The bird and the lion: arara

Paulo Roberto Massa Cereda

1 Prologue

There I was, back in 2011, with a huge project in my hands: a songbook. But it was far from any ordinary book due to the involved complexity: each song had several tags and at least 25 indexes, with different styles! Of course, TEX and friends were able to tackle this beast on their own, but I was not prepared. The lion was definitely hungry and I was the typographic meat provider.

My compilation workflow was striking: at least 30 to 40 steps in order to achieve the final result. As a first experiment, I wrote a nice `Makefile` and the problem had appeared to be solved once and for all. Suddenly, however, I found myself in need of a portable solution: I had to share my projects with at least three different operating systems (Windows, GNU/Linux and Mac OS X) and I should ensure that all the needed tools were in place for my workflow to work. Worse: I had to rely on system-dependent commands and other nuisances.

My first idea was to stand on the shoulders of giants and rely on the brilliant `latexmk` by John Collins; sadly, the workflow was too complicated for me to grasp at once, and my `.latexmkrc` shortly became a beast on its own. The second idea was to use `rubber` but, as my worst nightmares became true, at some point, I was writing ugly hacks and injecting Python code into the tool itself. Alas, no success, the songbook remained intractable.

When all else had failed, I decided to come up with a solution on my own. I sat in front of my computer with an open terminal and started to code while listening to Pink Floyd. In a couple of hours, a new tool was tackling my songbook.

I mentioned this journey in the chat room of the TEX community at StackExchange and Enrico Gregorio encouraged me to release this tool into the wild. Later on, Marco Daniel, Brent Longborough, Nicola Talbot and many, many others jumped in and a new project — arara — was born. The name was chosen as an homage to a Brazilian bird of the same name, which is a macaw. The word *arara* comes from the Tupian word *a'rara*, which means *big bird* (much to my chagrin, Sesame Street's iconic character Big Bird is not a macaw; according to some sources, he claims to be a golden condor). As I mention in the user manual, araras are colorful, noisy, naughty and very funny. Everybody loves araras. The name seemed catchy for a tool and, in the blink of an eye, arara was quickly spread to the whole TEX world. It is an interesting story of a bird and a lion living together.

2 The basics

I think the best way to explain how arara works is to provide a quick comparison with similar tools, like the ones I've mentioned in the prologue. Let us use the following file `hello.tex` as an example:

```
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

How would one successfully compile `hello.tex` with `latexmk` and `rubber`, for instance? It's quite straightforward: it is just a matter of providing the file to the tool and letting it do the hard work; a simple `latexmk hello` or `rubber -pdf hello` would do the trick. Now, if one tries `arara hello`, I'm afraid *nothing* will be generated; the truth is, arara doesn't know what to do with your file (and the tool will raise an error message complaining about this issue). You need to tell arara what to do.

That is the major difference of arara when compared to other tools: it is not an automatic process and the tool does not employ any guesswork on its own. You are in control of your documents; arara won't do anything unless you teach it how to do a task and explicitly tell it to execute the task.

How does one teach arara how to do a task? The answer is quite simple: we have to define rules. A rule is a formal description of how arara should handle a certain task. For example, if we want to use `pdflatex` with arara, we need a rule for that. Once a rule is defined, arara automatically provides an access layer to the user. The package provides dozens of predefined rules, so you already have several options out of the box to set up your workflow.

Once we know how to execute a task, we need to explicitly tell arara when to do it. This is done through a directive. A directive is a special comment inserted in the source file in which you indicate how arara should behave. You can insert as many directives as you want, and in any position of the file; arara will read the whole file and extract the directives. A directive should be placed in a line of its own, in the form

`% arara:` ⟨*directive*⟩

It is important to observe that a directive is not the command to be executed, but the name of the

rule associated with that directive (once arara finds a directive, it will look for the associated rule). That is basically how arara works: we teach the tool to do a task by providing a rule, and tell it to execute it via directives in the source code.

Sometimes, we need to provide additional information to the rule from the source code. That's why arara offers two types of directives:

empty directive An empty directive, as the name indicates, has only the rule identifier. The syntax for an empty directive is

`% arara:` ⟨*directive*⟩

parameterized directive A parameterized directive has the rule identifier followed by its arguments. It's very important to mention that the arguments are mapped by their identifiers and not by their positions. The syntax for such a directive is

`% arara:` ⟨*directive*⟩`: { `⟨*arglist*⟩` }`

An individual argument has the form

⟨*key*⟩`:` ⟨*value*⟩

and an ⟨*arglist*⟩ has keys with their respective values separated by commas. The arguments are defined according to the rule mapped by the directive (you cannot give an argument `foo` to a directive `bar` if it does not offer support for this named parameter).

If you want to disable a directive, there's no need to remove it from the source file. Simply replace

```
% arara:
```

by, for example,

```
% !arara:
```

or insert some other symbol before `arara:` and this directive will be ignored. The tool always looks for a line that, after removing the leading and trailing spaces, starts with a comment and contains '`arara:`' as a word of its own. The user manual shows how to override this search pattern, but the `arara:` keyword is always required.

Now that we know how to tell arara what to do with `hello.tex`, we need to modify it a little by including the proper `pdflatex` directive:

```
% arara: pdflatex
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

And that's it. Now, calling `arara hello` (or `arara hello.tex` — both will work), the document will be successfully compiled. Then, let's say we would like to enable shell escape for this particular compilation; we can achieve that by providing a parameterized directive, like this:

```
% arara: pdflatex: { shell: yes }
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Of course, `shell` is defined in the rule scope, otherwise arara would raise an error about an invalid key. The user manual has a list of all available keys for each predefined rule.

As we've noted, arara relies on the provided source file as the main document. The `pdflatex` rule above thus passes the provided filename to the `pdflatex` command. Let us see how to override such information in order to run programs on other files.

There's a reserved argument key named `files`, whose value is a list. If you want to override the default value of the main document for a specific rule, use this key in the directive, in the form

`% arara:` ⟨*directive*⟩`: { files: [`⟨*list*⟩`] }`

For example, if you need to run `makeindex` on files `a` and `b` instead of the default `hello`, you can use

```
% arara: makeindex: { files: [ a, b ] }
```

That is the trick I used when working with 25 indexes in my songbook: it was just a matter of providing their names and which styles to the `makeindex` directive.

There is much more to arara than what I've described in this section. For more complete coverage of available tools, please refer to the user manual. arara is already available in TeX Live and also as a standalone tool. Source code is available at

`https://github.com/cereda/arara`

It is also important to observe that a new version is in the works and this hopefully will fix a couple of nuisances found with the current official version (namely, version 3.0 of the tool). The new version also includes several improvements which will be unveiled as soon as the tool reaches its official release (as a bonus, a new article will be provided for readers).

3 Examples

Now that we know how arara works, let us see some examples. The first document, `ex1.tex`, requires two runs in order to set the labels correctly, so we write two directives.

```
% arara: pdflatex
% arara: pdflatex
```

```
\documentclass{article}
\begin{document}
\section{Introduction}
\label{sec:intro}
As seen in Section~\ref{sec:intro}\ldots
\end{document}
```

The second document, `ex2.tex`, has a citation (courtesy of `xampl.bib`, available in the TEX Live tree), so we need to specify a call to `bibtex` as well:

```
% arara: pdflatex
% arara: bibtex
% arara: pdflatex
% arara: pdflatex
\documentclass{article}
\begin{document}
As seen in \cite{book-full}\ldots
\bibliographystyle{plain}
\bibliography{xampl}
\end{document}
```

The third document, `ex3.tex`, has the same LATEX source as the previous example, but we want to use `biber` instead of `bibtex`; it's just a matter of replacing the directive:

```
% arara: pdflatex
% arara: biber
% arara: pdflatex
% arara: pdflatex
\documentclass{article}
\usepackage{biblatex}
\addbibresource{xampl.bib}
\begin{document}
As seen in \cite{book-full}\ldots
\printbibliography
\end{document}
```

The fourth document, `ex4.tex`, shows an example of a simple index, so we include a `makeindex` directive:

```
% arara: pdflatex
% arara: makeindex
% arara: pdflatex
\documentclass{article}
\usepackage{makeidx}
\makeindex
\begin{document}
Some text.\index{Apple}
\printindex
\end{document}
```

The fifth document, `ex5.tex`, shows a glossary, courtesy of the great `glossaries` package. We need to add a `makeglossaries` directive for this:

```
% arara: pdflatex
% arara: makeglossaries
% arara: pdflatex
\documentclass{article}
\usepackage{glossaries}
\newglossaryentry{equation}{name=equation,
description={an equation usually involves
at least one variable, and has two sides;
typically we will try to solve an
equation for one of the unknown
variables}}
\makeglossaries
\begin{document}
\glsaddall
\printglossary
\end{document}
```

The sixth document, `ex6.tex`, shows a good old plain TEX source, compiled with the `tex` directive. As expected, we will get `ex6.dvi` as output.

```
% arara: tex
Hello world.
\bye
```

The seventh document, `ex7.tex`, enhances the previous example by adding a conversion chain in order to obtain a PDF file; this is done by converting `ex7.dvi` to `ex7.ps` and then to `ex7.pdf` (the directive names are self-explanatory).

```
% arara: tex
% arara: dvips
% arara: ps2pdf
Hello world.
\bye
```

The eighth document, `ex8.tex`, uses a package (namely `minted`) which requires shell escapes to be enabled. We give the (parameterized) directive for that in order to achieve a proper compilation:

```
% arara: pdflatex: { shell: yes }
\documentclass{article}
\usepackage{minted}
\begin{document}
\begin{minted}{c}
int main() {
  printf("hello, world");
  return 0;
}
\end{minted}
\end{document}
```

The ninth document, `ex9.tex`, uses `multibib` in order to provide two separate bibliographies; we must run `bibtex` on the second auxiliary file `A.aux` as well, so we give the special `files` key to `bibtex`:

```
% arara: pdflatex
% arara: bibtex
% arara: bibtex: { files: [ A ] }
```

```
% arara: pdflatex
% arara: pdflatex
\documentclass{article}
\usepackage{multibib}
\newcites{A}{References 2}

\begin{document}
\cite{book-full}
\citeA{inproceedings-full}

\bibliographystyle{plain}
\begingroup
\bibliography{xampl}
\endgroup

\bibliographystyleA{plain}
\begingroup
\bibliographyA{xampl}
\endgroup
\end{document}
```

Observe the `\begingroup` and `\endgroup` around the `\bibliography` commands: this is because the sample bibliography file `xampl.bib` has a preamble field in which a couple of commands are defined which would otherwise cause some ugly definition errors (as both `.bbl` files contain `\newcommand`).

Alternatively, we could have used one `bibtex` directive with two `files`:

```
% arara: bibtex: { files: [ ex9, A ] }
```

instead of writing two `bibtex` directives. However, I would choose to write a separate line for each `bibtex` run, both to better organize my workflow, and to provide only the second auxiliary filename; otherwise, the main document filename would also have to be explicitly specified.

The tenth and last document, `ex10.tex`, has a `clean` directive to remove `ex10.log` after correctly generating the PDF file:

```
% arara: pdftex
% arara: clean: { files: [ ex10.log ] }
Hello world.
\bye
```

And that is it: arara is quite straightforward to use, provided that you know the available rules and keys, and also the compilation workflow needed.

4 Final remarks

As shown in this article, arara can be used in complex workflows, such as theses and books. You can tell the tool to compile a document, generate indexes and apply styles, remove temporary files, compile other documents, run METAFONT or METAPOST, create glossaries, call `pdfcrop`, `gnuplot`, move files, and much more. Furthermore, arara is platform-independent. It's all up to you.

That said, I believe that the warning featured in the user manual still applies: Hic Sunt Dracones. Hopefully the new version will exterminate a couple of nuisances and bugs found in the current official release; however, as with any non-trivial software, the tool is far from being bug-free. And you will learn that arara gives you plenty of rope. In other words, you will be responsible for how the tool behaves and all the consequences from your actions. Sorry to sound scary, but I really needed to tell you this. After all, one of arara's greatest features is the freedom it offers. But as you know, freedom always comes at a cost.

Feedback is surely welcome for me to improve this humble tool — just write an e-mail to me or any other member of the team and we will reply as soon as possible. The source code is fully available at

`https://github.com/cereda/arara`

Feel free to contribute to the project by forking it, submitting bugs, sending pull requests or even translating it to your language. If you want to support the LaTeX development with a donation, the best way to do this is by donating to the TeX Users Group.

Happy TeXing with arara!

◇ Paulo Roberto Massa Cereda
Analândia, São Paulo, Brazil
`cereda (at) users dot sf dot net`