

2.3.4 Converting WmlNodeSentence to PoCommentedString

When wmlxgettext parse a WML file, it must store WML nodes in memory. Each `WmlNode` object may contain (or not) one or more translatable strings, stored in `node.sentences` list (list of `WmlNodeSentence` objects).

Each time a WML node is closed, before removing the node from memory, wmlxgettext will look at the `WmlNode` object, checking if it contains `WmlNodeSentence` objects or not.

Every `WmlNodeSentence` object contained in `WmlNode` object will be converted in a temporary `PoCommentedString` thank of the `nodesentence_to_posentence` function provided by `WmlNode` class.

This function is very complex, since it must assemble a `PoCommentedString` searching the required values in different places:

- some infos are stored in the `WmlNode` itself
- other infos are stored in `WmlNode` itself, but must be *"assembled"*.
- other infos are stored in the single `WmlNodeSentece` contained in the `WmlNode` object.

PoCommentedString data infos

Now it is time to explain all data infos contained in a `PoCommentedString`:

- **sentence** = translatable string text.
- **wmlinfos** = list of wmlinfos.
- **addedinfos** = infos added with `# po: something` directives
- **finfos** = list of files and line number where any occurence of the string was found.
- **orderid = it is an (unique) tuple of three values:**
  - *fileno*: the file where the string was found the first time (file with lower fileno id value.
  - *lineno*: the line numeber, in *fileno*, where the string was found the first time.
  - *lineno_sub*: line_sub is a progressive value. It will be helpful to assign the correct order of the sentences, when two or more sentences were stored in the same file and in the same line.
- **ismultiline** = `True` if it is a multiline string.

The **orderid** tuple is very important, becouse, when wmlxgettext must write down all `PoCommentedString` objects from the "*postring*" dictionary to the .po file, it must print them in the right order (and not randomly):

```
# ./wmlxgettext:196
for posentence in sorted(sentlist.values(), key=lambda x: x.orderid):
```

When converting a `WmlNodeSentence` object to a `PoCommentedString` object, `WmlNode.assemble_orderid` create the tuple of three values to pass to `PoCommentedString.orderid` parameter:

- **fileno** (first value) –> comes from the `WmlNode` object containing the `WmlNodeSentence`.
- **lineno** and **lineno_sub** –> comes from the single `WmlNodeSentence`.

`PoCommentedString` and `WmlNode` both have a `wmlinfos` list, but they are conceptually different:

- `WmlNode.wlinfos` contains **single pieces** of infos captured on the WML node (example: `speaker=speaker_name` or `id=value`).
- Those single pieces must be assembled (with `WmlNode.assemble_wmlinfo`) to create a **single** `PoCommentedString.wmlinfos` element.
- So, when converting a `WmlNodeSentence` to a `PoCommentedString`, all *wmlinfos* contained in the `WmlNodeSentence` will add a **single** `PoCommentedString` *wmlinfo*.

About overrideinfo and addedinfos

A `WmlNodeSentence` object can contain an override info. This will happen if `# po-override: overrideinfo` directive was found in the WML/Lua file.

The override info, if exists, will be written directly on `PoCommentedString` as a `PoCommentedString.wmlinfos` element. `WmlNode` wmlinfos list will be ignored for that `WmlNodeSentence` and `assemble_wmlinfos` will not executed on that single conversion.

"*Addedinfos*", instead, behave in the same way both in `WmlNodeSentence` and in `PoCommentedString` objects. Those are additional infos to display to translator. If a `WmlNodeSentence` object contains elements in `addedinfos` list, those elements will be added in the `PoCommentedString-addedinfos` list. This will happen if one or more `# po: addedinfo` directive(s) was found in WML/Lua file.

Create a new dictionary key or update an existing one?

So, when closing a WML node, all `WmlNodeSentence` objects contained in that `WmlNode` object will be converted to temporary `PoCommentedString` objects.

Those temporary `PoCommentedString` objects will be not immediately stored in the dictionary, since the dictionary must contain **one instance only** of any sentence.

This why all temporary `PoCommentedString` objects created by `WmlNode.nodesentence_to_posentence` function will be "scanned".

- If a temporary `PoCommentedString` objects contains an instance of an **already existing** translatable string, the dictionary key will be updated (*no new key will be added*). The function `update_with_commented_string` of the `PoCommentedString` object contained in the dictionary key will be executed to update that `PoCommentedString` object.
- If a temporary `PoCommentedString` object contains a **new** translatable string not previously stored in the dictionary, this object will be simply added in the dictionary

```
# ./pywmlx/nodemanip.py:15
def _closenode_update_dict(podict):
    if nodes[-1].sentences is not None:
        for i in nodes[-1].sentences:
```

(continues on next page)

(continued from previous page)

```
            posentence = podict.get(i.sentence.lower())
            if posentence is None:
                podict[i.sentence.lower()] = (
                    nodes[-1].nodesentence_to_posentence(i) )
            else:
                posentence.update_with_commented_string(
                    nodes[-1].nodesentence_to_posentence(i) )
```

As you can see this check is actually performed inside the `./pywmlx/nodemanip.py` module, explained in the next paragraph.

2.3.5 The nodemanip module

Note: Until now this chapter explained:

- The structure of WML language and why wmlxgettext use `WmlNode` objects to store the WML tree structure in memory.
- `WmlNodeSentence` objects: the data type used by `WmlNode` objects to internally store translatable string(s) found inside the WML node stored in memory by that `WmlNode` object.
- `PoCommentedString` dictionary: where the translatable strings will be stored, as `PoCommentedString` objects (where a `PoCommentedString` object describe how actually the translatable string will be writte in .po file)
- `PoCommentedString` objects data infos
- How and when `WmlNodeSentence` objects will be converted into `PoCommentedString` objects

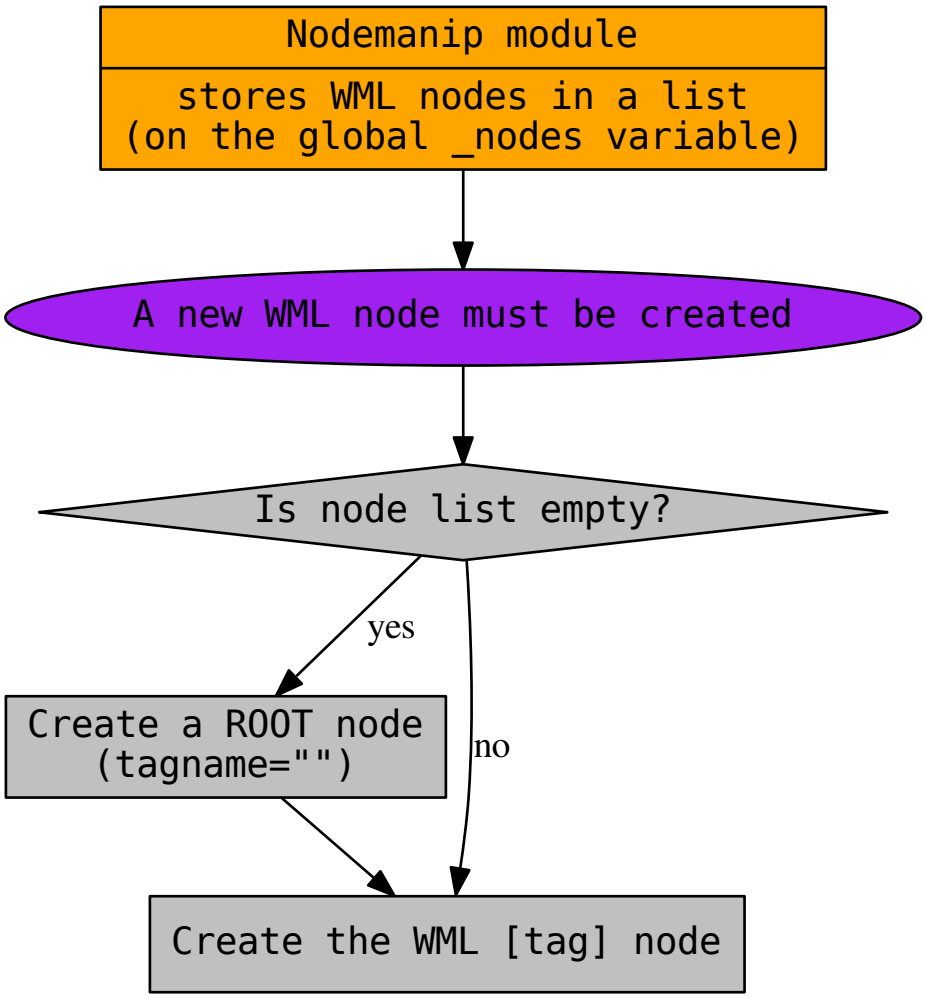
---

Now it is time to talk about `./pywmlx/nodemanip.py` module, the module wich actually manage when and how to store/clear WML nodes in memory.

Wmlxgettext main script file (*or better, the state machine*), infact, does not directly create/delete WML nodes in memory, but it delegates this job to the `./pywmlx/nodemanip.py` module (from now on: `nodemanip`).

This approach ensure that wmlxgettext internal code will be safer and easier to maintain than managing directly nodes in all the part of code where it will be required to manipulate WML nodes.

Storing a new WML node



`nodemanip` stores all WML nodes in a list, and not in a real tree structure. This becouse, as explained in the very beginning of this chapter, WML language is structured by **nested tag**, where any new *child* tag must be closed before its *parent* tag. Coming back to the WML sample code showed on the beginning of this chapter (with added comments):

```
# [scenario] is the first tag encountered in the WML.
# [scenario] tag is the parent of all following (nexted) WML tags and
#            it will be closed after all its child tags
[scenario]
   id=scenario_id
   name= _ "scenario name (translatable)"
   map_data = ...

   # [objective] tag does not have childs, so it will be closed immediately
   # after its opening
```

(continues on next page)

(continued from previous page)

```
   [objective]
       description= _ "objective text (translatable)"
   [/objective]

   # again... [event] tag will have a child: the tag [message].
   # the tag [message] must be closed before the parent [event] tag.
   [event]
     name = "start"

     [message]
        message = _"I am saying something (translatable)"
        speaker = id_of_the_speaker
     [/message]

   [/event]
[/scenario]
```

So why WML nodes can be stored in a list:

- everytime a new node is added, we can simply add an element in the list. The last item in the list is the last WML node opened.
- the last node in list, is the current node and it is the node we will must close before all other nodes in memory
- when the current node (last node in list) is closed, it will be removed by the list, so the last item on the list (the new current node) will be the parent node, for example, look at the WML sample code above:
  - when [event] tag is opened a new [event] node is added in node list.
  - when [message] tag is opened, a new [message] node is added in node list.
  - when [/message] found, then the [message] node is removed from list and the [event] tag will be now the last node in list (current node)

Coming back to the already displayed flow chart, we could notice that there is a special **ROOT** node that it will be created by nodemanip. It is a fake node required to avoid memory leaks and it will store all translatable strings stored outside any tag (for example a translatable string inside a macro definition). All captured `wmlinfos` in ROOT node will be ignored, since `autowml` is setted to `False`.

ROOT node is also special becouse, when created, cannot be deleted until the end of the WML file reached.

Deleting a WML node from memory

Clearing a WML node is the most important work performed by `nodemanip` module since, before actually clearing the node from memory, we must verify if the WML code is correctly written:

- the closing tag `[/tagname]` must be equal to the last `[tagname]` in list (current WML node). Else, a critical error must be returned (calling `wmlerr` function - wmlxgettext should stop execution)
- a critical error should be also returned when a close tag is unexpected at all, since no tags are openend (the list of WML node is still empty **or** the current WML node is the ROOT node).

All those checks is done by the `closenode` function on `nodemanip` module:

```
# ./pywmlx/nodemanip.py:73
def closenode(closetag, mydict, lineno):
```

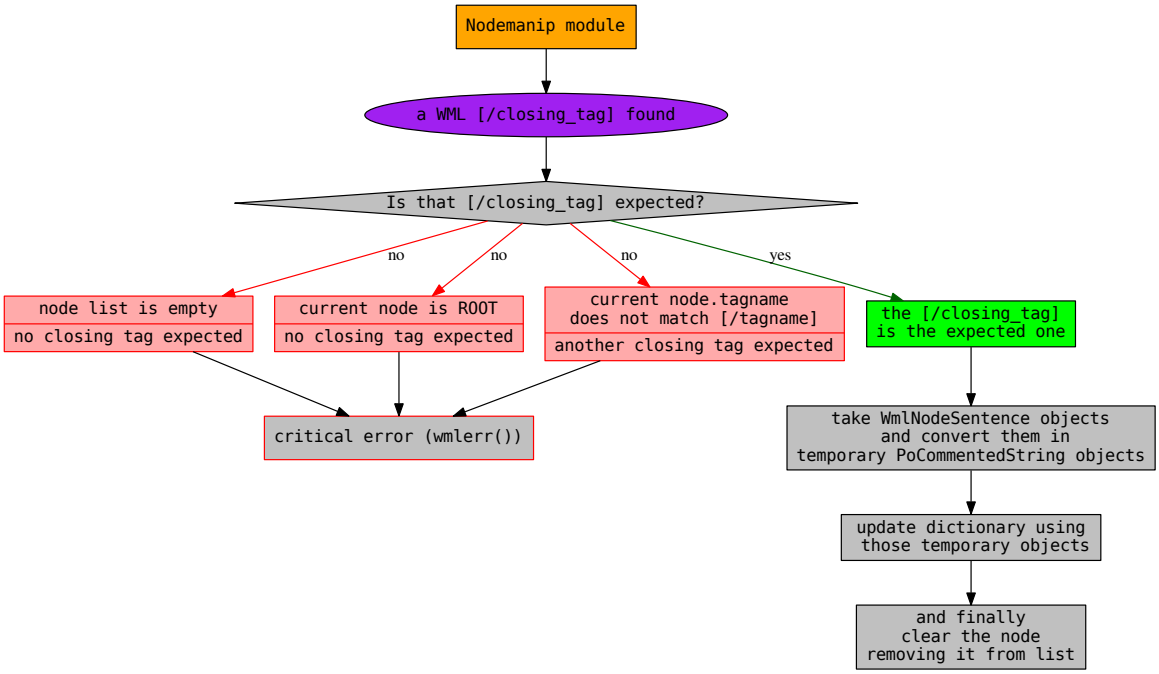
But, even if the closing tag `[/tagname]` is the expected one, `nodemanip` module does not immediately clear the node from the nodes' list.

```
# ./pywmlx/nodemanip.py:15
def _closenode_update_dict(podict):
    if nodes[-1].sentences is not None:
        for i in nodes[-1].sentences:
            posentence = podict.get(i.sentence.lower())
            if posentence is None:
                podict[i.sentence.lower()] = (
                    nodes[-1].nodesentence_to_posentence(i) )
            else:
                posentence.update_with_commented_string(
                    nodes[-1].nodesentence_to_posentence(i) )
```

Note: `_closenode_update_dict()` function is internally called by `closenode()` function of the `nodemanip` module.

As previously explained in *Converting WmlNodeSentence to PoCommentedString* and all its subparagraphs, infact, `nodemanip`, before closing the node:

- it will convert all `WmlNodeSentence` objects contained into the pending `WmlNode` object, before removing it from the list.
- all the `PoCommentedString` temporary values created by the conversion will be used to update the dictionary (more details about this process can be found at *Converting WmlNodeSentence to PoCommentedString* and all its subparagraphs).



Adding a new translatable string into the current WML node

Every translatable string found inside a WML file must be stored in the current WML node as a `WmlNodeSentence` object.

Every time a new WML file is opened, the node list `_nodes` on `nodemanip` module is empty (or better, is `None`).

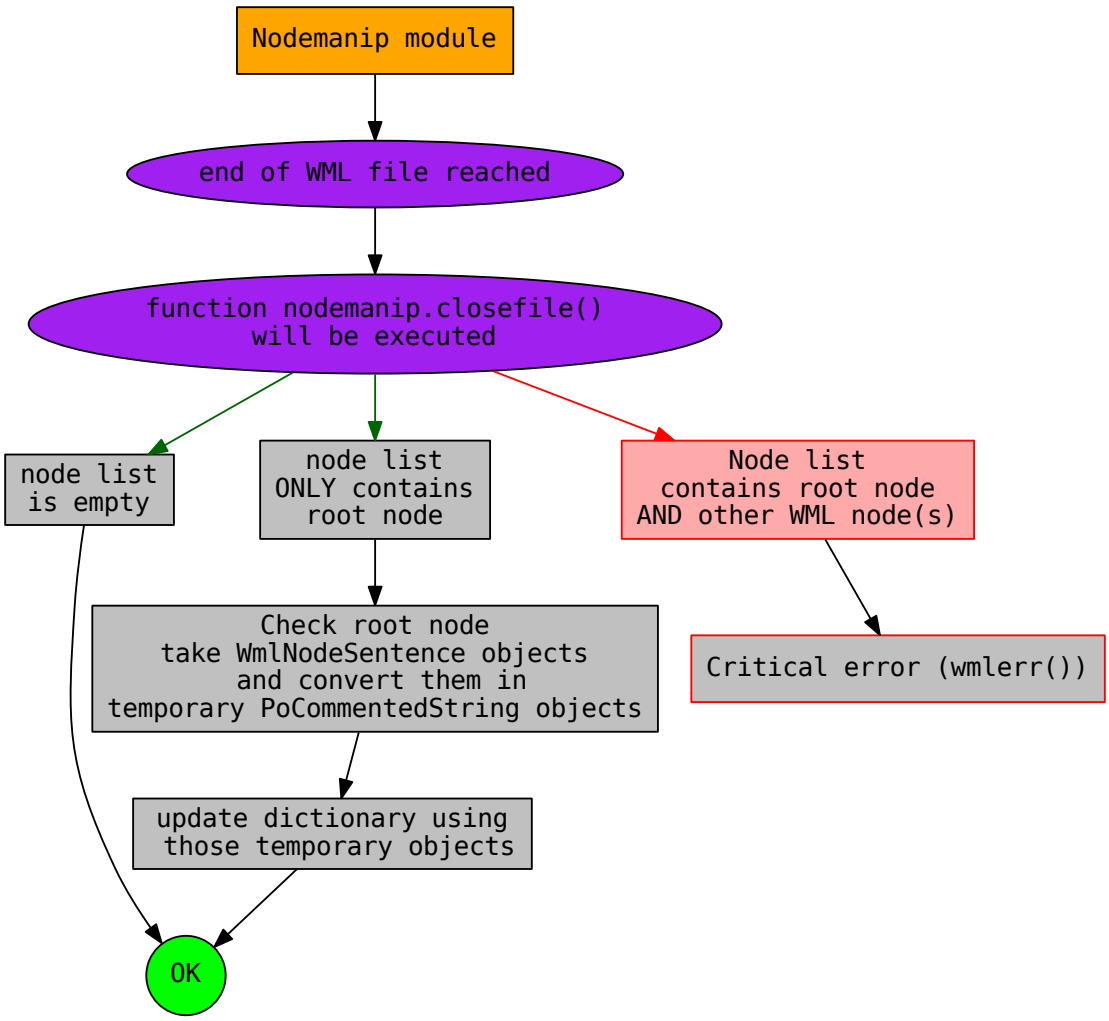
Usually, a ROOT WML node is created before creating the first actual WML node. This allows to store translatable strings located outside any tag.

But it could happen that a translatable string is found when node list is still empty (and when ROOT node does not still exist).

This why the `nodemanip.addNodeSentence` function, before trying to add the translatable string in current WML node, checks if the node list is not empty (or better, `is not None`). If the node list is empty, it creates the ROOT WML node and add the translatable string into that node.

What nodemanip does when end of WML file reached

When end of WML reached, `nodemanip` module will run its own `closefile()` function. There are three possible cases, as showed in the following flow chart:



Since the root node is not a standard WML node, and since it cannot be closed by any tag, `nodemanip` needs to explicitly explore it when the end of the WML file reached (otherwise the translatable strings stored in root node will be not added in dictionary).

2.3.6 Parsing Lua file (or lua code)

Parsing a lua file (or a .lua code inside a WML file) is somewhat *"easier"*. Here there is a sample .lua code (on an actual .lua file used by a wesnoth addon (Invasion from the Unknown)).

```
-- Invasion From The Unknown campaign
-- note: the original code is slightly different than this one we are
--       showing in this sample code
-- original code can be found on file: lua/gui/bug.lua:163
local function preshow()
     local _ = wesnoth.textdomain('wesnoth-Invasion_from_the_Unknown')
     local msg = _ "An inconsistency has been detected"
```

(continues on next page)

(continued from previous page)

```
    if report then
        msg = msg .. "\n\n" .. _ "Please report this to the maintainer!"
    end
    -- (other code here, omissed)
end
```

As the sample code shows, lua is a **procedural** language. Wmlxgettext does not *"parse"* .lua code, but:

- captures translatable strings, **directly** as `PoCommentedString` objects.
- the only *"wmlinfo"* captured inside a lua code is the last function name found in the .lua file

Lua code used on wesnoth add-ons can recognize those directives:

- `# po: <addedinfo>` to add infos to write to translators
- `# po-override: <override>` to override wmlinfo

Unlike WML code the textdomain is changed with the line

```
local _ = wesnoth.textdomain('wesnoth-Invasion_from_the_Unknown')
```

Note: All *"WML directives recognized by lua code"* showed above must be written inside lua comments (introduced by `--`), like the following code sample:

```
-- # po: my additional info
```

You must write **ONE** directive at time, into a new line:

```
-- this is a good example
-- # po: my additional info

-- this is, instead, a bad example
somecode = somevalue -- # po: my additional info
```

The directive `# wmlxgettext: <WML code>` is instead **not** supported in Lua code, since it is required by wmlxgettext only when parsing WML code (usually that directive is used when it is required to use unbalanced tags, avoiding error messages produced by unbalanced tags).

2.4 The State Machine

While developing this tool, one big issue was the WML parsing, since WML allow to add nested Lua code. The classical (perl) approach was to use two separate functions, one dedicated to lua code, and one for WML. The classical approach, however, can lead to some problems, when we face WML file with nested Lua code, so why another approach is used here.

This release has an unique "parser", using a finite state machine that reads every line of a file (Lua or WML) and perform the proper action (running a concrete state) when an important thing was found (for example, a translatable string).

```
# ./wmlxgettext:146
pywmlx.statemachine.setup(sentlist, args.initdom, args.domain)
for fx in args.filelist:
```

(continues on next page)

(continued from previous page)

```
    # omissing some code
    # ./wmlxgettext:157
    if fname[-4:].lower() == '.cfg':
        pywmlx.statemachine.run(filebuf=infile, fileref=fx,
                        fileno=fileno, startstate='wml_idle', waitwml=True)
    if fname[-4:].lower() == '.lua':
        pywmlx.statemachine.run(filebuf=infile, fileref=fx,
                        fileno=fileno, startstate='lua_idle', waitwml=False)
```

First of all, the state machine is initialized with the **statemachine.setup()** function (called one time only during all the script execution).

Then wmlxgettext will execute **statemachine.run()** function every times we open a new file (listed on args.filelist). This is the **statemachine.run()** parameters list:

- **filebuf**: the file buffer to read
- **fileref**: the name of file (relative path to –directory)
- **fileno**: a progressive (and unique) **id** value assigned to the file
- **startstate**: the name of the state where the state machine must start. Its value is assigned to '*wml_idle*' for WML (.cfg) files or assigned to '*lua_idle*' for .lua files
- **waitwml**: Its value is *True* if we are parsing a WML file. It is *False* if we are parsing a Lua file. Infact, only if a Lua code is indented in a WML file you could "expect" to exit from lua parsing and returning to WML parsing. In a .lua file, instead, you will have only Lua code.

2.4.1 The State class

Now it is time to start to explain more deeply how the state machine works. The State class has 3 properties:

- **regex**: it is the regular expression to match. If the regex matches, than the run function will be executed.
- **run (*self, xline, lineno, match*):**
  - *xline*: the line of the file we are parsing
  - *lineno*: current line number
  - *match*: the match object returned by `re.match(regex)`
- **iffail**: the state (state name) to reach if the regex does not match (usually the next state).

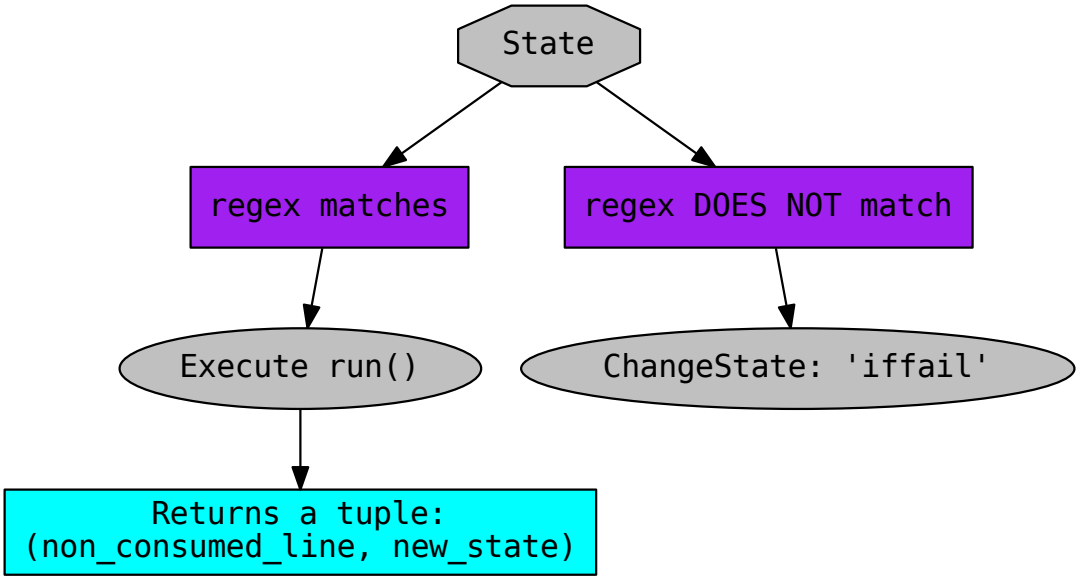
The State class prototype (*./pywmlx/state/state.py*) does not contain any actual code. The concrete states are defined in *./pywmlx/state/lua_states.py* and in *./pywmlx/state/wml_states.py* using temporary classes (for better code readability).

All states are stored in statemachine into a dictionary (**_states**) with:

- key = State name (example: *'wml_idle'*)
- value = concrete State object

Standard States

Standard states works exactly as previously explained:



The regexp is verified through `re.match`, so it maches only if the rule is True at the *very start* of the line. If it matches, than **run()** is executed.

Run() returns a pair of values (tuple):

- **xline**: the non-consumed part of the line. If the line is to be considered consumed, then xline will be setted to *None*
- **nextstate**: label of the next state to go. Usually it is *'wml_idle'* or *'lua_idle'* since the parsing is recursive.

If the regexp does not match, the **iffail** state will be reached. Usually the iffail is equal to the "*next state*". See *State Sequence*

Always-Run States

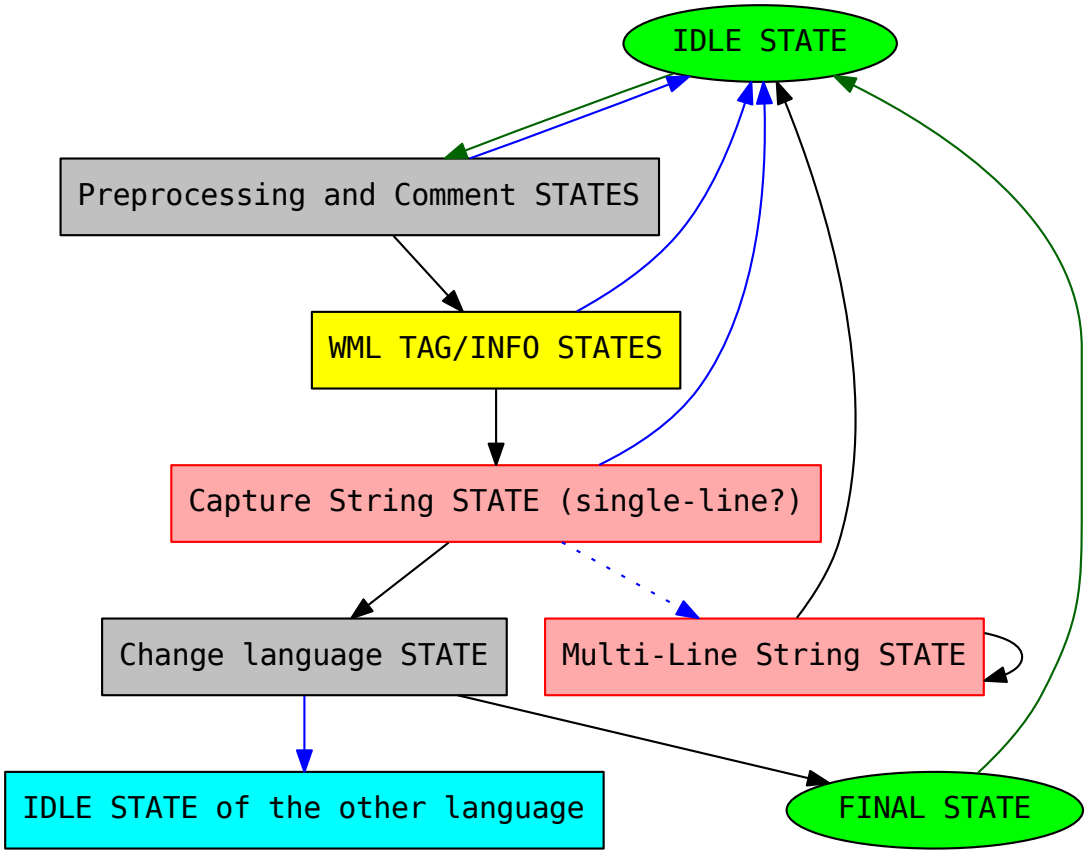
Always-run states are special states with `regexp = None`

Unlike standard states, an always-run state will **always** execute its own **run()** function. An example of always-run state is *'wml_idle'* state.

An always-run state does not actually require the **iffail** parameter. This is why always-run states have `iffail = None`

2.4.2 State Sequence

Now it is the time to show the generic state sequence:



This is, more or less, the design that is applied both for WML and Lua states. However the flow chart already displayed is mainly focused to WML states:

- **Arrows:**
  - green -> Always-run states (IDLE and FINAL) **always** go into the state pointed by the green arrow
  - blue -> Standard state reach the State pointed by blue arrow when the regex found a match.
  - black -> Standard state reach the State pointed by black arrow when the regex **DOESN'T** match
- **Boxes/Ellipses:**
  - IDLE and FINAL states are special states that appears both in WML and in Lua code. They are displayed in green circle since they are "always run" states.
  - Preprocessing States appears both in WML and Lua code, even if WML and Lua use different states (for example, *#wmlxgettext* is not needed in Lua code). They are standard states (grey box)
  - *'wml_getinf'* and *'wml_tag'* states appears only in WML states (yellow box)
  - String States (red boxes) behave very differently in WML and in Lua.
  - Change Language State checks if WML code switch to Lua or vice-versa. If the language is changed, the IDLE state of the other language will be reached (cyan box).

IDLE and FINAL States

Both IDLE and FINAL states check if there is a pending string, and if it is so, pending string will be stored in memory.

- **WML**: checks `pymlx.state.machine._pending_wmlstring`. If `pymlx.state.machine._pending_wmlstring is None` there is no pending WML string to store
- **Lua**: checks `pymlx.state.machine._pending_luastring`. If `pymlx.state.machine._pending_luastring is None` there is no pending Lua string to store

Both Lua and WML pending strings, before actually storing its own value, perform some cheks:

- verify if it is a translatable string
- verify if the current domain is the same of the addon domain name
- if so, it fixes the string for storage, and then store it

However WML pending string is stored in a very different way then Lua pending string:

- Lua pending string is directly stored, as a `PoCommentedString`, in the *"posentence dictionary"*.
- WML pending string is, instead, stored in the current WML node as a `WmlNodeSentence`. Only when the current WML node will be closed, all `WmlNodeSentence` objects contained in the node will be stored in the *"posentence dictionary"*. (See: *The nodemanip module* and *Converting WmlNodeSentence to PoCommentedString*)

WmlFinalState always return the pair `(xline, 'wml_idle')` while *LuaFinalState* always return the pair `(xline, 'lua_idle')`, where `xline` is setted to `None` in both cases. As previously explained, infact, when `xline` is `None`, the line is considered completely consumed and the statemachine will read the next line of the file.

Finally, the *'lua_final'* state perform another action, but it will be explain later. See *About storing the last Lua function name*.

Capture String States

When a string (translatable or not) is found, then the regexp of the proper "*Capture String*" state matches. Captured string will be stored as `pymlx.state.machine._pending_wmlstring` (WML string), or as `pymlx.state.machine._pending_luastring` (Lua string).

Now it is the time to discuss deeply about those capturing string states.

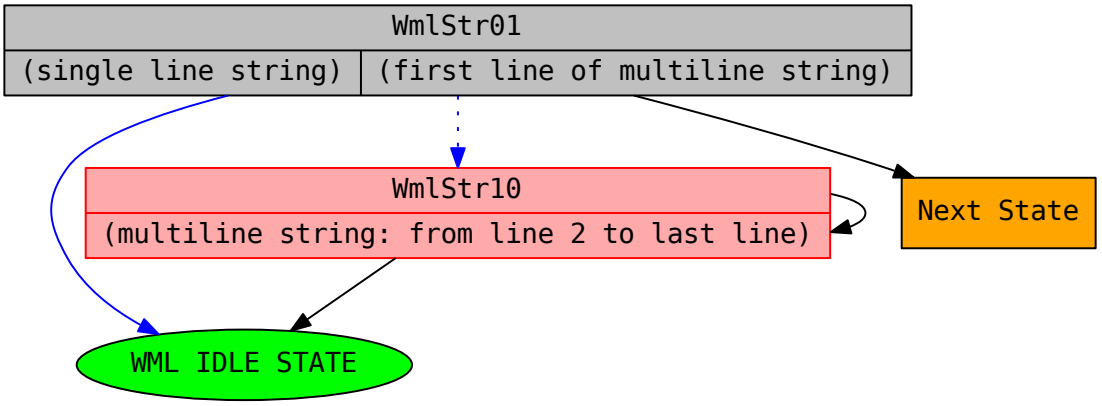
Capture String: WML States

WML accepts only one syntax:

```
_ "translatable_string"
```

Only two states, then, required to capture strings:

```
# ./pywmlx/state/wml_states:161
class WmlStr01:
    # ...
# ./pywmlx/state/wml_states:190
class WmlStr10:
```



More in details:

- **WmlStr01** (*'wml_str01'*): This state capture a single-line string and also capture the FIRST LINE of a WML multiline string.
  - If it is a single line string, the string will be stored in `pymlx.state.machine._pending_wmlstring`. (Change to *'wml_idle'* state).
  - If the closing quote `"` does not exist (*multiline string*) , then the matched string will be saved in `pymlx.state.machine._pending_wmlstring`. Following lines will be added to the pending string by the WmlStr10 State (change to *'wml_str10'* state)
- **WmlStr10** (*'wml_str10'*): All following lines of the multiline string will be added to pending string by this state until the closing quote `"` will be finded. This states recursively come back to itself, and, when the string ends, state will be changed again to *'wml_idle'*

Capture String: Lua States

Unlike WML, Lua accepts three different syntaxes:

```
"string: type 1"

'string: type 2'

[[string: type 3]]
```

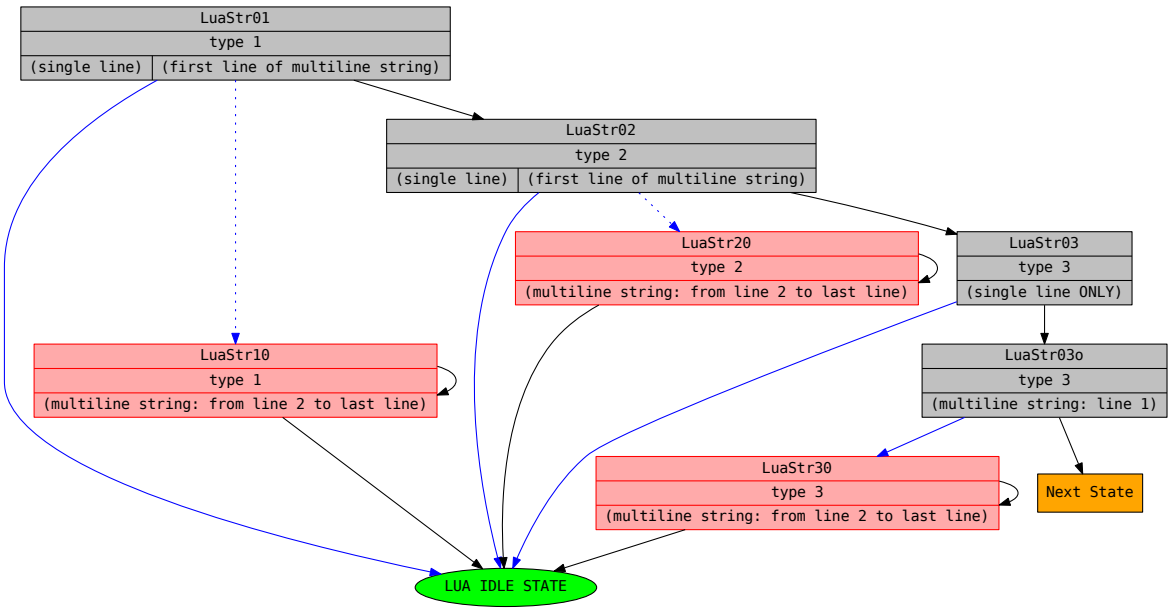
The third way (mostly suggested for multi-line lua strings) is even more flexible than showed in the sample code above, as you can type any number of equals symbols (from 0 to n) between the two brackets `[[` and `]]`

Note: In the example above, we wrote `[[string: type3]]`, since it is the most common way of defining a bracketed lua string, but we could also put any number of equals symobols between brackets.

For example, we could have printed `[==[string: type3]==]` placing the equal symbol two times. In that case, both opening and closing delimiter must use the same amount of equal symbols.

Coming back to wmlxgettext, we shoud now notice that all this flexibility allowed by the lua language (three ways to identify a string) means "*more states are required*". There are, infact, seven states this time:

```
# ./pywmlx/state/lua_states:71 (syntax "1": single-line or start multiline)
class LuaStr01:
    # ...
# ./pywmlx/state/lua_states:173 (syntax "1": multiline)
class LuaStr10:
    # ...
# ./pywmlx/state/lua_states:99 (syntax "2": single-line or start multiline)
class LuaStr02:
    # ...
# ./pywmlx/state/lua_states:193 (syntax "2": multiline)
class LuaStr20:
    # ...
# ./pywmlx/state/lua_states:127 (syntax "3": single-line ONLY)
class LuaStr03:
    # ...
# ./pywmlx/state/lua_states:149 (syntax "3": start multiline)
class LuaStr03o:
    # ...
# ./pywmlx/state/lua_states:211 (syntax "3": multiline [from line 2])
class LuaStr30:
    # ...
```



This time the flow chart is not so easy to understand at a first sight, so it requires a little explaination:

- **Boxes/Ellipses:**
  - green -> always-run states (green arrow rule applied)
  - orange -> used for "Next State", for a better look
  - red (LuaStr10 and LuaStr20): LuaStr10 and LuaStr20 are recursive standard states. They can go back to theirself, until the end of the multi-line string is matched (when the multi-line string ends, *'lua_idle'* state will be reached) (no arrow rule: all arrows are black)
  - red (LuaStr30): LuaStr30 is indeed an always-run state, but it acts like a recursive standard state.

The regular expression evaluation is moved in the run() function since the regexp rule is calculated on runtime. If the regexp doesn't match (current line of code does not end the multiline string) than LuaStr30 comes back to itself (recursive state). If the regexp does match, the multi-line string finished, and LuaStr30 goes to LuaIdleState.

  - grey -> standard states (black, blue and dotted blue arrow rules applied)
  - purple (ellipse) -> LuaStr30 can find (or not) the `]==]` symbol. Purple ellipses shows what happen if `]==]` was found and if `]==]` was NOT found (see where the black arrows will go).

- **Arrow rules (when applied):**
  - green -> LuaStr31 is an always-run state. LuaStr31 will **always** come back to LuaStr30 state
  - blue -> when the state finds what it is searching, go to the state pointed by blue arrow
  - blue (dotted) -> LuaStr01 and LuaStr02 regex rule can match both a single-line string AND the start of a multi-line string. If the a multi-line string matched, than go to the state pointed by the dotted blue arrow instead of the standard blue arrow
  - black -> When the regex rule of the state fails (the state does not find what it is searching). [except for red boxes]

About storing the last *Lua function name*

Unlike the WML states, there isn't any Lua state that captures lua infos. The only extra info that could be auto-cached inside a lua code is the name of the last function opened / defined.

This kind of search required to use a specific regexp search, using `re.search` instead of `re.match`. Unlike all other searches, infact, we need to capture function name at any point of the line we are parsing, or the regexp will not work properly.

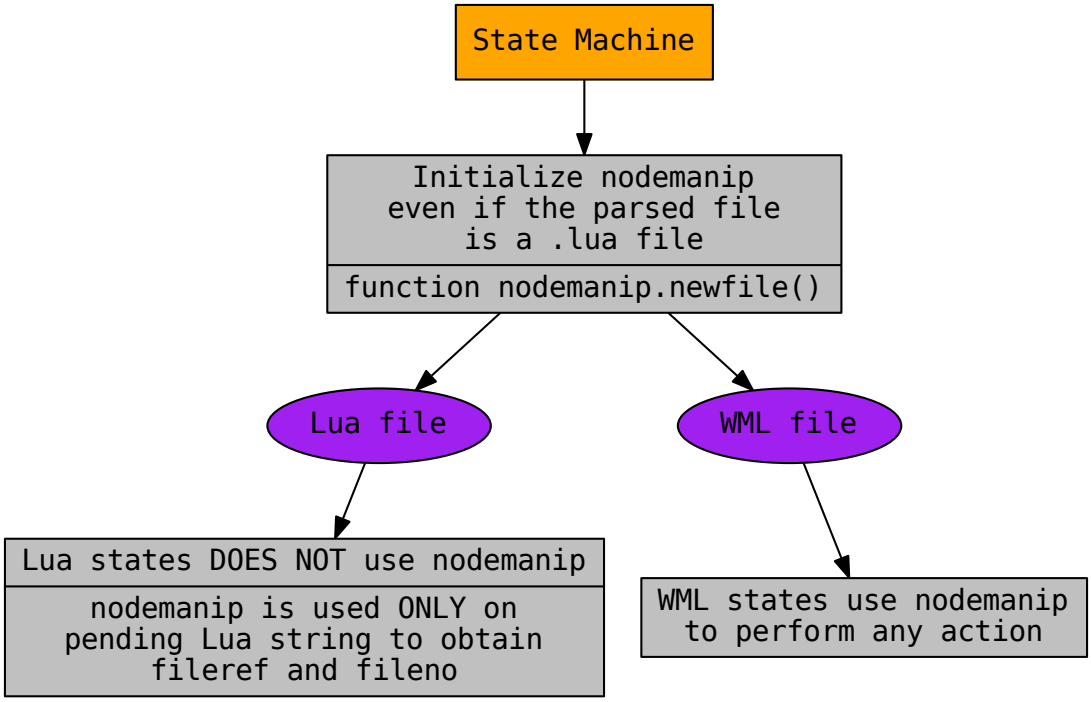
But, as explained at the beginning of this page, the state machine relies on `re.match` (best performance) to verify the regexp rule of every state. For this reason, `LuaFinalState` searches by itself if there is a function name somewhere, and, if so, stores the value into `pywmlx.state.machine._pending_luafuncname`.

2.4.3 State Machine and nodemanip

The previous chapter (*Introducing WML and Lua parser*) explained a lot of things, and expecially:

- how WML nodes are stored in memory
- how `nodemanip` module manage WML nodes (See: *The nodemanip module*).

But, an important thing was omissed: **nodemanip** is used by the statemachine.



When wmlxgettext `import pywmlx`, nodemanip module is **not** loaded in `pywmlx` namespace: `nodemanip` is only internally used by state machine (module `./pywmlx/state/machine.py`).

2.5 The last step: writing the .po file

When the dictionary of `PoCommentedString` object is done (the end of the last file is reached and nothing is left unparsed) it is the time to write all the dictionary into an actual .po file:

```
# ./wmlxgettext:164
outfile = None
if args.outfile is None:
    outfile = sys.stdout
else:
    outfile_name = os.path.realpath(os.path.normpath(args.outfile))
    outfile = open(outfile_name, 'w')
pkgversion = args.package_version + '\\n"'
print('msgid ""\nmsgstr ""', file=outfile)
print('"Project-Id-Version:', pkgversion, file=outfile)
print('"Report-Msgid-Bugs-To: http://bugs.wesnoth.org/\\n"', file=outfile)
now = datetime.now()
cdate = str(now.year) + '-'
if now.month < 10:
    cdate = cdate + '0'
cdate = cdate + str(now.month) + '-'
if now.day < 10:
    cdate = cdate + '0'
```

(continues on next page)

(continued from previous page)

```
cdate = cdate + str(now.day) + ' '
if now.hour < 10:
    cdate = cdate + '0'
cdate = cdate + str(now.hour) + ':'
if now.minute < 10:
    cdate = cdate + '0'
cdate = cdate + str(now.minute) + strftime("%z") + '\\n"'

print('"POT-Creation-Date:', cdate, file=outfile)
print('"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\\n"', file=outfile)
print('"Last-Translator: FULL NAME <EMAIL@ADDRESS>\\n"', file=outfile)
print('"Language-Team: LANGUAGE <LL@li.org>\\n"', file=outfile)
print('"MIME-Version: 1.0\\n"', file=outfile)
print('"Content-Type: text/plain; charset=UTF-8\\n"', file=outfile)
print('"Content-Transfer-Encoding: 8bit\\n"\n', file=outfile)
```

This part of code (into `wmlxgettext` main script file) writes down the .po header informations:

```
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: http://bugs.wesnoth.org/\n"
"POT-Creation-Date: 2016-02-19 17:59+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

After writing the header, it is the time to write the translatable strings:

```
# ./pywmlx/wmlxgettext:196
for posentence in sorted(sentlist.values(), key=lambda x: x.orderid):
    posentence.write(outfile, args.fuzzy)
    print('', file=outfile)
```

All `PoCommentedString` objects contained in dictionary will be written in the correct order (thank of `sorted()` that sorts `PoCommentedString` object by `orderid` value)

Every `PoCommentedString` object will be then written in .po file, calling the `PoCommentedString.write()` function.

The `PoCommentedString.write()` function will:

- write `wmlinfos` and `addedinfos` on `PoCommentedString` object, one by one, as `#. <message to translator>`
- write `finfos` on `PoCommentedString` object, one by one, as `#: path/to/file:x` infos
- put the *fuzzy flag* if `--fuzzy` option was used in wmlxgettext command line
- write translatable string into `msgid "..."` parameter in the proper way
- add an empty line `msgstr ""` (where the translator will put the translation into another language).

Now it is the time for the very last explaination:

```
# ./pywmlx/wmlxgettext:199
if args.outfile is not None:
    outfile.close()
```

If `args.outfile` is `None`, then the option `-o output-file-name` was not used (output should be written in `stdout` like in wmlxgettext 1.0, and it can be redirected to a file)

If `args.outfile` is **not** `None`, then an output file is directly created by wmlxgettext itself (and that file buffer must be closed).

2.6 Deep explaination of all regular expressions

This part of the source documentation is a kind of an "appendix", where all regular expressions used in source code will be explained deeply

2.6.1 Regexes used on WML States

WML IDLE State

```
m = re.match(r'\s*$', xline)
```

If the line is actually empty (only contains tabs/spaces) it will be consumed immediately. It is equal to that regular expression:

```
^\s*$
```

WmlCheckdomState

```
self.regex = re.compile(r'\s*#textdomain\s+(\S+)', re.I)
```

this is equal to that (case insensitive) regex:

```
^\s*#textdomain\s+(\S+)
```

At the **start of the string** will search for:

- spaces/tabs (from 0 to n)
- the character `#`.
- the word `textdomain` followed by one or more spaces
- one or more NO-SPACE characters, captured into group 1

WmlCheckpoState

```
rx = r'\s*#\s+(wmlxgettext|po-override|po):\s+(.+)'
self.regex = re.compile(rx, re.I)
```

wich is equal to that (case insensitive) regex:

```
^\s*#\s+(wmlxgettext|po-override|po):\s+(.+)
```

At the **start of the string** will search for:

- spaces/tabs (from 0 to n)
- the character #
- one or more space before an actual word
- one of those words: *wmlxgettext, po-override* or *po* captured into group 1.
- followed by the character : and one or more spaces/tabs.
- followed by any number of any characters (at least 1) captured on group 2.

WmlCommentState

```
self.regex = re.compile(r'\s*#.+')
```

At the **start of the string** will search for:

- spaces/tabs (from 0 to n)
- the character # followed by any character

WmlTagState

Note: Special Thanks to:

- **Soliton**
  - for pointing me that a tag name could, in theory, a number
  - for having a very nice idea about how to distinguish a tag from an array index (see the regexp explaination)
- **celticminstrel**
  - for providing me a good regexp rule, that allowed me to write down the regexp used in this state

---

```
rx = r'\s*(?:[^"]+\(\s*)?\s*\[\s*([\/+-]?)\s*([A-Za-z0-9_]+)\s*\]'
self.regex = re.compile(rx)
```

Before explaining what the regex searches, we need to explain why the regexp was written in this way.

We must take mind that a WML tag (we now focus on open tag, but the discussion is the same also on close tags) can appear in two different ways; this is the first one:

```
# first way: tagname can be defined at the start of the line
  [tagname]
```

In this case, the WML line we are parsing may have an arbitrary number of spaces (or tabs) before the tagname, but nothing else must appear before the [tagname]. This is the most common case where a tag is defined, but it is not the only one; a tag can be added also in the body of a WML macro call as a part of WML code passed as parameter to the macro.

So why a WML tag can also apper inside the body of a macro call, like showed in this example:

```
{MACRO ([foo]
            bar = "baz"
        [/foo])}
```

So, wmlxgettext had to face two corner problems:

- it should record the `[foo]` open tag inside the macro call, or it will return an error when closing `[/foo]` tag will be found
- it should, however avoid to collect array indexes, thinking they are tags, for example:

```
# [$i], here, is not a tag, but it is an index value of the array my_array
value = my_array[$i]
```

So... how to distinguish tag from an array using a regexp? Well... a tagname, when placed inside a WML macro call, should be ALWAYS immediately preceded by `(`; nothing else than spaces can be putted before the parenthesis and the tag definition.

After all those explainations we have almost all the informations required to understand why the regexp used on WmlTagState is:

```
^\s*(?:[^"]+\(\s*)?\[\s*([\/+-]?)\s*([A-Za-z0-9_]+)\s*\]
```

As usual, at the start of the string, an arbitrary number of spaces or tabs (`^\s*`) can be found.

After that the regexp will consider two different scenarios:

- first scenario: [tagname] is defined inside a macro call
- second scenario: [tagname] stays alone (most common case)

On the fist scenario, the [tagname] is contained into a MACRO CALL, so we must verify that the [tagname] definition immediately follows a parenthesis `(`, except for spaces or tabs that can separates `(` and `[tagname]`:

```
(?:[^"]+\(\s*)?
```

This check is performed by the non-capturing group written above, wich can occur one-time only (when tagname is contained in the macro definition) **or** it can occur **zero** times (when the tagname stays alone in the line, second scenario).

The non-capturing group will search for the last opening parenthesis encountered (and following spaces) that satisfies the remaining part of the regexp (explained later) wich search for [tagname].

This is, in particular, made by the second part of the non-capturing group:

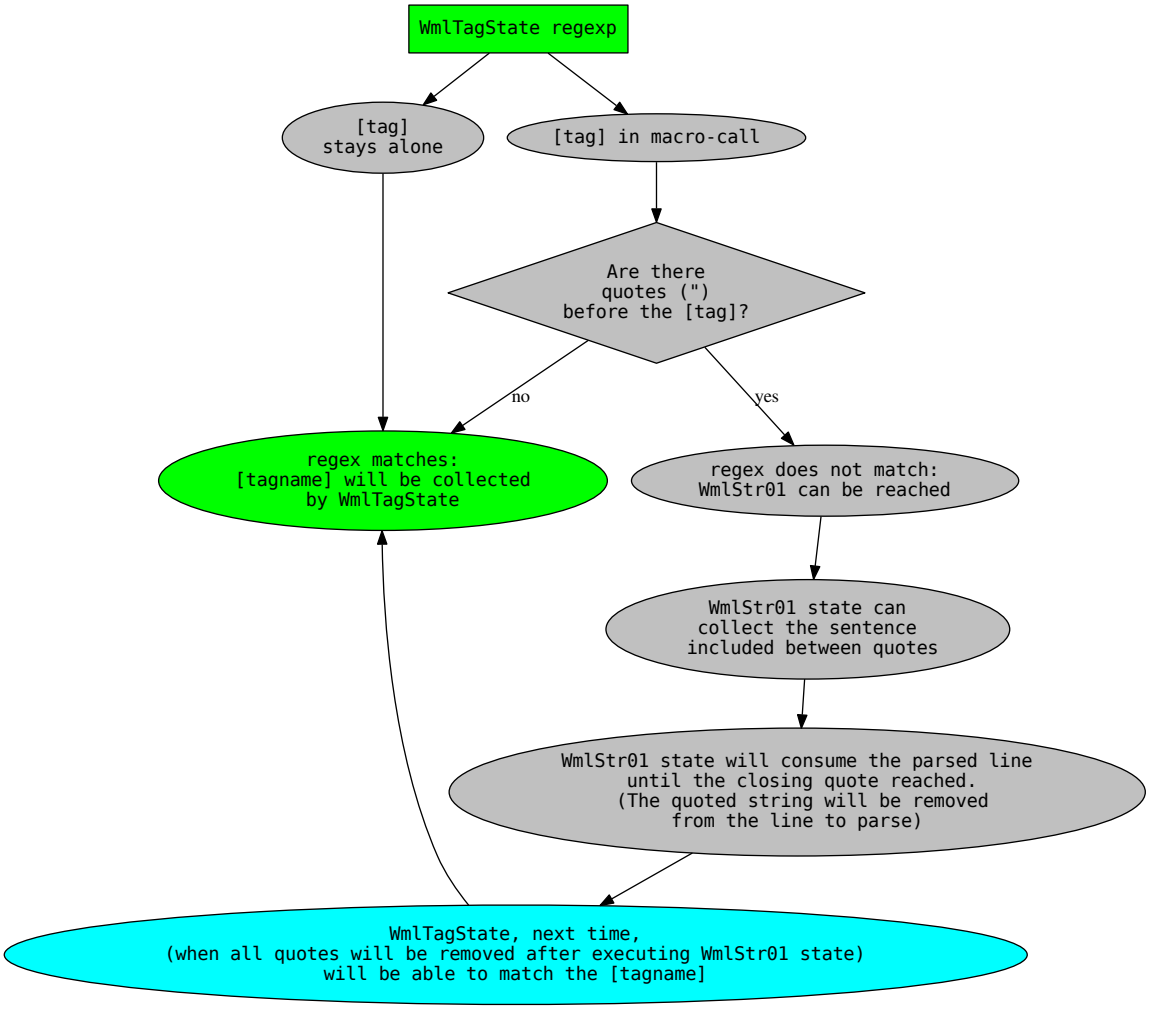
```
\(\s*
```

But the non-capturing group will verify that no quote symbols (`"`) were found in the meantime:

```
[^"]+
```

The reason of this exclusion is related to the wmlxgettext state machine design: the WmlTagState, infact, is evaluated before the WmlStr01 state (wich will search WML strings, translatable or not).

Wich means: if we allowed WmlTagState to match a line containing a quotation, we would let WmlTagState to consume all the matched line, including the WML string, wich will never been evalated by WmlStr01 State. But we don't want that this event could happen.



So, coming back to the regexp:

```
^\s*(?:[^"]+\(\s*)?\[\s*([\/+-]?)\s*([A-Za-z0-9_]+)\s*\]
```

We said:

- `^\s*` will search for arbitrary number of spaces (or tabs) at the start of the line
- `(?:[^"]+\(\s*)?` is the **zero or one** time non-campturing group that verifies if the tag is included inside a macro call. Wmlxgettext will search for a `[tagname]` wich is directly preceded by an opening parenthesis and an arbitrary number of spaces (or tabs). In the meantime it will verify that no quotations symbols (`"`) can be found in the meantime. If a quotation symbol will be found, the regexp will be fail, so the WmlStr01 state can do its work (see the flow chart here above).
- `\[\s*([\/+-]?)\s*([A-Za-z0-9_]+)\s*\]` is the final part of the regexp (valid both for tags placed alone and for tags placed inside a macro call) that actually identify the tag. It will discussed here now.

The final part of the regular expression will search for `[tagname]`, `[/tagname]`, `[+tagname]` or `[-tagname]` where any number of spaces can be placed between `[`, tagname and `]`.

If +, − or / symbol is used, any number of spaces can be placed between the symbol, the `[` and the tagname.

The regular expression, in this final part will also do those tasks:

- it will store, on group(1), the symbol +, − or /. If no symbol will be used, the group(1) will be an empty string.
- it will store, on group(2), the tagname. Characters allowed are only letters, numbers, or underscore, so why the expression `([A-Za-z0-9_]+)` is used there (note that tagname must contain at least one character, this is why the + quantifier was used).

Note: On group(1), as we said, we can find an empty string (no symbol used) or one of those symbols: +, − and /.

- if / is found, then the tag is a closing tag
- if + is found, the tag is considered like a normal open tag, ignoring the + symbol.
- if − is found, the tag is treated like [+tag].

Note that the `[-tag]` is not currently supported in WML code. Wmlxgettext included the rule for the − symbol if, in a future, also the [-tag] feature will ever included (thinking the chance of doing the opposite thing that is done by the [+tag]).

WmlGetinfState

```
rx = ( r'\s*(speaker|id|role|description|condition|type|race)' +
       r'\s*=\s*(.*)' )
self.regex = re.compile(rx, re.I)
```

This **case-insensitive** regex will be search, **start of the string**, for:

- spaces/tabs (from 0 to n)
- one of the following words: *speaker, id, role, description, condition, type* or *race*. The word will be captured into group 1.
- spaces/tabs (from 0 to n)
- the = character
- spaces/tabs (from 0 to n)
- any number of any character, captured by group 2. (this will be the value assigned to the parameter captured by group 1).

Note: The `WmlGetinfState` and the state `WmlStr01` could generate a bug, without the proper cautions. This is the reason why you can find this code into `WmlGetinfState`

```
if '"' in match.group(2):
    _nextstate = 'wml_str01'
    pywmlx.state.machine._pending_winfotype = match.group(1)
```

If a `"` sign was found in group 2, it means that the value assigned to the parameter (for example, `name="something"`) is a quoted string. This string must be managed then by the state `WmlStr01`. State Machine will remember that there is a pending wml info with quoted string. the `winfotype` will store only the parameter at the moment, waiting for `WmlStr01` (that will process the quoted string)

State WmlStr01

This is the state wich will capture a wml string type 1 (“quoted string”)

```
rx = r'(?:[^"]*?)\s*(_?)\s*"((?:""|[^"])*)("?)'
self.regex = re.compile(rx)
```

the regexp used here is a bit complex, so it will be atomized:

```
^(?:[^"]*?)
```

without creating group (`(?:` creates a non-capturing group), any number of characters **different than** `"` will be found. But the search will be less greedy than possible (thank the very last `?` putted after `*`). The “*less greedy than possible*” rule is necessary, othewhise the following rule will be ignored:

```
\s*(_?)\s*"
```

we need, infact, to know if a string is translatable or not. We must see if a _ sign was found before opening the quote. But the _ sign is different than `"` sign, so if the previous rule was greedy, the regexp could never capture on group 1 the _ sign. Instead, since the non-capturing group `(?:[^"]*?)` is “*less greedy than possible*” it will stops as soon the following rule `\s*(_?)\s*"` will be true.

Since the rule `\s*(_?)\s*"` will check:

- spaces/tabs (from 0 to n)
- **zero or one** _ sign, captured on group 1, followed by spaces/tabs (from 0 to n)
- followed by `"` sign

this means that the regexp, until now:

- is true even if something was found before _ `"translatable string"`
- will see if _ is used (group 1). Group 1 will be _ if the _ will be found, or it will be an empty string if the _ will not be found (string is not translatable)
- it will check for opening quote `"` where the string actually starts.

Finally, the regexp continues with:

```
((?:""|[^"])*)("?)
```

This part of the regexp must be explained a bit. A WML string can contain two following `"` signs if you want to use the `"` character inside your string (for example, using a `"` sign in a message). For this reason, if you find `""` into a WML string, the string is not yet finised.

So, this part of the regexp:

- create a new group 2 (with the **outer** parenthesis on `((?:""|[^"])*)`)
- that group 2 will capture any number of the things captured by the **inner** parenthesis, wich doesn't create any additional groups (thank of the starting `?:`).
- the “things” that can be captured on group 2, so, can be:
  - either “”
  - **or** any character **different than** `"`
- finally checks if there is the enclosing `"` sign and capture it to group 3.

This is how this complex regexp works.

Note: it is the time to remember what the regexp capturing groups:

- group 1 -> can be _ or an empty string (to understand it the string is translatable or not).
- group 2 -> it is the text
- group 3 -> can be " or an empty string. If it is an empty string, (closing " sign not found) than the string is multi-line.

State WmlStr02

This is the state wich will capture a **translatable** wml string type 2 (_ <<translatable capitalized string>>)

```
rx = r'[^"]*_\s*<<(?:(.*?)>>|(.*))'
self.regex = re.compile(rx)
```

WmlStr02 is evalued after WmlCommentState (so it is evalued **before** WmlStr01):

```
[^"]*_\s*<<
```

Unlike before, WmlStr02 will match ONLY if the string is translatable (so non-translatable <<string>> will be ignored by regex). The regex will also mach only if **no** quotes found before the underscore marker followed by the << marker.

We said that WmlStr02 is evalued **before** WmlStr01, and that is the reason why **no** quote should be found before WmlStr02 (the WmlStr01 must be evalued and not skipped; so the WmlStr02 regex will fail, and the WmlStr01 state can be reached to collect the WmlStr01).

```
(?:(.*?)>>|(.*))
```

The second (and last) part of the regex is a non-capturing group wich contains two alternatives:

- `(.*?)>>` the first alternative matches if the close marker >> is found (single line translatable string). The capture ends when the **first** >> occurrence is found (non-greedy capture). Text is captured on group 1.
- `(.*)` the second alternative matches all the text until the end of the line (multi-line translatable string). Text is captured on group 2.

State WmlStr10

This is the state wich will capture multi-line wml "quoted" string (type 1) from line 2 to the end

```
self.regex = re.compile(r'((?:""|[^"])*)("?)')
```

The regexp is musch more simplier than the one used by the state `WmlStr01` even if it works in a very similar way.

The basic idea of this regexp is: <<*we are parsing a multi line string and this is NOT the first line of the string, so the starting part of the file line must be contained into the string until the ending ‘‘”‘‘ will be found*>>.

It will save, on group 1 and group 2, what the regexp used by WmlStr01 capture on group 2 and group 3.

State WmlStr20

This is the state wich will capture multi-line wml <<capitalized>> string (type 2) from line 2 to the end

WmlStr20 is a very particular state: it is structured as an always-run state, but it works like a standard state.

There is a regex inside the `run` function wich is very simple:

```
(.*?)>>
```

This is a solution that allows WmlStr02 to stay there until the >> end marker will be found somewhere. Infact:

- If the regex fails, WmlStr20 will recursively change to itself (it stays to WmlStr20)
- If the regex matches, WmlStr20 will capture the text into group(1) and then the state will be changed to `wml_idle`

WmlGoluaState

```
self.regex = re.compile(r'.*?<<\s*')
```

It will be check, from the start part of the string, any number of any character (less greedy then possible) until << found (followed by any number of spaces/tabs - from 0 to n).

If the regexp will mach, the State will consume the line until the last space of the << symbol, and than switch to `lua_idle` state (parse Lua language).

2.6.2 Regexes used on Lua States

Unlike WML states, we will not explain **all** the regexp used, since most of them are **very similar** to the ones used on WML states

LuaCheckdomState

```
rx = (   r'\s*(local)?\s+_\s*=\s*wesnoth\s*\.\s*textdomain\s*'
       r'''(?:\(\s*)?(["'])(.*?)\2''')
self.regex = re.compile(rx, re.I)
```

The regular expression used by LuaCheckdomState is very long, and it is very different from the one used by WmlCheckdomState. Changing the current domain in lua code, infact, requires a very different syntax:

```
-- after executing the following line, the current domain
-- will be changed to: wesnoth-xyz
local _ = wesnoth.textdomain('wesnoth-xyz')
```

It is now the time to explain deeply the regexp used by LuaCheckdomState:

```
^\s*(local)?\s+_\s*=\s*wesnoth\s*\.\s*textdomain\s*(?:\(\s*)?(["'])(.*?)\2
```

The regexp can be dived in this way:

- `^\s*` –> Arbitrary number of spaces or tabs at the start of the line.
- `(local\s+)?` –> Optional `local` keyword. It is captured (if exists) in group(1). If `local` keyword is not used and the `--warnall` command line option is used, than a warning message is displayed.

- `_\s*=\s*` –> the underscore symbol (_) followed by equal (=). Any number of spaces or tab can be placed between underscore and equal; any mymber of spaces or tab can be also placed after the equal symbol.
- `wesnoth\s*\.\s*textdomain\s*` –> look for `wesnoth.textdomain`. Any number of spaces can be placed before and after the point symbol that divides `wesnoth` and `textdomain` ; any number of spaces can be placed after the `textdomain` word.
- `(?:\(\s*)?` –> This is a very important part of the regexp. This non-capturing group will ensure that the regexp will match when **zero or one** open paranthesis will follow after `wesnoth.textdomain`. The open parenthesis is, infact, optional.
- `(["'])` –> Then a single or a double quote is expected, and it will captured on group(2)
- `(.*?)` –> The actual textdomain will be captured on group(3)
- `\2` –> the closing quote (what it was captured on group2, wich opened the quote, must match be the same one that closes the quote)

Note: Special thanks to celticminstrel for providing me this regexp.

Lua *"Comment"* States

`LuaCheckpoState` and `LuaCommentState` use regexpes very similar to the ones used on *WmlCheckpoState* and *WmlCommentState*.

Here the differences:

- You can also use `-- po:` and `-- po-override:` or you can use `-- # po:` and `-- # po-override:` (both forms are allowed).
- `# wmlxgettext:` is **not** supported on lua code (it is useless)
- lua comment starts with `--` and not with #

LuaStr01 and LuaStr02 States

We will display the LuaStr01 python code

```
rx = r'''(?:[^["']*?)(_?)\s*"((?:\\"|[^"])*)("?)'''
self.regex = re.compile(rx)
```

wich is equal to the following regexp:

```
^(?:[^["']*?)(_?)\s*"((?:\\"|[^"])*)("?)
```

The regexp used by LuaStr02 is more or less the same, infact it is equal to the following regexp:

```
^(?:[^["']*?)(_?)\s*'((?:\\'|[^'])*)('?)
```

The basic logic of those regexp is more or less the same as the one used by *State WmlStr01*.

As the regexp used by *State WmlStr01*, it can be divided in three parts:

- *things* before the strings starts
- check if the string is translatable, searching for _ sign rigtly before the string starts (followed by any number of spaces-tabs). (group 1 = _ or empty string)

- check for start quote (" for LuaStr01, ' for LuaStr02).
- check for text (group 2)
- check for quotation end (group 3) (if empty, is a multiline string).

The actual difference from the regexp used by *State WmlStr01* is the **first** part of the regexp rule:

```
(?:[^["']*?)
```

Instead of searching of all characters different than only the " symbol, this regex will search all characters that will be **neither** ", nor ', nor [.

This will avoid conflicts from the three possible syntaxes and it will ensure that, if any of the regexp match, it will really match the first string, avoiding that a lua string will be skipped.

Another difference is that the "non enclosing quote" is not "" like WML, but it is escaped in a different way (\" or \'), this is why the rule is a bit different also in the third part of the regexp rule.

LuaStr10 and LuaStr20 States

The basic idea is the same as the one used by *State WmlStr01*.

(See also: *State WmlStr01* and *LuaStr01 and LuaStr02 States*).

LuaStr03 State

LuaStr03 regexp can is equal to the following regexp rule:

```
^(?:[^["']*?)(_?)\s*\[(=*)\[(.*?)]\2]
```

The first part of regexp (^(?:[^["']*?)) is already explained in *LuaStr01 and LuaStr02 States*.

The second part of regexp((_?)\s*) captures _ on group 1 and collect any following spaces/tabs (without storing them in groups).

The third part of regexp (\[(=*)\[) captures all equal symbols placed between the two brackets and store them into group 2.

The fourth part of regexp ((.*?)) captures all characters contained between the lua bracketed string delimiters (*ending delimiter is defined by the last part of the regexp*). It captures the less charcaters than possible until the end delimiter found

The last part of regexp (]\2]) will search the right lua bracketed string end delimiter, checking how many equals symbols were captured on group 2 (\2 will search exactly what group 2 matched). So, if the group 2 is an empty string, than]] will be the end delimiter searched by regexp. If the group 2 is === (3 equals symbols) than the end delimiter will be]===]... and so on.

Note: This regexp, unlike the one used on LuaStr01 and LuaStr02, does not match at all if the right end-delimiter will be not found in the parsed line. This is why lua bracketed strings (lua string type 3) require another state that explicitly tells when the lua string type 3 is multiline. And this is the rule defined on LuaStr03o, explained in the next subparagraph.

LuaStr03o State

LuaStr03o State will match when the beginning of a lua multiline bracketed string is found:

```
^(?:[^["']*?)(_?)\s*\[(=*)\[(.*)
```

The state LuaStr03o will capture:

- on group 1: the _ symbol (if is used)
- on group 2: how many equal symbols where placed in the *starting string delimiter* (for example the delimiter `[=[` will contain one equal symbol between the two brackets)
- on group 3: the text of the first line of the string. This time the group 3 use **greedy** rule, capturing all following characters. This is why, this time, the regexp will be **True** (will match) even if nothing follows the `[=[` marker (multiline string).

Note: LuaStr03o, when creating the pending string (PendingLuaString object on state machine), stores the amount of equals signs in the PendingLuaString.numequals variable, wich will be used by LuaState30 to calculate (on runtime) wich regexp should be actually used.

State LuaStr30

The LuaStr30 is a very particular state, wich is structured as an always-run state, but it works like a standard state.

The regexp definition, infact, is not placed (as usual) in the State.regexp parameter, defined in the __init__ function. This becouse all states are stored in the state machine during the setup phase, before starting to parse WML and Lua files. Wich means that all State.regexp values can be defined only on the setup phase itself and they cannot change anymore.

But, this time, we require to use a regexp rule that search exactly wich is the end delimiter for that one lua bracketed multiline string started on the previous LuaStr03o state.

This why the regexp is defined directly in the run() function, wich explicitly performs all actions usually done by statemachine when evaluating a State.regexp.

This is the regexp that will be evaluated in the run() function:

```
^(.*?)]={n}]
```

where `n` is the exact number of equals symbols stored in the PendingLuaString.numequals variable by LuaStr03o.

So, for example, if LuaStr03o.regex previously matched == on group 2 (wich means that `[==[` was the opening delimiter used), then the regexp searched by the run function will be:

```
^(.*?)]={2}]
```

Now it is the time to actually explain the regexp. We will focus the explaination around this last concrete example (end delimiter must have exactly two equal symbols between close brackets). So why, from now on, we will explain the regexp:

```
^(.*?)]={2}]
```

This regexp will match if the line contains somewhere the `]==]` delimiter. the final part of the regexp (`]={2}]`), infact, means:

- litteral `]`

- followed by = (two times)
- followed by]

If the delimiter]==] will be found, the regexp will match, the last part of the string will be stored on group 1, than it will be added to the pending string. LuaStr30 will go to LuaIdleState (parsed line will be not completely consumed. Only what it will be matched will be removed from the parsed line.

If the delimiter]==] will not be found, than the regexp will not mach. LuaStr30 will store all the parsed line into the pending lua string and consume it at all, so the statemachine will be able to read the next line of code. LuaStr30 will come back again to itself (it acts like a recursive state, in a very similar way like the LuaStr10 and LuaStr20 states).

Note: The first part of the regexp `(.*?)` capture all characters using the **less greedy than possible** rule, with the same effects explained on *State WmlStr01* (first part of the regexp where the *less greedy* rule was used).

LuaFinalState

Lua Final States checks if the current parsing line contains a function name:

```
rx_str = ( r'function\s+([a-zA-Z0-9_.]+)|' +
                r'([a-zA-Z0-9_.]+)\s*=\s*function'
         )
rx = re.compile(rx_str, re.I)
m = re.search(rx, xline)
```

So it use `re.search` and not `re.match` as usual. This mean that we don't have a sort of an implicit caret symbol at the start of the regexp rule, so the resulting regexp rule is:

```
function\s+([a-zA-Z0-9_.]+)|([a-zA-Z0-9_.]+)\s*=\s*function
```

Note: the regexp showed above is **case insensitive** (option `re.I` used on `re.compile` function).

---

the regex will search:

- **function** <*name_of_function*>: where <*name_of_function*> will be stored on group 1.

or it will search:

- <*name_of_function*> = **function**: this time <*name_of_function*> will be stored on group 2.

2.6.3 "*Escape*" regexp rules

Translatable strings will be "reformatted" two times. The first time when they will be stored from pending string to a `PoCommentedString` (or to a `WmlNodeSentece`) object.

```
# ./pywmlx/state/machine.py (class PendingLuaString, function store)
if self.luatype == 'luastr2':
    self.luastring = re.sub(r"\'", r"'", self.luastring)
self.luastring = re.sub(r'(?<!\\)"', r'\"', self.luastring)

# ---------------------------------------

# ./pywmlx/state/machine.py (class PendingWmlString, function store)
self.wmlstring = re.sub('""', '"', self.wmlstring)
```

Those part of code will be replace the *escaped quote* found in that kind of string (`""` on WML and `\"` on Lua type 1 for symbol `"`; `\'` on Lua type 2 for symbol `'`).

Those escape code will be replaced in those way:

- `""` found on WML will be replaced by `\"`
- `"`, if not preceded by `\` will be replaced by `\"`, on lua string
- `\'`, found on Lua type 2, will be replaced by `'`.

This becouse, in the final .po file the quote string `"` must be escaped by `\`, so the right escape code is `\"`. The `'` symbol, instead, don't require any escape.

So it's the time to explain the regexp used on lua to verify if a `"` symbol is not preceded by `\`:

```
(?<!\\)"
```

This is the regexp rule used by the last `re.sub` used by the function `PendingLuaString.store()`.

the `(?<!\\)` is a **negative look-before** rule. So the regex will match if the `"` is found, but if the previous character is not `\` infact:

- `(?<! )` identify the negative look-before
- `\\` checks for the litteral character `\`.

We said that the translatable string is "reformatted" two times.

- the first time, when a new `PoCommentedString` or `WmlNodeSentece` object is stored in memory.
- the second time when every single `PoCommentedString` object contained in the dictionary will be written in the .po file, rightly before actually writing it.

On this last step the sentence will be translated from:

```
this is the \"sentence\" before second formatting
```

to:

```
"this is the \"sentence\" before second formatting"
```

If the string is multiline, for example:

```
this is
a \"multiline\" string
stored here
```

they will be formatted to:

```
""
"this is\n"
"a \"multiline\" string\n"
"stored here"
```

It is possible to notice that, on multiline string, the new "formatting" will create a first line with only `""`. It is not an error: it is expected since, if the string is multiline, it is expected that `""` will follow `msgid` on the first line.

All other lines (except the very last one) will end with `\n` (new line code).

All lines (included the very last one) will be enclosed in quotes (`"`).

Wmlxgettext is a python3.x tool that create a gettext translation file (.po file) for a wesnoth campaign / era / add-on.

It will parse a list .cfg and .lua files and stores the info required to create a nice .po file, than create the files.

From now on, the (old) perl script will be called "wmlxgettext 1.0", while the new python3 script will be called "wmlxgettext 2.x"

CHAPTER 3

End-User documentation

The *Documentation for End-Users* eplains how to **use** the wmlxgettext tool. It will explain what you can use into your .cfg (WML) and .lua files and it will explain how to invoke wmlxgettext tool.

CHAPTER 4

Source Code Documentation

The *Source Code Documentation* is directed to those ones that wants to contribute to wmlxgettext development, or to those ones that wants to modify/fork this tool for his own purposes.

The *Source Code Documentation* is **not useful** if you only need to learn how to use wmlxgettext.

CHAPTER 5

Special Thanks To:

- Elvish Hunter (wesnoth developer) for his very precious help.
- |Wolf| (python Italia) for his deep knowledge on python and his precious help.
- celticminstrel (wesnoth developer) for explaining me how to use the original perl wmlxgettext tool and for helping me to improve the script.
- loonycyborg: for testing the script and reporting bugs.

Note: If you can find something crap in wmlxgettext code, it is **only** my fault. People listed above are not responsible :)

Nobun