NAME

efabulor - read text files through **espeak** in a controlled manner

SYNOPSIS

```
[python3] efabulor.py [options] [file]
```

DESCRIPTION

efabulor is a wrapper around **espeak**, a freely-available speech-synthesis engine. Its basic functionality is to read a (plain) text file, split it in sentences, and send them to **espeak** to be read aloud. The reading process can be controlled with the keyboard (go back and forth, stop, pause and resume the reading, perform a search, go to a specific line, etc.). Files other than plain text can also be read by using conversion commands within the program (see below).

efabulor is intended as a quality-assurance companion tool for people working in text creation. You can have the program read a file while you edit it in your preferred word-processing application, and receive auditory feedback of changes to the underlying file as they happen. A specific use scenario involves translators, who need to check their translated text (or 'target text') against the source text for mistranslations, omissions, etc. This usually involves reading back and forth between both languages, a process in which minor errors are easy to overlook. With **efabulor**, you can listen to the translated text while you read the source text along (or the other way around). It's yet another way to quality-check your translations.

Transformation rules can be applied to the input text, e.g., to add newline characters at selected points in order to control the splitting of sentences. They can also be used to apply any kind of preprocessing to the text that will be read by **efabulor**. Transformation rules are defined in a simple language which is interpreted by a companion tool called **efabtrans**. (You don't need to call **efabtrans** yourself; **efabulor** will take care of that.) There is also the option of running arbitrary external preprocessing filters (instead of builtin transformation rules). Preprocessing filters and transformation rules can be applied together.

Substitution rules can be applied to the input text as it is being read, e.g., if you want to improve or tweak the results of the text-to-speech algorithm. Substitution rules are defined in a simple language similar to the one used to define transformation rules, and they are interpreted directly by **efabulor**. The difference between transformation rules and substitution rules is that transformation rules are applied to the text as a whole before loading it as input for the program; substitution rules are applied on a sentence-by-sentence basis and only when they are actually needed (when **efabulor** is about to read a given sentence). Another difference involves 'tracking' of changes (see CHANGE TRACKING).

The output of any command can be used as input for **efabulor**. Using this feature with an external conversion program, you can define a text extraction command to read non-

plain text files. A companion tool called **efabconv.py** is provided to simplify the conversion of different file formats by assigning preconfigured commands to different file extensions and/or mimetypes. (**efabulor** will take care of calling **efabconv** when needed.)

Please note that currently **efabulor** is a command-line tool and there are no plans to provide a graphical interface for the time being. (It might be added in a future version.)

TARGET PLATFORMS

efabulor was developed and tested under Linux (Ubuntu and Debian). It might run in Mac OS X, BSD, etc., but it's not guaranteed to do so.

It has been adapted to run (but not thoroughly tested) in Windows. Performance in Windows may be lower.

Note: beginning with Windows 10, you can run Linux applications within Windows by using the new Windows Subsystem for Linux (WSL). There is also the option to install a Linux distribution within a virtual machine environment. Either option can help you solve any problem you find when trying to use **efabulor** directly under Windows.

LIMITATIONS

Runtime configuration

Currently, many runtime options can only be configured through arguments in the command line; the ability to put such options in a runtime configuration file is missing. It might be added in a future version.

File size

efabulor is not intended to be used with very large files (longer than a few hundred pages). There are ways to circumvent this limitation, but they require some scripting outside **efabulor**. Support for very large files might be included in future versions if the users require it, but it is not a priority right now.

Substitution and transformation rules

You can specify as many substitution rules and transformation rules files (see SPLITTING AND TRANSFORMATION RULES and SUBSTITUTION RULES) as you wish, but only the first nine of either class will be accessible through built-in menus.

Performance in Windows

The program has not been thorougly tested under Windows, and its performance there might be lower. The program depends crucially on the ability to start new processes quickly, which is one of the strengths of Linux (and the Unixes). I'm not sure how well the latest versions of Windows perform in this regard. If you perceive that the program takes too long to start reading a new sentence after finishing the previous one, the

problem might be that each new instance of **espeak** being invoked by **efabulor** is taking too long to start. A solution to this (potential) problem might be added in future versions. (You can also try the suggestions given above in TARGET PLATFORMS.)

If you would like to comment on any problem or ask for new functionalities, please open an issue at:

https://github.com/estebanflamini/efabulor/issues

KNOWN BUGS

The program has been thoroughly tested (in Linux), and no *obvious* bugs seem to exist. If you find any bug, please open an issue at:

https://github.com/estebanflamini/efabulor/issues

DEPENDENCIES

MINIMAL

Python >= 3.7 (installed in most modern Linux distributions; Windows users can download it from **python.org**)

espeak (available from standard repositories in Linux; Windows users can download it from **espeak.sourceforge.net** at the time of this writing)

RECOMMENDED

mbrola and the **mbrola** voices for the languages you intend to use, to get a more humanlike reading performance (**mbrola** is available in standard repositories for Linux; Windows users can download it from **github.com/numediart/MBROLA** at the time of this writing, or kindly do an Internet search for “mbrola”)

libreoffice and **unoconv**/**unoserver** to simplify extraction of text from non-plain text files (available from standard repositories in Linux; for Windows, see **libreoffice.org** and **github.com/unoconv/unoconv**).

(In Windows, after installing Python, it is recommended to add the **psutil** package:

```
pip install psutil
```

Alternatively, you can download **pssuspend** from Microsoft’s website and copy it to a directory within your PATH. Please follow instructions by Microsoft.)

CONTRIBUTING

If you would like to contribute to this project, please drop me a line at:

https://github.com/estebanflamini/efabulor/issues

You can check the file **contributing.md** for some hints. All contributions will be credited.

INSTALLATION

[Note: for the time being, no installation scripts are provided. You will have to install the required dependencies and copy the program's files yourself. I look forward to building a users' community who might be willing to help me create installation scripts and/or improve the installation instructions.]

Copy all six **efab*.py** files to a directory included in your PATH. In Linux, make them executable, so you can call the scripts by name. In Windows there is also a way to make scripts executable by name, but you can simply run **efabulor** by prefixing "python3":

```
python3 efabulor.py [options] [file]
```

Copy the ***.cfg** files to the same directory where you put the ***.py** files.

For the installation of the required dependencies (**python3**, **espeak**) and recommended dependencies (**mbrola**, **unoconv**/**unoserver**, **libreoffice**; **psutil**/**pssuspend** in Windows), kindly follow the installation instructions given by the authors of the required packages, and if you'd like to share detailed instructions for your platform (and get credited for doing so), please contact me by opening an issue at:

https://github.com/estebanflamini/efabulor/issues

TUTORIAL

This is a very simple tutorial to give you an idea of what you can do with the program. Detailed instructions for using the program are given after the tutorial.

PART 1:

Copy the **tutorial** directory to somewhere in your disk. Open a terminal (in Windows, you must go to **Start->Execute**, write **cmd.exe in** the search box and press **Enter**; if

you are not terminal-savvy, kindly look for advice in the Internet or in your Windows documentation).

Change to the tutorial's directory. Copy **source.txt** to **tmp.txt** (we will modify the file later, so it's a good idea to keep the original **source.txt** file).

Now, to start the tutorial, execute (the part in brackets is optional in Linux):

```
[python3] efabulor.py --lang en tmp.txt
```

If everything works as expected, **efabulor** will read the file aloud and stop after reading the last line.

Now press **b** to jump back one line.

Press **a** to start reading again.

Press **v** to jump back to the start of the file; if **efabulor** was reading when you pressed **v**, it will keep reading from there (or you can press **a** again to start the reading again).

Press **n** to jump to the next line (unless you are already at the last line).

Press **m** to jump to the last line.

Try going back and forth and restarting the reading a few times.

Now, while **efabulor** is reading, press the **spacebar**. The reading will be paused (in Windows, it only works if you installed **psutil** or **pssuspend**; see the INSTALLATION section). Press the **spacebar** again; the reading will be resumed where it left.

If instead of the **spacebar** you press **x**, the reading will be stopped and the next time you press the spacebar, it will be resumed from the beginning of the line. (Pressing **a** always starts or restarts the reading from the beginning of the line.)

Press **o**. After a short time, a new window should pop up and you should see the input file in your default editor for plain text files.

Go back to the window where **efabulor** is running, and press **m** to go to the last line.

Now, go to the editor window and make some changes to the file. After no more than two seconds, **efabulor** should detect that changes were made, jump to the first modified line, and start reading again from there. **efabulor** is designed to give you immediate auditory feedback of changes to the input file as they are being made. (I call this 'tracking'.)

At this point of the tutorial, changes are only being 'tracked' on lines before the current one (that's why I asked you to go to the last line before making any changes).

Press **v** to go to the first line. Press **?** to open a 'tracking-mode' menu. From the menu, select **3** to activate the 'forward' tracking mode.

Make some changes to the input file, after the first line. Now **efabulor** will also detect changes made after the current line.

When you feel you have experimented enough, press **q** to quit the program.

By the way, if you don't like the keys I've chosen for the commands, they are configurable; see [TO DO].

PART 2:

You might have noticed that **efabulor** was splitting lines at the end of paragraphs (or, more technically, wherever there is a 'newline' character). It would be nice to have it split the text in smaller sections, wouldn't it?

Copy **source.txt** to **tmp.txt** again, and execute (it's one line):

```
[python3] efabulor.py --lang en --transform transformrules.txt
tmp.txt
```

efabulor will read the file **tmp.txt** again, but now it will be split in shorter sentences.

When you are satisfied that **efabulor** is splitting the text in shorter sentences, press **:** (colon).

A new editor window should pop up, with the content of the **transformrules.txt** file. Have a look at it; you will find some introductory information on how **efabulor** can apply transformation rules to the text before splitting it in lines.

If you will be using regular expressions to write your transformation rules (you will!), please note that regular expressions are applied to the text as a whole; that is, they span over paragraphs boundaries (as if you used the **-z** option in **sed**); this gives you more flexibility when defining transformations. More information is given in SPLITTING AND TRANSFORMATION RULES. (And by the way, if you'd prefer to transform the text using other standard tools, such as **sed**, or using any program of your own writing, there is an option to do that; see PREPROCESSING FILTERS).

The transformation file provided here is long! Be sure to read it in full, as there is important information at the end. (By the way, in the previous part of this tutorial, you've learned that **efabulor** keeps track of changes you make to the input file. **efabulor** will also track changes to the input text indirectly caused by changes to transformation rules files; that way, you can edit your transformation rules and have immediate feedback of the effects.)

Also note that transformation rules are not the only way to configure the splitting of sentences in **efabulor**. See PREPROCESSING_FILTERS.

Ok, now quit the program (you already know how to do it) so we are ready to start the final part of this tutorial.

PART 3:

If you have modified **tmp.txt** since the last time, copy **source.txt** to **tmp.txt** once again. Now execute (it's one line):

```
[python3] efabulor.py --lang en --transform transformrules.txt
--subst substrules.txt tmp.txt
```

Pay close attention to the reading. Did you notice any change? If you didn't, go back to the first line and start the reading again.

Once you detect what was different this time (or if you don't) press **s**. Once again, an editor window should pop up, this time showing the content of the **substrules.txt** file. Substitution files are similar to transformation files, in that they instruct **efabulor** to

edit the input text before sending it to **espeak**. The difference is that transformation rules are applied to the text as a whole before splitting it in sentences; substitution rules are applied on a sentence-by-sentence basis, after the text was split. And if you change the substitution-rules file while **efabulor** is running, a 'tracking-event' will be triggered if and only if the new rules affect the current line.

By this time, you should be starting to have an idea of how you can use **efabulor** to support your workflows. The rest of the manual will provide you a full account of how it works.

USING EFABULOR

READ A PLAIN TEXT FILE

```
efabulor.py <plain_text_file>
```

Or (needed in Windows, unless you know how to make scripts executable in Windows):

```
python3 efabulor.py <plain_text_file>
```

efabulor will read the text file aloud and stop at the end waiting for instructions. A voice for the language specified in your default locale will be used (**mbrola** voices will be given preference if installed). The reading can be controlled with the keyboard (see KEYBOARD CONTROL below).

KEYBOARD CONTROL

(For a shorter explanation, see KEYBOARD CHEAT SHEET below.)

Key	Action
x	Stop the reading if the player is running, and reset it to start playing from the beginning of the line the next time it is started.
a	Restart the reading from the beginning of the current line.
A	Restart the reading from the beginning of the current line, scheduling the player to stop at the end of the line.

Key	Action
<spacebar>	Pause the reading if the player is running. Unpause the reading if the player is paused. Start the reading if the player is stopped, honoring a 'pause-before' option if set. If the input text changed while the player was not running and the loading of the input text was postponed, start reloading the text now. (Windows: the first two behaviours only work if either **pssuspend** or **psutil** was installed.)
v	Go to the first line in the current sequence mode. If the player was running and not paused, restart the reading at the new line. If the player already was at the first line, do nothing.
V	Go to the first line in the current sequence mode and stop there.
m	Go to the last line in the current sequence mode. If the player was running and not paused, restart the reading at the new line. If the player already was at the last line, do nothing.
M	Go to the last line in the current sequence mode and stop there.
n	Go to the next line in the current sequence mode. If the player was running and not paused, restart the reading at the new line. If the player was already at the last line, do nothing.
N	Go to the next line in the current sequence mode and stop there. If the player already was at the last line, only stop.
b	Go to the previous line in the current sequence mode. If the player was running and not paused, restart the reading at the new line. If the player was already at the first line, do nothing.
B	If the player is running and not paused, stop the reading and reset the player to start playing from the beginning of the line the next time it is started. If the player is stopped or paused, go to the previous line, unless the player is at the first line.
f	Stop the player if it is running and prompt for a search string. Search for the first line containing that string, case insensitively.
F	Stop the player if it is running and prompt for a search string. Search for the first line containing that string, case sensitively.
r	Stop the player if it is running and prompt for a search regular expression. Search for the first line matching that expression, case insensitively.
R	Stop the player if it is running and prompt for a search regular expression. Search for the first line matching that expression, case sensitively.

Key	Action
/	Stop the player if it is running and prompt for a search mode (plain text/regular expression), a case-sensitivity option, and a search string/regular expression. Search for the first line matching the given criteria.
t	Repeat the last search forward. If a new match is found, go there. If the player was running, start playing the new line. If no match is found, do nothing.
T	Repeat the last search forward. If a new match is found, go there and stop the player if it was running. If no match is found, do nothing.
e	Repeat the last search backward. If a new match is found, go there. If the player was running, start playing the new line. If no match is found, do nothing.
E	Repeat the last search backward. If a new match is found, go there and stop the player if it was running. If no match is found, do nothing.
g	Stop the player if it is running and prompt for a line number. Go to that line number.
<	If the text was modified, go to the nearest modified line before the current line and read it, then stop at the end of the line. If there are no modified lines before the current one, do nothing. (Note: the set of modified lines is updated each time the underlying file changes on disk.)
>	If the text was modified, go to the nearest modified line after the current line and read it, then stop at the end of the line. If there are no modified lines after the current one, do nothing. (Note: the set of modified lines is updated each time the underlying file changes on disk.)
*	Choose a random line, read it, and then stop at the end of the line.
.	Schedule the player to stop at the end of the current line. If the option is already set, unset it.
,	Schedule the player to stop at the end of each line. If the option is already set, unset it.
q	Stop the player if it is running or paused. Quit the program, asking the user for confirmation before.
Q	Stop the player if it is running or paused. Quit the program, withouth asking the user for confirmation.
Ctrl-C	Same as **Q**.
w	Print the current line again.
l	Cycle line number printing (no line number -> only line number -> line number and total number of lines).
D	Toggle an option for showing original text/substituted text if substitution rules were given and they are being applied.

Key	Action
S	Toggle an option for applying/not applying substitutions. If the current line being read has changed as a result of this, restart the player if it is running, or stop the player and reset it to start reading from the beginning of the line next time if it is paused.
s	Stop the player if it is running and present the user the list of substitution-rule files being used. Let the user choose one of the files and open it in the default editing application. If there is only one file, open it without asking. If no substitution rules are being used, do nothing.
:	Stop the player if it is running and present the user the list of transformation-rule files being used. Let the user choose one of the files and open it in the default editing application. If there is only one file, open it without asking. If no transformation rules are being used, do nothing.
_	Stop the player if it is running and present the user the list of additional monitored files that were specified in the command line, if any. Let the user choose one of the files and open it in the default editing application. If there is only one file, open it without asking. If no monitored files were specified in the command line, do nothing.
u	If substitutions rules were given, and substitutions were applied to the current line, stop the player and show a substitution log. Otherwise, do nothing.
j	If transformation rules were given, and they affected the input text, stop the player and show a transformation log. Otherwise, do nothing.
+	Increase the reading speed by 10 words/minute; if the player is running, restart reading the current line.
-	Decrease the reading speed by 10 words/minute; if the player is running, restart reading the current line.
o	Open the input file in its default editing application.
O	Stop the player if it is running and open the input file in its default editing application.
c	Stop the player if it is running and open the default shell. When you close the shell, you will return to **efabulor**.
L	Reload the input file.
C	Check the monitored files for modifications now. (This command allows you to override a checking interval if given in the command line.)
?	Stop the player if it is running and prompt the user to choose a tracking mode. (See TRACKING MODES)
)	Stop the player if it is running and prompt the user to choose a sequence mode. (See [TO DO])

Key	Action
=	Stop the player if it is running and prompt the user to choose a feedback mode. (See FEEDBACK MODES)
0	Stop the player if it is running and enter a special mode where tracking mode is ‘forward’, feedback mode is ‘full’, and the player stops after reading each line. (Pro tip: I use this mode for a final proofreading, to have full feedback of any changes I make before sending the file to the client.)
1	Stop the player if it is running and enter the normal mode where tracking mode is ‘backward’, feedback mode is ‘minimum’, and the player does not stop after reading each line.
i	Show the actual command being run to get the input text. This is for debugging purposes.
I	Stop the player if it is running and show the actual command being run to get the input text. This is for debugging purposes.

Note: the association between keys and actions (key bindings) is user-configurable; see DEFAULT ACTION NAMES and [TO DO].

KEYBOARD CHEAT SHEET

Key	Action
x	Stop
a	Restart
A	Restart and stop after current line
<spacebar>	Pause/unpause/restart
v	Go to the first line
V	Go to the first line and stop
m	Go to the last line
M	Go to the last line and stop
n	Go to the next line
N	Go to the next line and stop
b	Go to the previous line
B	Stop if playing, go the the previous line if not playing
f	Plain text case insensitive search
F	Plain text case sensitive search

Key	Action
r	Regular expression case insensitive search
R	Regular expression case sensitive search
/	Generic search
t	Repeat the last search forward
T	Repeat the last search forward and stop
e	Repeat the last search backward
E	Repeat the last search backward and stop
g	Go to line number
<	Read the previous modified line and stop
>	Read the next modified line and stop
*	Read a random line and stop
.	Stop/do not stop after the current line
,	Stop/do not stop after each line
q	Quit the program, with confirmation
Q	Quit the program, without confirmation
Ctrl-C	Quit the program, without confirmation
w	Print the current line
l	Cycle line number printing
D	Toggle showing the effect of substitutions
S	Toggle the application of substitutions
s	Open substitution-rule files
:	Open transformation-rule files
_	Open command-line monitored files
u	Show substitution log
j	Show transformation log
+	Increase the reading speed
-	Decrease the reading speed
o	Open the input file
O	Stop and open the input file
c	Stop and open the shell
L	Reload the input file
C	Check the monitored files for modifications
?	Choose a tracking mode
)	Choose a sequence mode

Key	Action
=	Choose a feedback mode
0	Set tracking mode=forward and feedback mode=full; stop after each line
1	Set tracking mode=backward and feedback mode=minimum; do not stop after each line
i	Show the input command
I	Stop and show the input command

SPECIFY THE LANGUAGE FOR READING

```
efabulor.py --lang <language ISO code> <file>
```

efabulor will use the default voice for the given language (**mbrola** voices will be given preference if installed). For example, to read a text in Spanish:

```
efabulor.py --lang es somefile.txt
```

SPECIFY THE VOICE FOR READING

```
efabulor.py --voice <voice> <file>
```

To query available voices:

```
espeak --voices
```

You can also configure a preferred voice for any language in a locale-specific configuration file. See LOCALE-SPECIFIC CONFIGURATION FILE.

READ THE OUTPUT OF A COMMAND

> CAUTION: Any command you provide with the `--do` option will be executed as given. It is YOUR responsibility to ensure the command does not have undesired side effects (e.g., erasing all of your files).

```
efabulor.py --do <command>
```

Read the output of any command. You might need to enclose the command in quotation marks. For example, to read aloud the names of all text files in the current directory (replace 'ls' with 'dir' in Windows):

```
efabulor.py --do 'ls *.txt'
```

A command can also consist of a pipeline. For example, to read all lines in **file.txt** containing the string 'foo', but changing it to 'bar' you could run (in Linux):

```
efabulor.py --do 'grep foo file.txt | sed "s/foo/bar/g"'
```

READ NON-PLAIN TEXT FILES

If you have **unoconv** (and LibreOffice) installed, you can use it to read text from non-plain text files:

```
efabulor.py --do 'unoconv -f txt --stdout file.rtf'
```

If you have **unoserver** instead of **unoconv**, the process should be similar. I still have to upgrade the program (and my computer) from **unoconv** to **unoserver**.

Sometimes, **unoconv** fails to convert the given file the first time you try it; instead, it terminates with an error. To avoid this error and to simplify the conversion of files, a wrapper around **unoconv** called **efabucw** is provided with **efabulor**. It will keep calling **unoconv** several times until it succeeds (and by the way, it will also take care of providing the necessary options to **unoconv** so you don't have to specify them each time):

```
efabulor.py --do 'efabucw.py file.rtf'
```

Currently, this tool only works with **unoconv**. Support for the new **unoserver** might be added in the future.

If you don't have **unoconv**, you may use whichever conversion tool you have installed to get similar results. There are several format conversion tools available for Linux in

standard repositories; Windows users might need to do some research and install a suitable one in their systems.

PRECONFIGURED CONVERSION COMMANDS

CAUTION: Any command provided in the 'do' part of the configuration file will be executed as given. It is YOUR responsibility to ensure the given command does not have undesired side effects (e.g., erasing all of your files). You MUST NEVER use the `--text-conversion-config` option (TO DO: unify options) with a configuration file you have received from outside without first checking its content.

Once you have a useful conversion command for a certain kind of files, you can preconfigure it as the default conversion command for files with a specific mimetype or filename extension. For example, suppose you use **efabucw.py** to extract text from RTF files, as explained above. You can configure it for all files with an RTF filename extension. Just create a configuration file containing the following (you can name the configuration file whichever way you want):

```
ext:   rtf
do: efabucw.py $file
```

(`$file` is a placeholder, which will be replaced by the name of the actual file.)

Then you can convert and read any RTF file with this command:

```
efabulor.py --text-conversion-config <the name of the config file>
somefile.rtf
```

Note: Be sure to specify a conversion command that prints the extracted text to the standard output; otherwise, **efabulor** will have nothing to read.

You can specify conversion commands for different types of files in a same configuration file, separated by at least one blank line. And you can include comments too:

```
ext:    rtf
do:     efabucw.py $file

ext:    doc
do:     efabucw.py $file

# This is just an example:
ext:    xls
do:     myFancySpreadSheetConverter --print-to-stdout $file
```

In Linux, you can also use mimetypes instead of filename extensions:

```
mime:   application/rtf
do:     efabucw.py $file

mime:   application/vnd.ms-word
do:     efabucw.py $file

mime:   application/vnd.ms-excel
do:     myFancySpreadSheetConverter --print-to-stdout $file
```

The following placeholders can be used in the 'do' subsection:

`$file`	will be replaced with the name of the file being read.
`$lang`	will be replaced with the language ISO code being used. This can be handy when processing multilingual files (e.g., TMX). For example: `do: myFancyTmxReader --extract-language=$lang $file`

SETTING A DEFAULT CONVERSION CONFIGURATION

> CAUTION: Any command provided in the 'do' part of a conversion-configuration file will be executed as given. It is YOUR responsibility to ensure such commands do not have undesired side effects (e.g., erasing all of your files). You MUST NEVER place a configuration file you have received from outside in the same directory where you installed **efabulor.py** and the companion tools without first checking its content.

To set a default conversion configuration file, name it **efabconv.cfg** and place it in the same directory where you installed **efabulor.py** and the companion tools. Then you can read a non-plain text file (e.g., an RTF file) with a command such as:

```
efabulor.py somefile.rtf
```

USING TEXT CONVERSION AS A STANDALONE APPLICATION

The text conversion is handled by a companion tool called **efabconv**. (That is the reason why the default configuration file has to be named **efabconv.cfg**.) You can also use **efabconv** as a standalone text extraction tool. (I do it all the time.) The only caveat is that the extracted text always goes to the standard output. If you want to save it to a

file, use a redirection.

To extract text from a non-plain text file (e.g., an RTF file) and print it to the standard output, using a default configuration file **efabconv.cfg** placed in the same directory as **efabconv.py**:

```
efabconv.py somefile.rtf
```

To extract text from a non-plain text file (e.g., an RTF file) and print it to the standard output, using a specified configuration file:

```
efabconv.py --config-file <configuration file> somefile.rtf
```

SPLITTING AND TRANSFORMATION RULES

By default, **efabulor** splits the text at paragraph boundaries (newline characters), strips spaces at the beginning and end of the resulting paragraphs, and removes empty paragraphs. You can control the splitting process by defining 'transformation rules'. A transformation rule is an instruction to search a given pattern within the input text and replace it with some other string. For example, if you would like to split the text after every occurrence of a period, an exclamation mark, or a question mark, you can create a 'transformation rules file' containing the following:

```
[DO]
s/([.!?])/$1\n/
```

The '[DO]' line marks the beginning of the transformation rules; when it is the first non-empty line in the file it is optional, but it is a good habit to include it anyway. The rule below that line is a regular expression (regex), similar to the ones used by **sed** in Linux. In this case, the regex asks the program to add a newline character right after any occurrence of a period, an exclamation mark, or a question mark. (Note that **efabulor** allows using a dollar sign to introduce backreferences. You can also use a backslash for the same purpose.)

A full explanation of regular expressions is beyond the scope of this manual, but there are plenty of tutorials and manuals on the topic online.

Supposing you save the transformation rule to a file named **transformrules.txt**, you can have **efabulor** apply the transformation rules to the input file with the following command:

```
efabulor.py --transform transformrules.txt inputfile.txt
```

You can include as many transformation rules in one file as you wish. And you can use several transformation-rules files in a single invocation of **efabulor**. (This allows for some modularity when you define transformation rules.)

There is another simplified syntax for transformation rules which does not involve regular expressions. You can use plain text rules consisting of two lines, such as:

```
foo
bar
```

This rule means ‘replace every occurrence of foo with bar’. Two-line transformation rules must be separated from the precedent rule and from the following one by at least an empty line.

The transformations rules will be applied before **efabulor** splits the text at newlines, so the net result is that the text will be split not only where there was a newline character in the original text, but also after the selected punctuation marks, where the transformation rule added a newline. (The newline character is not added to the underlying file itself, only to the input text that **efabulor** has loaded to memory.)

Transformation rules are applied to the text as a whole (just as if you used the -z option with **sed**), so for example the ^ anchor means the very beginning of the text and the $ anchor means the very end of the text. This default behavior can be modified by including compilation flags within the regular expression. For an explanation of the regex flags used by **efabulor**, see [TO DO]. Note also that regexes used as transformation rules are implicitly global; no need to add a ‘g’ flag.

PROTECTING TEXT FROM TRANSFORMATION RULES

If you would like to prevent the application of transformations rules in certain cases, you can define ‘protection rules’. Following with the previous example, suppose you would like to split the text at every occurrence of a period, an exclamation mark, or a question mark, but not when that period is part of an abbreviation or it is the decimal point in a number. You can create a transformation-rules file with the following content:

```
[DO]
s/([.!?])/$1\n/

[DO NOT]
/\d+\.\d+/
e.g.
etc.
```

The new ‘[DO NOT]’ section contains ‘protection rules’. Any text span matching one of these rules will be protected from the transformation rules given in the previous ‘[DO]’ section. Note you can use both regular expressions and plain text patterns as protection rules (in the preceding example, the first rule is a regex, and the remaining two are plain text rules). Protection rules of both types can be mixed, and there is no need to separate them with empty lines. You can have as many [DO]/[DO NOT] pairs as you wish; in each case, the protection rules contained in a [DO NOT] section protect only from the transformation rules contained in the previous [DO] section.

USING TRANSFORMATION RULES FOR PREPROCESSING

While the main purpose of transformation rules is to give you a way to add newlines in such places where you want the text to be split for reading, you can use transformation rules to do any kind of preprocessing. (That's why they are called 'transformation rules'.) For example, transformation rules could be used to extract text from markup files; or to spot errors and add a message at the place of the error to be read aloud by **espeak**. (I have a set of transformation rules to detect mismatched quotation marks, superfluous spacing, punctuation marks preceded by spaces, repeated words, etc.) Once you learn how to use transformation rules, the limit is your imagination!

TRANSFORMATION RULES AS A STANDALONE APPLICATION

The transformation rules are not applied by **efabulor** itself, but by a companion tool called **efabtrans**. This allows using the transformation rules as a standalone application. For example, to split a plain text file according to the set of transformation rules discussed above and copy the result to the standard output, you can use the following command:

```
efabtrans.py --transform transformrules.txt inputfile.txt
```

If you want to save the result to a file, use a redirection.

Note that unlike **efabulor**, after the transformation rules are applied, **efabtrans** will not strip spaces at the beginning and end of paragraphs nor remove empty paragraphs (an option to do this might be added in future versions). If you need that content removed, you should take care of that yourself, either by adding specific transformation rules to the transformation rules file, or by filtering **efabtrans**'s output through an external command, e.g. **sed**:

```
efabtrans.py --transform transformrules.txt inputfile.txt |
  sed 's/^\s*//' | sed 's/\s*//' | sed '/^$/d'
```

Because **efabtrans** works as a filter, you can apply it to non-plain text files as well:

```
efabconv.py file.rtf | efabtrans.py --transform transformrules.txt
```

PREPROCESSING FILTERS

CAUTION: Any command provided with `--preprocess` will be executed as given. It is YOUR responsibility to ensure the given command does not have undesired side effects (e.g., erasing all of your files).

As an alternative (or complement) to transformation rules, you can preprocess the input text by giving **efabulor** a filtering command or pipeline with the --preprocess option. The command or pipeline will receive the unprocessed input text as *its* input text and will pass its output as input text for **efabulor**. For example, another way to split the text at periods, exclamation signs and quotation signs is to use the following command:

```
efabulor.py --preprocess 'sed -r "s/([.!?])/\1\n/g"' inputfile.txt
```

Note the two sets of quotation marks: the outer one is to enclose the filtering command, the inner one is to enclose the regular expression used by **sed** (which happens to be the filtering command in this example).

Preprocessing filters give you full freedom to define a preprocessing stage when transformation rules are not enough (or when they are too much).

You can give several preprocessing filters, which will be applied in the order they are given. Supposing **filter1** and **filter2** are some external filtering scripts or programs:

```
efabulor.py --preprocess filter1 --preprocess filter2 inputfile.txt
```

You can also combine all the preprocessing filters into a single pipelined filter:

```
efabulor.py --preprocess 'filter1 | filter2' inputfile.txt
```

Preprocessing filters can be combined with transformation rules. In this case, transformation rules will be applied *after* preprocessing filters (the relative location of the --transform and --preprocess options in the command line is irrelevant). If you need to apply transformation rules *before* preprocessing filters, calling **efabtrans** as a filter before the other filters will do the trick:

```
efabulor.py --preprocess 'efabtrans.py --transform
transformrules.txt' --preprocess some_other_filter inputfile.txt
```

[TO DO: unificar mark/sign]

SEGMENTATION AND SEPARATOR OPTIONS

Transformation rules (and preprocessing filters) provide a rather simple and flexible way to control the way **efabulor** splits the input text in sentences. In certain cases, however, there is yet a simpler way.

efabulor splits text at newlines by default. You can tell the program to use any other string or pattern instead. For example, to have **efabulor** split the text at spaces instead of newlines, you can use the following command:

```
efabulor.py --separator ' ' inputfile.txt
```

To have **efabulor** split the text at spaces *and* newlines, you must use a regular expression:

```
efabulor.py --separator '/[ \n]/' inputfile.txt
```

A simple way to read CSV files (provided no commas appear within fields):

```
efabulor.py --separator '/[,\n]/' inputfile.csv
```

And there is yet another alternative. Instead of telling **efabulor** which string or patterns separate readable units [TO DO], you can provide a regular expression which defines how a readable unit looks like. For example, to tell **efabulor** to split the text in groups of two newline separated paragraphs, you can use the following command:

```
efabulor.py --segment '((?:.+?\n+){1,2})' inputfile.txt
```

As you see, it's not necessary to enclose the regex in regex delimiters when you use the --segment option, although you can do it if you want (or if you need to specify flags; see [TO DO]):

```
efabulor.py --segment '/((?:.+?\n+){1,2})/' inputfile.txt
```

Note the use of capturing groups. When the segmentation regex includes capturing groups, each capturing group will deliver one readable unit. The outer pair of parentheses in this example encloses a capturing group. The inner pair of parentheses delimits the pattern to which the quantifier '{1,2}' applies. The pattern starts with ?: to indicate it's not a capturing group; otherwise, we would get the second paragraph of each group twice. We use '\n+' for newlines instead of just '\n' to allow for the case where paragraphs in the original text are separated by more than one newline.

If there are no capturing groups, the readable unit will be just the text matched by the regular expression. So the segmentation regex of the previous example could be simplified a little more:

```
efabulor.py --segment '(?:.+?\n+){1,2}' inputfile.txt
```

The quantified pattern still needs to be a non-capturing group. The reason is that when you apply a quantifier to a capturing group, only the last match is returned. (You could use such a pattern to read every other paragraph in a file.)

Now a slightly simpler example. Suppose you have a TMX file (TMX is a common multilingual format based on XML used by translators) containing something like this:

```
...
    <tu>
      <tuv xml:lang="EN"><seg>First sentence</seg></tuv>
      <tuv xml:lang="ES"><seg>Primera oración</seg></tuv>
    </tu>
    <tu>
      <tuv xml:lang="EN"><seg>Second sentence</seg></tuv>
      <tuv xml:lang="ES"><seg>Segunda oración</seg></tuv>
    </tu>
    <tu>
      <tuv xml:lang="EN"><seg>Third sentence</seg></tuv>
      <tuv xml:lang="ES"><seg>Tercera oración</seg></tuv>
    </tu>
 …
```

You want to have **efabulor** read just the English sentences one after the other (that is: “First sentence”, “Second sentence”, …). One approach would be to create a script that extracts precisely those parts, and invoke it through the --do option (or configure it as a generic extracting script for TMX files in the **efabconv.cfg** configuration file). Another approach would be to write a transformation rules file to delete everything in the TMX files which is not part of the English sentences, taking care of leaving the sentences separated by newlines. Both solutions might be the right thing to do it if you will be reading TMX files such as this all the time. But if you need to read just this one TMX file, and because you know the exact structure of the file, there is a simpler ‘on-the fly’ solution in which you don’t need to write extraction scripts or transformation rules. Just enter:

CAUTION: Just in case you are reading only this part of the manual: remember that any command you provide with the `--do` option will be executed as given. It is YOUR responsibility to ensure such commands do not have undesired side effects (e.g., erasing all of your files).

```
efabulor.py --segment '<tuv xml:lang="EN"><seg>(.+?)<\/seg><\/tuv>'
--do 'cat inputfile.tmx'
```

(Replace ‘cat’ with ‘type’ in Windows.)

There is only one capturing group, and it will extract exactly the content of the English

sentences (which are identified by the ‘xml:lang="EN"’ attribute in the TMX file). Note also we had to escape with a backslash the forward slashes that mark the closing tags in TMX. Finally, we use the ‘--do 'cat inputfile.tmx’’ part because we assume that no specific conversion command has been configured for TMX files. We know however that a TMX file is just a special kind of text file, so we can ‘extract’ its content by simply copying it to the standard output (remember that the output of --do will be the input for **efabulor**). The command to do so is **cat** in Linux and **type** in Windows. If we had just specified the name of the file (instead of --do), **efabulor** would complain that no conversion rules were given for TMX files.

SUBSTITUTION RULES

Substitution rules can be applied to the text to further control the reading process. Substitution rules are similar to transformation rules, and in fact, the language used for defining them is almost the same (see SPLITTING AND TRANSFORMATION RULES). There are some differences though:

- In the current version of **efabulor**, protection rules cannot be defined for substitution rules, because substitution rules files cannot contain sections. Substitution rules files containing anything resembling a section header file will be ignored. This minor difference between substitution rules and transformation rules might be removed in future versions.
- Transformation rules are applied after plain text has been extracted from the input file (and possibly preprocessed; see PREPROCESSING), and before the text is split into segments. They affect the input text as a whole as seen by **efabulor**. (And when you use regular expressions to define transformation rules, they apply to the whole text by default.) They help to define *what* text will be read.
  Substitution rules are applied to each segment before it is sent to **espeak** for reading. They affect only one segment at a time. If you use regular expressions to define substitution rules, they apply only to the current segment. The result of the substitution is not shown onscreen by default; it is only seen by **espeak**. Substitution rules affect the actual input for **espeak**. They help to define *how* the input text will be read. (See an example below of when this can be useful.)
- When changes to transformation rules are detected, **efabulor** will reload the whole text (that is, execute the input command again) and possibly generate a tracking event (see CHANGE TRACKING). When changes to substitution rules are detected, the input text will not be reloaded. **efabulor** will only reload the substitution rules and apply the new rules only to the current segment. If the substituted text changes as a result of the new substitution rules, a tracking event will be generated only for the current segment.

Suppose you want **espeak** to say ‘bar’ aloud each time the input text contains ‘foo’. You can define a substitution rules file, say **substrules.txt**, with the following content:

```
foo
bar
```

Then you can use the substitution rules file with the following command:

```
efabulor.py --subst substrules.txt inputfile.txt
```

You can use more than one substitution rules file, and they will be applied in the order they are given.

You can also use regular expressions as substitution rules, but if you mix plain text (two-line) substitution rules and regex (one-line) substitution rules, plain text rules must be separated from each other and from regex rules by empty lines:

```
foo
bar

42
the Answer to the Ultimate Question of Life, etc.

s/(\w+)\s+\1\b/repeated word here/
```

You can refer to the section on SPLITTING AND TRANSFORMATION RULES, as most of its content also applies to substitution rules.

Why would you use substitution rules? One reason is that sometimes the result of the underlying text-to-speech conversion can be unsatisfactory or erroneous. For example, when using the **spanish-mbrola-1** voice to read text in Spanish, there is a minor glitch every time the 'll' digraph occurs in the input text (with certain versions of **espeak** and **mbrola** at least). I use the following substitution rule to solve this, as 'y' is virtually homophonous to 'll' in most variants of Spanish:

```
s/[Ll]l([aeiouáéíóú])/y$1/
```

Another use for substitution rules is to give you auditory feedback of common errors in your input text. For example, to have **efabulor** alert you of repeated spaces within text, you could have a substitution rule such as:

```
s/ {2,}/ repeated space here /
```

(Note that repeated spaces *between sentences* will be removed by **efabulor** when splitting the text into sentences.)

You can place quality-assurance rules such as this in a substitution rules file when they can be applied on a segment by segment basis; or place them in a transformation rules file when the regular expression needs to be matched against the whole text. Experience will teach you the proper way to do it.

CHANGE TRACKING

One of the main functionalities of **efabulor** is its ability to track changes to the underlying files being used within a reading session. The underlying files (also called monitored files) which are tracked for changes are:

- the input file

- the transformation rule files

- the substitution rule files

- additional files/directories specified with the --monitored-file option

By default, **efabulor** checks tracked files for changes every 2 seconds. The interval can be adjusted with the --monitoring-interval option.

A file is considered to be changed whenever its modification timestamp on disk has changed, irrespective of the actual contents of the file.

When **efabulor** detects a change to a monitored file which is not a substitution rules file, the following things happen by default (the actual behavior can be further modified with several runtime options and command line switches):

- The text extraction pipeline (see [TO DO]) is executed again.

- The new input text is compared with the previous version.

- If changes are detected:

    - A summary report is given onscreen.

    - If the first modified line is at or before the current line, a 'tracking event' is generated. A tracking event means that the player is stopped if it is running, the first modified line becomes the current line and the reading is resumed. A brief notification is read aloud before restarting the player, to act as an audible separator within the stream of reading.

When the change affects a substitution rules file, the substitution rules file is reloaded and the player is restarted if and only if the new substitution rules affect the current line.

TRACKING MODES

There are four tracking modes, which are: backward, forward, restart and none. efabulor starts in backward mode (unless called with the --tracking-mode option and a different choice).

The behaviour in **backward** mode was described in the preceding section.

In **forward** mode, a tracking event is generated also when the first modified line is after the current one. (That means that the player will advance to the first modified line and read from there.)

In **restart** mode, whenever a change to the input text is detected, the player will restart at the first line irrespective of the location of the first modified line.

Finally, tracking mode can be set to **none**. In this mode, a report of changes will be given onscreen, but the player will continue reading the current line. (If the feedback mode is set to **full**, the player may have to stop to give the auditory feedback; after that,

the player will restart reading the current line.)

FEEDBACK MODES

Feedback modes specify the amount of auditory feedback you will receive when the underlying files change and a tracking event is generated (see CHANGE TRACKING). There are three feedback modes: none, minimum and full. Minimum feedback mode is the default. In minimum feedback mode, when a tracking event occurs **efabulor** will only tell you whether the reading is being restarted or if the player is jumping to another sentence in the input text because of the tracking event. Minimum feedback messages are intended as a separator between the sentence that was being read when the tracking event occurred and the sentence that will be read afterwards. When feedback mode is 'none', you don't get that separator when a tracking event occurs; the sentence that was being read is interrupted abruptly, and the player continues reading the new current sentence. When feedback mode is 'full', **efabulor** will also read aloud several messages trying to describe the change that triggered the tracking event: whether there are changes to one or many lines, if the length of the file has decreased/increased, etc.

CONFIGURING FEEDBACK MESSAGES FOR THE LOCALE

The feedback messages which are read in response to changes to the underlying files are not hardcoded within **efabulor** but configured in a locale configuration file. See LOCALE-SPECIFIC CONFIGURATION FILE.

LOCALE-SPECIFIC CONFIGURATION FILE

Several options which depend on the language being used for reading can be configured in a locale-specific configuration file. Those options currently are: the preferred voice for the language and the feedback messages used when changes to the underlying files are detected (see CHANGE TRACKING and FEEDBACK MODES).

The current version of **efabulor** includes two predefined configuration files for English and Spanish. You can use any of them as a template for creating a configuration file for another language by simply replacing your preferred voice and translating the messages. For example, this is the predefined English configuration file:

```
voice:                  english-mb-en1
no-changes:             File changes do not affect the text
changes-reverted:       The last changes were reverted
many-changes:           Many lines have changed
changes-before:         Changes before current line
changes-here:           Changes in current line
```

```
changes-next:           Changes in next line
changes-after:          Changes after current line
blank-areas-changed:    Empty areas changed
jumping-back:           Going back
jumping-forward:        Going forward
continuing:             Continuing
restarting:             Starting again
starting-again:         Starting again from the beginning
subst-changed:          Substitution rule changed
reload-delayed:         File reload postponed
file-increased:         File size increased
file-decreased:         File size decreased
file-too-short:         File was reduced before the current line
file-is-empty:          File is empty
file-changed-again:     File changed again on disk. Reporting
                        changes since last modification
```

The 'voice' option defines the preferred voice for reading. (If you include the `--voice` option in the command line, it will take precedence over the voice configured in the locale configuration file.)

The other options define the feedback messages used with feedback modes of 'minimum' and 'full'. The current version of **efabulor** provides the following feedback messages in response to changes to the underlying files:

IN MINIMUM/FULL FEEDBACK MODE:

(Note: messages from this set are mutually exclusive)

Message key	**Played when …**
restarting	the reading will resume at the current line.
jumping-back	the reading will resume at a line before the current one.
jumping-forward	the reading will resume at a line after the current one.
continuing	the reading will resume at the line immediately after the current one.
subst-changed	a substitution rule has changed that affects the current line.
starting-again	the reading will resume at the first line because the tracking mode is set to 'restart'.

ONLY IN FULL FEEDBACK MODE:

(Note: messages from this set can be combined with each other and with the 'minimum' set.)

Message key	Played when ...
reload-delayed	the modification time of an underlying file has changed, but reloading the text and looking for changes is postponed until the playing is restarted by the user.
no-changes	the modification time of an underlying file has changed, but no actual changes to the input text have been detected.
changes-reverted	changes to the input text revert the previous change.
many-changes	changes to the input text affect many lines.
changes-before	changes to the input text occur before the current line.
changes-here	changes to the input text occur in the current line.
changes-after	changes to the input text occur after the current line.
changes-next	changes to the input text occur in the line following the current line.
file-increased	the file's line count has increased.
file-decreased	the file's line count has decreased.
file-too-short	the file's line count has decreased before the current line.
file-is-empty	the input text is empty after the change.
file-changed-again	an underlying file changed while feedback on a previous change was being played.
blank-areas-changed	changes to whitespace areas of the input text immediately before the end of a line have been detected.

Note: you can prevent a particular message to be played in any mode by simply commenting out or deleting the corresponding key-value pair from the locale configuration file you are using.

USING A LOCALE CONFIGURATION FILE

There are two ways to use a specific locale configuration file. One is to specify it in the command line by using the `--config-file` option. For example:

```
efabulor.py -c locale_config.txt input_file.txt
```

Alternatively, you can set a locale configuration file as a default by placing it in the same directory as **efabulor.py**. (This is probably the way you will use most.) The name of the configuration file has to obey the following format:

```
efabulor.<language ISO code>.cfg
```

For example (for German):

```
efabulor.de.cfg
```

Now **efabulor** will use whichever configuration file it finds whose language ISO code matches the `--lang` option given in the command line or the computer's current locale if no `--lang` option was given.

Whichever way you use, the configuration file is read when the program starts. If you introduce changes to the configuration file, you must restart the program for the changes to take effect.

ABOUT THE LOCALISATION OF FEEDBACK MESSAGES

In the current version of **efabulor**, the feedback modes and the lists of messages which are played in each mode are predefined; the only thing you can change by creating a new locale configuration file or editing the default ones is the feedback messages themselves. (You can also prevent a particular message to be played in any mode by simply commenting out or deleting the corresponding key-value pair from the locale configuration file you are using.)

The translation of the feedback messages is handled independently from the localisation system used to translate the program's interface (gettext) because the language you will be using for the interface is not necessarily the same as the language you will be using for reading a particular text. In future versions, a design based on gettext might also be used for the feedback submodule, and the locale configuration files would be removed.

RECONFIGURING KEY BINDINGS

The key bindings used by **efabulor** are reconfigurable. To do so, you need to define a key-binding configuration file and pass it in the command line.

The simplest way to create a key-bindings configuration file is to print the default key bindings to a file and edit the file with a standard text editor. To print the default key bindings, call **efabulor** with the `--save-default-key-bindings` option:

```
efabulor.py --save-default-key-bindings <file for key bindings>
```

If you omit the filename, **efabulor** will print the default key bindings to standard output.

The default key-bindings are as follows:

```
\x03        <quit-now>
Q           <quit-now>
q           <quit-ask>
\x20        <toggle>
a           <restart>
A           <restart-and-stop>
```

```
x           <stop-and-reset-pointer>
V           <first-stop>
v           <first>
M           <last-stop>
m           <last>
N           <next-stop>
n           <next>
B           <stop-or-previous>
b           <previous>
.           <toggle-stop-after-current-line>
,           <toggle-stop-after-each-line>
S           <toggle-apply-subst>
D           <toggle-show-subst>
u           <log-subst>
j           <log-transform>
l           <cycle-line-number>
w           <show-line>
f           <search-plain-case-insensitive>
F           <search-plain-case-sensitive>
r           <search-regex-case-insensitive>
R           <search-regex-case-sensitive>
/           <search>
t           <find-next>
T           <find-next-stop>
e           <find-prev>
E           <find-prev-stop>
g           <go-line>
<           <prev-change>
>           <next-change>
*           <random>
+           <faster>
-           <slower>
o           <open-input-file>
O           <open-input-file-stop>
s           <open-subst>
:           <open-transform>
_           <open-cl-monitored-file>
c           <open-shell>
L           <reload>
```

```
C           <check-files>
?           <choose-tracking-mode>
)           <choose-sequence-mode>
=           <choose-feedback-mode>
0           <special-mode>
1           <normal-mode>
I           <show-input-cmd-stop>
i           <show-input-cmd>
```

In the key-bindings configuration file, printable characters stand for themselves; hexadecimal codes such as `\x03` and special characters such as `\n` stand for control characters or characters according to their ASCII value. For example, `\x03` stands for the ASCII character no. 3, which you can enter by pressing **Ctrl+C** in most terminals. The code `\x20` stands for the **spacebar**. The code `\x0a` and the code `\n` stand for the **Enter** key; and so on.

The action names correspond to the default actions as explained in KEYBOARD CONTROL; the names should be self explanatory.

To reassign key bindings to the several actions, you can simply overwrite the key specifications in the configuration file. Be careful to separate the key specification and the name of the action with a tabulation character. Also be careful not to assign the same key to different actions, and to assign at least one key to at least one of the **<quit-now>** and **<quit-ask>** actions; if you fail to do so, the configuration file will not be usable.

If you want to use special non-printable keys, e.g. arrow keys, function keys, etc., see below CREATING KEY BINDINGS INTERACTIVELY.

Once you finish editing the key-bindings configuration file, you can use it by adding the `--key-bindings` option to the command you normally would use to call **efabulor**. For example:

```
efabulor.py --key-bindings <key bindings file> <other options…>
<file to read>
```

You can also create a partial key-binding configuration file and add it to the default key bindings by using the `--add-key-bindings` option with the `--key-bindings` option. For example, suppose you want to retain the default key bindings, but adding a new key binding to pause/unpause the player by using the **Enter** key. You can create a file with the following content:

```
\n          <toggle>
```

Then you can add the new key binding to the default one by calling **efabulor** with the `--add-key-bindings` option:

```
efabulor.py --key-bindings <key bindings file> --add-key-bindings
<other options…> <file to read>
```

CREATING KEY BINDINGS INTERACTIVELY

There is yet another way to create a key-binding configuration file. You can call **efabulor** with the `--edit-keycodes` option, followed by a filename, and enter an editing mode in which you will be asked to type the action names (as specified above) and then press the key you want to associate with that action. When you are done, press **Enter** instead of typing an action; **efabulor** will write the key bindings to the file you specified (or to standard output, if none was specified). This mode allows you to configure also non-printable keys (e.g., arrow keys, function keys) whose codes depend on the specific type of terminal you are using. Note that a key-binding configuration file containing codes for such non-printable keys might not work when copied to another computer with a different type of terminal. For example, if I were to create a key binding between the **down arrow key** and the **<toggle>** action in my computer (running Debian Linux with Gnome terminal) it would look as follows:

```
\x1b[B      <toggle>
```

OPERATION MODES

[TO DO: this section is to be removed when I explain just how to use scripted mode]

efabulor has three operation modes: normal mode, scripted mode, and interactive scripted mode.

In normal mode, the user controls the program with the keyboard. The program reads single keystrokes from the standard input and converts them into actions according to a binding table (either the default one –see below– or a key binding table provided by the user –see below). Informational messages and error messages are printed to standard output and standard error.

In scripted mode, the program reads and executes newline-terminated commands describing actions directly in its internal scripting language (see below). Informational messages and error messages are printed to standard output and standard error as in normal mode. In scripted mode, the normal commanding by single keystrokes is suppressed. Scripted mode is activated by calling the program with the --scripted command-line argument. In this mode, **efabulor** can be used as a backend for an external process running as a frontend (e.g., a graphical user interface). The external process controls the program by writing newline-terminated commands to its standard input.

The interactive scripted mode is like the scripted mode, but with the user acting as the ‘external process’ and typing commands to the standard input. Informational messages and error messages are printed to standard output and standard error. (The interactive mode is for debugging and experimentation.) Using the default key bindings, you can activate this mode by typing **!** (an exclamation point). Some functionalities do not work in interactive mode.

The key binding table associates keystrokes to actions; actions are simply a sequence of one or more commands in the internal scripting language.

[TO DO]

CONFIGURATION FILES

[TO DO: verify all configuration files are explained somewhere else and delete this heading.]

GLOSSARY

[TO DO]