OVERVIEW

efabulor is a Python script for reading text files through **espeak** in a controlled manner.

PROGRAM INVOCATION

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

DESCRIPTION

efabulor is a wrapper around **espeak**, a freely-available speech-synthesis engine (https://espeak.sourceforge.net). It reads a (plain) text file, splits it into sentences, and sends them to **espeak** to be read aloud one by one. The reading process can be controlled with the keyboard (moving back and forward, stopping/pausing/resuming the reading, searching, going to a specific line, etc.). Non-plain text files can also be read by defining conversion pipelines, which will be called internally by **efabulor**.

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 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 before it is read by **efabulor**, e.g., to add newline characters at selected points to control the splitting of sentences. Transformation rules can also be used to apply any kind of preprocessing to the text. Transformation rules are defined in a simple language which is interpreted by a companion tool called **efabtrans** (which is called internally by **efabulor**). Alternatively, you can apply an external preprocessing filter to the input text before it is read by **efabulor**. Preprocessing filters and transformation rules can be mixed.

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 used by **espeak**. Substitution rules are defined in a simple language similar to the one used to define transformation rules. However, they are interpreted directly by **efabulor**, not by a companion tool. Another difference between transformation rules and substitution rules is that transformation rules are applied to the text as a whole before **efabulor** reads and splits the text; substitution rules are applied on a sentence-by-sentence basis, and only when they are actually needed (just before reading a 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/pipeline 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 and have **efabulor** call the conversion tool internally.

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 (specifically, Ubuntu and Debian). It might run in macOS, BSD, etc., but it's not guaranteed to do so.

It has been adapted to run (but not thoroughly tested) in Windows. Some functionalities may be missing or have a lower performance. In case of trouble, perhaps you can run **efabulor** within the **Cygwin** environment (`https://www.cygwin.com`), but please note that I haven't tested this option myself. Alternatively, you can install a Linux distribution within a virtual machine environment (such as **VirtualBox**, `https://www.virtualbox.org`) or within the new **Windows Subsystem for Linux** (`https://learn.microsoft.com/en-us/windows/wsl/install`).

LIMITATIONS

Runtime configuration

Most runtime options are preconfigurable through arguments in the command line, but the option to read those options from a configuration file is missing. (It might be added in a future version.) If you don't want to write the same options each time you call the program, you can create an alias or a shell function (in Linux) or a batch file or script (Windows and Linux) to call **efabulor** with the desired options.

File size

efabulor is not meant to be used with very large files (longer than a few hundred pages). Support for very large files might be included in future versions if the users require it, but it is not a high priority right now.

Substitution and transformation rules

You can specify as many substitution rule- and transformation rule-files (see Splitting and transformation rules and Substitution rules) as you wish, but when you try to open those files from within **efabulor**, only the first nine of either class will be accessible through menus.

Performance in Windows

The program has *not* been thorougly tested under Windows. Some functionalities may be missing or have a lower performance. The program depends crucially on being able 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 experience long delays between consecutive sentences, the problem might be that new processes are not being created quickly enough. 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

Since Python 3.6, the L flag cannot be used with string patterns. Please abstain from using that flag (for example, with the `--regex-flags` option) until I find a way to fix the problem.

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

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

DEPENDENCIES

Minimal

Python (https://www.python.org). An up-to-date version is advisable; **efabulor** version 1 should work fine with Python 3.7; for version 2 it is better if you have at least Python 3.9.

espeak (https://espeak.sourceforge.net; any up-to-date version should work fine)

Recommended

mbrola (https://github.com/numediart/MBROLA) and the **mbrola** voices for the languages you intend to use, in order to get a more humanlike reading performance

libreoffice (https://www.libreoffice.org) and **unoconv** (https://github.com/unoconv/unoconv) to simplify extraction of text from non-plain text files

Only in Windows: either **psutil** or **pssuspend**, in order to be able to pause/unpause the reading.

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

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

In the following instructions, all URLs are given as of the time of this writing. If some link is broken, please use an Internet search engine and your best judgment.

1. Copy all **efab*.py** files to a directory included in the search path for executable files. It is recommended to make **efabulor.py**, **efabconv.py**, **efabtrans.py** and **efabucw.py** executable, so you can call the scripts by name.

   Linux: `chmod +x <name of Python file>`

   Windows: you can make scripts executable by name with the `PATHEXT` variable. Please refer to online documentation.

   In any case, making the scripts executable is not mandatory, as you can always run **efabulor** or any Python3 script by prefixing `python3`, e.g.:

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

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

3. Check that you have Python >= 3.7 (and preferrably >= 3.9) installed:

   ```
   python3 --version
   ```

   Linux: In the extremely unlikely event that you don't have Python3 installed, you can get it from a standard repository (if you have an older Python installed, you might want to install **python3-venv** and put the latest version of Python 3 inside a virtual environment, so it does not interfere with the older installation. See: https://docs.python.org/3/library/venv.html).

   Windows: you can get Python from https://www.python.org and follow the installation instructions there.

4. **Linux:** install **espeak** (either install it from the standard repositories using your default package manager or download it from https://espeak.sourceforge.net and follow the instructions there).

   For example, to install **espeak** from the standard repositories in Debian:

   ```
   sudo apt-get install espeak
   ```

   Windows: download **espeak** from https://espeak.sourceforge.net and follow the instructions there.

5. (Recommended to get a more humanlike reading.)

   Linux: install **mbrola** (either install it from the standard repositories using your default package manager or download it from https://github.com/numediart/MBROLA and follow the instructions there).

   For example, to install **mbrola** from the standard repositories in Debian:

   ```
   sudo apt-get install mbrola
   ```

   Do not forget to install the voices for the language(s) you will normally use.

   For example, to install a Spanish voice for **mbrola** from the standard repositories in Debian:

   ```
   sudo apt-get install mbrola-es1
   ```

   Windows: download **mbrola** from https://github.com/numediart/MBROLA and follow the instructions there. Do not forget to install the voices for the language(s) you will normally use.

   Note: sometimes getting **mbrola** to work with **espeak** is not easy. Please refer to the developers' website if you find any trouble.

6. (Recommended to be able to read non-plain text files such as ODT, DOC, DOCX and RTF.)

   Linux: install **libreoffice** and **unoconv** (either install them from the standard repositories using your default package manager or download them from https://www.libreoffice.org and https://github.com/unoconv/unoconv and follow the instructions there).

   For example, to install **libreoffice** and **unoconv** from the standard repositories in Debian:

   ```
   sudo apt-get install libreoffice
   ```

   ```
   sudo apt-get install unoconv
   ```

   Windows: download **libreoffice** from https://www.libreoffice.org and **unoconv** from https://github.com/unoconv/unoconv and follow the instructions there.

   Linux/Windows: You can also install other format conversion programs that you may find useful; for example, I sometimes use **pandoc** (https://pandoc.org) as an alternative to **unoconv**. I cannot provide you any particular instructions here.

   Linux/Windows: you might install the new **unoserver** (https://github.com/unoconv/unoserver) conversion program instead of **unoconv**. I do not have experience with it yet and cannot provide you any instructions for the time being.

7. (Only in Windows; not mandatory, but required in order to be able to pause/unpause the reading).

   Install either **psutil** or **pssuspend**.

   To install **psutil** (the first line is only to ensure **pip** is working):

   ```
   python3 -m ensurepip --upgrade
   ```

   ```
   python3 -m pip install psutil
   ```

   To install **pssuspend**, download it from Microsoft's website at https://docs.microsoft.com/en-us/sysinternals/downloads/pssuspend and copy it to the executable path. You don't need to pay attention to the instructions for using **pssuspend** given there, as **efabulor** will take care of calling **pssuspend** when needed.

If you find any error in these instructions, or if you would like to share detailed installation 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 do not know how to use a terminal in Windows, 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, provided you copied **efabulor.py** to a directory within the search path for executable files):

```
[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 Installation). 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, make some changes to the file and save it. 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 Reconfiguring key bindings.

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 (the part in brackets is optional in Linux, provided you copied **efabulor.py** to a directory whithin the search path for executable files):

```
[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 into 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 into sentences.

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 rule 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 will be 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 only one line, and the part in brackets is optional in Linux, provided you copied **efabulor.py** to a directory within the search path for executable files):

```
[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 rule files are similar to transformation rule files: both 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 into sentences; substitution rules are applied on a sentence-by-sentence basis, after the text was split. And if you change the substitution rule 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 (necessary 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 to wait 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).

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. If the 'stop after current line' option (see below) is set, reset it.
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.
<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 behaviors only work if either **pssuspend** or **psutil** are installed.)
v	Go to the first line in the current sequence mode (see Sequence modes). 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 (see Sequence modes) and stop there.
m	Go to the last line in the current sequence mode (see Sequence modes). 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 (see Sequence modes) and stop there.

Key	Action
n	Go to the next line in the current sequence mode (see Sequence modes). 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 (see Sequence modes) 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 (see Sequence modes). 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 in the current sequence mode (see Sequence modes), 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.
/	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. (The set of modified lines is updated each time the underlying file changes on disk.)

Key	Action
>	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. (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.
w	Print the current line again.
l	('el') 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.
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 show 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 show 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.
_	(Underscore) Stop the player if it is running and show the list of the 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.

Key	Action
+	Increase the reading speed by 10 words/minute; if the player is running, restart reading the current line.
-	(Hyphen) 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	(Big 'oh') 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, executing again the whole input pipeline.
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 Sequence modes)
=	Stop the player if it is running and prompt the user to choose a feedback mode. (See Feedback modes)
0	(Zero) 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	(One) 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 command being run to get the input text (the 'input pipeline'). This is for debugging purposes.
I	Stop the player if it is running and show the command being run to get the input text (the 'input pipeline'). This is for debugging purposes.

Note: the association between keys and actions (key bindings) is user-configurable; see Default key bindings and Reconfiguring key bindings.

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

Key	Action
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
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
w	Print the current line
l	(‘el’) 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
_	(Underscore) Open command-line monitored files
u	Show substitution log
j	Show transformation log
+	Increase the reading speed
-	(Hyphen) Decrease the reading speed

Key	Action
o	Open the input file
O	(Big ‘oh’) 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
=	Choose a feedback mode
0	(Zero) Set tracking mode=forward and feedback mode=full; stop after each line
1	(One) 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, you'll just have to modify the command a little (please read **unoserver**'s manual).

Sometimes, **unoconv** may fail to convert the given file the first time, and 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 can use whichever conversion tool you have installed and 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 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 standard output; otherwise, **efabulor** will have nothing to read.

You can specify conversion commands for different types of files in the 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 ISO code of the language 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 the following command:

```
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 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 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 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 rule file' containing the following:

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

The `[DO]` line marks the beginning of the transformation rules; it is optional when it is the first non-empty line in the file, 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.)

For more information about regular expressions as used by efabulor, see Regular expressions.

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 rule 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 one 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 loaded by **efabulor** into 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 Regular expressions. 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 rule 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. 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 those 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 print the result to 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 rule 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 marks and quotation marks 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
```

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]/' --do 'cat inputfile.csv'
```

(In Windows, write `type` instead of `cat`.)

> You must use `--do 'cat inputfile.csv'` instead of just the name of the file swhen no specific conversion command has been configured for CSV files. A CSV file is just a special kind of text file, so you can 'extract' its content by simply copying it to 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 CSV files.

And there is yet another alternative. Instead of telling **efabulor** which string or patterns separate readable units, you can provide a regular expression which defines what 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 Regular expressions):

```
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 used by translators which is based on XML) 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 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 rule 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'
```

(In Windows, write `type` instead of `cat`.)

You must use `--do 'cat inputfile.tmx'` instead of just the name of the file when no specific conversion command has been configured for TMX files. A TMX file is just a special kind of text file, so you can 'extract' its content by simply copying it to 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.

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 the forward slashes that mark the closing tags in TMX with a backslash.

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 rule files cannot contain sections. Substitution rule 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 filters), 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' each time the input text contains 'foo'. You can define a substitution rule file, say **substrules.txt**, with the following content:

```
foo
bar
```

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

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

You can use more than one substitution rule 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 what is said there applies also to substitution rules.

Why would you use substitution rules? One reason is that sometimes you may find that the result of the underlying text-to-speech conversion algorithm is unsatisfactory or wrong. For example, when using the **spanish-mbrola-1** voice to read text in Spanish (with the current version as of this writing), there is a minor glitch every time the 'll' digraph occurs in the input text. 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.)

Quality assurance rules (such as the one given above) can be stored in a substitution rule file when they can be applied on a segment-by-segment basis. If on the other hand, the regular expression has to be matched against the whole text, you should place such rules in a transformation rule file. Experience will teach you the proper way to do it.

Temporarily disabling substitution rules

When using substitution rules, you can temporarily disable their application by pressing **S**.

Press **S** again to re-enable substitutions.

Showing the effect of substitution rules

When using substitution rules, you can ask **efabulor** to show the effect of substitution rules, instead of the 'raw' input text, by pressing **D**. Press **D** again to go back to the normal mode of showing the input text before substitutions.

Editing substitution rules on the fly

When using substitution rules, you can ask **efabulor** to open the substitution rule file in an editor window by pressing **s**. If there is only one substitution rule file being used, efabulor will try to open it in the application registered in your system for editing plain text files. If more than one substitution rule file is being used, you will be presented with a menu of files to choose from.

Change tracking

One of the core functionalities of **efabulor** is its ability to monitor the text being read in real time and provide instant feedback of changes. The monitored files whose modification on disk could result in a change to the text being read are:

- the input file itself
- the transformation rule files
- the substitution rule files

You can also ask the program to monitor other files (and even directories) with the `--monitored-file` option.

Understanding the default monitoring behavior, and the ways you can adapt it to your needs, is essential for a proper use of **efabulor**.

Default monitoring and tracking behavior

Once started, **efabulor** will check the monitored files for changes every 2 seconds. A file is considered potentially changed when its timestamp on disk is modified.

When the timestamp of a monitored file changes on disk, **efabulor** executes the input pipeline again and loads the new version of the input text.

The new text is compared with the previous version. If (a) there are no actual changes, or (b) the first changed line is after the current one, nothing happens. If the player was stopped, it will remain so; if it was playing, it will keep playing. You will hear the newly changed line(s), if any, once the player gets there.

If changes are detected at or before the current line, the player is repositioned at the first changed line (this is called *tracking*), a brief notification is read aloud, and the reading is (re)started. If the file only gets shorter (but no other changes are detected) and the new file ends before the current position, the player is repositioned to the end of the file and stops there.

In either case, information on the detected changes (or lack thereof) will also be printed onscreen.

Modifying the monitoring and tracking behavior

You can adjust several parts of the default monitoring/tracking behavior.

For starters, you can add files or directories to be monitored to the default list with the

`--monitored-file` option. This can be useful, for example, if your input pipeline depends on an external program whose behavior is controlled by a configuration file; you can monitor the configuration file to re-execute the input pipeline as needed.

You can set a different monitoring interval (in seconds) with the `--monitoring-interval` option followed by a positive integer. Note that setting too big an interval will prevent **efabulor** from providing you timely feedback on changes (you can use a very big number to *disable* monitoring if you wish so).

When a timestamp modification is detected, the input pipeline is executed immediately. You can use the `--no-reload-when-stopped` command line option to ask **efabulor** not to execute the input pipeline when the player is stopped. The input pipeline will be executed when you restart the player. (Currently, there is no way to prevent the input pipeline from being executed if the player is playing.)

If the player is stopped when a tracking event occurs, the reading is restarted automatically after repositioning the player. You can use the `--no-restart-after-change` option to tell **efabulor** only to reposition the player, without automatic restarting.

Conversely, you can use the `--restart-on-touch` option to have the reading (re)started whenever the timestamp of a monitored file changes on disk, even if there are no actual changes. The reading will restart at the current line, unless the player was repositioned because of an actual change, or unless the tracking mode (see below) is set to **restart**, in which case the reading will restart at the first line. Note: this option takes precedence over `--no-restart-after-change`.

Finally, when the player is stopped and a substitution rule file changes, with effect on the current line, the reading is not automatically restarted. You can use the `--restart-after-substitution-change` to have the reading automatically restarted in response to changes to the substitution rules.

You can also configure the tracking response and the feedback level as described in the following sections.

Tracking modes

There are four tracking modes: **backward**, **forward**, **restart** and **none**. The default mode is **backward**; you can choose another mode by calling **efabulor** with the `--tracking-mode` option. The tracking mode can also be changed while the program is running by pressing **?**.

In **backward** mode, the player will be repositioned to the first changed line if it is at or before the current line.

In **forward** mode, the player will be repositioned to the first changed line whatever its location.

In **restart** mode, the player will always be repositioned to the first line of the input text.

If tracking mode is set to **none**, the player will not be repositioned in response to changes, but auditory feedback will still be provided according to the chosen feedback mode, as explained below.

Feedback modes

Feedback modes specify the amount of auditory feedback you will receive when the underlying files change. There are three feedback modes: **none**, **minimum** and **full**. Minimum feedback mode is the default. In minimum feedback mode, **efabulor** will only play a short message to tell you that the reading is being restarted or that the player must jump to another sentence. When feedback mode is **none**, no informative message is played; if the player was running and will continue at another sentence, the reading is interrupted abruptly and restarted without

notice. With feedback mode set to **full**, the auditory feedback will include the 'minimum' messages plus information about the detected changes: whether there are changes to one or many lines, if the length of the file has decreased/increased, etc.

Configuring locale-specific feedback messages

The feedback messages which are read in response to changes are not hardcoded within **efabulor** but configured in a locale configuration file as described below.

Locale-specific configuration file

Several options which depend on the language being used for reading can be configured in a locale-specific configuration file. Currently you can only configure the preferred voice for the language and the feedback messages used when changes 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: simply replace your preferred voice and translate 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:

Feedback messages given 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**.

* With `--no-restart-after-change` activated, the `changes-here` message will be played

instead of the `restarting` message.

** With `--no-restart-after-change` activated, the `jumping-forward` message will be played instead of the `continuing` message.

Feedback messages given 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 input text's line count has increased.
`file-decreased`	the input text's line count has decreased.
`file-too-short`	the input text'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.

* These messages are also played with feedback mode set to **minimum** if the player will not restart after the tracking event.

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 with the `--config-file` (or `-c`) option. For example:

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

Alternatively, you can set a default locale configuration file 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 must be in the following format:

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

For example (for German):

```
efabulor.de.cfg
```

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 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.)

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. Future versions might include a design based on gettext also for the feedback submodule and the locale configuration files might be removed.

Sequence modes

In addition to the normal sequence mode, in which sentences are read in the order they appear in the input file, **efabulor** supports two other sequence modes: **modified** and **random**. You can choose the sequence mode by pressing the **)** key. You can also specify a sequence mode in the command line with the `--sequence-mode` option followed by one of `normal`, `modified`, or `random`.

Sequence mode: random

The **random** sequence mode is what its name implies: sentences are read in a random order. In this mode, if you ask **efabulor** to go to the next sentence (pressing **n** with the normal key binding), a new random sentence will be chosen. **efabulor** keeps track of the last 100 sentences read in random mode (there is no option to change that limit in this version). If you ask **efabulor** to go back (pressing **b** or **B** with the normal key binding), **efabulor** will go to the previous sentence read in random mode, or to a new random sentence if you go past the history's limit.

Note: in normal mode, you can also explore the sentences in random order using the * key.

Sequence mode: modified

The **modified** sequence mode is based on the change tracking feature of **efabulor** (see Change tracking). Whenever the underlying file is modified, **efabulor** checks the new version against the old one and computes a list of sentences that were changed. In modified sequence mode, sentences will be read in the order they appear in the latest list of changes; unchanged sentences will be skipped. Going backward and forward refers to the order of the sentences in the list of modified sentences.

Note: in normal mode, you can also explore the sentences according to the latest list of changes using the **<** and **>** keys.

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 binding 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.

Default key bindings

The default key-bindings are as follows (`\x20` is the code for space; more information on the codes is given following the table). The meaning of the actions is described in Keyboard control.

Key	Action
`Q`	`quit-now`
`q`	`quit-ask`
`\x20`	`toggle`
`a`	`restart`
`A`	`restart-and-stop`
`x`	`stop-and-reset`
`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`

Key	Action
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 binding configuration file, printable characters stand for themselves; hexadecimal codes such as `\x01` and escape sequences such as `\n` stand for control characters or normal characters, according to their ASCII value. For example, `\x01` stands for the ASCII character codified as 1 in the ASCII chart (which you can enter by pressing **Ctrl+A** in most terminals). The code `\x20` stands for the **spacebar**. The code `\x0a` and the escape sequence `\n` both stand for the **Enter** key; and so on. (Information on control characters and escape sequences can be found at: https://en.wikipedia.org/wiki/ASCII). Note: you cannot use the code `\x03` (**Ctrl-C**) because it is reserved for unconditional termination of the program.

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

To define different key bindings for the actions, you can simply overwrite the key specifications in the configuration file. Separate the key specification and the name of the action with

whitespace (any combination of one or more spaces and/or tabulation characters will work, but one tabulation character is recommended for visual alignment). 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, **efabulor** will refuse to use the configuration file.

To use special non-printable keys, e.g. arrow keys, function keys, etc., see Creating key bindings interactively below.

Once you finish editing the key binding 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 with the `--add-key-bindings` option instead of the `--key-bindings` option. For example, suppose you want to retain the default key bindings, but add a new key binding to pause/unpause the player with the **Enter** key. You can create a file with the following content:

```
\n     toggle
```

To add the new key binding to the default one, call **efabulor** with the `--add-key-bindings` option:

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

Creating key bindings interactively

To create a key binding configuration file interactively, call **efabulor** with the `--edit-key-bindings` option followed by a filename. You will 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 include non-printable keys (e.g., arrow keys, function keys) whose codes depend on the specific type of terminal you are using (the current version of the program does not allow to use mnemonic names for keys, such as `Ctrl-C`; that option might be added in a future version).

Note that a key binding configuration file containing codes for non-printable keys might not work in 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
```

Regular expressions

Several features of **efabulor** allow using regular expressions to look for text patterns:

- To define the sentence separator (`--separator` command-line option).
- To define sentences themselves (`--segment` command-line option).
- Within substitution rules (`--subst` command-line option).
- Within transformation rules (`--transform` command-line option).
- To search for text.

Because **efabulor** is written in Python, the regular expression language used is basically the same as Python's. For a full presentation of Python's regular expressions, see https://docs.python.org/3/library/re.html. (You might need to read that to understand what follows.)

The following points must be taken into account when using regular expressions with **efabulor**:

Delimiters: In places where a regular expression must be surrounded by delimiters to distinguish it from a plain text pattern (e.g., in transformation rules and substitution rules), any of the following characters can be used as a delimiter:

```
/      _      :      %      "      !      @
```

The same character must be used at the beginning and at the end of the regular expression; when defining a substitution regular expression (in transformation rules and substitution rules), the same character must be used to separate the pattern from the replacement (e.g., `%substitute this%with this%`). This behavior is similar to **sed**.

Scope: regular expressions are matched against the whole input text (before segmentation) or against separate lines of text (after segmentation). This affects the meaning of the `^` and `$` anchors and whether the `s` (single line) flag must be used to make `.` (the dot pattern) match against newline characters.

Regular expressions used in a…	**are matched against…**
search pattern	each separate sentence in the text
segment pattern	the whole input text
separator pattern	the whole input text
substitution rule	each separate sentence in the text
transformation rule	the whole input text

Default flags: to set a default compilation flag to be applied to all regular expressions within transformation rules, substitution rules and search patterns, call the program with the `--regex-flags` switch, followed by the desired flags. For example:

```
efabulor.py --regex-flags=sm <other options…> <file to read>
```

The available flags are:

Flag	**Meaning**
`i`	regular expressions are case insensitive
`s`	`.` (dot pattern) matches newline
`m`	`^` and `$` match at the beginning and end of a line
`u`	Unicode matching
~~`l`~~	~~Locale dependent matching~~

Note: the default flags do not apply to regular expressions declared in the command line, that is, with the `--segment` and `--separator` switches. To apply flags to these regular expressions, you must include them within the regular expression (see below).

Note: Since Python 3.6, the L flag cannot be used with string patterns. Please abstain from using that flag until I find a way to fix the problem.

Flags within a regular expression: you can use the same ~~five~~ four flags discussed above within a regular expression, to apply that flag only to that regular expression, perhaps overriding a default flag. You can use either Python's syntax:

```
/(?s)pattern/
```

or Perl/sed-like syntax:

```
/pattern/s
```

When using the second syntax, you can prefix a - (minus sign) to a flag to unset that flag for this regular expression if it was set by `--regex-flags`; or you can include a special `0` (zero) flag to unset all default flags. For example:

`/pattern/-su`	Unset the `s` flag if set by `--regex-flags`, and set the `u` flag.
`/pattern/-s-u`	Unset the `s` and `u` flags if set by `--regex-flags`
`/pattern/0`	Unset all flags set by `--regex-flags`
`/pattern/0i`	Unset all flags set by `--regex-flags`, and set the `i` flag.

Passing options to espeak

You can call **efabulor** with the `--opt` command line switch and a string containing options that will be passed directly to **espeak**. For example, to adjust the pitch and have the names of punctuation marks spoken out, you could call **efabulor** with the command line shown below:

> CAUTION: **efabulor** will try to detect obviously wrong options, but in most cases the string you provide with `--opt` will be tokenized and passed to **espeak** as provided. For example, if the line being read is 'Mary had a little lamb.', the voice is 'english', the speed is the default speed (180), and you give the options shown below, **efabulor** will create a subprocess with the following call:
>
> `espeak -s 180 -v english --punct -p 99 'Mary had a little lamb.'`
>
> Be careful to include only valid, well-formed options that **espeak** will accept; a malformed `--opt` might have undesired side-effects (most probably, prevent **espeak** to run properly). The current version of **efabulor** doesn't call **espeak** through a shell, but better be safe than sorry: pay extra attention when using this command switch, as it *could* be passed to the shell in a future version.

```
efabulor.py --opt='--punct -p 99' <file>
```

Scripted mode

efabulor includes a scripted operation mode, in which it can be invoked and controlled by an external program. This mode is not intended for normal use; it is included to have a relatively easy way to create an external graphical user interface (GUI) for the program.

In scripted mode, normal keypress-driven operation is disabled. Instead, **efabulor** will read newline-terminated commands (actions) from standard input (while sending normal output and errors to standard output and standard error as usual). The available commands (actions) are described in Default key bindings.

To invoke scripted mode, simply add the `--scripted` option to the program's normal invocation. E.g.:

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

If you want to experiment with scripted mode, call **efabulor** with the `--scripted` option as above and start typing commands as described in Default key bindings (one command per line followed by **Enter**). Beware that your typing may get intermingled with **efabulor**'s output; don't pay attention to the output. Also, the output will show special prefixes which are not shown in normal mode (see Writing a GUI below). To quit the program, type `quit-now` followed by **Enter**. In case of trouble, you can always terminate the program by pressing **Ctrl-C**.

Functionalities disabled/modified in scripted mode

Some functionalities are disabled or changed in scripted mode.

The `open-shell` action is disabled in scripted mode (if the user were to invoke a shell from a GUI, the code to do so could be implemented within the GUI).

The following functionalities are changed in scripted mode:

- `quit-now` and `quit-ask` both quit the program without asking for confirmation.
- The menus shown when invoking the `choose-tracking-mode`, `choose-sequence-mode`, `choose-feedback-mode` and `search` actions accept the same numeric replies as in normal mode, but they must be followed by a newline character (**Enter**).
- In normal mode, the substitution and transformation logs are sent through a pager; in scripted mode, they are sent directly to standard output (each line is prefixed with the string `[NormEx]`; see below).

Writing a GUI

The current version of **efabulor** is a command-line tool, and there are no plans to provide a built-in graphical user interface (GUI). A GUI can be provided by writing an external program to act as an interface between the user and **efabulor**. The external GUI should invoke **efabulor** with the `--scripted` option, send commands to **efabulor**'s standard input as described above and process any information printed by **efabulor** to standard output and standard error.

efabulor sends messages to either standard output or standard error according to the type of message. In normal mode, a simplified prefix such as > and `[!]` is added before the message, just for the user's convenience; the same prefix may be used for more than one type of message. In scripted mode, a special prefix (followed by a space) is added to the output, and prefixes are unique, to allow for easy identification of the type of message by the GUI. It is the GUI's responsibility to filter out **efabulor**'s messages, strip the prefixes if desired, and either ignore the messages or show them in different text fields within the GUI.

Note: the surrounding brackets are part of the prefix:

Type of message	**Sent to**	**Prefix**

Normal output (lines being read):	Standard output	`[Normal]`
Normal extended output (only used for substitution and transformation logs)	Standard output	`[NormalEx]`
Informational output (describing what **efabulor** is doing at a certain time)	Standard output	`[Info]`
Informational extended output (reserved for a future version)	Standard output	`[InfoEx]`
Errors	Standard error	`[Error]`
Extended error messages (error messages printed by external processes, e.g. by **espeak**)	Standard error	`[ErrorEx]`
Prompts (shown to the user when expecting user's input)	Standard output	`[Prompt]`
Pause prompt (shown when **efabulor** is honoring a pause before or between lines)	Standard output	`[Pause]`

Command line options

```
efabulor [-h] [--version] [--lang LANG] [--encoding ENCODING]
[--config-encoding CONFIG_ENCODING]
[-c <configuration file for voice and feedback messages>]
[-C <configuration file for text conversion> SECURITY WARNING: read the
 manual before using this option!]
[-v <name of the voice to be used>]
[-p [<pause length in seconds>]]
[-P [<pause length in seconds>]]
[-m <time to wait before checking files for modifications, in seconds>]
[-M <name of a file that will be checked for modifications>]
[--no-reload-when-stopped] [--no-restart-after-change]
[--restart-after-substitution-change] [--restart-on-touch]
[--no-restarting-message-when-not-playing]
[-t {none,backward,forward,restart} | --sequence-mode
 {normal,modified,random}]
[-F {none,minimum,full}] [-u] [-l] [-L] [-E] [-f]
[-s <speed for reading, in words/minute>] [-,] [-Q]
[-S <file with substitution rules>] [-r]
[--transformation-rules <file with transformation rules>]
[--preprocess <a filtering command to preprocess the input> SECURITY
 WARNING: read the manual before using this option!]
[--separator <string or regular expression to be used as a separator> |
 --segment <regular expression for segmenting>]
[--regex-flags <global flags for regular expressions>]
[--no-echo] [--no-showline] [--no-info] [--no-update-player]
[--no-reset-scheduled-stop-after-moving]
[--left-indent [<how many spaces>]]
[--right-indent [<how many spaces>]]
[--window-width-adjustment [{0,1}]]
[-k <configuration file for converting keystrokes to commands> |
 --add-key-bindings <additive configuration file for converting
 keystrokes to commands> | --scripted]
[--opt <options that will be passed to espeak> NOTE: read the manual and
 use with caution]
[-K [<file where the default configuration for keystrokes will be
 saved>]]
```

```
[--edit-key-bindings [<file where the default configuration for
 keystrokes will be saved/appended>]]
[--do <command to create text to be read> SECURITY WARNING: read the
 manual before using this option!]
[<file to be read>]

positional arguments:
<file to be read>

optional arguments:
-h, --help            show this help message and exit
--version             show program's version number and exit
--lang LANG
--encoding ENCODING
--config-encoding CONFIG_ENCODING
-c <configuration file for voice and feedback messages>, --config-file
 <configuration file for voice and feedback messages>
-C <configuration file for text conversion> SECURITY WARNING: read the
 manual before using this option!, --text-conversion-config
 <configuration file for text conversion> SECURITY WARNING: read the
 manual before using this option!
-v <name of the voice to be used>, --voice <name of the voice to be used>
-p [<pause length in seconds>], --pause-before [<pause length in
 seconds>]
-P [<pause length in seconds>], --pause-between [<pause length in
 seconds>]
-m <time to wait before checking files for modifications, in seconds>,
 --monitoring-interval <time to wait before checking files for
 modifications, in seconds>
-M <name of a file that will be checked for modifications>,
 --monitored-file <name of a file that will be checked for
 modifications>
This option can appear multiple times.
--no-reload-when-stopped
--no-restart-after-change
--restart-after-substitution-change
--restart-on-touch
--no-restarting-message-when-not-playing
```

```
-t {none,backward,forward,restart}, --tracking-mode
 {none,backward,forward,restart}
--sequence-mode {normal,modified,random}
-F {none,minimum,full}, --feedback-mode {none,minimum,full}
-u, --show-subst
-l, --show-line-number
-L, --show-total-lines
-E, --close-at-end
-f, --force-execution
-s <speed for reading, in words/minute>, --speed <speed for reading, in
 words/minute>
-,, --stop-after-each-line
-Q, --quit-without-prompt
-S <file with substitution rules>, --substitution-rules <file with
 substitution rules>, --subst <file with substitution rules>
This option can appear multiple times. The files will
be processed in the given order.
-r, --raw, --do-not-apply-substitutions
Apply substitutions only when asked by the user.
--transformation-rules <file with transformation rules>,
--transform <file with transformation rules>
This option can appear multiple times. The files will
be processed in the given order.
--preprocess <a filtering command to preprocess the input> SECURITY
 WARNING: read the manual before using this option!
This option can appear multiple times. The filters
will be applied in the given order.
--separator <string or regular expression to be used as a separator>
--segment <regular expression for segmenting>
--regex-flags <global flags for regular expressions>
--no-echo
--no-showline
--no-info
--no-update-player
--no-reset-scheduled-stop-after-moving
--left-indent [<how many spaces>]
--right-indent [<how many spaces>]
--window-width-adjustment [{0,1}]
Default is 1 in Windows, 0 in Linux.
```

```
-k <configuration file for converting keystrokes to commands>,
 --key-bindings <configuration file for converting keystrokes to
 commands>
--add-key-bindings <additive configuration file for converting keystrokes
 to commands>
--scripted
--opt <options that will be passed to espeak> NOTE: read the manual and
 use with caution
This option must use the following syntax:
--opt='-opt1 -opt2 ...'
-K [<file where the default configuration for keystrokes will be saved>],
 --save-default-key-bindings [<file where the default configuration for
 keystrokes will be saved>]
--edit-key-bindings [<file where the default configuration for keystrokes
 will be saved/appended>]
--do <command to create text to be read> SECURITY WARNING: read the
 manual before using this option!
```

Detail of command line options

Mandatory mutually-exclusive arguments (you must enter exactly one of them)	**Meaning**
<an input file>	An input file to be read
-K [FILE] --save-default-key-bindings [FILE]	Save default key bindings to this file. Print default key bindings to standard output if no file is given. See Reconfiguring key bindings.
--edit-key-bindings [FILE]	Create key bindings interactively and save them to FILE. Print created key bindings to standard output if no file is given. See Creating key bindings interactively.
--do COMMAND	Execute the given command to get the input text. See Read the output of a command. SECURITY WARNING: read that section of the manual before using this option!

Optional arguments	**Meaning**
-h --help	Show a help message and exit
--version	Show the program's version number and exit

Optional arguments	**Meaning**
`--lang LANG`	Specify the language of the input text, as an ISO language code (e.g., `ES`, `EN`, etc.). The language will be used to select the voice for reading. If this option is not given, the system's locale is used.
`--encoding ENCODING`	Encoding for the input text (e.g., `utf8`, `iso_9959-1`, etc.). If this option is not given, the system's default encoding is used.
`--config-encoding FILE`	Encoding (e.g., `utf8`, `iso_9959-1`, etc.) used for substitution rule files, transformation rule files, key bindings files, locale configuration files. If this option is not given, the system's default encoding is used.
`-c FILE` `--config-file FILE`	Locale configuration file, specifying the voice to be used for a specific language and/or the feedback messages to be used for a specific language. See Locale-specific configuration file.
`-C FILE` `--text-conversion-config FILE`	Text conversion configuration file, specifying text extraction pipelines to be used when the input file is not plain text. See Preconfigured conversion commands. SECURITY WARNING: read that section of the manual before using this option!
`-v VOICE` `--voice VOICE`	Name of the voice to be used. The names of the voices are specific to **espeak**/**mbrola**, and they can be found by executing: `espeak --voices` If this option is given, it supersedes the voice configured in the locale configuration file.
`-p [LENGTH]` `--pause-before [LENGTH]`	Pause LENGTH seconds before starting to read, when the player is restarted from being stopped. If the option is given and no length is specified, a default pause of 3 seconds will be set.
`-P [LENGTH]` `--pause-between [LENGTH]`	Pause LENGTH seconds between lines. If the option is given and no length is specified, a default pause of 3 seconds will be set.
`-m LENGTH` `--monitoring-interval LENGTH`	Interval, in seconds, for checking files for modifications. See Change tracking.
`-M FILE` `--monitored-file FILE`	Add this file to the default list of files checked for modifications. See Change tracking.

Optional arguments	Meaning
`--no-reload-when-stopped`	When the modification timestamp of a monitored file changes on disk, the input file is reloaded again to check for actual changes to the input text. This option disables file reloading if the timestamp of the file being monitored changes while the player is stopped. See Change tracking.
`--no-restart-after-change`	When a tracking event occurs while the player is stopped, reading is restarted by default. This option disables automatic restarting. See Change tracking.
`--restart-after-substitution-change`	By default, when changes to substitution rules occur affecting the current line while the player is stopped, the reading is not restarted. When this option is set, substitution rule changes affecting the current line will trigger a restart. See Change tracking.
`--restart-on-touch`	By default, automatic restarting of reading is not honored when a monitored file modification timestamp changes on disk if the input text has not actually changed. When this option is set, the reading will be restarted when the file is 'touched' (the timestamp changes) even if there are no actual changes to the input text. See Change tracking.
`--no-restarting-message-when-not-playing`	By default, when the player is stopped and the reading is restarted after a tracking event, a 'restarting' feedback message will be played (except in 'no feedback' mode). Setting this option disables that feedback message (except if the tracking mode is 'restart'). See Change tracking, Feedback modes and Configuring locale-specific feedback messages.
`-t MODE` `--tracking-mode MODE`	Set the tracking mode at startup. MODE can be one of: `none`, `backward`, `forward`, `restart`. Default mode is `backward`. See Tracking modes.
`--sequence-mode`	Set the sequence mode at startup. MODE can be one of: `normal`, `modified`, `random`. Default mode is `normal`. See Sequence modes.
`-F` `--feedback-mode`	Set the feedback mode at startup. MODE can be one of: `none`, `minimum`, `full`. Default mode is `minimum`. See Feedback modes.

Optional arguments	**Meaning**
`-u` `--show-subst`	Set the ‘show substitutions’ mode at startup. Default mode is ‘do not show the effect of substitutions’. See Showing the effect of substitution rules.
`-l` `--show-line-number`	Set the ‘show line number’ mode at startup. Default mode is ‘do not show line number’.
`-L` `--show-total-lines`	Set the ‘show total lines’ mode at startup. Default mode is ‘do not show total lines’.
`-E` `--close-at-end`	Set the ‘close at end’ mode. When set, the program will exit without asking for confirmation after reading the last line.
`-f` `--force-execution`	Allow multiple instances to run. By default, **efabulor** will try to see if another instance is running, and will exit if it finds such an instance. (Note that sometimes **efabulor** might fail to find a running instance even if there is one, depending on how the other instance was started.) Use this command-line switch to have **efabulor** run even if there is another running instance.
`-s SPEED` `--speed SPEED`	Set the speed for reading in words/minute. Default speed is 180 words/minute.
`-,` `--stop-after-each-line`	Set the ‘stop after each line’ mode at startup.
`-Q` `--quit-without-prompt`	Set the ‘quit without prompt’ option. When set, **efabulor** will exit without asking for confirmation when the user presses the **q** or **Q** keys.
`-S FILE` `–substitution-rules FILE`	Use this substitution rule file. See Substitution rules.

`-r`
`--raw`
`--do-not-apply-substitutions`
	Set the ‘do not apply substitutions’ option at startup. Substitutions will only be applied after the user asks so. See Temporarily disabling substitution rules.

`--transformation-rules FILE` `--transform FILE`	Use this transformation rule file. See Splitting and transformation rules.
`--preprocess COMMAND`	Filter the input text through COMMAND before reading it. See Preprocessing filters. SECURITY WARNING: read that section of the manual before using this option!

Optional arguments	Meaning
`--separator SEPARATOR`	Set a string or regular expression as a separator between lines. See Segmentation and separator options.
`--segment PATTERN`	Set a regular expression to extract lines to be read from the input text. See Segmentation and separator options.
`--regex-flags FLAGS`	Set default flags to be used when compiling regular expressions used in substitution rules, transformation rules, and searching. See Regular expressions.
`--no-echo`	Set the 'no echo' option at startup. When this option is set, lines being read are not printed unless specifically asked by the user (by pressing the **w** key).
`--no-showline`	Set the 'no showline' option at startup. When this option is set, lines being read are not printed even if asked by the user.
`--no-info`	Suppress informational messages.
`--no-update-player`	Allow current line to be changed without restarting the player at the new line. Note: this is an experimental feature and it is likely to be changed or removed in future versions.
`--no-reset-scheduled-stop-after-moving`	By default, when the player is scheduled to stop after reading the current line, and the current line is changed, the scheduled stop is unset. When this option is set, a scheduled stop is not unset after changing the current line.
`--left-indent NUMBER`	Set a left indent for showing text. (Note: this indent does not apply to informational messages and error messages.)
`--right-indent NUMBER`	Set a right indent for showing text. (Note: this indent does not apply to informational messages and error messages.)
`--window-width-adjustment MODE`	Set the window width adjustment mode. MODE can be either 1 or 0. Default is 1 for Windows, 0 for Linux. Set this option if you see empty lines in the output after lines extending right up to the right end of the screen.
`-k FILE` `--key-bindings FILE`	Use this key bindings file. See Reconfiguring key bindings.

Optional arguments	Meaning
`--add-key-bindings FILE`	Add this key bindings file. See Reconfiguring key bindings.
`--scripted`	Run **efabulor** in scripted mode. See Scripted mode.
`--opt OPTIONS`	Specify a string containing additional options to be passed to the **espeak** instances initiated by **efabulor**. See Passing options to espeak. NOTE: read that section of the manual and use with caution.

GLOSSARY

Term	Meaning
change tracking	See *tracking*
espeak	A free speech-synthesis engine, used internally by **efabulor** to read the *source file* aloud.
feedback	Short messages that **efabulor** speaks out loud to report changes to the *source text* (*tracking events*). *Tracking* and feedback are essential for using **efabulor** as a text quality assurance tool while you edit the *source file*.
input pipeline	The actual command efabulor executes to extract readable text from the *source file*
key binding	The association between **efabulor**'s commands and keys (or combinations thereof). **efabulor**'s user-friendliness as a quality assurance tool depends mostly on being keypress-driven. Key bindings are configurable.
locale	Configurations which depend on the user's language and region; e.g., US-EN for English as spoken in the United States, ES-AR for Spanish as spoken in Argentina, etc.
mbrola	An utility that allows postprocessing the speech-synthesis output from **espeak** to produce a more humanlike reading. It is not mandatory to have it installed in order to run **efabulor** (nor **espeak**), but it is a recommended package for installation.
pipeline	See *input pipeline*
readable unit	See *segment*
regular expression	A rule, codified in a special language, used to search for text or match patterns of text. **efabulor** uses regular expressions to define *substitution rules* and *transformation rules*, and also to search within the *source text*.

Term	Meaning
segment	The minimum unit of text that **efabulor** sends to be read separately by **expeak**; a *sentence*. By default, **efabulor** splits segments at newline characters, but you can configure that behavior with the `--separator` and/or `--segment` command-line options.
segmenting	Dividing the source text into *segments*
sentence	See *segment*
sequence mode	A selection and ordering of *segments* (*sentences*) within the *source text*, used by **efabulor** when reading it (normal: read all the sentences as given in the source text; random: read the *sentences* in random order; modified: read only the *sentences* changed since the last *tracking event*).
source file	A file given to be read by **efabulor**.
source text	The readable content of the *source file*
splitting	See *segmenting*
substitution rule	A modification **efabulor** applies to each *segment* before reading it. You can use substitution rules to improve or tweak the reading of text by **efabulor**.
text conversion/extraction pipeline	See *input pipeline*
tracking	The fact that **efabulor** monitors the *source file* (and possibly others) for changes and responds to such changes. Tracking and *feedback* are essential for using **efabulor** as a quality assurance tool while you edit the *source file*.
tracking event	An event occurring in response to a change in the *source text* that might force **efabulor** to restart the reading, perhaps in another *segment*.
transformation rule	A modification **efabulor** applies to text immediately after *text extraction* but before *splitting* the text in *segments*. You can use transformation rules to alter the *splitting* of text into *segments*, or for any other modification you want to apply to the *source text*.
voice	The voice used by **espeak** to convert text into speech. Voices are language-specific.