

arara

The cool TEX automation tool

User manual

Paulo Cereda, Marco Daniel,
Brent Longborough, and Nicola Talbot



Version 4.0

No birds were harmed in the making of this manual.



Foreword

> That deserves no less than a "Holy guacamole!".
>
> GONZALO MEDINA

Creating a PDF from LaTeX code can be quite tiresome. Suppose I am using TeXworks and I have a document that has a bibliography, glossary and index, then I need to select the `pdflatex` tool and click on the typeset button, then select the `bibtex` tool and click on the typeset button, then select the `makeindex` tool and click on the typeset button, then select the `makeglossaries` tool (which I may need to add first) and click on the typeset button, then select the `pdflatex` tool and click on the typeset button, and once more to ensure all the cross-references are up to date. Then I edit the document and have to go through that whole process all over again!

Automation makes life so much simpler. Instead of all those tools that I need to keep selecting, I just need one tool, in this case arara, which will do all the necessary work for me behind the scenes.

Some automation tools try to be clever, but there are invariably exceptions that trip them up. arara does not try to be clever; it just does what it is told to do. The instructions are provided as special comments in the source code that TeX ignores, but they are human-readable and can also provide a hint to non-arara co-authors as to what tools are required in order to complete the document build.

The new improved arara version 4.0 now comes with some exciting features, such as the ability to use conditionals, and it definitely ranks as my favourite automation tool for document creation. Paulo has done a great job, and I would like to take this opportunity to thank him for his patience in dealing with my many feature requests!

Nicola Louise Cecilia Talbot
on behalf of the arara *team*



Prologue

> Moral of the story: never read the documentation, bad things happen.
>
> DAVID CARLISLE

Writing software is easy. Writing *good* software is extremely difficult. When the counter stopped at version 3.0, Brent, Marco and I decided it was time for arara to graduate and finally be released in TeX Live. My life had changed.

It was a success. A lot of people liked the idea of explicitly telling our tool how to compile their TeX documents instead of relying on guesswork. It was indeed a cool concept! But then, the inevitable happened: a lot of bugs had emerged from the dark depths of my humble code.

In all seriousness, *I was about to give up*. My code was not awful, but there were a couple of critical and blocking bugs. Something very drastic had to be done in order to put arara back on track. Then, walking on faith, I decided to rewrite the tool entirely from scratch. In order to achieve this goal, I created a sandbox and started working on the new code.

It was my redemption. Nicola helped me with the new version, writing code, fixing bugs and suggesting new features. Soon, we all achieved a very pleasant result. It was like arara was about to hatch again. Version 4.0 was definitely at our hands. Now, it is up to you.

Surprisingly, this humble user manual is not the best resource for learning about our tool. If you really want to see arara in action, I strongly recommend LaTeX for administrative work, an amazing book freely available for download. The author is, of course, Nicola herself! She explains how LaTeX can be used for administrative work, such as writing correspondence, performing repetitive tasks or typesetting problem sheets on exam papers. And arara is there!

Enjoy the new version. Happy TeXing with arara!

Paulo Roberto Massa Cereda
on behalf of the arara team



License

> Anything that prevents you from being friendly, a good neighbour, is a terror tactic.
>
> RICHARD STALLMAN

arara is licensed under the New BSD License. It is important to observe that the New BSD License has been verified as a GPL-compatible free software license by the Free Software Foundation, and has been vetted as an open source license by the Open Source Initiative.



New BSD License



Copyright © 2012–2018, Paulo Roberto Massa Cereda
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

This software is provided by the copyright holders and contributors “as is” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright holder or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.



Glossary

GUI (Graphical User Interface)
: An interface that allows users to interact through graphical components, such as buttons and menus.

IETF (Internet Engineering Task Force)
: An organisation that develops and promotes Internet standards.

JVM (Java Virtual Machine)
: A virtual machine that enables Java programs to be run.

MVEL (MVFLEX Expression Language)
: A hybrid, dynamic, statically typed, embeddable expression language and runtime for the Java platform. (See Chapter 11.)

Orb tag
: A dynamic element of an MVEL template which is evaluated at runtime. (See Section 11.7.)

REPL (Read–Eval–Print Loop)
: A simple computer programming environment that takes a single expression (input), evaluates it and results the result.

SQL (Structured Query Language)
: A database language.

XML (Extensible Markup Language)
: A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

YAML (YAML Ain't Markup Language)
: Human-friendly data, commonly used for configuration files but also used for data storage or transmission. (See Chapter 10.)

To Marco's son Niclas.



Contents

Glossary **v**

1 Introduction **1**
1.1 What is this tool? . . . 1
1.2 Core concepts . . . 3
1.3 Operating system remarks . . . 6
1.4 Support . . . 6

I — The application

2 Important concepts **8**
2.1 Rules . . . 8
2.2 Directives . . . 18
2.3 Migration guide . . . 24

3 Command line **30**
3.1 User interface design . . . 30
3.2 Options . . . 32
3.3 File name lookup . . . 39

4 Configuration file **42**
4.1 File lookup . . . 42
4.2 Basic structure . . . 43

5 Logging **52**
5.1 System information . . . 52
5.2 Directive extraction . . . 53
5.3 Directive normalization . . . 54
5.4 Rule interpretation . . . 55

6 Methods **58**
6.1 Files . . . 58
6.2 Conditional flow . . . 69
6.3 Strings . . . 73
6.4 Operating systems . . . 76
6.5 Type checking . . . 80
6.6 Classes and objects . . . 82
6.7 Dialog boxes . . . 84

6.8 Commands and triggers 91
6.9 Others 95

7 The official rule pack **100**

II — Development and deployment

8 Building from source **136**
8.1 Requirements 136
8.2 Compiling the tool 138

9 Deploying the tool **142**
9.1 Directory structure 142
9.2 Defining a location 143
9.3 Tool wrapping 144

III — A primer on formats and scripting

10 YAML **150**
10.1 Collections 150
10.2 Scalars 152
10.3 Tags 153
10.4 Further reading 153

11 MVEL **154**
11.1 Basic usage 154
11.2 Inline lists, maps and arrays 156
11.3 Property navigation 157
11.4 Flow control 158
11.5 Projections and folds 160
11.6 Assignments 160
11.7 Basic templating 161
11.8 Further reading 162



Introduction

Hello there, welcome to arara, the cool TEX automation tool! This chapter is actually a quick introduction to what you can (and cannot) expect from arara. For now, concepts will be informally presented and will be detailed later on, in the next chapters.

1.1 What is this tool?

Good question! arara is a TEX automation tool based on rules and directives. It is, in some aspects, similar to other well-known tools like `latexmk` and `rubber`. The key difference (and probably the selling point) might be the fact that arara aims at explicit instructions in the source code (in the form of comments) in order to determine what to do instead of relying on other resources, such as log file analysis. It is a different approach for an automation tool, and we have both advantages and disadvantages of such design. Let us use the following file `hello.tex` as an example:



Source file

`hello.tex`

```
\documentclass{article}

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

How would one successfully compile `hello.tex` with `latexmk` and `rubber`, for instance? It is quite straightforward: it is just a matter of providing the file to the tool and letting it do the hard work:



Terminal

```
$ latexmk -pdf mydoc.tex
$ rubber --pdf mydoc.tex
```

The mentioned tools perform an analysis on the file and decide what has

to be done. However, if one tries to invoke `arara` on `hello.tex`, I am afraid *nothing* will be generated; the truth is, arara does not know what to do with your file, and the tool will even raise an error message complaining about this issue:

Terminal

```
$ arara hello.tex
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'hello.tex' (size: 86 bytes, last modified: 05/03/2018
07:28:30), please wait.

It looks like no directives were found in the provided file. Make
sure to include at least one directive and try again.

Total: 0.00 seconds
```

Quite surprising. However, this behaviour is not wrong at all, it is completely by design: arara needs to know what you want. And for that purpose, you need to tell the tool what to do.

A very important concept

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 will not do anything unless you *teach it how to do a task and explicitly tell it to execute the task*.

Now, how does one tell arara to do a task? That is actually the easy part, provided that you have everything up and running. We accomplish the task by adding a special comment line, hereafter known as *directive*, somewhere in our `hello.tex` file (preferably in the first lines):

Source file `hello.tex`

```
% arara: pdflatex
\documentclass{article}

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

For now, do not worry too much about the terms, we will come back to

them later on, in Chapter 2, on page 8. It suffices to say that arara expects *you* to provide a list of tasks, and this is done by inserting special comments in the source file. Let us see how arara behaves with this updated code:

Terminal

```
$ arara hello.tex

  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'hello.tex' (size: 86 bytes, last modified: 05/03/2018
07:28:30), please wait.

(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS

Total: 0.73 seconds
```

Hurrah, we finally got our document properly compiled with a TEX engine by the inner workings of our beloved tool, resulting in an expected `hello.pdf` file created using the very same system call that typical automation tools like `latexmk` and `rubber` use. Observe that arara works practically on other side of the spectrum: you need to tell it how and when to do a task.

1.2 Core concepts

When adding a directive in our source code, we are explicitly telling the tool what we want it to do, but I am afraid that is not sufficient at all. So far, arara knows *what* to do, but now it needs to know *how* the task should be done. If we want arara to run `pdflatex` on `hello.tex`, we need to have instructions telling our tool how to run that specific application. This particular sequence of instructions is referred as a *rule* in our context.

Note on rules

Although the core team provides a lot of rules shipped with arara out of the box, with the possibility of extending the set by adding more rules, some users might find this decision rather annoying, since other tools have most of their rules hard-coded, making the automation process even more transparent. However, since arara does not rely on a specific automation or compilation scheme, it becomes more extensible. The use of directives in the source code make the automation steps more fluent, which allows the specification of complex workflows very easy.

Despite the inherited verbosity of automation steps not being suitable for small documents, arara really shines when you have a document which needs

full control of the automation process (for instance, a thesis or a manual). Complex workflows are easily tackled by our tool.

Rules and directives are the core concepts of arara: the first dictates how a task is done, and the latter is the proper instance of the associated rule on the current document, i.e, when and where the commands must be executed.

The name



Do you like araras? We do, specially our tool which shares the same name of this colorful bird.

The tool 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). 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.

Now that we informally introduced rules and directives, let us take a look on how arara actually works given those two elements. The whole idea is pretty straightforward, and I promise to revisit these concepts later on in this manual for a comprehensive explanation (more precisely, in Chapter 2).

First and foremost, we need to add at least one instruction in the source code to tell arara what to do. This instruction is named a *directive* and it will be parsed during the preparation phase. Observe that arara will tell you if no directive was found in a file, as seen in our first interaction with the tool.

An arara directive is usually defined in a line of its own, started with a comment (denoted by a percent sign in TEX and friends), followed by the word `arara:` and task name:

A typical directive

```
% arara: pdflatex
\documentclass{article}
...
```

Our example has one directive, referencing `pdflatex`. It is important to observe that the `pdflatex` identifier *does not represent the command to be executed*, but *the name of the rule associated with that directive*.

New feature in version 4.0

Multiline directives – Later on, in Section 2.2, on page 18, we will discover that a directive can also span several lines in order to provide a better code organization. For now, let us assume a typical directive occupies only one line.

Once arara finds a directive, it will look for the associated *rule*. In our example, it will look for a rule named `pdflatex` which will evidently run the `pdflatex` command line application. Rules are YAML files named according to their identifiers followed by the `yaml` extension and follow a strict structure. This concept is covered in Section 2.1, on page 8.

New feature in version 4.0

REPL workflow – arara now employs a REPL workflow for rules and directives. In previous versions, directives were extracted, their corresponding rules were analyzed, commands were built and added to a queue before any proper execution or evaluation. I decided to change this workflow, so now arara evaluates each rule on demand, i.e, there is no *a priori* checking. A rule will *always* reflect the current state, including potential side effects from previous executed rules.

Now, we have a queue of pairs (*directive, rule*) to process. For each pair, arara will map the directive to its corresponding rule, evaluate it and run the proper command. The execution chain requires that command i was successfully executed to then proceed to command $i + 1$, and so forth. This is also by design: arara will halt the execution if any of the commands in the queue had raised an error. How does one know if a command was successfully executed? arara checks the corresponding *exit status* available after a command execution. In general, a successful execution yields 0 as its exit status.

New feature in version 4.0

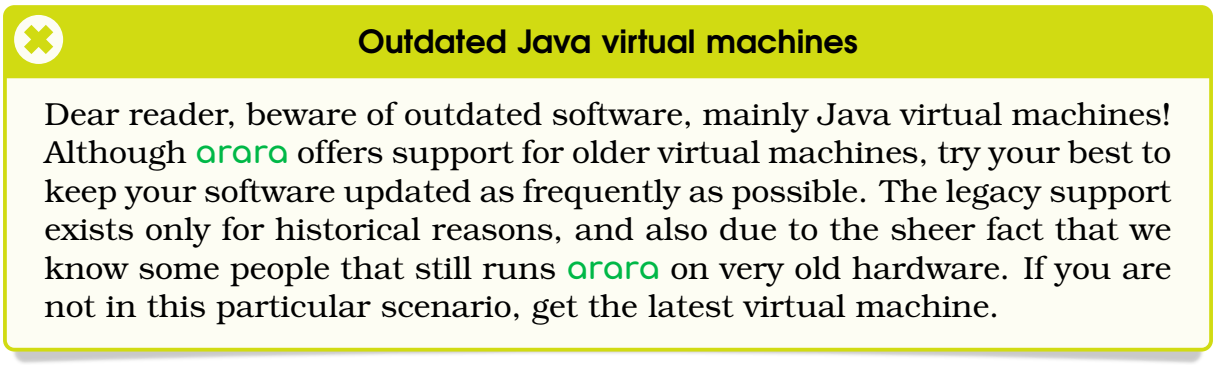
Custom exit status checking – In previous versions, there was no way of customizing the exit status checking of a command. A command was successful if, and only if, its resulting exit status was 0 and no other value. From now on, we can define any value, or even forget about it and make it always return a valid status regardless of execution (for instance, in a rule that always is successful – see, for instance, the `clean` rule).

That is pretty much how arara works: directives in the source code are mapped to rules. These pairs are added to a queue. The queue is then executed and the status is reported. More details about the expansion process are presented in Chapter 2, on page 8. In short, we teach arara to do a task by providing a rule, and tell it to execute it through directives in the source

code.

1.3 Operating system remarks

The application is written using the Java language, so arara runs on top of a Java virtual machine, available on all the major operating systems – in some cases, you might need to install the proper virtual machine. We tried very hard to keep both code and libraries compatible with older virtual machines or from other vendors. Currently, arara is known to run on Oracle's Java 5 to 10, and OpenJDK 5 to 10. We also have reports of users successfully using the tool with virtual machines provided by Azul Systems, so your mileage might vary greatly.



In Chapter 8, on page 136, we provide instructions on how to build arara from sources using Apache Maven. Even if you use multiple operating systems, arara should behave the same, including the rules. There are helper functions available in order to provide support for system-specific rules based on the underlying operating system.

1.4 Support

If you run into any issue with arara, please let us know. We all have very active profiles in the TeX community at StackExchange, so just use the arara tag in your question and we will help you the best we can (also, take a look at their starter guide). We also have a Gitter chat room, in which we occasionally hang out. Also, if you think the report is worthy of an issue, open one in our GitHub repository. And last, but not least, feel free to poke us by good old electronic mail (please try the other approaches first).

We really hope you like our humble contribution to the TeX community. Let arara enhance your TeX experience, it will help you when you will need it the most. Enjoy the manual.



part one

The application



two

Important concepts

Time for our first proper contact with arara! I must stress that is very important to understand a few concepts in which arara relies before we proceed to the usage itself. Do not worry, these concepts are easy to follow, yet they are vital to the comprehension of the application and the logic behind it.

2.1 Rules

A *rule* is a formal description of how arara handles a certain task. For instance, if we want to use `pdflatex` with our tool, we should have a rule for that. Directives are mapped to rules, so a call to a non-existent rule `foo`, for instance, will indeed raise an error:



Terminal

```
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc1.tex' (size: 83 bytes, last modified: 05/03/2018
12:10:33), please wait.

I could not find a rule named 'foo' in the provided rule paths.
Perhaps a misspelled word? I was looking for a file named
'foo.yaml' in the following paths in order of priority:
(/opt/paulo/arara/rules)

Total: 0.09 seconds
```

Once a rule is defined, arara automatically provides an access layer to that rule through directives in the source code, a concept to be formally introduced later on, in Section 2.2. Observe that a directive reflects a particular instance of a rule of the same name (i.e, a `foo` directive in a certain source code is an instance of the `foo` rule).

In short, a rule is a plain text file written in the YAML format, described in Chapter 10, on page 150. I opted for this format because back then it was cleaner and more intuitive to use than other markup languages such as

XML, besides of course being a data serialization standard for programming languages.



Animal jokes

As a bonus, the acronym *YAML* rhymes with the word *camel*, so arara is heavily environmentally friendly. Speaking of camels, there is the programming reference as well, since this amusing animal is usually associated with Perl and friends.

The default rules, i.e, the rules shipped with arara, are placed inside a special subdirectory named `rules/` inside another special directory named `ARARA_HOME` (the place where our tool is installed). We will learn later on, in Section 4.2, on page 43, that we can add an arbitrary number of paths for storing our own rules, in order of priority, so do not worry too much about the location of the default rules, although it is important to understand and acknowledge their existence.

The following list describes the basic structure of an arara rule by presenting the proper elements (or keys, if we consider the proper YAML nomenclature). Observe that elements marked as `M` are mandatory (i.e, the rule *has* to have them in order to work). Similarly, elements marked as `O` are optional, so you can safely ignore them when writing a rule for our tool. A key preceded by `context→` indicates a context and should be properly defined inside it.

`M` `!config`

This keyword is mandatory and must be the first line of any arara rule. It denotes the object mapping metadata to be internally used by the tool. Actually, the tool is not too demanding on using it (in fact, you could suppress it entirely and arara will not complain), but it is considered good practice to start all rules with a `!config` keyword regardless.

`M` `identifier`

This key acts as a unique identifier for the rule (as expected). It is highly recommended to use lowercase letters without spaces, accents or punctuation symbols, as good practice (again). As a convention, if you have an identifier named `pdflatex`, the rule filename must be `pdflatex.yaml` (like our own instance). Please note that, although `yml` is known to be a valid YAML extension as well, arara only considers files ending with the `yaml` extension. This is a deliberate decision.



Example

```
1 identifier: pdflatex
```

`M` `name`

This key holds the name of the *task* (a rule instantiated through a direc-

tive) as a plain string. When running arara, this value will be displayed in the output enclosed in parentheses.

Example

```
1 name: PDFLaTeX
```

O `authors`

We do love blaming people, so arara features a special key to name the rule authors (if any) so you can write stern electronic communications to them! This key holds a list of strings. If the rule has just one author, add it as the first (and only) element of the list.

Example

```
1 authors:
2 - Marco Daniel
3 - Paulo Cereda
```

M `commands`

This key is introduced in version 4.0 of arara and denotes a potential list of commands. From the user perspective, each command is called a *subtask* within a task (rule and directive) context. A task may represent only a single command (a single subtask), as well as a sequence of commands (subtasks). For instance, the `frontespizio` rule requires at least two commands. So, as a means of normalizing the representation, a task composed of a single command (single subtask) is defined as the only element of the list, as opposed to previous versions of arara, which had a specific key to hold just one command.

Incompatibility with older versions

Dear reader, note that rules from version 4.0 are incompatible with older versions of arara. If you are migrating from old versions to version 4.0, we need to replace `command` by `commands` and specify a contextual element, as seen in the following descriptions. Please refer to Section 2.3, on page 24, for a comprehensible migration guide.

In order to properly set a subtask, the keys used in this specification are defined inside the `commands→` context and presented as follows.

O `commands→` `name`

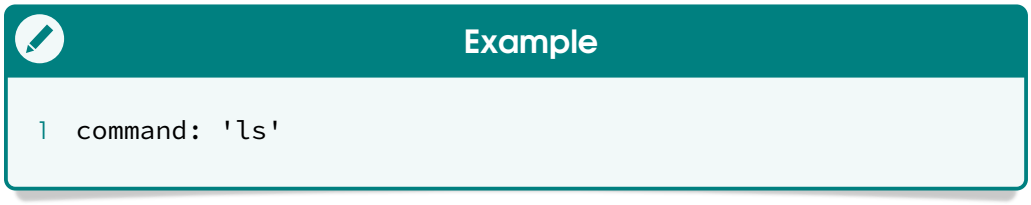
This key holds the name of the subtask as a plain string. When running arara, this value will be displayed in the output. Subtask names are displayed after the main task name. By the way, did you notice

that this key is entirely optional? That means that a subtask can simply be unnamed, if you decide so. However, such practice is not recommended, as it's always good to have a visual description of what arara is running at the moment, so name your subtasks properly.

M commands→ command

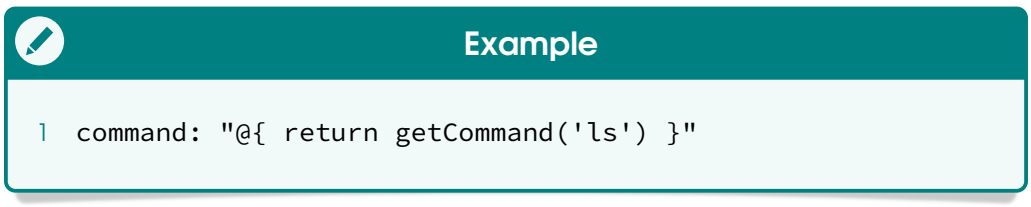
This key holds the action to be performed, typically a system command. In previous versions, arara would rely solely on a string. For this version on, as a means to enhance the user experience (and also fix serious blockers when handling spaces in file names, as seen in previous issues reported in the repository), the tool offers four types of returned values:

- A plain string: this is the default (and only) behaviour in older versions of arara. The plain string is processed as it is by the underlying execution engine. However, automatic argument parsing is problematic, so this approach, although supported, is not recommended any more.



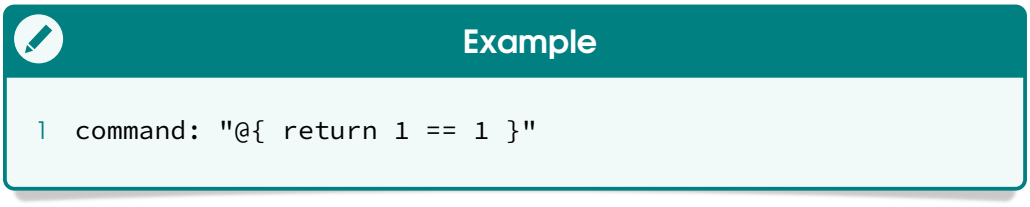
It is important to observe that you can use either a plain string directly or use an orb tag with an explicit `return` command (as seen in Section 11.1, on page 154). Personally, I favour the explicit indication for a quick understanding.

- A `Command` object: arara 4.0 features a new approach for handling system commands based on a high level structure with explicit argument parsing named `Command` (for our curious users, it is a plain Java object). In order to use this approach, we need to rely on orb tags and use a helper method named `◊getCommand` to obtain the desired result. We will detail this method later on, in Section 6.8, on page 91. We highly recommend the adoption of this approach for rule writing instead of using plain strings.



- A boolean value: it is also possible to exploit the expressive power of the underlying scripting language available in the rule context (see Chapter 11, on page 154, for more details) for writing complex code. In this particular case, since the computation is being done by arara itself and not the underlying operating system, there will not be a command to be executed, so simply return a boolean

value – either an explicit `true` or `false` value or a logical expression – to indicate whether the computation was successful.

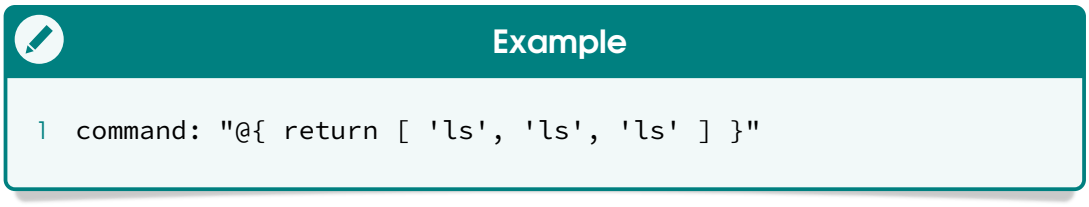


Example

```
1  command: "@{ return 1 == 1 }"
```

- A `Trigger` object: this is surely the least common type of returned value and it is mentioned here just for documentation purposes. In simple terms, a `Trigger` object constitutes a special command that changes the internal workings of arara at runtime. We have not worked much on this concept, so there is only one trigger available, seen in action in the official `halt` rule. In order to use this approach, we need to rely on orb tags and use a helper method named `◊getTrigger` to obtain the desired result.

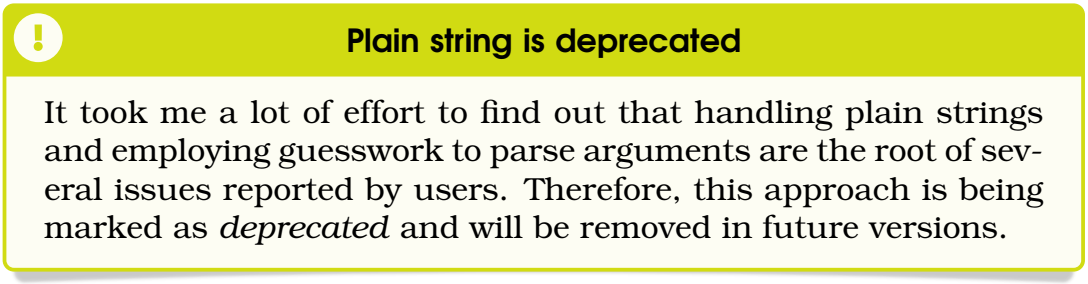
It is also worth mentioning that arara also supports lists of commands represented as plain strings, `Command` or `Trigger` objects, boolean values or a mix of them. This is useful if your rule has to decide whether more actions are required in order to accomplish a task. In this case, our tool will take care of the list and execute each element in the specified order.



Example

```
1  command: "@{ return [ 'ls', 'ls', 'ls' ] }"
```

As an example, please refer to the official `clean` rule for a real scenario where a list of commands is successfully employed: for each provided extension, the rule creates a new cleaning command and adds it to a list of removals to be processed later.



Plain string is deprecated

It took me a lot of effort to find out that handling plain strings and employing guesswork to parse arguments are the root of several issues reported by users. Therefore, this approach is being marked as *deprecated* and will be removed in future versions.

There are at least two variables available in the `command` context and are described as follows (note that MVEL variables and orb tags are discussed in Chapter 11). A variable will be denoted by `◊variable` in this list. For each rule argument (defined later on), there will be a corresponding variable in the `command` context, directly accessed through its unique identifier.

`◊file`
: This variable holds the file name, without any path reference, as a plain string. It is usually composed from the base name and the extension. This variable is available since the first release of arara.

`◊reference`
: This variable is introduced in version 4.0 of arara and holds the canonical, absolute path representation of the `◊file` variable as a `File` object. This is useful if it's necessary to know the hierarchical structure of a project. Since the reference is a Java object, we can use all methods available in the `File` class.



Quote handling

The YAML format disallows key values starting with `@` without proper quoting. This is the reason we had to use double quotes for the value and internally using single quotes for the command string. Also, we could use the other way around, or even using only one type and then escaping them when needed. This is excessively verbose but needed due to the format requirement. Thankfully, arara offers two solutions for removing the quoting verbosity when writing commands.

The first solution is used in previous versions and it still works like a charm in modern days. We need to precede our command with a special keyword `<arara>` which will be removed afterwards. This solution works on virtually every key in the rule context, so it is a bonus. The new code will look like this:



Example

```
1 command: <arara> @{ return getCommand('ls') }
```

The second approach is more of a YAML feature rather than a tool exclusive, although we have to do a couple of checks under the hood in order to ensure the correct execution. The idea here is to use the scalar content in folded style, as seen in Section 10.2, on page 152. The new code will look like this:

Quote handling (ctd.)

Example

```
command: >
  @{
    return getCommand('ls')
  }
```

Mind the indentation, as YAML requires it to properly identify blocks. I personally recommend this approach for longer code, as it provides a better visual representation. You will see the second solution all around the default rules, but feel free to use the one you feel more comfortable with.

0 `commands→` `exit`

This key holds a special purpose in arara 4.0, as it represents a custom exit status evaluation for the corresponding command. In general, a successful execution has zero as an exit status, but sometimes we end up with tools or situations where we need to override this check for whatever reason. For this purpose, simply write a MVEL expression *without orb tags* as plain string and use the special variable `◊value` if you need the actual exit status returned by the command, available at runtime. For example, if the command returns a non-zero value indicating a successful execution, we can write this key as:

Example

```
exit: value > 0
```

If the execution should be marked as successful by arara regardless of the actual exit status, you can simply write `true` as the key value and this rule will never fail, for obvious reasons.

For instance, consider a full example of the `commands` key, defined with only one command, presented as follows. The hyphen denotes a list element, so mind the indentation for correctly specifying the component keys. Also, note that, in this case, the `exit` key was completely optional, as it does the default checking, and it was included for didactic purposes.

Example

```
commands:
- name: The PDFLaTeX engine
  command: >
    @{
      return getCommand('pdflatex', file)
    }
  exit: value == 0
```

M `arguments`

This key holds a list of arguments for the current rule, if any. The arguments specified in this list will be available to the user later on for potential completion through directives. Once instantiated, they will become proper variables in the `command` contexts. This key is mandatory, so even if your rule does not have arguments, you need to specify a list regardless. In this case, use the empty list notation:

Example

```
arguments: []
```

In order to properly set an argument, the keys used in this specification are defined inside the `arguments→` context and presented as follows.

M `arguments→` `identifier`

This key acts as a unique identifier for the argument. It is highly recommended to use lowercase letters without spaces, accents or punctuation symbols, as a good practice. This key will be used later on to set the corresponding value in the directive context.

Example

```
identifier: shell
```

It is important to mention that not all names are valid as argument identifiers. arara has restrictions on three names, described as follows, which cannot be used.

Reserved names for rule arguments

Our tool has three names reserved for internal use: `file`, `files`, and `reference`. Do not use them as argument identifiers!

0 `arguments→` `flag`

This key holds a plain string and is evaluated when the corresponding argument is defined in the directive context. After being evaluated, the result will be stored in a variable of the same name to be later accessed in the `command` context. In the scenario where the argument is not defined in the directive, the variable will hold an empty string.

Example

```
1 flag: >
2   @{
3       isTrue(parameters.shell, '--shell-escape',
4              '--no-shell-escape')
5   }
```

There are three variables available in the `flag` context, described as follows. Note that are also several helper methods available in the rule context (for instance, `◊isTrue` presented in the previous example) which provide interesting features for rule writing. They are detailed later on, in Chapter 6, on page 58.

`◊parameters`

This variable holds a map of directive parameters available at runtime. For each argument identifier listed in the `arguments` list in the rule context, there will be an entry in this variable. This is useful to get the actual values provided during execution and take proper actions. If a parameter is not set in the directive context, the reference will still exist in the map, but it will be mapped to an empty string.

`◊file`

This variable holds the file name, without any path reference, as a plain string. It is usually composed from the base name and the extension. This variable is available since the first release of arara.

`◊reference`

This variable is introduced in version 4.0 of arara and holds the canonical, absolute path representation of the `◊file` variable as a `File` object. This is useful if it's necessary to know the hierarchical structure of a project. Since the reference is a Java object, we can use all methods available in the `File` class.

In the previous example, observe that the MVEL expression defined in the `flag` key checks if the user provided an affirmative value regarding shell escape, through comparing `◊parameters.shell` with a set of predefined affirmative values. In any case, the corresponding command flag is defined as result of such evaluation.

0 `arguments→` `default`

As default behaviour, if a parameter is not set in the directive context,

the reference will be mapped to an empty string. This key exists for the exact purpose of overriding such behaviour.

Example

```
1 default: ''
```

There are three variables available in the `default` context, described as follows. Note that are also several helper methods available in the rule context (for instance, `◊isTrue` presented in the previous example) which provide interesting features for rule writing. They are detailed later on, in Chapter 6, on page 58.

`◊parameters`
: This variable holds a map of directive parameters available at runtime. For each argument identifier listed in the `arguments` list in the rule context, there will be an entry in this variable. This is useful to get the actual values provided during execution and take proper actions. If a parameter is not set in the directive context, the reference will still exist in the map, but it will be mapped to an empty string.

`◊file`
: This variable holds the file name, without any path reference, as a plain string. It is usually composed from the base name and the extension. This variable is available since the first release of arara.

`◊reference`
: This variable is introduced in version 4.0 of arara and holds the canonical, absolute path representation of the `◊file` variable as a `File` object. This is useful if it's necessary to know the hierarchical structure of a project. Since the reference is a Java object, we can use all methods available in the `File` class.

`0` `arguments→` `required`
: There might be certain scenarios in which a rule could make use of required arguments (for instance, a copy operation in which source and target must be provided). The `required` key acts as a boolean switch to indicate whether the corresponding argument should be mandatory. In this case, set the key value to `true` and the argument becomes required. Later on at runtime, arara will throw an error if a required parameter is missing in the directive.

Example

```
1 required: false
```

Note that setting the `required` key value to `false` corresponds to omitting the key completely in the rule context, which resorts to the default behaviour (i.e, all arguments are optional).



Note on argument keys

As seen previously, both `flag` and `default` are marked as optional, but at least one of them must occur in the argument specification, otherwise arara will throw an error, as it makes no sense to have no argument handling at all. Please make sure to specify at least one of them for a consistent behaviour!

For instance, consider a full example of the `arguments` key, defined with only one argument, presented as follows. The hyphen denotes a list element, so mind the indentation for correctly specifying the component keys. Also, note that, in this case, keys `required` and `default` were completely optional, and they were included for didactic purposes.



Example

```
1  arguments:
2  - identifier: shell
3    flag: >
4      @{
5          isTrue(parameters.shell,
6                 '--shell-escape',
7                 '--no-shell-escape')
8      }
9    required: false
10   default: ''
```

This is the rule structure in the YAML format used by arara. Keep in mind that all subtasks in a rule are checked against their corresponding exit status. If an abnormal execution is detected, the tool will instantly halt and the rule will fail. Even arara itself will return an exit code different than zero when this situation happens (detailed in Chapter 3, on page 30).

2.2 Directives

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. The tool will read the whole file and extract the directives.

New features in version 4.0

Partial directive extraction – From version 4.0 of arara on, it is now possible to extract directives only available in the file preamble, i.e, all lines from the beginning that are comments until reaching the first line that is not a comment (excluding blank lines). To this end, a new command line flag is introduced in Section 4.2, on page 43.

Predefined preambles – Common preambles can be predefined and used with files that require the same automation steps, then arara can be invoked based on such specifications. This feature is covered in Section 4.2, on page 43.

There are two types of directives in arara which determine the way the corresponding rules will be instantiated. They are listed as follows. Note that directives are always preceded by the `arara:` pattern.

empty directive

This type of directive has already been mentioned in Chapter 1, on page 1, it has only the rule name (which refers to the `identifier` key from the rule of the same name). All rule arguments are mapped to empty strings, except the ones with `default` values.

Empty directive

```
1 % arara: pdflatex
```

parametrized directive

This type of directive also has the rule name (which refers to the `identifier` key from the rule of the same name), and also contains a map of parameters in order to provide additional information to the corresponding rule. This map is defined in the YAML format, based on the inline style.

Parametrized directive

```
1 % arara: pdflatex: { shell: yes }
```

Observe that arara relies on named parameters, so they are mapped by their corresponding argument identifiers and not by their positions. The syntax for a parameter is described as follows. Please refer to the map definition in Section 10.1, on page 150.

Parameter syntax

```
key : value
```

Note that virtually any type of data can be used as parameter value, so lists, integers, booleans, sets and other maps are available as well. However, there must be the correct handling of such types in the rule context.

When handling parametrized directives, arara always checks if directive parameters and rule arguments match. If we try to inject a non-existent parameter in a parametrized directive, the tool will raise an error about it:

Terminal

```
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'hello.tex' (size: 103 bytes, last modified:
05/03/2018 15:40:16), please wait.

I have spotted an error in rule 'pdflatex' located at
'/opt/paulo/arara/rules'. I found these unknown keys
in the directive: (foo). This should be an easy fix,
just remove them from your map.

Total: 0.21 seconds
```

As the message suggests, we need to remove the unknown parameter key from our directive or rewrite the rule in order to include it as an argument. The first option is, of course, easier.

New feature in version 4.0

Helpful messages – It is a staple of arara to have friendly and helpful messages. From version 4.0 on, we decided to make messages even friendlier and include suggestions for correcting errors or improving usage. So, whenever possible, make sure to read all messages our tool provides, they will help you!

Sometimes, directives can span several columns of a line, particularly the ones with several parameters. From arara 4.0 on, we can split a directive into multiple lines by using the `arara: -->` mark (also known as *arrow notation* during development) to each line which should compose the directive. We call it a *multiline directive*. Let us see an example:

Multiline directive

```
% arara: pdflatex: {
% arara: --> shell: yes,
% arara: --> synctex: yes
% arara: --> }
```

It is important to observe that there is no need of them to be in contiguous lines, i.e, provided that the syntax for parametrized directives hold for the line composition, lines can be distributed all over the code. In fact, the log file (when enabled) will contain a list of all line numbers that compose a directive. This feature is discussed later on, in Section 5.2, on page 53.

Keep lines together

Although it is possible to spread lines of a multiline directive all over the code, it is considered good practice to keep them together for easier reading and editing. In any case, you can always see which lines compose a directive by inspecting the log file.

arara 4.0 provides logical expressions, written in the MVEL language, and special operators processed at runtime in order to determine whether and how a directive should be processed. This feature is named *directive conditional*, or simply *conditional* as an abbreviation. The following list describes all conditional operators available in the directive context.

a priori `*if`

The associated MVEL expression is evaluated beforehand, and the directive is interpreted if, and only if, the result of such evaluation is true. This directive, when the conditional holds true, is executed at most once.

Conditional

```
% arara: pdflatex if missing('pdf') || changed('tex')
```

a posteriori `*until`

The directive is interpreted the first time, then the associated MVEL expression evaluation is done. While the result holds false, the directive is interpreted again and again. There are no guarantees of proper halting.

Conditional

```
% arara: pdflatex until !found('log', 'undefined references')
```

a priori `*unless`

Technically an inverted `*if` conditional, the associated MVEL expression is evaluated beforehand, and the directive is interpreted if, and only if, the result is false. This directive, when the conditional holds false, is executed at most once.

Conditional

```
1 % arara: pdflatex unless unchanged('tex') && exists('pdf')
```

a priori `*while`

The associated MVEL expression is evaluated beforehand, the directive is interpreted if, and only if, the result is true, and the process is repeated while the result still holds true. There are no guarantees of proper halting.

Conditional

```
1 % arara: pdflatex while missing('pdf') ||
2 % arara: --> found('log', 'undefined references')
```

Several methods are available in the directive context in order to ease the writing of conditionals, such as `◊missing`, `◊changed`, `◊found`, `◊unchanged`, and `◊exists` featured in the previous examples. They will be properly detailed later on, in Section 6.1, on page 58.

No infinite loops

Although there are no conceptual guarantees for proper halting of unbounded loops, we have provided a technical solution for potentially infinite iterations: arara has a predefined maximum number of loops. The default value is set to 10, but it can be overridden either in the configuration file or with a command line flag. We discuss this feature later on, in Sections 3.2 and 4.2, on pages 32 and 43, respectively.

All directives, regardless of their type, are internally mapped with both `file` and `reference` parameters, discussed earlier on, in Section 1.2, on page 3, as special variables in the rule context. When inspecting the log file, you will find all map keys and values for each extracted directive (actually, there is an entire log section devoted to detailing directives found in the code). This feature is covered in Section 5.3, on page 54. See, for instance, the report of the directive extraction and normalization process performed by arara when inspecting `doc2.tex`, available in the log file. Note that timestamps were deliberately removed in order to declutter the output, and line breaks were included in order to easily spot the log entries.

Source file

doc2.tex

```
% arara: pdflatex
% arara: pdflatex: { shell: yes }
\documentclass{article}

\begin{document}
Hello world.
\end{document}
```

An excerpt of the log file (directive section)

```
Directive: { identifier: pdflatex, parameters:
{reference=/home/paulo/doc2.tex, file=doc2.tex},
conditional: { NONE }, lines: [1] }

Directive: { identifier: pdflatex, parameters:
{shell=yes, file=doc2.tex, reference=/home/paulo/doc2.tex},
conditional: { NONE }, lines: [2] }
```

The directive context also features another special parameter named `files` which expects a non-empty list of file names as plain string values. For each element of this list, arara will replicate the current directive and point the element being iterated as current `file` and `reference` values (being the latter resolved to a proper absolute, canonical path of the former). See, for instance, the report of the directive extraction and normalization process performed by arara when inspecting `doc3.tex`, available in the log file.

Source file

doc3.tex

```
% arara: pdflatex: { files: [ doc1.tex, doc2.tex ] }
Hello world.
\bye
```

An excerpt of the log file (directive section)

```
Directive: { identifier: pdflatex, parameters:
{reference=/home/paulo/doc1.tex, file=doc1.tex},
conditional: { NONE }, lines: [1] }

Directive: { identifier: pdflatex, parameters:
{reference=/home/paulo/doc2.tex, file=doc2.tex},
conditional: { NONE }, lines: [1] }
```

It is important to observe that, in this case, `doc3.tex` is a plain TEX file, but `pdflatex` is actually being called on two LATEX documents, first `doc1.tex` and then, at last, `doc2.tex`.

Even when a directive is interpreted with a file other than the one being processed by arara (through the magic of the `files` parameter), it is possible to use helper methods in the rule context to get access to the original file and reference. Such methods are detailed later on, in Section 6.1, on page 58.

2.3 Migration guide



A note to users

If this is your first time using arara or you do not have custom rules in the old format, you can safely ignore this section. All rules shipped with our tool are already written in the new format.

As previously discussed in Section 2.1, on page 8, version 4.0 of arara introduces a new rule format. As a result, user-defined rules in the old format are incompatible with the new version of our tool and thus have to be updated. In short, we need to replace `command` by `commands` and specify a contextual element. As an example, consider the following hypothetical rule `ls` written in the old format:



A rule in the old format

```
!config
identifier: ls
name: LS
command: ls @{details}
arguments:
- identifier: details
  flag: '@{ isTrue(parameters.details, "-l", "") }'
```

This rule does nothing too important, it simply runs the system command `ls` which lists the contents of the current directory. However, when we try to run arara on a file which contains a directive referencing this rule, we get the following error in the terminal:



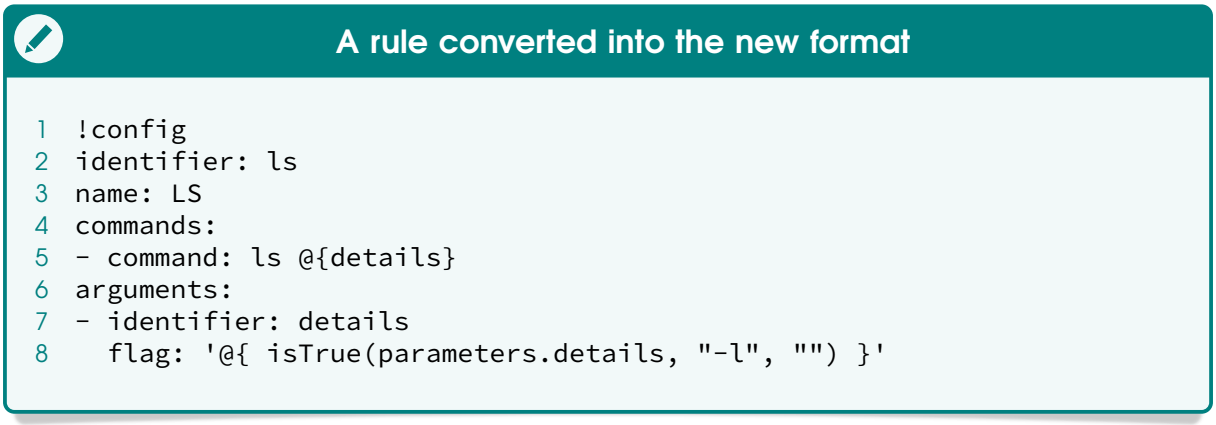
Terminal

```
   __ _ _ __ __ _ _ __ __ _
  / _  | '__/ _  | '__/ _  |
 | (_| | | | (_| | | | (_| |
  \__,_|_|  \__,_|_|  \__,_|

Processing 'doc4.tex' (size: 31 bytes, last modified: 05/27/2018
```



The above terminal output shows the usual error arara raises when a rule in the old format is used, and thus the corresponding YAML file is considered invalid. In order to fix the rule, we need to move the `command` key inside a `commands→` context as a list element, as seen as follows.



Note that this fix is sufficient to make the rule valid in the new format. Also, it is interesting to observe that the subtask will be unnamed during the execution, as there is no corresponding `name` key in the list element.

Terminal

```
(LS) Unnamed task ..................................... SUCCESS
```

Now, let us consider an example containing a list of commands, also based on the old format. The `ls` rule was updated to include two runs of the system command of the same name in the current directory:

A rule in the old format

```
!config
identifier: ls
name: LS
commands:
- ls @{details}
- ls @{details}
arguments:
- identifier: details
  flag: '@{ isTrue(parameters.details, "-l", "") }'
```

Observe that the old format directly represents commands as a list of plain strings. When trying to run arara on a file which contains a directive referencing the updated `ls` rule, we get the following error in the terminal:

Terminal

```
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc4.tex' (size: 31 bytes, last modified: 05/27/2018
21:05:57), please wait.

I have spotted an error in rule 'ls' located at '/home/paulo'. I
could not parse the rule, something bad happened. Apparently, the
provided YAML file is invalid. I will do my best to help you in
any way I can. There are more details available on this
exception:

DETAILS ------------------------------------------------------------
Cannot create property=commands for
JavaBean=com.github.cereda.arara.model.Rule@91161c7
 in 'reader',
line 1, column 1:
    !config
    ^
No single argument
constructor found for class
```

Terminal (ctd.)

```
com.github.cereda.arara.model.RuleCommand
 in 'reader', line 5,
column 1:
    - ls @{details}
    ^


Total: 0.02 seconds
```

The above terminal output shows a slightly different message, but the error is practically the same to the one arara raised before when a rule in the old format was used. The difference relies on the missing rule property being set at the moment, but the idea remains the same. In order to fix the rule, we need to precede every list element in the `commands→` context with the `command` key, as seen as follows.

A rule converted into the new format

```
!config
identifier: ls
name: LS
commands:
- command: ls @{details}
- command: ls @{details}
arguments:
- identifier: details
  flag: '@{ isTrue(parameters.details, "-l", "") }'
```

This fix is sufficient to make the rule valid in the new format. Also, as mentioned before, it is interesting to observe that the subtasks will be unnamed during the execution, as there are no corresponding `name` keys in the list elements.

Terminal

```
(LS) Unnamed task ..................................... SUCCESS
(LS) Unnamed task ..................................... SUCCESS
```

There is a helper tool available in the release section of our project repository that attempts to automatically convert rules in the old format to the new one. If you want to try it, download the `rc.jar` file from the repository and put it in the same directory where the old rules are located. You can also provide a full path instead. It is important to note that, although the tool might indicate a successful conversion, there are no guarantees that the resulting rule is fully compliant with the new format, due to potential changes in the internal work-

ings of arara, so your mileage may vary. In general, it should work. The rule converter is written in Java and requires a virtual machine to run. The tool has a straightforward workflow and takes just one parameter referring to the rule to be converted. The entire process should happen without intervention. When invoked without the file name, this is the expected output:

Terminal

```
$ java -jar rc.jar
          _                                  _
 ___ _ _| |___    ___ ___ ___ _ _ ___ ___| |_ ___ ___
|  _| | | | -_|  |  _| . |   | | | -_|  _|  _| -_|  _|
|_| |___|_|___|  |___|___|_|_|\_/|___|_| |_| |___|_|

version 1.0 (rules < 4.0)

OH NO! ---------------------------------------------------------
This tool expects the YAML rule from previous versions of
arara. Please, provide a proper YAML file containing the old
rule as a parameter and try again. I will do my best to
convert the rule to the new version 4.0 format.
```

Let us invoke the tool with the first version of our hypothetical `ls` rule, still in the old format. The tool removes all comments from the original file, if any, and constructs a new file with a `_v4` suffix attached to the name. The original file is preserved. Just keep in mind that the new rule must be renamed afterwards, as the base name and the corresponding `identifier` key must match. The output is presented as follows.

Terminal

```
$ java -jar rc.jar ls.yaml
          _                                  _
 ___ _ _| |___    ___ ___ ___ _ _ ___ ___| |_ ___ ___
|  _| | | | -_|  |  _| . |   | | | -_|  _|  _| -_|  _|
|_| |___|_|___|  |___|___|_|_|\_/|___|_| |_| |___|_|

version 1.0 (rules < 4.0)

The provided YAML rule looks OK. I will try my best to
convert it to the new version 4.0 format adopted by arara.
The new rule name will be written in the same directory of
the original one and will have a '_v4' suffix to it. Keep in
mind that the base name must match the identifier!

YAY! ------------------------------------------------------------
Good news, everybody! The provided YAML rule was updated
successfully to the new version 4.0 format of arara! Of
course, there are no guarantees this new rule will work out
of the box, so fingers crossed! Take a closer look at the
```

Terminal (ctd.)

```
20  manual and update your rule to use the new enhancements of
21  arara. Have a great time!
```

The resulting rule is identical to the one manually converted in this section. Just note that, when creating the file, the resulting YAML file might write the keys in alphabetical order. That means that, although both files semantically represent the same rule, the positions of the keys differ. However, that poses no issue at all, as long as the keys are correctly defined. Also, it is important to note that, due to a conversion policy of the underlying YAML library, folded scalars in the old format are transcribed as literal scalars in the new format. We could force a folded style as default, but the resulting rule would be unnecessarily verbose, so we opted for the simpler, cleaner solution. As a direct consequence, we strongly recommend a subsequent verification and potential fix of existing literal scalars into folded ones, if any. Please refer to Section 10.2, on page 152, for more details on scalars.

Replace plain strings in commands

As plain strings are known to be problematic when defining commands, they are marked as deprecated in version 4.0 of arara and will likely be removed in future releases. Since you are migrating from an old format to a new one, please consider replacing plain strings in command by proper `Command` objects. The helper methods available in the rule context, including the indispensable `◊getCommand` method, for obvious reasons, are detailed in Section 6.8, on page 91. It is highly advisable to update your rules on this regard, if applicable.

This section pretty much covered the basics for correctly migrating rules in the old format to the new one. Of course, it is highly advisable to make use of the new features available in arara 4.0 for achieving better results. If you need any help, please do not hesitating in contacting us. See Section 1.4, on page 6, for more details on how to get help.



Command line

arara is a command line tool. It can be used in a plethora of command interpreter implementations, from bash to a Windows prompt, provided that the Java runtime environment is accessible within the current session. This chapter covers the user interface design, as well as options (also known as flags or switches) that modify the underlying application behaviour.

3.1 User interface design

The goal of a user interface design is to make the interaction as simple and efficient as possible. Good user interface design facilitates finishing the task at hand without drawing unnecessary attention to itself. For arara 4.0, we redesigned the interface in order to look more pleasant to the eye, after all, we work with TEX and friends:



Terminal

```
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc5.tex' (size: 307 bytes, last modified: 05/29/2018
08:57:30), please wait.

(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS
(BibTeX) The BibTeX reference management software ....... SUCCESS
(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS
(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS

Total: 1.45 seconds
```

First of all, we have the nice application logo, displayed using ASCII art. The entire layout is based on monospaced font spacing, usually used in terminal prompts. Hopefully you follow the conventional use of a monospaced font in your terminal, otherwise the visual effect will not be so pleasant. First and foremost, arara displays details about the file being processed, including size and modification status:

Terminal

```
Processing 'doc5.tex' (size: 307 bytes, last modified: 05/29/2018
08:57:30), please wait.
```

The list of tasks was also redesigned to be fully justified, and each entry displays both task and subtask names (the former being displayed enclosed in parentheses), besides of course the usual execution result:

Terminal

```
(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS
(BibTeX) The BibTeX reference management software ....... SUCCESS
(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS
(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS
```

As previously mentioned in Section 2.1, on page 8, if a task fails, arara will halt the entire execution at once and immediately report back to the user. This is an example of how a failed task looks like:

Terminal

```
(PDFLaTeX) PDFLaTeX engine ............................. FAILURE
```

Also, observe that our tool displays the execution time before terminating, in seconds. The execution time has a very simple precision, as it is meant to be easily readable, and should not be considered for command profiling.

Terminal

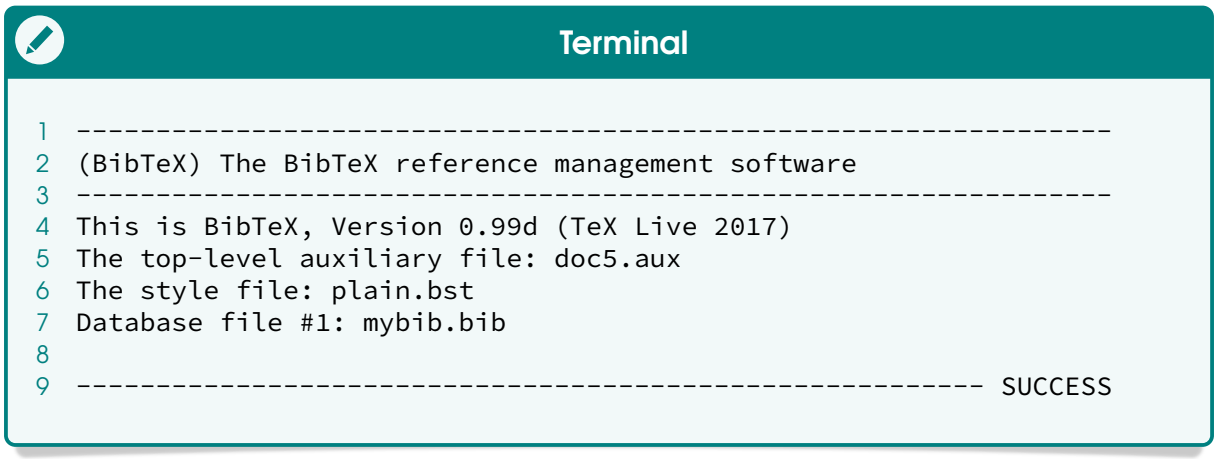
```
Total: 1.45 seconds
```

The tool has two execution modes: *silent*, which is the default, and *verbose*, which prints as much information about tasks as possible. When in silent mode, arara will simply display the task and subtask names, as well as the execution result. Nothing more is added to the output. For instance:

Terminal

```
(BibTeX) The BibTeX reference management software ....... SUCCESS
```

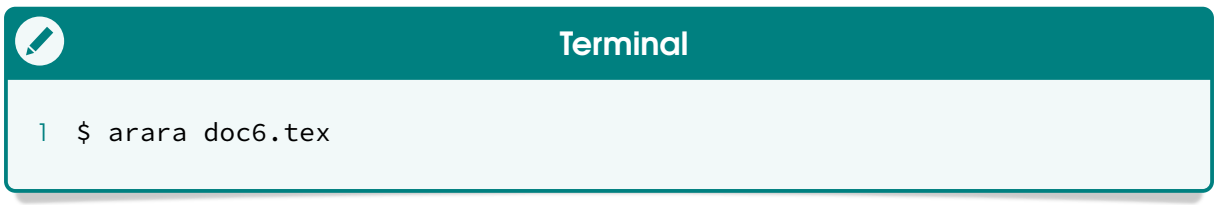
When executed in verbose mode, arara will display the underlying system command output as well, when applied. In version 4.0 of our tool, this mode was also entirely redesigned in order to avoid unnecessary clutter, so it would be easier to spot each task. For instance:



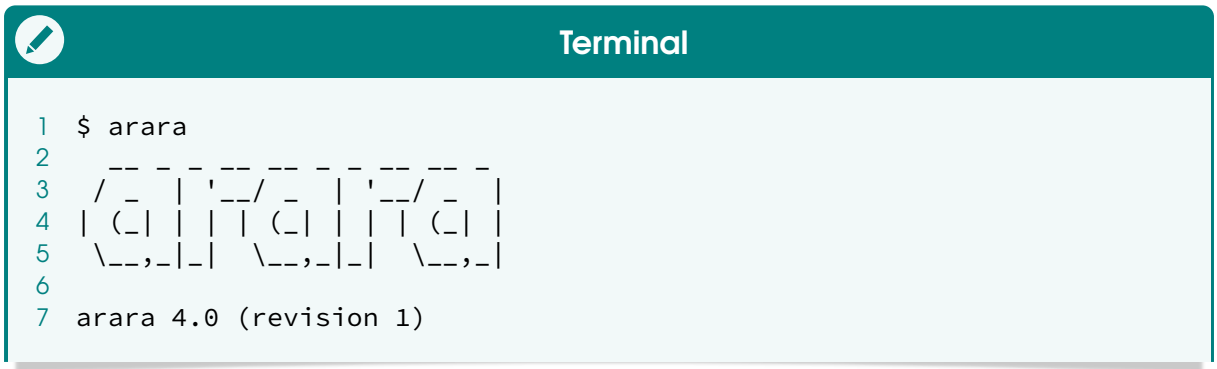
It is important to observe that, when in verbose mode, arara can offer proper interaction if the system command requires user intervention. However, when in silent mode, the tool will simply discard this requirement and the command will almost surely fail.

3.2 Options

In order to run arara on your TEX file, the simplest possible way is to provide the file name to the tool in your favourite command interpreter session, provided that the file has at least one directive:



The tool has a set of command line options (also known as flags or switches) that modify the underlying execution behaviour or enhance the execution workflow. If you do not provide any parameters, arara will display the tool usage and the available options:



Terminal (ctd.)

```
Copyright (c) 2012-2017, Paulo Roberto Massa Cereda
All rights reserved

usage: arara [file [--dry-run] [--log] [--verbose | --silent]
             [--timeout N] [--max-loops N] [--language L]
             [ --preamble P ] [--header] | --help | --version]
 -h,--help                  print the help message
 -H,--header                extract directives only in the file header
 -l,--log                   generate a log output
 -L,--language <code>       set the application language
 -m,--max-loops <number>    set the maximum number of loops
 -n,--dry-run               go through all the motions of running a
                            command, but with no actual calls
 -p,--preamble <name>       set the file preamble based on the
                            configuration file
 -s,--silent                hide the command output
 -t,--timeout <number>      set the execution timeout (in milliseconds)
 -V,--version               print the application version
 -v,--verbose               print the command output
```

The available options for our tool are detailed as follows. Each option contains short and long variations, which are denoted by `-o` and `--option` in the command line, respectively. Additionally, when a parameter is required by the current option, it will be denoted by `▷parameter` in the description.

`-h` `--help`

As the name indicates, this option prints the help message containing the tool usage and the list of all available options. The tool exits afterwards. When running arara without any options or a file to be processed, this is the default behaviour. This option has the highest priority over the others.

`-H` `--header`

This option changes the mechanics of how arara extracts the directives from the code. The tool always reads the entire file and extracts every single directive found throughout the code. However, by activating this switch, arara will extract all directives from the beginning of the file until it reaches a line that is not empty and it is not a comment (hence the option name). Consider the following example:

Source file `doc7.tex`

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

% arara: pdftex
```

When running arara without this option, two directives will be extracted (the ones found in lines 1 and 5). However, if executed with `--header`, the tool will only extract one directive (from line 1), as it will stop the extraction process as soon as it reaches line 2. This option can also be activated by default in the configuration file (see Section 4.2, on page 43).

`-l` `--log`

This option enables the logging feature of our tool. All streams from all system commands will be logged and, at the end of the execution, a consolidated log file named `arara.log` will be generated. This option can also be activated by default in the configuration file (see Section 4.2, on page 43). Refer to Chapter 5, on page 52, for more details on the logging feature.

`-L` `--language` `▷code`

This option sets the language of the current execution of arara according to the language code identified by the `▷code` value provided as the parameter. The language code tries to follow the ISO 639 norm, standardized nomenclature used to classify languages. For example, this is our tool speaking Dutch:

Terminal

```
1  $ arara -L nl doc5.tex
2    __ _ _ __ __ _ _ __ __ _
3   / _  | '__/ _  | '__/ _  |
4  | (_| | | | (_| | | | (_| |
5   \__,_|_|  \__,_|_|  \__,_|
6
7  Verwerken van 'doc5.tex' (grootte: 307 bytes, laatst gewijzigd:
8  05/29/2018 11:50:32), een ogenblik geduld.
9
10 (PDFLaTeX) PDFLaTeX engine ........................... SUCCESVOL
11 (BibTeX) The BibTeX reference management software ..... SUCCESVOL
12 (PDFLaTeX) PDFLaTeX engine ........................... SUCCESVOL
13 (PDFLaTeX) PDFLaTeX engine ........................... SUCCESVOL
14
15 Totaal: 1,59 seconden
```

Navis volitans mihi anguillis plena est

At time of writing, arara is able to speak English, German, Dutch, Italian and Brazilian Portuguese out of the box. There is also a special dialect named Broad Norfolk, spoken by those living in the county of Norfolk in England.

`en` English	`de` German	`qn` Broad Norfolk
`it` Italian	`nl` Dutch	`ptbr` Portuguese (BR)

Navis volitans mihi anguillis plena est (ctd.)

Would you like to make arara speak your own language? Splendid! We would love to have you in the team! Just send us an electronic mail, join our dedicated chatroom or open an issue about it. The localization process is quite straightforward, we can help you. Any language is welcome!

This option can also be specified in the configuration file (see Section 4.2, on page 43). However, one can always override this setting by running the tool with an explicit `-L` option.

`-m` `--max-loops` `▷number`

As a means to avoid infinite iterations, arara has a predefined maximum number of loops, with the default set to 10, as a technical solution (seen in Section 2.2, on page 18). For instance, consider the following directive:

A naughty directive

```
% arara: pdftex while true
```

The `--max-loops` option is used to redefine the maximum number of loops our tool will allow for potentially infinite iterations. Any positive integer can be used as the `▷number` value for this option. An execution of the previous directive with a lower maximum number of loops is shown as follows:

Terminal

```
$ arara -m 2 doc8.tex
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc8.tex' (size: 45 bytes, last modified: 05/29/2018
12:32:14), please wait.

(PDFTeX) PDFTeX engine ................................ SUCCESS
(PDFTeX) PDFTeX engine ................................ SUCCESS

Total: 0.58 seconds
```

This option can also be specified in the configuration file (see Section 4.2, on page 43). However, one can always override this setting by running the tool with an explicit `-m` option.

`-n` `--dry-run`

This option makes arara go through all the motions of running tasks and subtasks, but with no actual calls. It is a very useful feature for testing the sequence of underlying system commands to be performed on a file. For instance, consider the following execution:



Terminal

```
$ arara -n doc5.tex
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc5.tex' (size: 307 bytes, last modified: 05/29/2018
11:50:32), please wait.

[DR] (PDFLaTeX) PDFLaTeX engine
------------------------------------------------------------------
Authors: Marco Daniel, Paulo Cereda
About to run: [ pdflatex, doc5.tex ]

[DR] (BibTeX) The BibTeX reference management software
------------------------------------------------------------------
Authors: Marco Daniel, Paulo Cereda
About to run: [ bibtex, doc5 ]

[DR] (PDFLaTeX) PDFLaTeX engine
------------------------------------------------------------------
Authors: Marco Daniel, Paulo Cereda
About to run: [ pdflatex, doc5.tex ]

[DR] (PDFLaTeX) PDFLaTeX engine
------------------------------------------------------------------
Authors: Marco Daniel, Paulo Cereda
About to run: [ pdflatex, doc5.tex ]

Total: 0.27 seconds
```

Note that the rule authors are displayed (so they can be blamed in case anything goes wrong), as well as the system command to be executed. It is an interesting approach to see everything that will happen to your document and in which order.

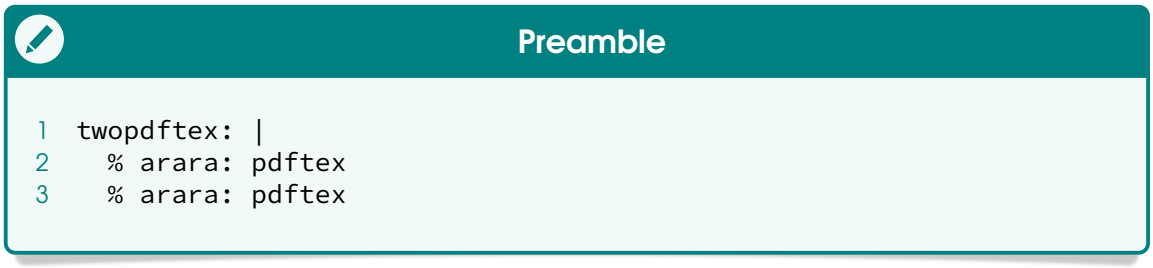


Conditionals and boolean values

It is very important to observe that conditionals are not evaluated when our tool is executed in the `--dry-run` mode, although they are properly listed. Also, when a rule returns a boolean value, the code is executed regardless of this mode.

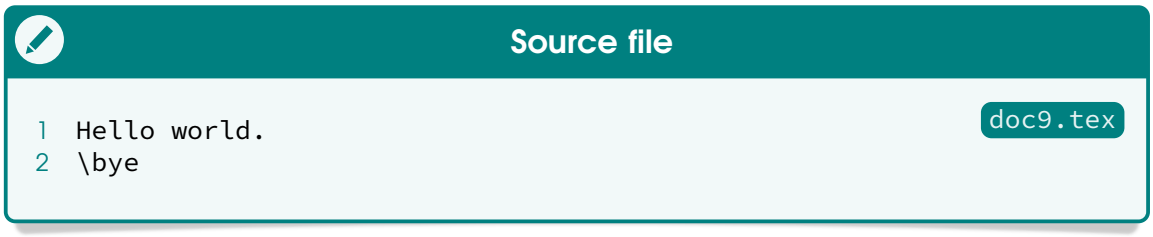
`-p` `--preamble` `▷name`

Some TEX documents require the same automation steps, e.g, a set of articles. To this end, so as to avoid repeating the same preamble over and over in this specific scenario, arara has the possibility of setting predefined preambles in a special section of the configuration file identified by a unique key for later use. This command line option prepends the predefined preamble referenced by the `▷name` key to the current document and then proceeds to extract directives, as usual. For instance:



Preamble

```
1  twopdftex: |
2    % arara: pdftex
3    % arara: pdftex
```

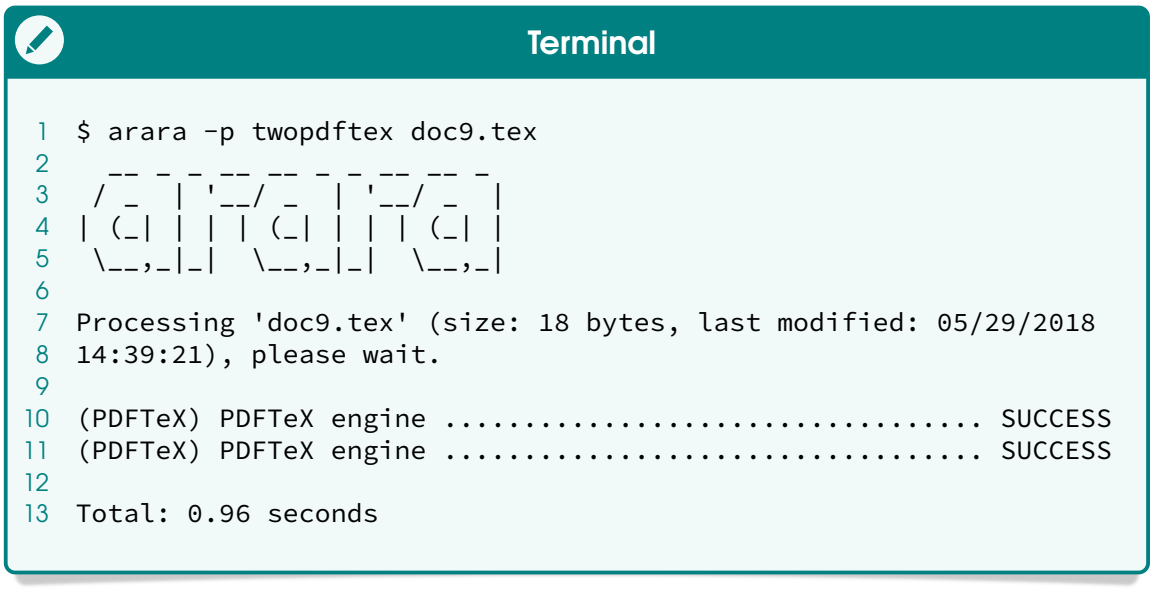


Source file

`doc9.tex`

```
1  Hello world.
2  \bye
```

In this example, we have a preamble named `twopdftex` and a TEX file named `doc9.tex` with no directives. Of course, our tool will complain about missing directives, unless we deliberately inject the two directives from the predefined preamble into the current execution:



Terminal

```
1  $ arara -p twopdftex doc9.tex
2    __ _ _ __ __ _ _ __ __ _
3   / _  | '__/ _  | '__/ _  |
4  | (_| | | | (_| | | | (_| |
5   \__,_|_|  \__,_|_|  \__,_|
6
7  Processing 'doc9.tex' (size: 18 bytes, last modified: 05/29/2018
8  14:39:21), please wait.
9
10 (PDFTeX) PDFTeX engine ..................................... SUCCESS
11 (PDFTeX) PDFTeX engine ..................................... SUCCESS
12
13 Total: 0.96 seconds
```

It is important to note that this is just a directive-based preamble and nothing else, so a line other than a directive is discarded. Line breaks and conditionals are supported. Trying to exploit this area for other purposes will not work. The preamble specification in the configuration file is

detailed in Section 4.2, on page 43.

`-t` `--timeout` `▷number`

This option sets an execution timeout for every task, in milliseconds. If the timeout is reached before the task ends, arara will kill it and halt the execution. Any positive integer can be used as the `▷number` value for this option. Of course, use a sensible value to allow proper time for a task to be executed. For instance, consider the following recursive call:

Source file `doc10.tex`

```
% arara: pdftex
\def\foo{\foo}
This will go \foo forever.
\bye
```

Terminal

```
$ arara --timeout 3000 doc9.tex
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc10.tex' (size: 63 bytes, last modified: 05/29/2018
15:24:06), please wait.

(PDFTeX) PDFTeX engine ..............................   ERROR

The system command execution reached the provided timeout value
and was aborted. If the time was way too short, make sure to
provide a longer value. There are more details available on this
exception:

DETAILS ------------------------------------------------------------
Timed out waiting for java.lang.UNIXProcess@6b53e23f to finish,
timeout: 3000 milliseconds, executed command [pdftex, doc10.tex]

Total: 3.37 seconds
```

If left unattended, this particular execution would never finish (and probably crash the engine at a certain point), as expected by the recursive calls without a proper fixed point. The `--timeout` option was set at 3000 milliseconds and the task was aborted when the time limit was reached. Note that the tool raised an error about it.

`-V` `--version`

This option, as the name indicates, prints the current version. It also prints the current revision and a list of libraries with their corresponding

licenses. Finally, it simply exits the application. Note that this option has the second highest priority over the others.

`-v` `--verbose`
: This option enables the verbose mode of arara, as seen in Section 3.1, on page 30. It also enables all streams to be flushed directly to the terminal, including potential user input interactions (the exact opposite of silent mode). This option can also be activated by default in the configuration file (see Section 4.2, on page 43).

`-s` `--silent`
: This option disables the verbose mode of arara (thus activating the default silent mode), if previously enabled by a proper configuration file (see Chapter 4, on page 42). It is important to note that this command line option has higher priority over the `--verbose` counterpart.

You can combine options, use long or short variations interchangeably and write them in any order, provided that a file name is given at some point in the command line, otherwise the usage will be printed. Use the provided features in order to enhance and optimize your automation workflow.

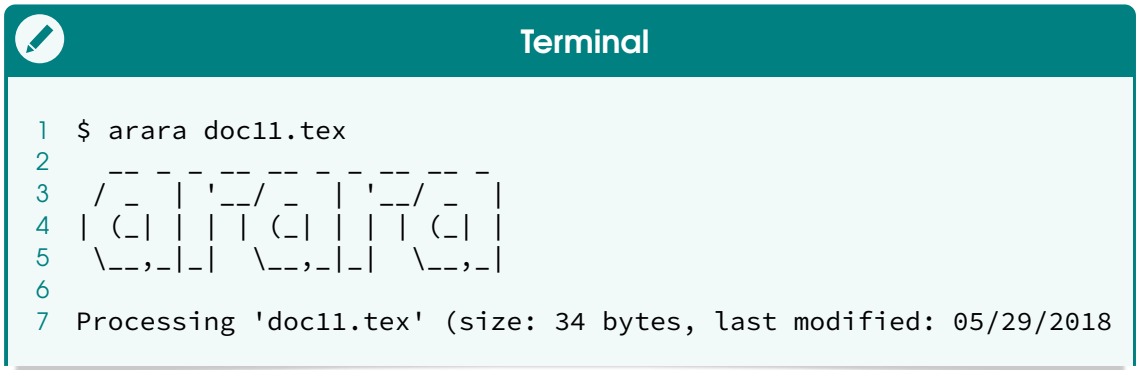
3.3 File name lookup

arara, as a command line application, provides support for a restricted range of file types. In particular, the tool recognizes five file types based on their extensions. These types are presented as follows, as well as the lookup order.

attempt 1	*attempt 2*	*attempt 3*	*attempt 4*	*attempt 5*
`tex`	`dtx`	`ltx`	`drv`	`ins`

Note that other extensions can be added through a proper mapping in the configuration file, as well as modifying the lookup order. This feature is detailed later on, in Section 4.2, on page 43. arara employs the following scheme for file name lookup:

– First and foremost, if the provided file name already contains a valid extension, the tool attempts an exact match. If the file exists, it will be selected. This is the best approach if your working directory contains other files sharing the same base name.



Terminal (ctd.)

```
19:40:35), please wait.

(PDFTeX) PDFTeX engine ................................ SUCCESS

Total: 0.69 seconds
```

– If the provided file name has an unsupported extension or no extension at all, the tool iterates through the list of default extensions, appending the current element to the file name and attempting an exact match. If the file exists, it will be selected.

Terminal

```
$ arara doc11
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing 'doc11.tex' (size: 34 bytes, last modified: 05/29/2018
19:40:35), please wait.

(PDFTeX) PDFTeX engine ................................ SUCCESS

Total: 0.69 seconds
```

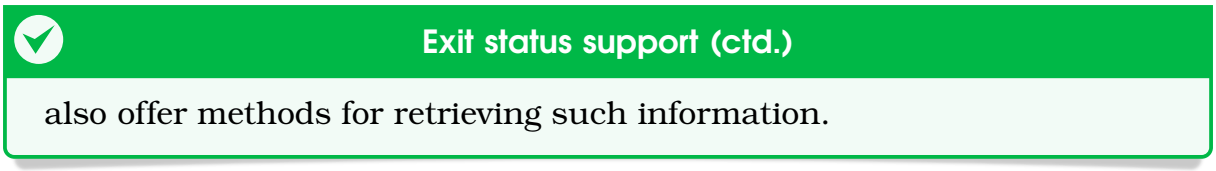
It is highly recommended to use complete file names with our tool, in order to ensure the correct file is being processed. If your command line interpreter features tab completion, you can use it to automatically fill partially typed file names from your working directory.

Exit status support

arara follows the good practices of software development and provides three values for exit status, so our tool can be programmatically used in scripts and other complex workflows.

0	Successful execution
1	One of the rules failed
2	An exception was raised

Please refer to the documentation of your favourite command line interpreter to learn more about exit status captures. Programming languages


Exit status support (ctd.)
also offer methods for retrieving such information.



Configuration file

arara provides a persistent model of modifying the underlying execution behaviour or enhancing the execution workflow through the concept of a configuration file. This chapter provides the basic structure of that file, as well as details on the file lookup in the operating system.

4.1 File lookup

Our tool looks for the presence of at least one of four very specific files before execution. These files are presented as follows. Observe that the directories must have the correct permissions for proper lookup and access. The lookup order is also presented.

attempt 1	*attempt 2*	*attempt 3*	*attempt 4*
`.araraconfig.yaml`	`araraconfig.yaml`	`.arararc.yaml`	`arararc.yaml`

From version 4.0 on, arara provides two approaches regarding the location of a configuration file. They dictate how the execution should behave and happen from a user perspective, and are described as follows.

global configuration file
: For this approach, the configuration file should be located at `USER_HOME` which is the home directory of the current user. All subsequent executions of arara will read this configuration file and apply the specified settings accordingly. However, it is important to note that this approach has the lowest lookup priority, which means that a local configuration, presented as follows, will always supersede a global counterpart.

local configuration file
: For this approach, the configuration file should be located at `USER_DIR` which is the working directory associated with the current execution. This directory can also be interpreted as the one relative to the processed file. This approach offers a project-based solution for complex workflows, e.g, a thesis or a book. However, arara must be executed within the working directory, or the local configuration file lookup will fail. Observe that this approach has the highest lookup priority, which means that it will always supersede a global configuration.

Beware of empty configuration files

A configuration file should never be empty, otherwise arara will complain about it. Make sure to populate it with at least one key, or do not write a configuration file at all. The available keys are described in Section 4.2, on page 43.

If the logging feature is properly enabled, arara will indicate in the corresponding `arara.log` file whether a configuration file was used during the execution and, if so, the corresponding canonical, absolute path. Logging is detailed later on, in Chapter 5, on page 52.

4.2 Basic structure

The following list describes the basic structure of an arara configuration file by presenting the proper elements (or keys, if we consider the proper YAML nomenclature). Observe that elements marked as M are mandatory (i.e, the configuration file *has* to have them in order to work). Similarly, elements marked as O are optional, so you can safely ignore them when writing a configuration file for our tool.

M `!config`

This keyword is mandatory and must be the first line of a configuration file. It denotes the object mapping metadata to be internally used by the tool. Actually, the tool is not too demanding on using it (in fact, you could suppress it entirely and arara will not complain), but it is considered good practice to start a configuration file with a `!config` keyword regardless.

O *string list* `paths`

When looking for rules, arara always searches the default rule path, which consists of a special subdirectory named `rules/` inside another special directory named `ARARA_HOME` (the place where our tool is installed). If no rule is found, the execution halts with an error. The `paths` key specifies a list of directories, represented as plain strings, in which our tool should search for rules. The default path is appended to the list. Then the search happens from the first to the last element, in order.

Example

```
1 paths:
2 - '/home/paulo/rules'
3 - '/opt/paulo/rules'
```

There are three variables available in the `paths` context and are described as follows (note that MVEL variables and orb tags are discussed in Chapter 11.1). A variable will be denoted by `◊variable` in this list.

◊user.home

This variable, as the name implies, holds the value of the absolute, canonical path of USER_HOME which is the home directory of the current user, as plain string. Note that the specifics of the home directory (such as name and location) are defined by the operating system involved.

Example

```
1 paths:
2 - '@{user.home}/rules'
```

◊user.dir

This variable, as the name implies, holds the value of the absolute, canonical path of USER_DIR which is the working directory associated with the current execution, as plain string. Note that the working directory approach requires a user execution strategy to ensure the correct path value.

Example

```
1 paths:
2 - '@{user.dir}/rules'
```

◊user.name

This variable, as the name implies, holds the value of the current user account name, as plain string. On certain operating systems, this value is used to build the home directory structure.

Example

```
1 paths:
2 - '/home/@{user.name}/rules'
```

Observe that the ◊user variable actually holds a map containing three keys (resulting in a map within a map). However, for didactic purposes, it is easier to use the property navigation feature of MVEL, detailed in Section 11.3, on page 157, and consider the map references as three independent variables. You can use property navigation styles interchangeably. Note that you can also precede the path with the special keyword <arara> and save some quotes (see Section 2.1, on page 8). In this specific scenario, the special keyword will be automatically removed afterwards.

Avoid folded and literal styles for scalars in a path

Do not use folded or literal styles for scalars in a path! The orb tag resolution for a path in plain string should be kept as simple as possible, so *always* use the inline style.

0 *string* `language` *default:* `en`

This key sets the language of all subsequent executions of arara according to the provided language code value, as plain string. The default language is set to English. Also, it is very important to observe that the `--language` command line option can override this setting.

Example

```
language: nl
```

0 *integer* `loops` *default:* `10`

This key redefines the maximum number of loops arara will allow for potentially infinite iterations. Any positive integer can be used as the value for this variable. Also, it is very important to observe that the `--max-loops` command line option can override this setting.

Example

```
loops: 30
```

0 *boolean* `verbose` *default:* `false`

This key activates or deactivates the verbose mode of arara as default mode, according to the associated boolean value. Also, it is very important to observe that the `--verbose` command line option can override this setting if, and only if, this variable holds `false` as the value. Similarly, the `--silent` command line option can override this setting if, and only if, this variable holds `true` as the value.

Example

```
verbose: true
```

0 *boolean* `logging` *default:* `false`

This key activates or deactivates the logging feature of arara as the default behaviour, according to the associated boolean value. Also, it is very important to observe that the `--log` command line option can override

this setting if, and only if, this variable holds `false` as the value.

Example

```
1 logging: true
```

0 *boolean* `header` *default:* `false`

This key modifies the directive extraction, according to the associated boolean value. If enabled, arara will extract all directives from the beginning of the file until it reaches a line that is not empty and it is not a comment. Otherwise, the tool will resort to the default behaviour and extract all directives from the entire file. It is very important to observe that the `--header` command line option can override this setting if, and only if, this variable holds `false` as the value.

Example

```
1 header: false
```

0 *string* `logname` *default:* `arara`

This key modifies the default log file name, according to the associated plain string value, plus the `log` extension. The value cannot be empty or contain invalid characters. There is no orb tag evaluation in this specific context, only a plain string value. The log file will be written by our tool if, and only if, the `--log` command line option is used.

Example

```
1 logname: mylog
```

0 *string* `dbname` *default:* `arara`

This key modifies the default XML database file name, according to the associated plain string value, plus the `xml` extension. The value cannot be empty or contain invalid characters. There is no orb tag evaluation in this specific context, only a plain string value. This database is used by file hashing operations, detailed in Section 6.1, on page 58.

Example

```
1 dbname: mydb
```

0 *string* `laf` *default:* `none`

This key modifies the default look and feel class reference, i.e, the appearance of GUI widgets provided by our tool, according to the associated plain string value. The value cannot be empty or contain invalid characters. There is no orb tag evaluation in this specific context, only a plain string value. This look and feel setting is used by UI methods, detailed in Section 6.7, on page 84. Note that this value is used by the underlying Java runtime environment, so a full qualified class name is expected.

Example

```
1 laf: 'javax.swing.plaf.nimbus.NimbusLookAndFeel'
```

Special keywords for the look and feel setting

Look and feel values other than the default provided by Java offer a more pleasant visual experience to the user, so if your rules or directives employ UI methods (detailed in Section 6.7, on page 84), it might be interesting to provide a value to the `laf` key. At the time of writing, arara provides two special keywords that are translated to the corresponding fully qualified Java class names:

`none`	Default look and feel
`system`	System look and feel

The system look and feel, of course, offers the best option of all since it mimics the native appearance of graphical applications in the underlying system. However, some systems might encounter slow rendering times when this option is used, so your mileage might vary.

0 *string map* `preambles`

This key holds a string map containing predefined preambles for later use with the `--preamble` option (see Section 3.2, on page 32). Note that each map key must be unique. Additionally, it it is highly recommended to use lowercase letters without spaces, accents or punctuation symbols, as key values. Only directives, line breaks and conditionals are recognized.

Example

```
1 preambles:
2   twopdftex: |
3     % arara: pdftex
4     % arara: pdftex
```

Literal style when defining a preamble

When defining preambles in the configuration file, *always* use the literal style for scalar blocks. The reason for this requirement is the proper retention of line breaks, which are significant when parsing the strings into proper directive lines. Using the folded style in this particular scenario will almost surely be problematic.

O *file type list* `filetypes`

This key holds a list of file types supported by arara when searching for a file name, as well as their corresponding directive lookup patterns. In order to properly set a file type, the keys used in this specification are defined inside the `filetypes→` context and presented as follows.

M `filetypes→` `extension`

This key, as the name implies, holds the file extension, represented as a plain string and without the leading dot (unless it is part of the extension). An extension is an identifier specified as a suffix to the file name and indicates a characteristic of the corresponding content or intended use. Observe that this key is mandatory when specifying a file type, as our tool does not support files without a proper extension.

Example

```
1 extension: c
```

M O `filetypes→` `pattern`

This key holds the directive lookup pattern as a regular expression (which is, of course, represented as a plain string). When introducing a new file type, arara must know how to interpret each line and how to properly find and extract directives, hence this key. Observe that this key is marked as optional and mandatory. The reason for such an unusual indication highly depends on the current scenario and is illustrated as follows.

– The `pattern` key is entirely *optional* for known file types (presented in Section 3.3, on page 39, and henceforth named *default* file types), in case you just want to modify the file name lookup order. It is important to observe that default file types already have their directive lookup patterns set, which incidentally are the same, presented as follows.

Default regular expression pattern for known file types

```
1 ^\s*%\s+
```

– The `pattern` key is *mandatory* for new file types and for overriding existing patterns for default file types. Make sure to provide a valid regular expression as key value. It is very important to note that, regardless of the underlying pattern (default or provided through this key), the special `arara:` keyword is immutable and thus included by our tool in every directive lookup pattern.

Example

```
pattern: ^\s*//\s*
```

For instance, let us reverse the default file name lookup order presented in Section 3.3, on page 39. Since the default lookup patterns will be preserved, the corresponding `pattern` keys can be safely omitted. Now it is just a matter of rearranging the entries in the desired order, presented as follows.

Example

```
filetypes:
- extension: ins
- extension: drv
- extension: ltx
- extension: dtx
- extension: tex
```

If a default file type is included in the `filetypes` list but others from the same tier are left out, these file types not on the list will implicitly have the lowest priority over the explicit list element during the file name lookup, although still respecting their original lookup order modulo the specified file type. For instance, consider the following list:

Example

```
filetypes:
- extension: ins
- extension: drv
```

According to the previous example, three out of five default file types were deliberately left out of the `filetypes` list. As expected, the two default file types provided to this list will have the highest priority during the file name lookup. It is important to note that arara will always honour the original lookup order for omitted default file types, yet favouring the explicit elements. The following list is semantically equivalent to the previous example.

Example

```
1 filetypes:
2 - extension: ins
3 - extension: drv
4 - extension: tex
5 - extension: dtx
6 - extension: ltx
```

The following example introduces the definition of a new file type to support `c` files. Observe that, for this specific scenario, the `pattern` key is mandatory, as previously discussed. The resulting list is presented as follows, including the corresponding regular expression pattern.

Example

```
1 filetypes:
2 - extension: c
3   pattern: ^\s*//\s*
```

It is important to note that, if no default file type is explicitly specified, as seen in previous example, the original list of default file types will have the highest priority over the `filetypes` values during the file name lookup. The following list is semantically equivalent to the previous example.

Example

```
1 filetypes:
2 - extension: tex
3 - extension: dtx
4 - extension: ltx
5 - extension: drv
6 - extension: ins
7 - extension: c
8   pattern: ^\s*//\s*
```

Do not escape backslashes

When writing a file type pattern, there is no need for escaping backslashes as one does for strings in a typical programming language (including MVEL expressions). In this specific scenario, key values are represented as plain, literal strings.

However, please note that character escaping might be required by



Do not escape backslashes (ctd.)

the underlying regular expression in some scenarios (i.e, a literal dot in the pattern). It is highly recommended to consult a proper regular expression documentation for a comprehensive overview.

Since arara allows four different names for configuration files, as well as global and local approaches, it is highly advisable to run our tool with the `--log` command line option enabled, in order to easily identify which file was considered for that specific execution. The logging feature is discussed later on, in Chapter 5, on page 52.



five

Logging

The logging feature of arara, as discussed earlier on, is activated through either the `--log` command line option (Section 3.2, on page 32) or the equivalent key in the configuration file (Section 4.2, on page 43). This chapter covers the basic structure of a typical log file provided by our tool, including the important blocks that can be used to identify potential issues. The following example is used to illustrate this feature:



Source file

`doc12.tex`

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

When running our tool on the previous example with the `--log` command line option (otherwise, the logging framework will not provide a file at all), we will obtain the expected `arara.log` log file containing the most significant events that happened during this particular execution, as well as details regarding the underlying operating system. The contents of this file are discussed below. Note that timestamps were deliberated removed from the log entries in order to declutter the output, and line breaks were included in order to easily spot each entry.

5.1 System information

The very first entry to appear in the log file is the current version of arara followed by a revision number. The revision number acts as a counter for the last review on the major version. The counter starts at 1 to denote the first release in the version 4.0 series. The revision number is also important to indicate possible new features introduced later on, in the application.



Log file

```
Welcome to arara 4.0 (revision 1)!
```

The following entries in the log file are the absolute path of the current deployment of arara (line 1), details about the current Java virtual machine (namely, vendor and absolute path, in lines 2 and 3, respectively), the underlying operating system information (namely, system name, architecture and eventually the kernel version, in line 4), home and working directories (lines 5 and 6, respectively), and the absolute path of the applied configuration file, if any (line 7). This block is very important to help with tracking possible issues related to the underlying operating system and the tool configuration itself.



Log file

```
1 ::: arara @ /opt/paulo/arara
2 ::: Java 1.8.0_171, Oracle Corporation
3 ::: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.171-4.b10.fc28.x86_64/jre
4 ::: Linux, amd64, 4.16.12-300.fc28.x86_64
5 ::: user.home @ /home/paulo
6 ::: user.dir @ /home/paulo/Documents
7 ::: CF @ [none]
```



A privacy note

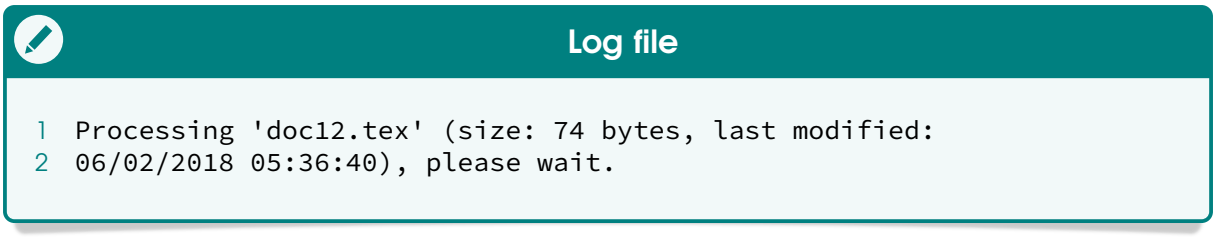
I understand that the previous entries containing information about the underlying operating system might pose as a privacy threat to some users. However, it is worth noting that arara does not share any sensitive information about your system, as entries are listed in the log file for debugging purposes only, locally in your computer.

From experience, these entries greatly help our users to track down errors in the execution, as well as learning more about the underlying operating system. However, be mindful of sharing your log file! Since the log file contains structured blocks, it is highly advisable to selectively choose the ones relevant to the current discussion.

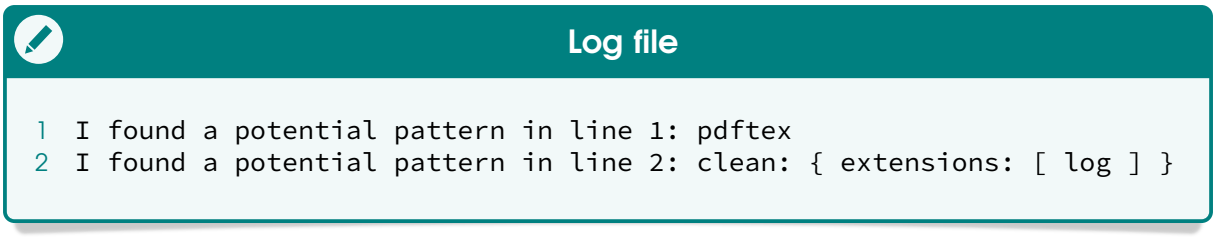
It is important to observe that localized messages are also applied to the log file. If a language other than English is selected, either through the `--language` command line option or the equivalent key in the configuration file, the logging framework will honour the current setting and entries will be available in the specified language. Having a log file in your own language might mitigate the traumatic experience of error tracking for TEX newbies.

5.2 Directive extraction

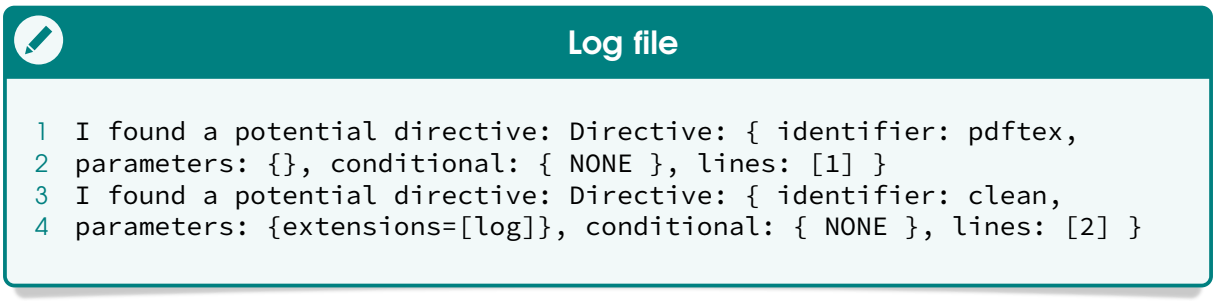
The following block in the log file refers to file information and directive extraction. First, as with the terminal output counterpart, the tool will display details about the file being processed, including size and modification status:



The next entries refer to finding potential directive patterns in the code, including multiline support. All matching patterns contain the corresponding line numbers. Note that these numbers might refer to incorrect lines in the code if the `--preamble` command line option is used.



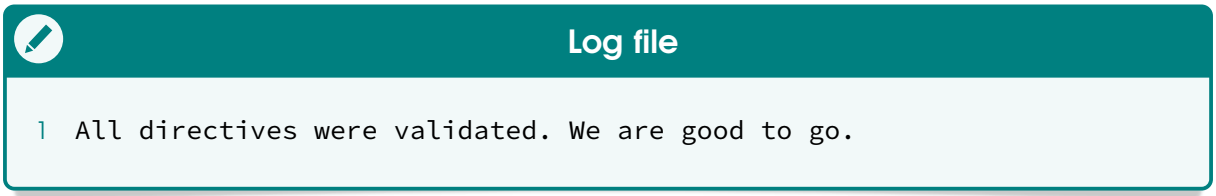
When all matching patterns are collected from the code in the previous phase, arara composes the directives accordingly, including potential parameters and conditionals. Observe that all directives have an associated list of line numbers from which they were originally composed. This phase is known as *directive extraction*.



In this phase, directives are correctly extracted and composed, but are yet to be validated regarding invalid or reserved parameter keys. The tool then proceeds to validate parameters and normalize such directives.

5.3 Directive normalization

Once all directives are properly composed, the tool checks for potential inconsistencies, such as invalid or reserved parameter keys. Then all directives are validated and internally mapped with special parameters, as previously described in Section 5.3, on page 54.



After validation, all directives are listed in a special block in the log file, including potential parameters and conditionals. This phase is known as *directive normalization*. Note that the special parameters are already included, regardless of the directive type. This particular block can be used specially for debugging purposes, since it contains all details regarding directives.



Log file

```
1 -------------------------- DIRECTIVES --------------------------
2 Directive: { identifier: pdftex, parameters:
3 {reference=/home/paulo/Documents/doc12.tex,
4 file=doc12.tex}, conditional: { NONE }, lines: [1] }
5 Directive: { identifier: clean, parameters: {extensions=[log],
6 file=doc12.tex, reference=/home/paulo/Documents/doc12.tex},
7 conditional: { NONE }, lines: [2] }
8 ----------------------------------------------------------------
```

Note, however, that potential errors in directive conditionals, as well as similar inconsistencies in the corresponding rules, can only be caught at runtime. The next phase covers proper interpretation based on the provided directives.

5.4 Rule interpretation

Once all directives are normalized, arara proceeds to interpret the potential conditionals, if any, and the corresponding rules. Note that, when available, the conditional type dictates whether the rule should be interpreted first or not. For each rule, the tool informs the identifier and the absolute path of the corresponding YAML file. In this specific scenario, the rule is part of the default rule pack released with our tool:



Log file

```
1 I am ready to interpret rule 'pdftex'.
2 Rule location: '/opt/paulo/arara/rules'
```

For each task (or subtask, as it is part of a rule task) defined in the rule context, arara will interpret it and return the corresponding system command. The return types can be found in Section 2.1, on page 8. In this specific scenario, there is just one task associated with the `pdftex` rule. Both task name and system command are shown:



Log file

```
1 I am ready to interpret task 'PDFTeX engine' from rule 'PDFTeX'.
2 System command: [ pdftex, doc12.tex ]
```

After proper task interpretation, the underlying execution library of arara executes the provided system command and includes the output from both output and error streams in an *output buffer* block inside the log file.



Log file

```
---------------------- BEGIN OUTPUT BUFFER ----------------------
This is pdfTeX, Version 3.14159265-2.6-1.40.19 (TeX Live 2018)
(preloaded format=pdftex)
 restricted \write18 enabled.
entering extended mode
(./doc12.tex [1{/usr/local/texlive/2018/texmf-var/fonts/map/
pdftex/updmap/pdfte
x.map}] )</usr/local/texlive/2018/texmf-dist/fonts/type1/
public/amsfonts/cm/cmr
10.pfb>
Output written on doc12.pdf (1 page, 11849 bytes).
Transcript written on doc12.log.
----------------------- END OUTPUT BUFFER -----------------------
```

Observe that the above output buffer block contains the relevant information about the `pdftex` execution on the provided file. It is possible to write a shell script to extract these blocks from the log file, as a means to provide individual information on each execution. Finally, the task result is shown:



Log file

```
Task result: SUCCESS
```

The execution proceeds to the next directive in the list and then interprets the `clean` rule. The same steps previously described are applied in this scenario. Also note that the output buffer block is deliberately empty due to the nature of the underlying system command, as removal commands such as `rm` do not provide output at all when successful.



Log file

```
I am ready to interpret rule 'clean'.
Rule location: '/opt/paulo/arara/rules'
I am ready to interpret task 'Cleaning feature' from rule 'Clean'.
System command: [ rm, -f, doc12.log ]
---------------------- BEGIN OUTPUT BUFFER ----------------------

----------------------- END OUTPUT BUFFER -----------------------
Task result: SUCCESS
```



Empty output buffer

If the system command is simply a boolean value, the corresponding block will remain empty. Also note that not all commands from the underlying operating system path provide proper stream output, so the output buffer block might be empty in certain corner scenarios. This is the case, for example, of the provided `clean` rule.

Finally, as the last entry in the log file, the tool shows the execution time, in seconds. As previously mentioned, the execution time has a very simple precision and should not be considered for command profiling.



Log file

```
1 Total: 0.33 seconds
```

The logging feature provides a consistent framework for event recording. It is highly recommended to include at least the `--log` command line option (or enable it in the configuration file) in your typical automation workflow, as relevant information is gathered into a single consolidated report.



six

Methods

arara features several helper methods available in directive conditional and rule contexts which provide interesting features for enhancing the user experience, as well as improving the automation itself. This chapter provides a list of such methods. It is important to observe that virtually all classes from the Java runtime environment can be used within MVEL expressions, so your mileage might vary.



A note on writing code

As seen in Chapter 11, on page 154, Java and MVEL code be used interchangeably within expressions and orb tags, including instantiation of classes into objects and invocation of methods. However, be mindful of explicitly importing Java packages and classes through the classic `import` statement, as MVEL does not automatically handle imports, or an exception will surely be raised. Alternatively, you can provide the full qualified name to classes as well.

Methods are listed with their complete signatures, including potential parameters and corresponding types. Also, the return type of a method is denoted by `△type` and refers to a typical Java data type (either class or primitive). Do not worry too much, as there are illustrative examples. A method available in the directive conditional context will be marked by C next to the corresponding signature. Similarly, an entry marked by R denotes that the corresponding method is available in the rule context.

6.1 Files

This section introduces methods related to file handling, searching and hashing. It is important to observe that no exception is thrown in case of an anomalous method call. In this particular scenario, the methods return empty references, when applied.

R `◊getOriginalFile()` `△String`

This method returns the original file name, as plain string, regardless of a potential override through the special `files` parameter in the directive mapping, as seen in Section 2.2, on page 18.

Example

```
1 if (file == getOriginalFile()) {
2     System.out.println("The 'file' variable
3         was not overridden.");
4 }
```

R ◊getOriginalReference() △File

This method returns the original file reference, as a File object, regardless of a potential reference override indirectly through the special files parameter in the directive mapping, as seen in Section 2.2, on page 18.

Example

```
1 if (reference.equals(getOriginalFile())) {
2     System.out.println("The 'reference' variable
3         was not overridden.");
4 }
```

C R ◊currentFile() △File

This method returns the file reference, as a File object, for the current directive. It is important to observe that, from version 4.0 on, arara replicates the directive when the special files parameter is detected amongst the parameters, so each instance will have a different reference.

Example

```
1 % arara: pdflatex if currentFile().getName() == 'thesis.tex'
```

C R ◊toFile(String reference) △File

This method returns a file (or directory) reference, as a File object, based on the provided string. Note that the string can refer to either a relative entry or a complete, absolute path. It is worth mentioning that, in Java, despite the curious name, a File object can be assigned to either a file or a directory.

Example

```
1 f = toFile('thesis.tex');
```

R ◊getBasename(File file) △String

This method returns the base name (i.e, the name without the associated

extension) of the provided `File` reference, as a string. Observe that this method ignores a potential path reference when extracting the base name. For a complete base name extraction with full path support, please refer to the `◊getFullBasename` methods. Also, this method will throw an exception if the provided reference is not a proper file.

Example

```
1 basename = getBasename(toFile('thesis.tex'));
```

R `◊getBasename(String reference)` `△String`

This method returns the base name (i.e, the name without the associated extension) of the provided `String` reference, as a string. Observe that this method ignores a potential path reference when extracting the base name. For a complete base name extraction with full path support, please refer to the `◊getFullBasename` methods.

Example

```
1 basename = getBasename('thesis.tex');
```

R `◊getFullBasename(File file)` `△String`

This method returns the full base name (i.e, the name without the associated extension, as well as the potential path reference) of the provided `File` reference, as a string. This method will throw an exception if the provided reference is not a proper file.

Example

```
1 basename = getFullBasename(toFile('/home/paulo/thesis.tex'));
```

R `◊getFullBasename(String reference)` `△String`

This method returns the full base name (i.e, the name without the associated extension, as well as the potential path reference) of the provided `String` reference, as a string. As the path discovery requires an underlying file conversion, this method will throw an exception if the provided reference is not a proper file.

Example

```
1 basename = getFullBasename('/home/paulo/thesis.tex');
```

R ◊getFiletype(File file) △String

This method returns the file type (i.e, the associated extension specified as a suffix to the name, typically delimited with a full stop) of the provided File reference, as a string. This method will throw an exception if the provided reference is not a proper file. An empty string is returned if, and only if, the provided file name has no associated extension.

Example

```
1 extension = getFiletype(toFile('thesis.pdf'));
```

R ◊getFiletype(String reference) △String

This method returns the file type (i.e, the associated extension specified as a suffix to the name, typically delimited with a full stop) of the provided String reference, as a string. An empty string is returned if, and only if, the provided file name has no associated extension.

Example

```
1 extension = getFiletype('thesis.pdf');
```

C R ◊exists(File file) △boolean

This method, as the name implies, returns a boolean value according to whether the provided File reference exists. Observe that the provided reference can be either a file or a directory.

Example

```
1 % arara: bibtex if exists(toFile('references.bib'))
```

C R ◊exists(String extension) △boolean

This method returns a boolean value according to whether the base name of the ◊currentFile reference (i.e, the name without the associated extension) as a string concatenated with the provided String extension exists. This method eases the checking of files which share the current file name modulo extension (e.g, log and auxiliary files). Note that the provided string refers to the extension, not the file name.

Example

```
1 % arara: pdftex if exists('tex')
```

C R `◊missing(File file)` `△boolean`

This method, as the name implies, returns a boolean value according to whether the provided `File` reference does not exist. It is important to observe that the provided reference can be either a file or a directory.

Example

```
1 % arara: pdftex if missing(toFile('thesis.pdf'))
```

C R `◊missing(String extension)` `△boolean`

This method returns a boolean value according to whether the base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the provided `String` extension does not exist. This method eases the checking of files which share the current file name modulo extension (e.g, log and auxiliary files). Note that the provided string refers to the extension, not the file name.

Example

```
1 % arara: pdftex if missing('pdf')
```

C R `◊changed(File file)` `△boolean`

This method returns a boolean value according to whether the provided `File` reference has changed since last verification, based on a traditional cyclic redundancy check. The file reference, as well as the associated hash, is stored in a XML database file named `arara.xml` located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in Section 4.2, on page 43). The method semantics (including the return values) is presented as follows.

file exists?	*entry exists?*	*has changed?*	*DB action*	*result*
☑	☑	☑	update	☑
☑	☑	☐	—	☐
☑	☐	—	insert	☑
☐	☐	—	—	☐
☐	☑	—	remove	☑

It is important to observe that this method *always* performs a database operation, either an insertion, removal or update on the corresponding entry. When using `◊changed` within a logical expression, make sure the evaluation order is correct, specially regarding the use of short-circuiting operations. In some scenarios, order does matter.

Example

```
1 % arara: pdflatex if changed(toFile('thesis.tex'))
```

Short-circuit evaluation

According to the Wikipedia entry, a *short-circuit evaluation* is the semantics of some boolean operators in some programming languages in which the second argument is executed or evaluated only if the first argument does not suffice to determine the value of the expression. In Java (and consequently MVEL), both short-circuit and standard boolean operators are available.

CRC as a hashing algorithm

arara internally relies on a CRC32 implementation for file hashing. This particular choice, although not designed for hashing, offers an interesting trade-off between speed and quality. Besides, since it is not computationally expensive as strong algorithms such as MD5 and SHA1, CRC32 can be used for hashing typical TEX documents and plain text files with little to no collisions.

C R ◊changed(String extension) △boolean

This method returns a boolean value according to whether the base name of the ◊currentFile reference (i.e, the name without the associated extension) as a string concatenated with the provided String extension has changed since last verification, based on a traditional cyclic redundancy check. The file reference, as well as the associated hash, is stored in a XML database file named arara.xml located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in Section 4.2, on page 43). The method semantics (including the return values) is presented as follows.

file exists?	*entry exists?*	*has changed?*	*DB action*	*result*
☑	☑	☑	update	☑
☑	☑	☐	—	☐
☑	☐	—	insert	☑
☐	☐	—	—	☐
☐	☑	—	remove	☑

It is important to observe that this method *always* performs a database operation, either an insertion, removal or update on the corresponding entry. When using ◊changed within a logical expression, make sure the

evaluation order is correct, specially regarding the use of short-circuiting operations. In some scenarios, order does matter.

Example

```
1 % arara: pdflatex if changed('tex')
```

C R `◊unchanged(File file)` `△boolean`

This method returns a boolean value according to whether the provided `File` reference has not changed since last verification, based on a traditional cyclic redundancy check. The file reference, as well as the associated hash, is stored in a XML database file named `arara.xml` located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in Section 4.2, on page 43). The method semantics (including the return values) is presented as follows.

file exists?	*entry exists?*	*has changed?*	*DB action*	*result*
☑	☑	☑	update	☐
☑	☑	☐	—	☑
☑	☐	—	insert	☐
☐	☐	—	—	☑
☐	☑	—	remove	☐

It is important to observe that this method *always* performs a database operation, either an insertion, removal or update on the corresponding entry. When using `◊unchanged` within a logical expression, make sure the evaluation order is correct, specially regarding the use of short-circuiting operations. In some scenarios, order does matter.

Example

```
1 % arara: pdflatex if !unchanged(toFile('thesis.tex'))
```

C R `◊unchanged(String extension)` `△boolean`

This method returns a boolean value according to whether the base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the provided `String` extension has not changed since last verification, based on a traditional cyclic redundancy check. The file reference, as well as the associated hash, is stored in a XML database file named `arara.xml` located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in Section 4.2, on page 43). The method semantics (including the return values) is presented as follows.

file exists?	*entry exists?*	*has changed?*	*DB action*	*result*
☑	☑	☑	update	☐
☑	☑	☐	—	☑
☑	☐	—	insert	☐
☐	☐	—	—	☑
☐	☑	—	remove	☐

It is important to observe that this method *always* performs a database operation, either an insertion, removal or update on the corresponding entry. When using `◊unchanged` within a logical expression, make sure the evaluation order is correct, specially regarding the use of short-circuiting operations. In some scenarios, order does matter.

Example

```
1 % arara: pdflatex if !unchanged('tex')
```

R `◊writeToFile(File file, String text, boolean append)` `△boolean`

This method performs a write operation based on the provided parameters. In this case, the method writes the `String` text to the `File` reference and returns a boolean value according to whether the operation was successful. The third parameter holds a `boolean` value and acts as a switch indicating whether the text should be appended to the existing content of the provided file. Keep in mind that the existing content of a file is always overwritten if this switch is disabled. Also, note that the switch has no effect if the file is being created at that moment. It is important to observe that this method does not raise any exception.

Example

```
1 result = writeToFile(toFile('foo.txt'), 'hello world', false);
```

Read and write operations in Unicode

arara *always* uses Unicode as the encoding format for read and write operations. This decision is deliberate as a means to offer a consistent representation and handling of text. Unicode can be implemented by different character encodings. In our case, the tool relies on UTF-8, which uses one byte for the first 128 code points, and up to 4 bytes for other characters. The first 128 Unicode code points are the ASCII characters, which means that any ASCII text is also UTF-8 text.

File system permissions

Most file systems have methods to assign permissions or access rights to specific users and groups of users. These permissions control the ability of the users to view, change, navigate, and execute the contents of the file system. Keep in mind that read and write operations depend on such permissions.

R ◊writeToFile(String reference, String text, boolean append) △boolean

This method performs a write operation based on the provided parameters. In this case, the method writes the `String` text to the `String` reference and returns a boolean value according to whether the operation was successful. The third parameter holds a `boolean` value and acts as a switch indicating whether the text should be appended to the existing content of the provided file. Keep in mind that the existing content of a file is always overwritten if this switch is disabled. Also, note that the switch has no effect if the file is being created at that moment. It is important to observe that this method does not raise any exception.

Example

```
result = writeToFile('foo.txt', 'hello world', false);
```

R ◊writeToFile(File file, List<String> lines, boolean append) △boolean

This method performs a write operation based on the provided parameters. In this case, the method writes the `List<String>` lines to the `File` reference and returns a boolean value according to whether the operation was successful. The third parameter holds a `boolean` value and acts as a switch indicating whether the text should be appended to the existing content of the provided file. Keep in mind that the existing content of a file is always overwritten if this switch is disabled. Also, note that the switch has no effect if the file is being created at that moment. It is important to observe that this method does not raise any exception.

Example

```
result = writeToFile(toFile('foo.txt'),
          [ 'hello world', 'how are you?' ], false);
```

R ◊writeToFile(String reference, List<String> lines, boolean append) △boolean

This method performs a write operation based on the provided parameters. In this case, the method writes the `List<String>` lines to the `String` reference and returns a boolean value according to whether the operation

was successful. The third parameter holds a `boolean` value and acts as a switch indicating whether the text should be appended to the existing content of the provided file. Keep in mind that the existing content of a file is always overwritten if this switch is disabled. Also, note that the switch has no effect if the file is being created at that moment. It is important to observe that this method does not raise any exception.

Example

```
result = writeToFile('foo.txt', [ 'hello world',
          'how are you?' ], false);
```

R `◊readFromFile(File file)` `△List<String>`

This method performs a read operation based on the provided parameter. In this case, the method reads the content from the `File` reference and returns a `List<String>` object representing the lines as a list of strings. If the reference does not exist or an exception is raised due to access permission constraints, the `◊readFromFile` method returns an empty list. Keep in mind that, as a design decision, UTF-8 is *always* used as character encoding for read operations.

Example

```
lines = readFromFile(toFile('foo.txt'));
```

R `◊readFromFile(String reference)` `△List<String>`

This method performs a read operation based on the provided parameter. In this case, the method reads the content from the `String` reference and returns a `List<String>` object representing the lines as a list of strings. If the reference does not exist or an exception is raised due to access permission constraints, the `◊readFromFile` method returns an empty list. Keep in mind that, as a design decision, UTF-8 is *always* used as character encoding for read operations.

Example

```
lines = readFromFile('foo.txt');
```

R
```
◊listFilesByExtensions(File file,
   List<String> extensions, boolean recursive)
```
`△List<File>`

This method performs a file search operation based on the provided parameters. In this case, the method list all files from the provided `File` reference according to the `List<String>` extensions as a list of strings,

and returns a `List<File>` object representing all matching files. The leading full stop in each extension must be omitted, unless it is part of the search pattern. The third parameter holds a `boolean` value and acts as a switch indicating whether the search must be recursive, i.e, whether all subdirectories must be searched as well. If the reference is not a proper directory or an exception is raised due to access permission constraints, the `◊listFilesByExtensions` method returns an empty list.

Example

```
1  files = listFilesByExtensions(toFile('/home/paulo/Documents'),
2          [ 'aux', 'log' ], false);
```

R `◊listFilesByExtensions(String reference, List<String> extensions, boolean recursive)` `△List<File>`

This method performs a file search operation based on the provided parameters. In this case, the method list all files from the provided `String` reference according to the `List<String>` extensions as a list of strings, and returns a `List<File>` object representing all matching files. The leading full stop in each extension must be omitted, unless it is part of the search pattern. The third parameter holds a `boolean` value and acts as a switch indicating whether the search must be recursive, i.e, whether all subdirectories must be searched as well. If the reference is not a proper directory or an exception is raised due to access permission constraints, the `◊listFilesByExtensions` method returns an empty list.

Example

```
1  files = listFilesByExtensions('/home/paulo/Documents',
2          [ 'aux', 'log' ], false);
```

R `◊listFilesByPatterns(File file, List<String> patterns, boolean recursive)` `△List<File>`

This method performs a file search operation based on the provided parameters. In this case, the method lists all files from the provided `File` reference according to the `List<String>` patterns as a list of strings, and returns a `List<File>` object representing all matching files. The pattern specification is described below. The third parameter holds a `boolean` value and acts as a switch indicating whether the search must be recursive, i.e, whether all subdirectories must be searched as well. If the reference is not a proper directory or an exception is raised due to access permission constraints, the `◊listFilesByPatterns` method returns an empty list. It is very important to observe that this file search operation might be slow depending on the provided directory. It is highly advisable to not rely on recursive searches whenever possible.

Patterns for file search operations

arara employs wildcard filters as patterns for file search operations. Testing is case sensitive by default. The wildcard matcher uses the characters `?` and `*` to represent a single or multiple wildcard characters. This is the same as often found on typical terminals.

Example

```
files = listFilesByPatterns(toFile('/home/paulo/Documents'),
        [ '*.tex', 'foo?.txt' ], false);
```

R ◇ `listFilesByPatterns(String reference, List<String> patterns, boolean recursive)` △ `List<File>`

This method performs a file search operation based on the provided parameters. In this case, the method lists all files from the provided `String` reference according to the `List<String>` patterns as a list of strings, and returns a `List<File>` object representing all matching files. The pattern specification follows a wildcard filter. The third parameter holds a `boolean` value and acts as a switch indicating whether the search must be recursive, i.e, whether all subdirectories must be searched as well. If the reference is not a proper directory or an exception is raised due to access permission constraints, the `◇listFilesByPatterns` method returns an empty list. It is very important to observe that this file search operation might be slow depending on the provided directory. It is highly advisable to not rely on recursive searches whenever possible.

Example

```
files = listFilesByPatterns('/home/paulo/Documents',
        [ '*.tex', 'foo?.txt' ], false);
```

As the methods presented in this section have transparent error handling, the writing of rules and conditionals becomes more fluent and not too complex for the typical user.

6.2 Conditional flow

This section introduces methods related to conditional flow based on *natural boolean values*, i.e, words that semantically represent truth and falsehood signs. Such concept provides a friendly representation of boolean values and eases the use of switches in directive parameters. The tool relies on the following set of natural boolean values:

☑ `yes` `true` `1` `on` ☐ `no` `false` `0` `off`

All elements from the provided set of natural boolean values can be used interchangeably in directive parameters. It is important to observe that, from version 4.0 on, arara throws an exception if a value absent from the set is provided to the methods described in this section.

R ◊isTrue(String string) △boolean

This method returns a boolean value according to whether the provided `String` value is contained in the sub-set of natural true boolean values. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the provided value is an empty string, the method returns false.

Example

```
1 result = isTrue('yes');
```

R ◊isFalse(String string) △boolean

This method returns a boolean value according to whether the provided `String` value is contained in the sub-set of natural false boolean values. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the provided value is an empty string, the method returns false.

Example

```
1 result = isFalse('off');
```

R ◊isTrue(String string, Object yes) △Object

This method checks if the first parameter is contained in the sub-set of natural true boolean values. If the result holds true, the second parameter is returned. Otherwise, an empty string is returned. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the first parameter is an empty string, the method returns an empty string.

Example

```
1 result = isTrue('on', [ 'ls', '-la' ]);
```

R ◊isFalse(String string, Object yes) △Object

This method checks if the first parameter is contained in the sub-set of

natural false boolean values. If the result holds true, the second parameter is returned. Otherwise, an empty string is returned. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the first parameter is an empty string, the method returns an empty string.

Example

```
1 result = isFalse('0', 'pwd');
```

R ◊isTrue(String string, Object yes, Object no) △Object

This method checks if the first parameter is contained in the sub-set of natural true boolean values. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the first parameter is an empty string, the method returns the third parameter.

Example

```
1 result = isTrue('on', [ 'ls', '-la' ], 'pwd');
```

R ◊isFalse(String string, Object yes, Object no) △Object

This method checks if the first parameter is contained in the sub-set of natural false boolean values. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the first parameter is an empty string, the method returns the third parameter.

Example

```
1 result = isFalse('0', 'pwd', 'ps');
```

R ◊isTrue(String string, Object yes, Object no, Object fallback) △Object

This method checks if the first parameter is contained in the sub-set of natural true boolean values. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the first parameter is an empty string, the method returns the fourth parameter as default value.

Example

```
1 result = isTrue('on', 'ls', 'pwd', 'who');
```

R ◊ `isFalse(String string, Object yes, Object no, Object fallback)` △ `Object`

This method checks if the first parameter is contained in the sub-set of natural false boolean values. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned. It is worth mentioning that the verification is case insensitive, i.e, upper case and lower case symbols are treated as equivalent. If the first parameter is an empty string, the method returns the fourth parameter as default value.

Example

```
1 result = isFalse('0', 'pwd', 'ps', 'ls');
```

R ◊ `isTrue(boolean value, Object yes)` △ `Object`

This method evaluates the first parameter as a boolean expression. If the result holds true, the second parameter is returned. Otherwise, an empty string is returned.

Example

```
1 result = isTrue(1 == 1, 'yes');
```

R ◊ `isFalse(boolean value, Object yes)` △ `Object`

This method evaluates the first parameter as a boolean expression. If the result holds false, the second parameter is returned. Otherwise, an empty string is returned.

Example

```
1 result = isFalse(1 != 1, 'yes');
```

R ◊ `isTrue(boolean value, Object yes, Object no)` △ `Object`

This method evaluates the first parameter as a boolean expression. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned.

Example

```
1 result = isTrue(1 == 1, 'yes', 'no');
```

R ◊isFalse(boolean value, Object yes, Object no) △Object

This method evaluates the first parameter as a boolean expression. If the result holds false, the second parameter is returned. Otherwise, the third parameter is returned.

Example

```
1 result = isFalse(1 != 1, 'yes', 'no');
```

Supported by the concept of natural boolean values, the methods presented in this section ease the use of switches in directive parameters and can be adopted as valid alternatives for traditional conditional flows, when applied.

6.3 Strings

String manipulation constitutes one of the foundations of rule interpretation in our tool. This section introduces methods for handling such types, as a means to offer high level constructs for users.

R ◊isEmpty(String string) △boolean

This method returns a boolean value according to whether the provided `String` value is empty, i.e, the string length is equal to zero.

Example

```
1 result = isEmpty('not empty');
```

R ◊isNotEmpty(String string) △boolean

This method returns a boolean value according to whether the provided `String` value is not empty, i.e, the string length is greater than zero.

Example

```
1 result = isNotEmpty('not empty');
```

R ◊isEmpty(String string, Object yes) △boolean

This method checks if the first parameter is empty, i.e, if the string length

is equal to zero. If the result holds true, the second parameter is returned. Otherwise, an empty string is returned.

Example

```
result = isEmpty('not empty', 'ps');
```

R ◊isNotEmpty(String string, Object yes) △boolean

This method checks if the first parameter is not empty, i.e, if the string length is greater than zero. If the result holds true, the second parameter is returned. Otherwise, an empty string is returned.

Example

```
result = isNotEmpty('not empty', 'ls');
```

R ◊isEmpty(String string, Object yes, Object no) △boolean

This method checks if the first parameter is empty, i.e, if the string length is equal to zero. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned.

Example

```
result = isEmpty('not empty', 'ps', 'ls');
```

R ◊isNotEmpty(String string, Object yes, Object no) △boolean

This method checks if the first parameter is not empty, i.e, if the string length is greater than zero. If the result holds true, the second parameter is returned. Otherwise, the third parameter is returned.

Example

```
result = isNotEmpty('not empty', 'ls', 'ps');
```

R ◊buildString(Object... objects) △String

This method returns a string based on the provided array of objects, separating each element by one blank space. It is important to observe that empty values are not considered. Also, note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters.

Example

```
1 result = buildString('a', 'b', 'c', 'd');
```

R ◊trimSpaces(String string) △String

This method trims spaces from the provided parameter, i.e, leading and trailing spaces in the `String` reference are removed, and returns the resulting string. It is important to observe that non-boundary spaces inside the string are not removed at all.

Example

```
1 result = trimSpaces('   hello world   ');
```

R ◊addQuotes(String string) △String

This method returns the provided parameter enclosed in double quotes, as a plain string. It is important to observe that there is no automatic quote handling.

Example

```
1 result = addQuotes('to be or not to be');
```

R ◊replicatePattern(String pattern, List<Object> values) △List<Object>

This method replicates the provided pattern to each element of the second parameter and returns the resulting list. The pattern must contain exactly one placeholder. For instance, `%s` denotes a string representation of the provided argument. Please refer to the `Formatter` class reference in the Java documentation for more information on placeholders. This method raises an exception if an invalid pattern is applied.

Example

```
1 names = replicatePattern('My name is %s', [ 'Brent', 'Nicola' ]);
```

C R ◊found(File file, String regex) △boolean

This method returns a boolean value according to whether the content of the provided `File` reference contains at least one match of the provided `String` regular expression. It is important to observe that this method raises an exception if an invalid regular expression is provided as the parameter or if the provided file reference does not exist.

Example

```
1 % arara: pdflatex while found(toFile('article.log'),
2 % arara: --> 'undefined references')
```

C R ◊found(String extension, String regex) △boolean

This method returns a boolean value according to whether the content of the base name of the ◊currentFile reference (i.e, the name without the associated extension) as a string concatenated with the provided String extension contains at least one match of the provided String regular expression. It is important to observe that this method raises an exception if an invalid regular expression is provided as the parameter or if the provided file reference does not exist.

Example

```
1 % arara: pdflatex while found('log', 'undefined references')
```

The string manipulation methods presented in this section constitute an interesting and straightforward approach to handling directive parameters without the usual verbosity in writing typical Java constructs.

6.4 Operating systems

This section introduces methods related to the underlying operating system detection, as a means of providing a straightforward approach to writing cross-platform rules.

R ◊isWindows() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is Microsoft Windows.

Example

```
1 if (isWindows()) { System.out.println('Running Windows.'); }
```

R ◊isLinux() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is a Linux instance.

Example

```
1 if (isLinux()) { System.out.println('Running Linux.'); }
```

R ◊isMac() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is Apple Mac OS.

Example

```
1 if (isMac()) { System.out.println('Running Mac OS.'); }
```

R ◊isUnix() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is any Unix variation.

Example

```
1 if (isUnix()) { System.out.println('Running Unix.'); }
```

R ◊isAIX() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is IBM AIX.

Example

```
1 if (isAIX()) { System.out.println('Running AIX.'); }
```

R ◊isIrix() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is Silicon Graphics Irix.

Example

```
1 if (isIrix()) { System.out.println('Running Irix.'); }
```

R ◊isOS2() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is IBM OS/2 Warp.

Example

```
1 if (isOS2()) { System.out.println('Running OS/2 Warp.'); }
```

R ◊isSolaris() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is Oracle Solaris.

Example

```
1 if (isSolaris()) { System.out.println('Running Solaris.'); }
```

R ◊isCygwin() △boolean

This method returns a boolean value according to whether the underlying operating system vendor is Microsoft Windows and arara is being executed inside a Cygwin environment.

Example

```
1 if (isCygwin()) { System.out.println('Running Cygwin.'); }
```

R ◊isWindows(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is Microsoft Windows. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isWindows('del', 'rm');
```

R ◊isLinux(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is a Linux instance. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isLinux('rm', 'del');
```

R ◊isMac(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is Apple Mac OS. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isMac('ls', 'dir');
```

R ◊isUnix(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is any Unix variation. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isUnix('tree', 'dir');
```

R ◊isAIX(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is IBM AIX. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isAIX('pwd', 'ls');
```

R ◊isIrix(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is Silicon Graphics Irix. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isIrix('ls', 'pwd');
```

R ◊isOS2(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is IBM OS/2 Warp. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isOS2('ls', 'cd');
```

R ◊isSolaris(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is Oracle Solaris. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isSolaris('ls', 'cat');
```

R ◊isCygwin(Object yes, Object no) △Object

This method checks if the underlying operating system vendor is Microsoft Windows and if arara is being executed inside a Cygwin environment. If the result holds true, the first parameter is returned. Otherwise, the second parameter is returned.

Example

```
1 command = isCygwin('ls', 'dir');
```

The methods presented in the section provide useful information to help users write cross-platform rules and thus enhance the automation experience based on specific features of the underlying operating system.

6.5 Type checking

In certain scenarios, a plain string representation of directive parameters might be inadequate or insufficient given the rule requirements. To this end, this section introduces methods related to type checking as a means to provide support and verification for common data types.

R ◊isString(Object object) △boolean

This method returns a boolean value according to whether the provided `Object` object is a string or any extended type.

Example

```
1 result = isString('foo');
```

R `◊isList(Object object)` `△boolean`

This method returns a boolean value according to whether the provided `Object` object is a list or any extended type.

Example

```
1 result = isList([ 1, 2, 3 ]);
```

R `◊isMap(Object object)` `△boolean`

This method returns a boolean value according to whether the provided `Object` object is a map or any extended type.

Example

```
1 result = isMap([ 'Paulo' : 'Palmeiras', 'Carla' : 'Inter' ]);
```

R `◊isBoolean(Object object)` `△boolean`

This method returns a boolean value according to whether the provided `Object` object is a boolean or any extended type.

Example

```
1 result = isBoolean(false);
```

R `◊checkClass(Class clazz, Object object)` `△boolean`

This method returns a boolean value according to whether the provided `Object` object is an instance or a subtype of the provided `Class` class. It is interesting to note that all methods presented in this section internally rely on `◊checkClass` for type checking.

Example

```
1 result = checkClass(List.class, [ 'a', 'b' ]);
```

The methods presented in this section cover the most common types used in directive parameters and should suffice for expressing the rule requirements. If a general approach is needed, please refer to the `◊checkClass` method for checking virtually any type available in the Java environment.

6.6 Classes and objects

From version 4.0 on, arara can be extended at runtime with code from JVM languages, such as Groovy, Scala, Clojure and Kotlin. The tool can load classes from `class` and `jar` files and even instantiate them. This section introduces methods related to class loading and object instantiation.



Ordered pairs

According to the Wikipedia entry, in mathematics, an *ordered pair* (a, b) is a pair of objects. The order in which the objects appear in the pair is significant: the ordered pair (a, b) is different from the ordered pair (b, a) unless $a = b$. In the ordered pair (a, b), the object a is called the *first* entry, and the object b the *second* entry of the pair. arara relies on this concept with the helper `Pair<A, B>` class, in which `A` and `B` denote the component classes, i.e, the types associated to the pair elements. In order to access the pair entries, the class provides two methods:

`◊first()` `△A`

This method, as the name implies, returns the first entry of the ordered pair, as an `A` object. It is important to observe that, from the MVEL context, as the method constitutes a property accessor (namely, a getter), the parentheses can be safely omitted.

`◊second()` `△B`

This method, as the name implies, returns the second entry of the ordered pair, as a `B` object. It is important to observe that, from the MVEL context, as the method constitutes a property accessor (namely, a getter), the parentheses can be safely omitted.

Keep in mind that the entries in the `Pair` class, once defined, cannot be modified to other values. The initial values are set during instantiation and, therefore, only entry getters are available to the user during the object life cycle.



Status for class loading and instantiation

The class loading and instantiation methods provided by arara typically return a pair composed of an integer value and a class or object reference. This integer value acts as a status of the underlying operation itself and might indicate potential issues. The possible values are:

`0`	Successful execution	`3`	Class was not found
`1`	File does not exist	`4`	Access policy violation
`2`	File URL is incorrect	`5`	Instantiation exception

Status for class loading and instantiation (ctd.)

Please make sure to *always* check the returned integer status when using class loading and instantiation methods in directive and rule contexts. This feature is quite powerful yet tricky and subtle!

C R `◊loadClass(File file, String name)` `△Pair<Integer, Class>`

This method loads a class based on the canonical name from the provided `File` reference and returns an ordered pair containing the status and the class reference itself. The file must contain the Java bytecode, either directly accessible from a `class` file or packaged inside a `jar` file. If an exception is raised, this method returns the `Object` class reference as second entry of the pair.

Example

```
1 result = loadClass(toFile('mymath.jar'),
2           'com.github.cereda.mymath.Arithmetic');
```

C R `◊loadClass(String reference, String name)` `△Pair<Integer, Class>`

This method loads a class based on the canonical name from the provided `String` reference and returns an ordered pair containing the status and the class reference itself. The file must contain the Java bytecode, either directly accessible from a `class` file or packaged inside a `jar` file. If an exception is raised, this method returns the `Object` class reference as second entry of the pair.

Example

```
1 result = loadClass('mymath.jar',
2           'com.github.cereda.mymath.Arithmetic');
```

C R `◊loadObject(File file, String name)` `△Pair<Integer, Object>`

This method loads a class based on the canonical name from the provided `File` reference and returns an ordered pair containing the status and a proper corresponding object instantiation. The file must contain the Java bytecode, either directly accessible from a `class` file or packaged inside a `jar` file. If an exception is raised, this method returns an `Object` object as second entry of the pair.

Example

```
1 result = loadObject(toFile('mymath.jar'),
2           'com.github.cereda.mymath.Trigonometric');
```

C R ◊loadObject(String reference, String name) △Pair<Integer, Object>

This method loads a class based on the canonical name from the provided `String` reference and returns an ordered pair containing the status and a proper corresponding object instantiation. The file must contain the Java bytecode, either directly accessible from a `class` file or packaged inside a `jar` file. If an exception is raised, this method returns an `Object` object as second entry of the pair.

Example

```
1 result = loadObject('mymath.jar',
2           'com.github.cereda.mymath.Trigonometric');
```

This section presented class loading and instantiation methods which may significantly enhance the expressiveness of rules and directives. However, make sure to use such feature with great care and attention.

6.7 Dialog boxes

A *dialog box* is a graphical control element, typically a small window, that communicates information to the user and prompts them for a response. This section introduces UI methods related to such interactions.

UI elements

The graphical elements are provided by the Swing toolkit from the Java runtime environment. Note that the default look and feel class reference can be modified through a key in the configuration file, as seen in Section 4.2, on page 43. It is important to observe that the methods presented in this section require a graphical interface. If arara is being executed in a headless environment (i.e, an environment with no graphical display available), an exception will be thrown when trying to use such UI methods in either directive or rule contexts.

Each dialog box provided by the UI methods of arara requires the specification of an associated icon. An *icon* is a pictogram displayed on a computer screen in order to help the user quickly identify the message by conveying its meaning through a visual resemblance to a physical object. Our tool features five icons, illustrated below, to be used with dialog boxes. Observe that each

icon is associated with a unique integer value which is provided later on to the actual method call. Also, it is worth mentioning that the visual appearance of such icons is based on the underlying Java virtual machine and the current look and feel, so your mileage might vary.



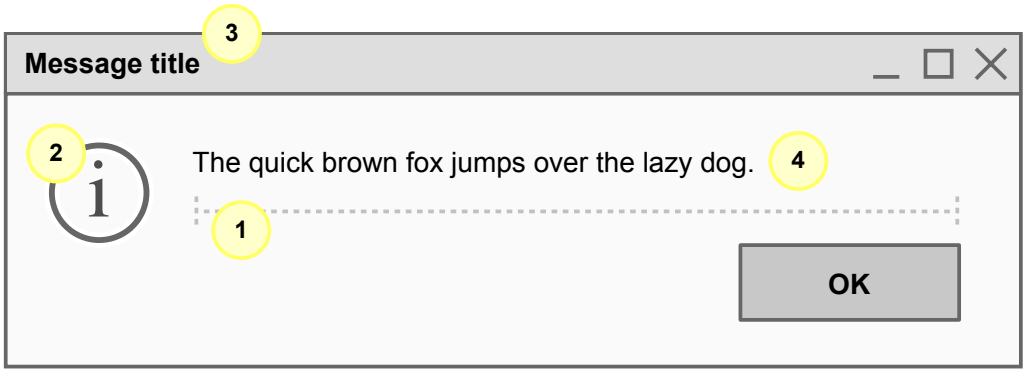
As good practice, make sure to provide descriptive messages to be placed in dialog boxes in order to ease and enhance the user experience. It is also highly advisable to always provide an associated icon, so avoid the plain option whenever possible.

Message text width

arara sets the default message text width to 250 pixels. Feel free to override this value according to your needs. Please refer to the appropriate method signatures for specifying a new width.

The UI method signatures are followed by a visual representation of the provided dialog box. For the sake of simplicity, each parameter index refers to the associated number in the figure.

R ◊showMessage(int width, int icon, String title, String text) △void

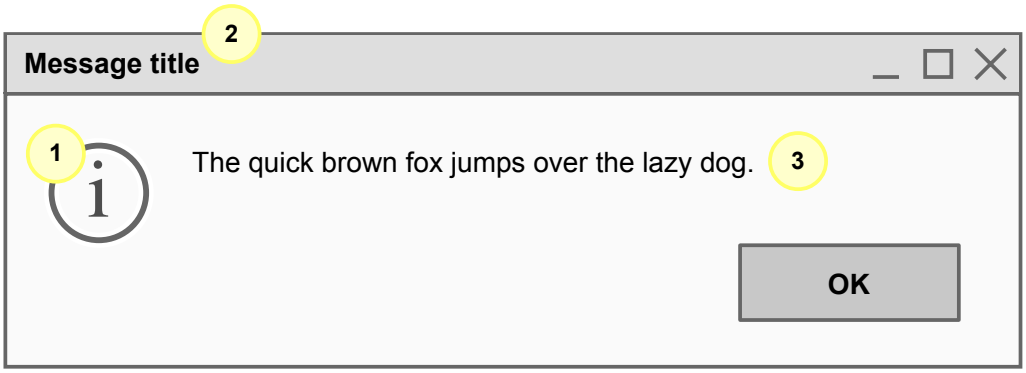


This method shows a message box according to the provided parameters. The dialog box is disposed when the user either presses the confirmation button or closes the window. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

Example

```
1 showMessage(250, 2, 'My title', 'My message');
```

R ◊showMessage(int icon, String title, String text) △void

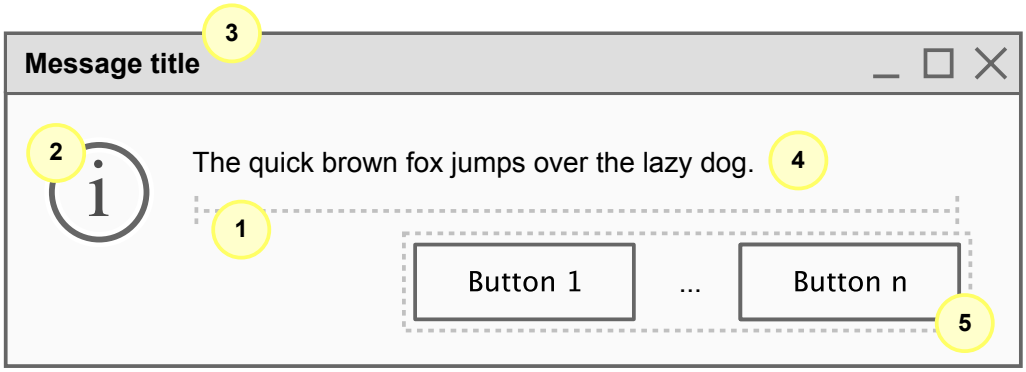


This method shows a message box according to the provided parameters. The dialog box is disposed when the user either presses the confirmation button or closes the window. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

Example

```
1 showMessage(2, 'My title', 'My message');
```

C R ◊showOptions(int width, int icon, String title, String text, Object... options) △int



This method shows a message box according to the provided parameters, including options represented as an array of `Object` objects. This array is portrayed in the dialog box as a list of buttons. The dialog box is disposed when the user either presses one of the buttons or closes the window. The method returns the natural index of the selected button, starting from `1`. If no button is pressed (e.g, the window is closed), `0` is returned. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

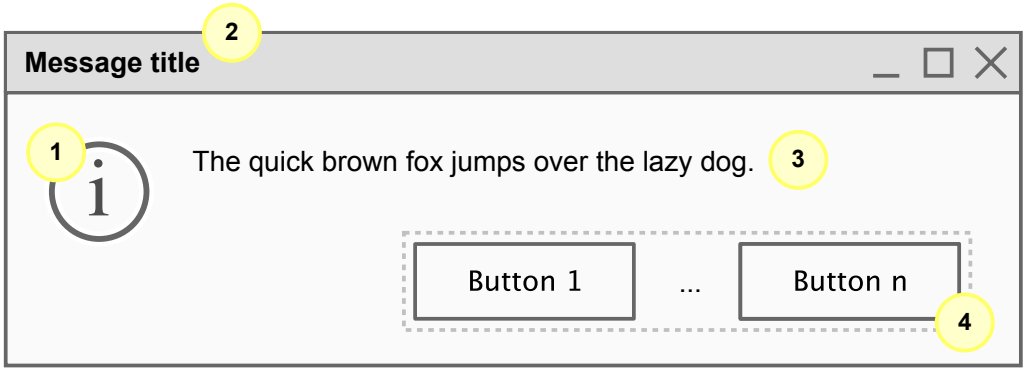
Example

```
% arara: pdflatex if showOptions(250, 4, 'Important!',
% arara: --> 'Do you like ice cream?', 'Yes!', 'No!') == 1
```

Button orientation

Keep in mind that your window manager might render the button orientation differently than the original arrangement specified in your array of objects. For instance, I had a window manager that rendered the buttons in the reverse order. However, note that the visual appearance should not interfere with the programming logic! The indices shall remain the same, pristine as ever, regardless of the actual rendering. Trust your code, not your eyes.

C R ◇ `showOptions(int icon, String title, String text, Object... options)` △ `int`

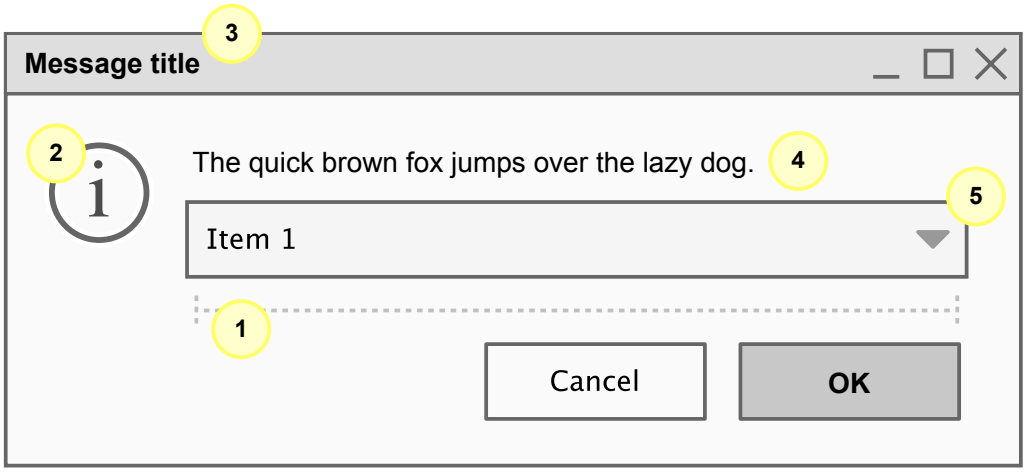


This method shows a message box according to the provided parameters, including options represented as an array of `Object` objects. This array is portrayed in the dialog box as a list of buttons. The dialog box is disposed when the user either presses one of the buttons or closes the window. The method returns the natural index of the selected button, starting from `1`. If no button is pressed (e.g, the window is closed), `0` is returned. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

Example

```
% arara: pdflatex if showOptions(4, 'Important!',
% arara: --> 'Do you like ice cream?', 'Yes!', 'No!') == 1
```

C R ◇ `showDropdown(int width, int icon, String title, String text, Object... options)` △ `int`



This method shows a dialog box according to the provided parameters, including options represented as an array of `Object` objects. This array is portrayed in the dialog box as a dropdown list. The first element from the array is automatically selected. The dialog box is disposed when the user either presses one of the buttons or closes the window. The method returns the natural index of the selected item, starting from `1`. If the user cancels the dialog or closes the window, `0` is returned. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

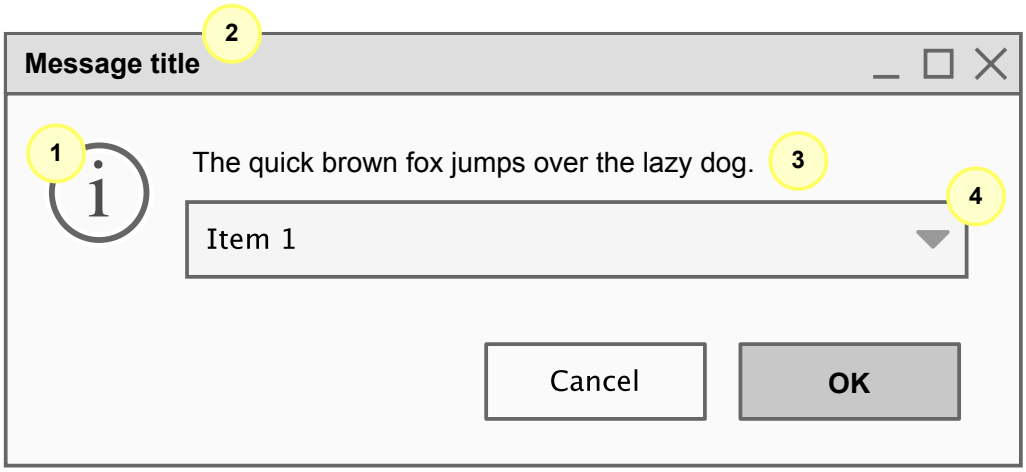
Example

```
% arara: pdflatex if showDropdown(250, 4, 'Important!',
% arara: --> 'Who deserves the tick?', 'David Carlisle',
% arara: --> 'Enrico Gregorio', 'Joseph Wright',
% arara: --> 'Heiko Oberdiek') == 2
```

Combo boxes and dropdown lists

According to the Wikipedia entry, a *combo box* is a combination of a dropdown list or list box and a single line editable textbox, allowing the user to either type a value directly or select a value from the list. The term is sometimes used to mean a dropdown list, but in Java, the term is definitely not a synonym! A dropdown list is sometimes clarified with terms such as non-editable combo box to distinguish it from the original definition of a combo box.

C R ◇ `showDropdown(int icon, String title, String text, Object... options)` △ `int`



This method shows a dialog box according to the provided parameters, including options represented as an array of `Object` objects. This array is portrayed in the dialog box as a dropdown list. The first element from the array is automatically selected. The dialog box is disposed when the user either presses one of the buttons or closes the window. The method returns the natural index of the selected item, starting from `1`. If the user cancels the dialog or closes the window, `0` is returned. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

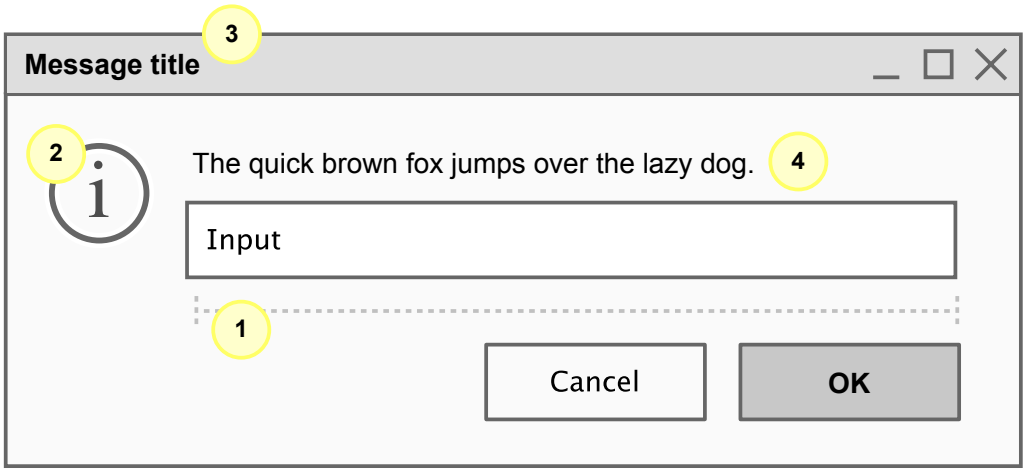
Example

```
% arara: pdflatex if showDropdown(4, 'Important!',
% arara: --> 'Who deserves the tick?', 'David Carlisle',
% arara: --> 'Enrico Gregorio', 'Joseph Wright',
% arara: --> 'Heiko Oberdiek') == 2
```

Swing toolkit

According to the Wikipedia entry, the Swing toolkit was developed to provide a more sophisticated set of GUI components than the earlier AWT widget system. Swing provides a look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components, such as scroll panes, trees, tables, and lists.

C R ◊showInput(int width, int icon, String title, String text) △String

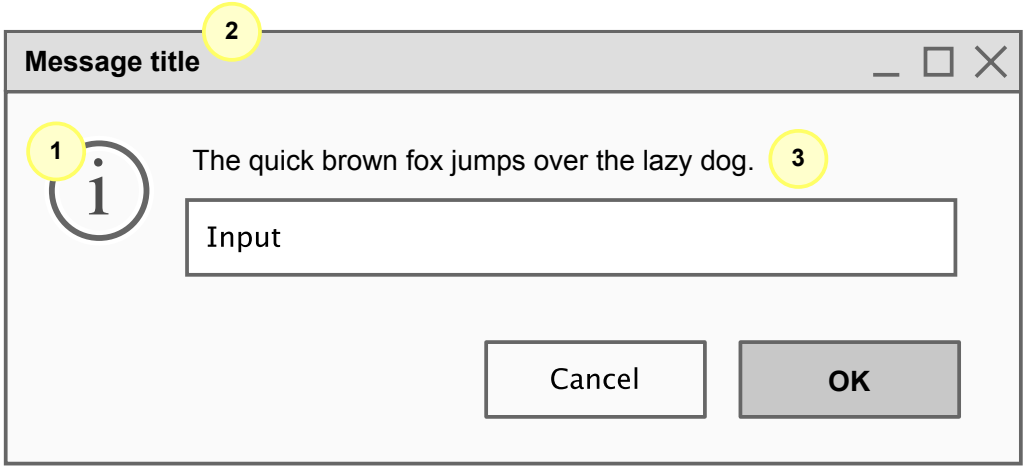


This method shows an input dialog box according to the provided parameters. The dialog box is disposed when the user either presses one of the buttons or closes the window. The method returns the content of the input text field, as a trimmed `String` object. If the user cancels the dialog or closes the window, an empty string is returned. It is important to observe that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

Example

```
1 % arara: pdflatex if showInput(250, 4, 'Important!',
2 % arara: --> 'Who wrote arara?') == 'Paulo'
```

C R ◊showInput(int icon, String title, String text) △String



This method shows an input dialog box according to the provided parameters. The dialog box is disposed when the user either presses one of the buttons or closes the window. The method returns the content of the input text field, as a trimmed `String` object. If the user cancels the dialog or closes the window, an empty string is returned. It is important to observe

that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

Example

```
% arara: pdflatex if showInput(4, 'Important!',
% arara: --> 'Who wrote arara?') == 'Paulo'
```

The UI methods presented in this section can be used for writing TEX tutorials and assisted compilation workflows based on user interactions, including visual input and feedback through dialog boxes.

6.8 Commands and triggers

From version 4.0 on, arara features the `Command` object, a new approach for handling system commands based on a high level structure with explicit argument parsing. Similarly, there is also the mystical `Trigger` object that constitutes a very special command that changes the inner workings of our tool at runtime. This section introduces methods for generating such objects.

The anatomy of a command

From the user perspective, a `Command` object is simply a good old list of `Object` objects, in which the list head (i.e, the first element) is the underlying system command, and the list tail (i.e, the remaining elements), if any, contains the associated command line arguments. For instance:

head *tail (associated command line arguments)*

`pdflatex` `--shell-escape` `--synctex=1` `thesis.tex`

From the previous example, it is important to observe that a potential file name quoting is not necessary. The underlying system command execution library handles the provided arguments accordingly.

Behind the scenes, however, arara employs a different workflow when constructing a `Command` object. The tool sets the working directory path for the current command to `USER_DIR` which is based on the current execution. The working directory path can be explicitly set through specific method calls, described later on in this section.

The list of objects is then completely flattened and all elements are mapped to their string representations through corresponding `◊toString` calls. Finally, the proper `Command` object is constructed. Keep in mind that, although a command takes a list (or even an array) of objects, which can be of any type, the internal representation is *always* a list of strings.

A list of objects might contain nested lists, i.e, a list within another list. As previously mentioned, arara employs *list flattening* when handling a list of objects during a `Command` object instantiation. As a means to illustrate this handy feature, consider the following list of integers:

A list with nested lists

```
[ 1, 2, [ 3, 4 ], 5, [ [ 6, 7 ], 8 ], 9, [ [ 10 ] ]
```

Note that the above list of integers contains nested lists. When applying list flattening, arara recursively adds the elements of nested lists to the original list and then removes the nested occurrences. Please refer to the source code for implementation details. The new flattened list is presented as follows.

A flattened list

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

List flattening and string mapping confer expressiveness and flexibility to the `Command` object construction, as users can virtually use any data type to describe the underlying rule logic and yet obtain a consistent representation.

The anatomy of a trigger

A `Trigger` object constitutes a special command that changes the internal workings of arara at runtime. It is a highly experimental feature. A trigger is basically a function call defined in terms of a `String` reference representing the actual function name followed by an optional list of `Object` arguments. For instance, consider this hypothetical trigger that multiplies an arbitrary number of integer terms:

name	*list of arguments*		
`multiply`	`12`	`34`	`65`

This description is, in some aspects, very much like a typical `Command` construction. However, a trigger is less forgiving on data types and does not apply transformations on the provided list of arguments. Therefore, the argument types *must match* the trigger signature.

So far, the tool provides only one trigger, seen in action in one of the official rules, under the name `halt` and with no parameters. This particular trigger halts the current interpretation workflow, such that subsequent directives are ignored. We have not worked much on triggers yet, and the concept is mentioned here for documentation purposes only.

R ◊getCommand(List<String> elements) △Command

This method, as the name implies, returns a `Command` object according to the provided list of `String` elements. If the list is empty, the tool will ignore the execution.

Example

```
1 return getCommand([ 'ls', '-l' ]);
```

R ◊getCommand(Object... elements) △Command

This method, as the name implies, returns a `Command` object according to the provided array of `Object` elements. If the array is empty, the tool will ignore the execution. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters.

Example

```
1 return getCommand('pdflatex', '--shell-escape', 'thesis.tex');
```

R ◊getTrigger(String name) △Trigger

This method, as the name implies, returns a `Trigger` object according to the provided `String` reference as function name. It is important to observe that this particular trigger instance does not have parameters.

Example

```
1 return getTrigger('halt');
```

R ◊getTrigger(String name, Object... parameters) △Trigger

This method, as the name implies, returns a proper `Trigger` object according to the provided `String` reference as function name and the `Object` array as associated parameters. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters.

Example

```
1 return getTrigger('multiply', 12, 34, 65);
```

R ◇ `getCommandWithWorkingDirectory(File directory, List<String> elements)` △ Command

This method, as the name implies, sets the working directory based on the provided `File` reference and returns a proper `Command` object according to the provided list of `String` elements. If the list is empty, the tool will ignore the execution.

Example

```
1 return getCommandWithWorkingDirectory(toFile('/home/paulo'),
2         [ 'ls', '-l' ]);
```

R ◇ `getCommandWithWorkingDirectory(String path, List<String> elements)` △ Command

This method, as the name implies, sets the working directory based on the provided `String` reference and returns a proper `Command` object according to the provided list of `String` elements. If the list is empty, the tool will ignore the execution.

Example

```
1 return getCommandWithWorkingDirectory('/home/paulo',
2         [ 'ls', '-l' ]);
```

R ◇ `getCommandWithWorkingDirectory(File directory, Object... elements)` △ Command

This method, as the name implies, sets the working directory based on the provided `File` reference and returns a proper `Command` object according to the provided array of `Object` elements. If the array is empty, the tool will ignore the execution. Note that the object array is denoted by a comma-separated sequence of elements in the actual method call, resulting in a variable number of parameters.

Example

```
1 return getCommandWithWorkingDirectory(toFile('/home/paulo'),
2         'pdflatex', '--shell-escape', 'thesis.tex');
```

R ◇ `getCommandWithWorkingDirectory(String path, Object... elements)` △ Command

This method, as the name implies, sets the working directory based on the provided `String` reference and returns a proper `Command` object according to the provided array of `Object` elements. If the array is empty, the tool will ignore the execution. Note that the object array is denoted by a comma-

separated sequence of elements in the actual method call, resulting in a variable number of parameters.

Example

```
1 return getCommandWithWorkingDirectory('/home/paulo',
2         'pdflatex', '--shell-escape', 'thesis.tex');
```

The methods presented in this section constitute the foundations of underlying system command execution. In particular, whenever possible, it is highly advisable to use `Command` objects through proper `◊getCommand` method calls, as the plain string approach used in previous versions of our tool is marked as deprecated and will be removed in future versions.

6.9 Others

This section introduces assorted methods provided by arara as a means to improve the automation itself with expressive rules and enhance the user experience. Such methods are properly described as follows.

Session

Rules are designed under the *encapsulation* notion, such that the direct access to internal workings of such structures is restricted. However, as a means to support framework awareness, arara provides a mechanism for data sharing across rule contexts, implemented as a `Session` object. In practical terms, this particular object is a global, persistent map composed of `String` keys and `Object` values available throughout the entire execution. The public methods are described as follows:

`◊insert(String key, Object value)` `△void`

This method, as the name implies, inserts an object into the session, indexed by the provided key. Observe that, if the session previously contained a mapping for the provided key, the old value is replaced by the specified value.

`◊remove(String key)` `△void`

This method, as the name implies, removes the mapping for the provided key from the session. Be mindful that an attempt to remove a mapping for a nonexistent key will raise an exception.

`◊exists(String key)` `△boolean`

This method, as the name implies, returns a boolean value according to whether the session contains a mapping for the provided key. It is highly advisable to use this method before attempting to remove a mapping from the session.

Session (ctd.)

◊obtain(String key) △Object

This method, as the name implies, returns the object value to which the specified key is mapped. Be mindful that an attempt to return a value for a nonexistent key will raise an exception.

◊forget() △void

This method, as the name implies, removes all of the existing mappings from the session. The session object will be effectively empty after this call returns.

It is important to observe that the `Session` object provided by our tool follows the *singleton* pattern, i.e, a software design pattern that restricts the instantiation of a class to one object. Therefore, the same session is consistently shared across rule contexts.

R ◊getSession() △Session

This method, as the name implies, returns the `Session` object for data sharing across rule contexts. Keep in mind that a session cannot contain duplicate keys. Each key can map to at most one value.

Example

```
1 name = getSession().obtain('name');
```

R ◊throwError(String message) △void

This method deliberately throws an error to be intercepted later on during execution. Consider using such method for an explicit notification about unexpected or unsought scenarios, e.g, wrong parameter types or values. The raised error has an associated message which is displayed in the terminal and added to the log file.

Example

```
1 options = 'not a list';
2 if (!isList(options)) {
3     throwError('I was expecting a list.');
4 }
```

R ◊isVerboseMode() △boolean

This method, as the name implies, returns a boolean value according to whether arara is being executed in verbose mode, enabled through either the `--verbose` command line option or the corresponding key in the con-

figuration file (detailed in Sections 3.2 and 4.2, respectively). Note that the logical negation of such method indicates whether the tool is being executed in silent mode.

Example

```
1 verbose = isVerboseMode();
```

R ◊isOnPath(String name) △boolean

This method, as the name implies, returns a boolean value according to whether the provided `String` reference representing a command name is reachable from the system path. For portability reasons, there is no need to provide extensions to Microsoft Windows command names, as arara will look for common patterns. This behaviour is expected and by design. However, be mindful that the search is case sensitive.

Example

```
1 result = isOnPath('pdftex');
```

Path inspection

According to the Wikipedia entry, `PATH` is an environment variable on Unix-like operating systems and Microsoft Windows, specifying a set of directories where executable programs are located. arara performs a file search operation based on all directories specified in the system path, filtering files by name (and extensions, when in Microsoft Windows). When an exact match is found, the search is concluded. Notwithstanding the great effort, it is very important to note that there is no guarantee that our tool will be able to correctly reach the command in all scenarios.

R ◊unsafelyExecuteSystemCommand(Command command) △Pair<Integer, String>

This method, which has a very spooky name, unsafely executes the provided `Command` reference and returns an ordered pair containing the exit status and the command output. Note that, if an exception is raised during the command execution, `-99` is assigned as exit status and an empty string is defined as command output. Please make sure to always check the returned integer status when using this method.

Example

```
1  result = unsafelyExecuteSystemCommand(getCommand('ls'));
```

Hic sunt leones

Please *do not abuse* this method! Keep in mind that this particular feature is included for very specific scenarios in which the command streams are needed ahead of time for proper decision making.

R ◊ `isSubdirectory(File directory)` △ `boolean`

This method checks whether the provided `File` reference is a valid subdirectory under the project hierarchy, return a corresponding boolean value. This is a check to impose a possible restriction in the rule scope, so that users can change down to subdirectories in their projects but not up, outside of the root directory.

Example

```
1  valid = isSubdirectory(toFile('chapters/'));
```

R ◊ `mergeVelocityTemplate(File input, File output, Map<String, Object> map)` △ `void`

This method, as the name implies, merges the provided `File` template reference written in the Velocity Template Language 1.7 specification with the `Map` data object in order to produce a corresponding `File` output. It is important to observe that this method will raise an exception if the provided input file does not exist or if there is an error with the underlying template language.

Source file `input.txt`

```
1  Hello, my name is ${name} and
2  I am from ${country}!
```

Example

```
1  mergeVelocityTemplate(toFile('input.txt'), toFile('output.txt'),
2       [ 'name' : 'Paulo', 'country' : 'Brazil' ])
```



Velocity Template Language 1.7

As of 2017, the Apache Foundation has released the new 2.0 version for the Velocity Template Language. However, this particular version introduces behavioural and syntactic changes that may cause problems with older versions.

In order to maintain compatibility with older Java virtual machines, arara works with the VTL 1.7 specification, so it is highly recommended to strictly adhere to this reference when writing templates for the corresponding method or the official `velocity` rule.

The methods presented in this section provide interesting features for persistent data sharing, error handling, early command execution, and templating. It is important to note that more classes, objects and methods can be incorporated into arara through class loading and object instantiation, extending the features and enhancing the overall user experience.



The official rule pack

arara ships with a pack of default rules, placed inside a special subdirectory named `rules/` inside another special directory named `ARARA_HOME` (the place where our tool is installed). This chapter introduces the official rules, including proper listings and descriptions of associated parameters whenever applied. Note that such rules work off the shelf, without any special installation, configuration or modification. An option marked by `S` after the corresponding identifier indicates a natural boolean switch. Similarly, the occurrence of an `R` mark indicates that the corresponding option is required.



Can my rule be distributed within the official pack?

As seen in Section 4.2, on page 43, the default rule path can be extended to include a list of directories in which our tool should search for rules. However, if you believe your rule is comprehensive enough and deserves to be in the official pack, please contact us! We will be more than happy to discuss the inclusion of your rule in forthcoming updates.

`animate` *Chris Hughes, Paulo Cereda*

This rule creates an animated `gif` file from the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `pdf` suffix, using the `convert` command line utility from the ImageMagick suite.

`delay` *default:* `10`

This option regulates the number of ticks before the display of the next image sequence, acting as a pause between still frames.

`loop` *default:* `0`

This option regulates the number of repetitions for the animation. When set to zero, the animation repeats itself an infinite number of times.

`density` *default:* `300`

This option specifies the horizontal and vertical canvas resolution while rendering vector formats into a proper raster image.

`program` *default:* `convert`

This option specifies the command utility path as a means to avoid potential clashes with underlying operating system commands.

Microsoft Windows woes

According to the ImageMagick website, the Windows installation routine adds the program directory to the system path, such that one can call command line tools directly from the command prompt, without providing a path name. However, `convert` is also the name of Windows system tool, located in the system directory, which converts file systems from one format to another.

The best solution to avoid possible future name conflicts, according to the ImageMagick team, is to call such command line tools by their full path in any script. Therefore, the `convert` rule provides the `program` option for this specific scenario.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: animate: { delay: 15, density: 150 }
```

`bib2gls` *Nicola Talbot, Paulo Cereda*

This rule executes the `bib2gls` command line application which extracts glossary information stored in a `bib` file and converts it into glossary entry definitions in resource files. This rule passes the base name of the `◊currentFile` reference (i.e, the name without the associated extension) as the mandatory argument.

`dir`
: This option sets the directory used for writing auxiliary files. Note that this option does not change the current working directory.

`trans`
: This option sets the extension of the transcript file created by `bib2gls`. The value should be just the file extension without the leading dot. The default is `glg`.

`locale`
: This option specifies the preferred language resource file. Please keep in mind that the provided value must be a valid IETF language tag. If omitted, the default is obtained by `bib2gls` from the JVM.

`group` `S`
: This option sets whether `bib2gls` will try to determine the letter group for each entry and add it to a new field called `group` when sorting. Be mindful that some `sort` options ignore this setting. The default value is off.

`interpret` S

This option sets whether the interpreter mode of `bib2gls` is enabled. If the interpreter is on, `bib2gls` will attempt to convert any LaTeX markup in the sort value to the closest matching Unicode characters. If the interpreter is off, the `log` file will not be parsed for recognised packages. The default value is on.

`breakspace` S

This option sets whether the interpreter will treat a tilde character as a non-breaking space (as with TeX) or a normal space. The default behaviour treats it as non-breakable.

`trimfields` S

This option sets whether `bib2gls` will trim leading and trailing spaces from field values. The default behaviour does not trim spaces.

`recordcount` S

This option sets whether the record counting will be enabled. If activated, `bib2gls` will add record count fields to entries. The default behaviour is off.

`recordcountunit` S

This option sets whether `bib2gls` will add unit record count fields to entries. These fields can then be used with special commands. The default behaviour is off.

`cite` S

This option sets whether `bib2gls` will treat citation instances found in the `aux` file as though it was actually an ignored record. The default behaviour is off.

`verbose` S

This option sets whether `bib2gls` will be executed in verbose mode. When enabled, the application will write extra information to the terminal and transcript file. This option is unrelated to arara's verbose mode. The default behaviour is off.

`merge` S

This option sets whether the program will merge `wrglossary` counter records. If disabled, one may end up with duplicate page numbers in the list of entry locations, but linking to different parts of the page. The default is on.

`uniscript` S

This option sets whether text superscript and subscript will use the corresponding Unicode characters if available. The default is on.

`packages`

This option instructs the interpreter to assume the packages from the provided list have been used by the document.

`ignore`

This option instructs `bib2gls` to skip the check for any package from the provided list when parsing the corresponding log file.

`custom`
: This option instructs the interpreter to parse the package files from the provided list. The package files need to be quite simple.

`mapformats`
: This option takes a list and sets up the rule of precedence for partial location matches. Each element from the provided list must be another list of exactly two entries representing a conflict resolution.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: bib2gls: { group: true }
% arara: --> if found('aux', 'glsxtr@resource')
```

`biber` *Marco Daniel, Paulo Cereda*

This rule runs `biber`, the backend bibliography processor for `biblatex`, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: biber: { options: [ '--wraplines' ] }
```

`bibtex` *Marco Daniel, Paulo Cereda*

This rule runs the `bibtex` program, a reference management software, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: bibtex: { options: [ '-terse' ] }
% arara: --> if exists(toFile('references.bib'))
```

`bibtex8` *Marco Daniel, Paulo Cereda*

This rule runs `bibtex8`, an enhanced, portable C version of `bibtex`, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string. It is important to note that this tool can read a character set file containing encoding details.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: bibtex8: { options: [ '--trace', '--huge' ] }
```

`bibtexu` *Marco Daniel, Paulo Cereda*

This rule runs the `bibtexu` program, an enhanced version of `bibtex` with Unicode support and language features, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: bibtexu: { options: [ '--language', 'fr' ] }
```

`clean` *Marco Daniel, Paulo Cereda*

This rule removes the provided file reference through the underlying system command, which can be `rm` in a Unix environment or `del` in Microsoft Windows. As a security lock, this rule will always throw an error if `◊currentFile` is equal to `◊getOriginalFile`, so the main file reference cannot be removed. It is highly recommended to use the special `files` parameter to indicate removal candidates. Alternatively, a list of file extensions can be provided as well. Be mindful that the security lock also applies to file removals based on extensions.

`extensions`

This option, as the name indicates, takes a list of extensions and constructs a new list of removals commands according to the base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with each extension from the original list as suffixes. Keep in mind that, if the special `files` parameter is used with this option, the resulting list will contain the cartesian product of file base names and extensions. An error is thrown if any data structure other than a proper list is provided as the value.



Better safe than sorry!

When in doubt, always remember that the `--dry-run` command line option is your friend! As seen in Section 3.2, on page 32, this option makes arara go through all the motions of running tasks and subtasks, but with no actual calls. It is a very useful feature for testing the sequence of removal commands without actually losing your files! Also, as good practice, always write plain, simple, understandable `clean` directives and use as many as needed in your TEX documents.



Example

```
% arara: clean: { extensions: [ aux, log ] }
```

`csplain` *Paulo Cereda*

This rule runs the `csplain` TEX engine, a conservative extension of Knuth's plain TEX with direct processing characters and hyphenation patterns for Czech and Slovak, on the provided `◊currentFile` reference.

`interaction`

This option alters the underlying engine behaviour. When such option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

`batchmode`

In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

`nonstopmode`

In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` S
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
1 % arara: csplain: { interaction: batchmode, shell: yes }
```

`datatooltk` *Nicola Talbot, Paulo Cereda*

This rule runs `datatooltk`, an application that creates `datatool` databases in raw format from several structured data formats, in batch mode. This rule requires `output` and one of the import options.

`output` R
: This option provides the database name to be saved as output. To guard against accidentally overwriting a document file, `datatooltk` now forbids the `tex` extension for output files. This option is required.

`csv`
: This option, as the name indicates, imports data from the `csv` file reference provided as a plain string value.

`sep`
: This option specifies the character used to separate values in the `csv` file. Defaults to a comma.

`delim`
: This option specifies the character used to delimit values in the `csv` file. Defaults to a double quote.

`name`
: This option, as the name indicates, sets the label reference of the newly created database according to the provided value.

`sql`
: This option imports data from an SQL database where the provided value refers to a proper `select` SQL statement.

`sqldb`
: This option, as the name indicates, sets the name of the SQL database according to the provided value.

`sqluser`
: This option, as the name indicates, sets the name of the SQL user according to the provided value.

`noconsole` *default:* `gui`
: This action dictates the password request action if such information was not provided earlier. If there is no console available, the action is determined by the following values:

  `error`
  : As the name indicates, this action issues an error when no password was previously provided through the proper option.

  `stdin`
  : This action requests the password via the standard input stream, which is less secure than using a console.

  `gui`
  : This action displays a dialog box in which the user can enter the password for the SQL database.

`probsoln`
: This option, as the name indicates, imports data in the `probsoln` format from the file name provided as the value.

`input`
: This option, as the name indicates, imports data in the `datatool` format from the file name provided as the value.

`sort`
: This option, as the name indicates, sorts the database according to the column whose label is provided as the value. The value may be preceded by `+` or `-` to indicate ascending or descending order, respectively. If the sign is omitted, ascending is assumed.

`sortlocale`
: This option, as the name indicates, sorts the database according to alphabetical order rules of the locale provided as the value. If the value is set to `none` strings are sorted according to non-locale letter order.

`sortcase` `S`
: This option sets whether strings will be sorted using case-sensitive comparison for non-locale letter ordering. The default behaviour is case-insensitive.

`seed`
: This option, as the name indicates, sets the random generator seed to the provided value. The seed is cleared if an empty value is provided.

`shuffle` S
: This option sets whether the database will be properly shuffled. Shuffle is always performed after sort, regardless of the option order.

`csvheader` S
: This option sets whether the `csv` file has a header row. The spreadsheet import functions also use this setting.

`debug` S
: This option, as the name indicates, sets whether the debug mode of `datatooltk` is activated. The debug mode is disabled by default.

`owneronly` S
: This option sets whether read and write permissions when saving `dbtex` files should be defined for the owner only. This option has no effect on some operating systems.

`maptex` S
: This option sets whether TEX special characters will be properly mapped when importing data from `csv` files or SQL databases.

`xls`
: This option, as the name indicates, imports data from a Microsoft Excel `xls` file reference provided as a plain string value.

`ods`
: This option, as the name indicates, imports data from an Open Document Spreadsheet `ods` file reference provided as a plain string value.

`sheet`
: This option specifies the sheet to select from the Excel workbook or Open Document Spreadsheet. This may either be an index or the name of the sheet.

`filterop`
: This option specifies the logical operator to be associated with a given filter. Filtering is always performed after sorting and shuffling. Possible values are:

  `or`
  : This value, as the name indicates, uses the logical `or` operator when filtering. This is the default behaviour. Note that this value has no effect if only one filter is supplied.

  `and`
  : This value, as the name indicates, uses the logical `and` operator when filtering. Note that this value has no effect if only one filter is supplied.

`filters`
: This option takes a list and sets up a sequence of filters. Each element from the provided list must be another list of exactly three entries representing a key, an operator and a value, respectively.

`truncate`

This option truncates the database to the number of rows provided as the value. Truncation is always performed after any sorting, shuffling and filtering, but before column removal.



Example

```
% arara: datatooltk: {
% arara: --> output: books.dbtex,
% arara: --> csv: booklist.csv }
```

`dvipdfm` *Marco Daniel, Paulo Cereda*

This rule runs `dvipdfm`, a command line utility for file format translation, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `dvi` suffix, generating a Portable Document Format `pdf` file.

`output`

This option, as the name indicates, sets the output name for the generated `pdf` file. There is no need to provide an extension, as the value is always normalized with `◊getBasename` such that only the name without the associated extension is used. The base name of the current file reference is used as the default value.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: dvipdfm: { output: thesis }
```

`dvipdfmx` *Marco Daniel, Paulo Cereda*

This rule runs `dvipdfmx`, an extended version of `dvipdfm` created to support multibyte character encodings and large character sets for East Asian languages, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `dvi` suffix, generating a Portable Document Format `pdf` file.

`output`

This option, as the name indicates, sets the output name for the generated `pdf` file. There is no need to provide an extension, as the value is always normalized with `◊getBasename` such that only the name without the associated extension is used. The base name of the current file reference is used as the default value.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: dvipdfmx: { options: [ '-K', '40' ] }
```

`dvips` *Marco Daniel, Paulo Cereda*

This rule runs `dvips` on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `dvi` suffix, generating a PostScript `ps` file.

`output`

This option, as the name indicates, sets the output name for the generated `ps` file. There is no need to provide an extension, as the value is always normalized with `◊getBasename` such that only the name without the associated extension is used. The base name of the current file reference is used as the default value.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: dvips: { output: thesis }
```

`dvipspdf` *Marco Daniel, Paulo Cereda*

This rule runs `dvips` in order to obtain a corresponding `ps` file from the initial `dvi` reference, and then runs `ps2pdf` on the previously generated `ps` file in order to obtain a `pdf` file. Note that all base names are acquired from the `◊currentFile` reference (i.e, the name without the associated extension) and used to construct the resulting files.

`output`

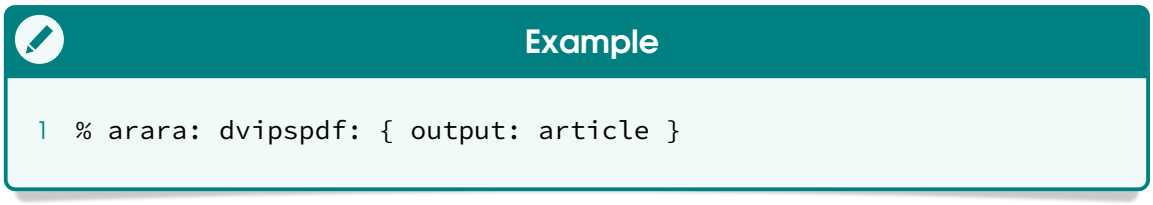
This option, as the name indicates, sets the output name for the generated `pdf` file. There is no need to provide an extension, as the value is always normalized with `◊getBasename` such that only the name without the associated extension is used. The base name of the current file reference is used as the default value.

`options1`

This option, as the name indicates, takes a list of raw command line

options and appends it to the `dvips` program call. An error is thrown if any data structure other than a proper list is provided as the value.

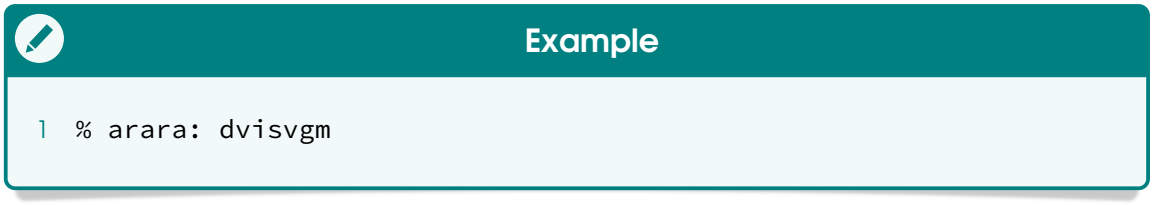
`options2`
: This option, as the name indicates, takes a list of raw command line options and appends it to the `ps2pdf` program call. An error is thrown if any data structure other than a proper list is provided as the value.



`dvisvgm` *samcarter*

This rule runs `dvisvgm` in order to obtain a corresponding `svg` file, a vector graphics format based on XML, from the initial `dvi` reference. It is important to observe that the base name is acquired from the `◊currentFile` reference (i.e, the name without the associated extension) and used to construct the resulting file.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



`etex` *Marco Daniel, Paulo Cereda, Nicola Talbot*

This rule runs the `etex` extended (plain) TEX engine on the provided `◊currentFile` reference, generating a corresponding file in a device independent format.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

  `batchmode`
  : In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

  `nonstopmode`
  : In this mode, the diagnostic message will appear on the terminal,

but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: etex: { shell: yes }
```

`frontespizio` *Francesco Endrici, Enrico Gregorio, Paulo Cereda*

This rule automates the steps required by the `frontespizio` package in order to help Italian users generate the frontispiece to their thesis. First and foremost, the frontispiece is generated. If `latex` is used as the underlying engine, there is an additional intermediate conversion step to a proper `eps` file. Finally, the final document is compiled.

`engine` *default:* `pdflatex`
: This option, as the name indicates, sets the underlying TEX engine to be used for both compilations (the frontispiece and the document itself). Possible values are:

  `latex`
  : This value, as the name indicates, sets the underlying TEX engine to `latex` for both compilations (frontispiece and document).

  `pdflatex`
  : This value, as the name indicates, sets the underlying TEX engine to `pdflatex` for both compilations (frontispiece and document).

  `xelatex`
  : This value, as the name indicates, sets the underlying TEX engine to `xelatex` for both compilations (frontispiece and document).

  `lualatex`
  : This value, as the name indicates, sets the underlying TEX engine to `lualatex` for both compilations (frontispiece and document).

`shell` S
: This option sets whether the possibility of running underlying system commands from within the selected TEX engine is activated.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

  `batchmode`
  : In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

  `nonstopmode`
  : In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

  `scrollmode`
  : In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

  `errorstopmode`
  : In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual TEX engine call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
1 % arara: frontespizio: { engine: xelatex,
2 % arara: --> shell: yes, interaction: nonstopmode }
```

`halt` *Heiko Oberdiek, Paulo Cereda*

This rule, as the name suggests, calls the `halt` trigger, which stops the current interpretation workflow, such that subsequent directives are ignored. This rule contains no associated options. Please refer to Section 6.8, on page 91, for more information on triggers.



Example

```
1 % arara: halt
```

`indent` *Chris Hughes, Paulo Cereda*

This rule runs `latexindent`, a Perl script that indents TEX files according to an indentation scheme, on the provided `◊currentFile` reference. Environments, including those with alignment delimiters, and commands, including those that can split braces and brackets across lines, are usually handled correctly by the script.

`silent` S

This option, as the name indicates, sets whether the script will operate in silent mode, in which no output is given to the terminal.

`overwrite` S

This option, as the name indicates, sets whether the `◊currentFile` reference will be overwritten. If activated, a copy will be made before the actual indentation process.

`trace`

This option, as the name indicates, enables the script tracing mode, such that a verbose output will be given to the `indent.log` log file. Possible values are:

`default`

This value, as the name indicates, refers to the default tracing level. Note that, especially for large files, this value does affect performance of the script.

`complete`

This value, as the name indicates, refers to the detailed, complete tracing level. Note that, especially for large files, performance of the script will be significantly affected when this value is used.

`screenlog` S

This option, as the name indicates, sets whether `latexindent` will output the log file to the screen, as well as to the specified log file.

`modifylinebreaks` S

This option, as the name indicates, sets whether the script will modify line breaks, according to specifications written in a configuration file.

`cruft`

This option sets the provided value as a cruft location in which the script will write backup and log files. The default behaviour sets the working directory as cruft location.

`logfile`

This option, as the name indicates, sets the name of the log file generated by `latexindent` according to the provided value.

`output`

This option, as the name indicates, sets the name of the output file. Please note that this option has higher priority over some switches, so options like `overwrite` will be ignored by the underlying script.

`options`

This option, as the name indicates, takes a list of raw command line

options and appends it to the actual script call. An error is thrown if any data structure other than a proper list is provided as the value.

`settings`
: This option, as the name indicates, dictates the indentation settings to be applied in the current script execution. Two possible values are available:

  `local`
  : This value, as the name implies, acts a switch to indicate a local configuration. In this scenario, the script will look for a proper settings file in the same directory as the `◊currentFile` reference and add the corresponding content to the indentation scheme. Optionally, a file location can be specified as well. Please refer to the `where` option for more details on such feature.

  `onlydefault`
  : This value, as the name indicates, ignores any local configuration, so the script will resort to the default indentation behaviour.

`where`
: This option, as the name indicates, sets the file location containing the indentation settings according to the provided value. This option can only be used if, and only if, `local` is set as the value for the `settings` option, otherwise the rule will throw an error.

`replacement`
: This option, as the name indicates, implements the replacement mode switches. Three possible values are available:

  `full`
  : This value, as the name indicates, performs indentation and replacements, not respecting verbatim code blocks.

  `noverb`
  : This value, as the name indicates, performs indentation and replacements, and will respect verbatim code blocks.

  `noindent`
  : This value, as the name implies, will not perform indentation, and will perform replacements not respecting verbatim code blocks.



Example

```
1  % arara: indent: { overwrite: yes }
```

`latex` *Marco Daniel, Paulo Cereda*

This rule runs the `latex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in a device independent format.

`interaction`
: This option alters the underlying engine behaviour. If this option is

omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

`batchmode`
: In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

`nonstopmode`
: In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

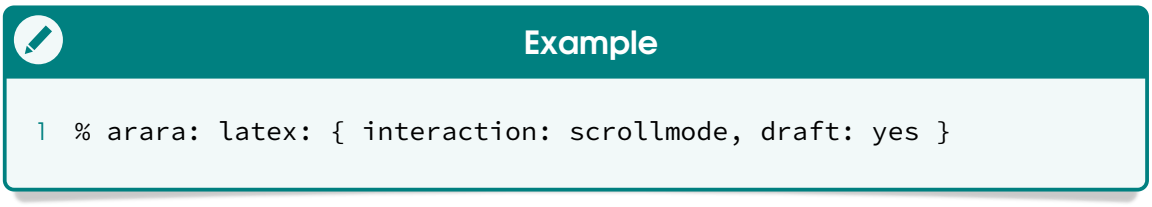
`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` `S`
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` `S`
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` `S`
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



`latexmk` *Marco Daniel, Brent Longborough, Paulo Cereda*

This rule runs `latexmk`, a fantastic command line tool for fully automated TEX document generation, on the provided `◊currentFile` reference.

`clean`
: This option, as the name indicates, removes all temporary files generated after a sequence of intermediate calls for document generation. Two possible values are available:

`all`
: This value, as the name indicates, removes all temporary, intermediate files, as well as resulting, final formats such as PostScript and Portable Document File. Only relevant source files are kept.

`partial`
: This value, as the name indicates, removes all temporary, intermediate files and keeps the resulting, final formats such as PostScript and Portable Document File.

`engine`
: This option, as the name indicates, sets the underlying TEX engine of `latexmk` to be used for the compilation sequence. Possible values are:

  `latex`
  : This value, as the name indicates, sets the underlying TEX engine of the script to `latex` for the compilation sequence.

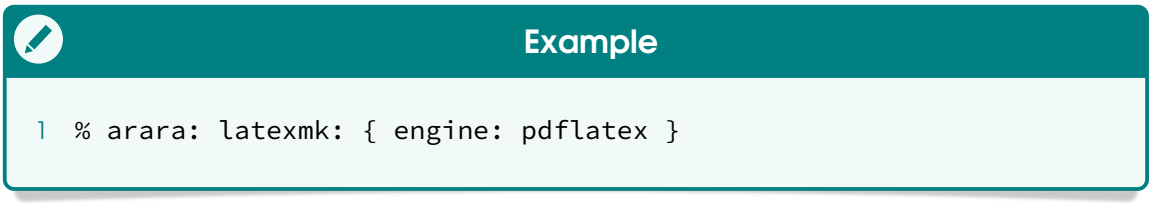
  `pdflatex`
  : This value, as the name indicates, sets the underlying TEX engine of the script to `pdflatex` for the compilation sequence.

  `xelatex`
  : This value, as the name indicates, sets the underlying TEX engine of the script to `xelatex` for the compilation sequence.

  `lualatex`
  : This value, as the name indicates, sets the underlying TEX engine of the script to `lualatex` for the compilation sequence.

`program`
: This option, as the name suggests, sets the TEX engine according to the provided value. It is important to note that this option has higher priority over `engine` values, so the latter will be discarded.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual script call. An error is thrown if any data structure other than a proper list is provided as the value.



`lualatex` *Marco Daniel, Paulo Cereda*

This rule runs the new `lualatex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in the Portable Document File format, as expected.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

`batchmode`
: In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

`nonstopmode`
: In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` S
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: lualatex: { interaction: errorstopmode,
% arara: --> synctex: yes }
```

`luatex` *Marco Daniel, Paulo Cereda*

This rule runs the `luatex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in the Portable Document File format, as expected.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

`batchmode`
: In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

`nonstopmode`
: In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

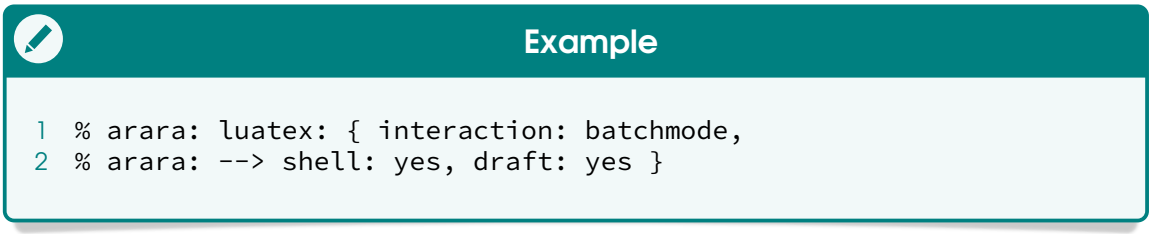
`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` S
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



`make` *Marco Daniel, Paulo Cereda*

This rule runs `make`, a build automation tool that automatically builds executable programs and libraries from source code, according to a special file which specifies how to derive the target program.

`targets`
: This option takes a list of targets. Note that `make` updates a target if it depends on files that have been modified since the target was last modified, or if the target does not exist.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: make: { targets: [ compile, package ] }
```

`makeglossaries` *Marco Daniel, Nicola Talbot, Paulo Cereda*

This rule runs `makeglossaries`, an efficient Perl script designed for use with TEX documents that work with the `glossaries` package. All the information required to be passed to the relevant indexing application should also be contained in the auxiliary file. The script takes the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as the mandatory argument.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual script call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: makeglossaries if found('aux', '@istfilename')
```

`makeglossarieslite` *Marco Daniel, Nicola Talbot, Paulo Cereda*

This rule runs `makeglossaries-lite`, a lightweight Lua script designed for use with TEX documents that work with the `glossaries` package. All the information required to be passed to the relevant indexing application should also be contained in the auxiliary file. The script takes the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as the mandatory argument.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual script call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: makeglossarieslite if found('aux', '@istfilename')
```

`makeindex` *Marco Daniel, Paulo Cereda*

This rule runs `makeindex`, a general purpose hierarchical index generator, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `idx` suffix, generating an index as a special `ind` file.

`style`

This option, as the name indicates, sets the underlying index style file. Make sure to provide a valid `ist` file when using this option.

`german` `S`

This option, as the name indicates, sets whether German word ordering should be used when generating the index, according to the rules set forth in DIN 5007.

`order`

This option, as the name indicates, sets the default ordering scheme for the `makeindex` program. Two possible values are available:

`letter`

This value, as the name indicates, activates the letter ordering scheme. In such scheme, a blank space does not precede any letter in the alphabet.

`word`

This value, as the name indicates, activates the word ordering scheme. In such scheme, a blank space precedes any letter in the alphabet.

`input` *default:* `idx`

This option, as the name indicates, sets the default extension for the input file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`output` *default:* `ind`

This option, as the name indicates, sets the default extension for the output file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`log` *default:* `ilg`

This option, as the name indicates, sets the default extension for the log file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
1 % arara: makeindex: { style: book.ist }
```

`nomencl` *Marco Daniel, Nicola Talbot, Paulo Cereda*

This rule runs `makeindex` in order to automatically generate a nomenclature list from TeX documents that work with the `nomencl` package. The program itself is a general purpose hierarchical index generator and takes the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `nlo` suffix and a special style file in order to generate the nomenclature list as a special `nls` file.

`style` *default:* `nomencl.ist`

This option, as the name indicates, sets the underlying index style file. The default value is set to the one automatically provided by the `nomencl` package, so it is highly recommended to not override it.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
1 % arara: nomencl
```

`pdfcsplain` *Paulo Cereda*

This rule runs the `pdfcsplain` TeX engine, a conservative extension of Knuth's plain TeX with direct processing characters and hyphenation patterns for Czech and Slovak, on the provided `◊currentFile` reference.

`interaction`

This option alters the underlying engine behaviour. If this option is omitted, TeX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

`batchmode`

In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TeX tries to input or request from keyboard input cause the job to abort.

`nonstopmode`

In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` S
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
% arara: pdfcsplain: { shell: yes, synctex: yes }
```

`pdflatex` *Marco Daniel, Paulo Cereda*

This rule runs the `pdflatex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in the Portable Document File format, as expected.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

  `batchmode`
  : In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

  `nonstopmode`
  : In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` S
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
1 % arara: pdflatex: { interaction: batchmode }
2 % arara: --> if missing('pdf') || changed('tex')
```

`pdftex` *Marco Daniel, Paulo Cereda*

This rule runs the `pdftex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in the Portable Document File format, as expected.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

  `batchmode`
  : In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

  `nonstopmode`
  : In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`
: In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`
: In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`draft` S
: This option sets whether the draft mode, i.e, a mode that produces no output, so the engine can check the syntax, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: pdftex: { draft: yes }
```

`pdftk` *Nicola Talbot, Paulo Cereda*

This rule runs `pdftk`, a command line tool for manipulating Portable Document Format documents, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `pdf` suffix.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: pdftk: { options: [ burst ] }
```

`ps2pdf` *Marco Daniel, Paulo Cereda*

This rule runs `ps2pdf`, a tool that converts PostScript to Portable Document File, on the corresponding base name of the `◊currentFile` reference

(i.e, the name without the associated extension) as a string concatenated with the `ps` suffix.

`output`
: This option, as the name indicates, sets the output name for the generated `pdf` file. There is no need to provide an extension, as the value is always normalized with `◊getBasename` such that only the name without the associated extension is used. The base name of the current file reference is used as the default value.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
1 % arara: ps2pdf: { output: article }
```

`sketch` *Sergey Ulyanov, Paulo Cereda*

This rule runs `sketch`, a system for producing line drawings of solid objects and scenes, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `sk` suffix. Note that one needs to add support for this particular file type, as seen in Section 4.2, on page 43.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



Example

```
1 % arara: sketch
```

`songidx` *Francesco Endrici, Paulo Cereda*

This rule runs `songidx`, a song index generation script for the `songs` package, on the file reference provided as parameter, generating a proper index as a special `sbx` file. It is very important to observe that, at the time of writing, this script is not available off the shelf in TeX Live or MiKTeX distributions, so a manual deployment is required. The script execution is performed by the underlying `texlua` interpreter.

`input` `R`
: This required option, as the name indicates, sets the input name for the song index file specified within the TeX document. There is no

need to provide an extension, as the value is always normalized with ◊getBasename such that only the name without the associated extension is used.

script *default:* songidx.lua

This option, as the name indicates, sets the script path. The default value is set to the script name, so either make sure songidx.lua is located in the same directory of your TEX document or provide the correct location (preferably a full path).

options

This option, as the name indicates, takes a list of raw command line options and appends it to the actual script call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: songidx: { input: songs }
```

tex *Marco Daniel, Paulo Cereda*

This rule runs the tex TEX engine on the provided ◊currentFile reference, generating a corresponding file in a device independent format.

interaction

This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

batchmode

In this mode, nothing is printed on the terminal, and errors are scrolled as if the return key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

nonstopmode

In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

scrollmode

In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

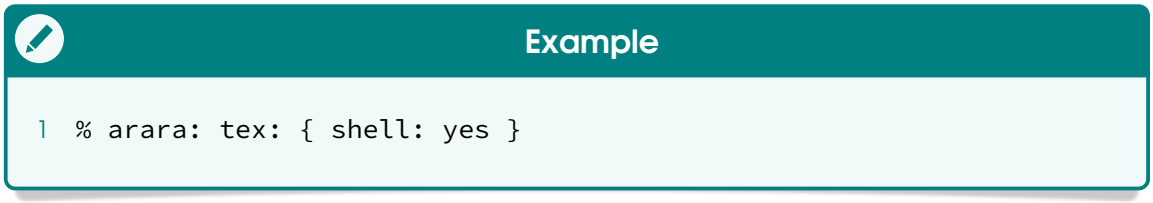
errorstopmode

In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

shell S

This option sets whether the possibility of running underlying system commands from within TEX is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.



`texindy` *Nicola Talbot, Paulo Cereda*

This rule runs `texindy`, a variant of the `xindy` indexing system focused on LaTeX documents, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `idx` suffix, generating an index as a special `ind` file.

`quiet` `S`
: This option, as the name indicates, sets whether the tool will output progress messages. It is important to observe that `texindy` always outputs error messages, regardless of this option.

`codepage`
: This option, as the name indicates, specifies the encoding to be used for letter group headings. Additionally, it specifies the encoding used internally for sorting, but that does not matter for the final result.

`language`
: This option, as the name indicates, specifies the language that dictates the rules for index sorting. These rules are encoded in a module.

`markup`
: This option, as the name indicates, specifies the input markup for the raw index. The following values are available:

  `latex`
  : This value, as the name implies, is emitted by default from the LaTeX kernel, and the raw input is encoded in the LaTeX Internal Character Representation format.

  `xelatex`
  : This value, as the name implies, acts like the previous `latex` markup option, but without `inputenc` usage. Raw input is encoded in the UTF-8 format.

  `omega`
  : This value, as the name implies, acts like the previous `latex` markup option, but with Omega's special notation as encoding for characters not in the ASCII set.

`modules`
: This option, as the name indicates, takes a list of module names. Modules are searched in the usual application path. An error is thrown if any data structure other than a proper list is provided as the value.

`input` *default:* `idx`

This option, as the name indicates, sets the default extension for the input file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`output` *default:* `ind`

This option, as the name indicates, sets the default extension for the output file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`log` *default:* `ilg`

This option, as the name indicates, sets the default extension for the log file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: texindy: { markup: latex }
```

`tikzmake` *Robbie Smith, Paulo Cereda*

This rule runs `make` on a very specific build file generated by the `tikzmake` package, as a means to simplify the externalization of TikZ pictures. This build file corresponds to the base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `makefile` suffix.

`force` `S`

This option, as the name indicates, sets whether all targets specified in the corresponding build file should be unconditionally made.

`jobs`

This option, as the name indicates, specifies the number of jobs (commands) to run simultaneously. Note that the provided value must be a positive integer. The default number of job slots is one, which means serial execution.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: tikzmake: { force: yes, jobs: 2 }
```

`velocity` *Paulo Cereda*

This rule, as the name suggests, calls the `◊mergeVelocityTemplate` method, merging an input template file written according to the Velocity Template Language 1.7 specification with the provided `Map` data object in order to produce a corresponding `File` output. Be mindful that this particular rule returns `true` if, and only if, the aforementioned method is successfully executed. Otherwise, an exception is raised.

`input`

This option, as the name indicates, sets the input template file, written according to the Velocity Template Language 1.7 specification, as a proper `File` reference. Please note that the `◊currentFile` reference is used as default input when this option is not set.

`output` `R`

This required option, as the name indicates, sets the output `File` reference. Be mindful that, if the reference exists, it will be overwritten without any warning.

`context` `R`

This required option, as the name indicates, sets the `Map` data object to be used as context to the method call, according to the provided value. An error is thrown if any data structure other than a proper map is specified.

Example

```
% arara: velocity: { input: input.txt, output: output.txt,
% arara: --> context: { name: Paulo, country: Brazil } }
```

`xdvipdfmx` *Marco Daniel, Paulo Cereda*

This rule runs `xdvipdfmx`, the back end for the `xetex` TEX engine (and not intended to be invoked directly), on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `dvi` suffix, generating a Portable Document Format `pdf` file.

`output`

This option, as the name indicates, sets the output name for the generated `pdf` file. There is no need to provide an extension, as the value is always normalized with `◊getBasename` such that only the name without the associated extension is used. The base name of the current file reference is used as the default value.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: xdvipdfmx: { output: thesis }
```

`xelatex` *Marco Daniel, Paulo Cereda*

This rule runs the new `xelatex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in the Portable Document File format, as expected.

`interaction`
: This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

  `batchmode`
  : In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

  `nonstopmode`
  : In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

  `scrollmode`
  : In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

  `errorstopmode`
  : In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` `S`
: This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` `S`
: This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`options`
: This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: xelatex: { shell: yes, synctex: yes }
```

`xetex` *Marco Daniel, Paulo Cereda*

This rule runs the `xetex` TEX engine on the provided `◊currentFile` reference, generating a corresponding file in the Portable Document File format, as expected.

`interaction`

This option alters the underlying engine behaviour. If this option is omitted, TEX will prompt the user for interaction in the event of an error. Possible values are, in order of increasing user interaction (courtesy of our master Enrico Gregorio):

`batchmode`

In this mode, nothing is printed on the terminal, and errors are scrolled as if the `return` key is hit at every error. Missing files that TEX tries to input or request from keyboard input cause the job to abort.

`nonstopmode`

In this mode, the diagnostic message will appear on the terminal, but there is no possibility of user interaction just like in batch mode, previously described.

`scrollmode`

In this mode, as the name indicates, TEX will stop only for missing files to input or if proper keyboard input is necessary. TEX fixes errors itself.

`errorstopmode`

In this mode, TEX will stop at each error, asking for proper user intervention. This is the most user interactive mode available.

`shell` S

This option sets whether the possibility of running underlying system commands from within TEX is activated.

`synctex` S

This option sets whether `synctex`, an input and output synchronization feature that allows navigation from source to typeset material and vice versa, available in most TEX engines, is activated.

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
1 % arara: xetex: { interaction: scrollmode, synctex: yes }
```

`xindy` *Nicola Talbot, Paulo Cereda*

This rule runs `xindy`, a flexible and powerful indexing system, on the corresponding base name of the `◊currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `idx` suffix, generating an index as a special `ind` file.

`quiet` S

This option, as the name indicates, sets whether the tool will output progress messages. It is important to observe that `xindy` always outputs error messages, regardless of this option.

`codepage`

This option, as the name indicates, specifies the encoding to be used for letter group headings. Additionally, it specifies the encoding used internally for sorting, but that does not matter for the final result.

`language`

This option, as the name indicates, specifies the language that dictates the rules for index sorting. These rules are encoded in a module.

`markup`

This option, as the name indicates, specifies the input markup for the raw index. The following values are available:

`latex`

This value, as the name implies, is emitted by default from the LaTeX kernel, and the raw input is encoded in the LaTeX Internal Character Representation format.

`xelatex`

This value, as the name implies, acts like the previous `latex` markup option, but without `inputenc` usage. Raw input is encoded in the UTF-8 format.

`omega`

This value, as the name implies, acts like the previous `latex` markup option, but with Omega's special notation as encoding for characters not in the ASCII set.

`xindy`

This value, as the name implies, uses the `xindy` input markup as specified in the `xindy` manual.

`modules`

This option, as the name indicates, takes a list of module names. Modules are searched in the usual application path. An error is thrown if any data structure other than a proper list is provided as the value.

`input` *default:* `idx`

This option, as the name indicates, sets the default extension for the

input file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`output` *default:* `ind`

This option, as the name indicates, sets the default extension for the output file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`log` *default:* `ilg`

This option, as the name indicates, sets the default extension for the log file, according to the provided value. Later, this value will be concatenated as a suffix for the base name of the `◊currentFile` reference (i.e, the name without the associated extension).

`options`

This option, as the name indicates, takes a list of raw command line options and appends it to the actual system call. An error is thrown if any data structure other than a proper list is provided as the value.

Example

```
% arara: xindy: { markup: xelatex }
```

It is highly advisable to browse the relevant documentation about packages and tools described in this chapter as a means to learn more about features and corresponding advanced usage. For TeX Live users, we recommend the use of `texdoc`, a command line program to find and view documentation. For example, this manual can be viewed through the following command:

Terminal

```
$ texdoc arara
```

The primary function of the handy `texdoc` tool is to locate relevant documentation for a given keyword (typically, a package name) on your disk, and open it in an appropriate viewer. For MiKTeX users, the distribution provides a similar tool named `mthelp` to find and view documentation. Make sure to use these tools whenever needed!



part two

Development and deployment



Building from source

arara is a Java application licensed under the New BSD License, a verified GPL-compatible free software license, and the source code is available in the project repository at GitHub. This chapter provides detailed instructions on how to build our tool from source.

8.1 Requirements

In order to build our tool from source, we need to ensure that our development environment has the minimum requirements for a proper compilation. Make sure the following items are available:

- On account of our project being hosted at GitHub, an online source code repository, we highly recommend the installation of `git`, a version control system for tracking changes in computer files and coordinating work on those files among multiple people. Alternatively, you can directly obtain the source code by requesting a source code download in the repository. In order to check if `git` is available in your operating system, run the following command in the terminal (version numbers might vary):

  Terminal

  ```
  $ git --version
  git version 2.17.1
  ```

  Please refer to the `git` project website in order to obtain specific installation instructions for your operating system. In general, most recent Unix systems have `git` installed out of the shelf.

- Our tool is written in the Java programming language, so we need a proper Java Development Kit, a collection of programming tools for the Java platform. Our source code is known to be compliant with several vendors, including Oracle, OpenJDK, and Azul Systems. In order to check if your operating system has the proper tools, run the following command in the terminal (version numbers might vary):



Terminal

```
$ javac -version
javac 1.8.0_171
```

The previous command, as the name suggests, refers to the `javac` tool, which is the Java compiler itself. The most common Java Development Kit out there is from Oracle. However, several Linux distributions (as well as some developers, yours truly included) favour the OpenJDK vendor, so your mileage may vary. Please refer to the corresponding website of the vendor of your choice in order to obtain specific installation instructions for your operating system.

- ☑ As a means to provide a straightforward and simplified compilation workflow, arara relies on Apache Maven, a software project management and comprehension tool. Based on the concept of a project object model, Maven can manage builds, reporting and documentation from a central piece of information. In order to check if `mvn`, the Maven binary, is available in your operating system, run the following command in the terminal (version numbers might vary):



Terminal

```
$ mvn --version
Apache Maven 3.5.2 (Red Hat 3.5.2-5)
Maven home: /usr/share/maven
Java version: 1.8.0_171, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-1.8.0-openjdk-
    1.8.0.171-4.b10.fc28.x86_64/jre
Default locale: pt_BR, platform encoding: UTF-8
OS name: "linux", version: "4.16.16-300.fc28.x86_64",
    arch: "amd64", family: "unix"
```

Please refer to the Maven project website in order to obtain specific installation instructions for your operating system. In general, most recent Linux distributions have the Maven binary, as well the proper associated dependencies, available in their corresponding repositories.

- ☑ For a proper repository cloning, as well as the first Maven build, an active Internet connection is required. In particular, Maven dynamically downloads Java libraries and plug-ins from one or more online repositories and stores them in a local cache. Be mindful that subsequent builds can occur offline, provided that the local Maven cache exists.

arara can be easily built from source, provided that the aforementioned requirements are available. The next section presents the compilation details, from repository cloning to a proper Java archive generation.

One tool to rule them all

For the brave, there is the Software Development Kit Manager, an interesting tool for managing parallel versions of multiple software development kits on most Unix based systems. In particular, this tool provides off the shelf support for several Java Development Kit vendors and versions, as well as most recent versions Apache Maven.

Personally, I prefer the packaged versions provided by my favourite Linux distribution (Fedora), but this tool is a very interesting alternative to set up a development environment with little to no effort.

8.2 Compiling the tool

First and foremost, we need to clone the project repository into our development environment, so we can build our tool from source. The cloning will create a directory named `arara/` within the current working directory, so remember to first ensure that you are in the appropriate directory. For example:

Terminal

```
$ mkdir git-projects
$ cd git-projects
```

Run the following command in the terminal to clone the arara project:

Terminal

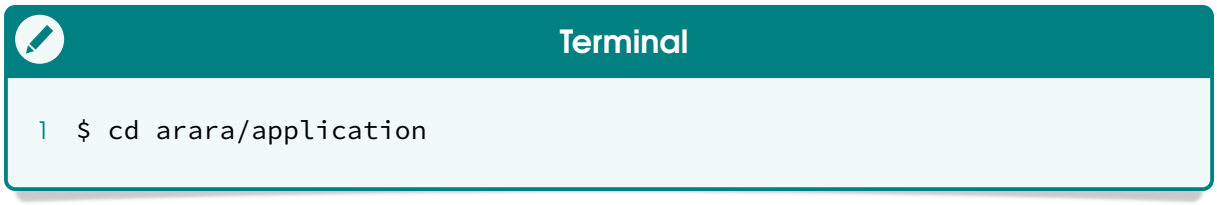
```
$ git clone https://github.com/cereda/arara
```

Wait a couple of seconds (or minutes, depending on your Internet connection) while the previous command clones the project repository hosted at GitHub. Be mindful that this operation pulls down every version of every file for the history of the project. Fortunately, the version control system has the notion of a *shallow clone*, which is a more succinctly meaningful way of describing a local repository with history truncated to a particular depth during the clone operation. If you want to get only the latest revision of everything in our repository, run the following command in the terminal:

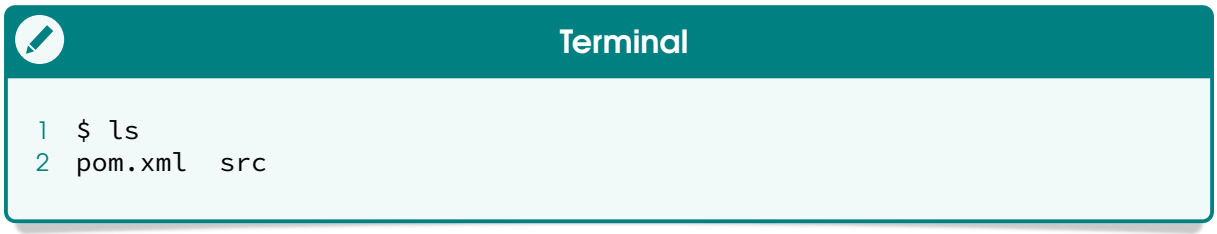
Terminal

```
$ git clone https://github.com/cereda/arara --depth 1
```

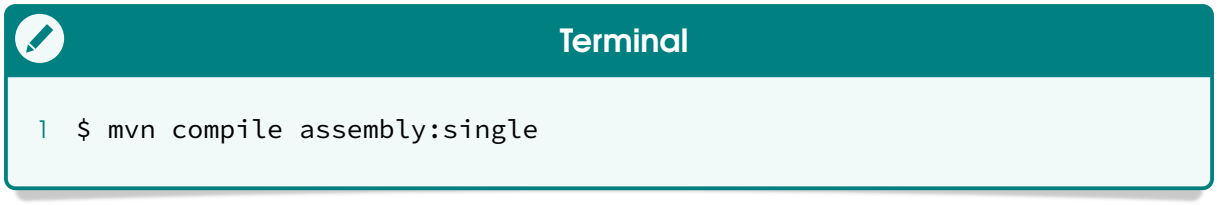
This operation is way faster than the previous one, for obvious reasons. Unix terminals typically start at `USER_HOME` as working directory, so, if you did not `cd` to another directory (as in the earlier example), the newly cloned `arara/` directory is almost certain to be accessible from that level. Now, we need to navigate to a directory named `application/` inside our project structure, where the source code and the corresponding build file are located. Run the following command in the terminal:



The previous command should take us inside the `application/` directory of our project, where the source code and the corresponding build file are located. Let us make sure we are in the correct location by running the following command in the terminal:



Great, we are in the correct location! From the previous output, let us inspect the directory contents. The `src/` directory, as the name suggests, contains the source code organized in an established package structure, whereas `pom.xml` is the corresponding build file written in the Project Object Model format, a special XML file that contains information about the project and configuration details used by Apache Maven to build the project. In order to build our tool, run the following command in the terminal:



Apache Maven is based around the central concept of a build life cycle. The `compile` phase, as the name suggests, compiles the source code of the project using the underlying Java compiler. From the previous command, bound to this particular build phase, note that there is an `assembly:single` plug-in goal which aggregates the project output along with its dependencies and other files into a single distributable archive. An *assembly* is a group of files, directories and dependencies that are assembled into an archive format and distributed. In our case, the resulting file will be a typical Java archive file, with the `jar` extension. The first Maven build will take a couple of seconds (or minutes, depending on your Internet connection), as the tool will download

all dependencies and required plug-ins for proper compilation and packaging. Subsequent builds will be significantly faster. Finally, after some time, Maven will output the following message as result (please note that the entire compilation and packaging only took 10 seconds on my development machine due to an existing local cache):



Terminal

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 10.371 s
[INFO] Finished at: 2018-06-22T18:38:02-03:00
[INFO] Final Memory: 39M/184M
[INFO] ------------------------------------------------------------
```

On account of a successful build in our previous interaction, there is now a newly created `target/` directory containing compiled classes, a established package structure and, at last but not least, our assembly. Now, let us move the resulting Java archive file from that particular directory to `application/` which is our current directory. It is important to note that the aforementioned assembly plug-in adds the `jar-with-dependencies` reference to the file name in order to differentiate a proper assembly from a typical Java archive file. Run the following command in the terminal (please note that the Java archive file was also renamed during the move operation):



Terminal

```
$ mv target/arara-4.0-jar-with-dependencies.jar arara.jar
```

Now, our current directory contains the final `arara.jar` Java archive file properly built from source. This file can be safely distributed and deployed, as seen later on, in Chapter 9, on page 142. You can also test the resulting file by running the following command in the terminal:



Terminal

```
$ java -jar arara.jar
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

arara 4.0 (revision 1)
Copyright (c) 2012-2018, Paulo Roberto Massa Cereda
All rights reserved
```

Terminal (ctd.)

```
10
11 usage: arara [file [--dry-run] [--log] [--verbose | --silent]
12              [--timeout N] [--max-loops N] [--language L]
13              [ --preamble P ] [--header] | --help | --version]
14  -h,--help                  print the help message
15  -H,--header                extract directives only in the file header
16  -l,--log                   generate a log output
17  -L,--language <code>       set the application language
18  -m,--max-loops <number>    set the maximum number of loops
19  -n,--dry-run               go through all the motions of running a
20                             command, but with no actual calls
21  -p,--preamble <name>       set the file preamble based on the
22                             configuration file
23  -s,--silent                hide the command output
24  -t,--timeout <number>      set the execution timeout (in milliseconds)
25  -V,--version               print the application version
26  -v,--verbose               print the command output
```

The following optional Maven phase is used to handle the project cleaning, including the complete removal of the `target/` directory. As a result, the project is then restored to the initial state without any generated Java bytecode. Run the following command in the terminal:

Terminal

```
1 $ mvn clean
```

This section covered the compilation details for building arara from source. The aforementioned steps are straightforward and can be automated in order to generate snapshots and daily builds. If you run into any issue, please let us know. Happy compilation!



Deploying the tool

As previously mentioned, arara runs on top of a Java virtual machine, available on all major operating systems – in some cases, you might need to install the proper virtual machine. This chapter provides detailed instructions on how to properly deploy the tool in your computer from either the official package available in our project repository or a personal build generated from source (as seen in Section 8.2, on page 138).

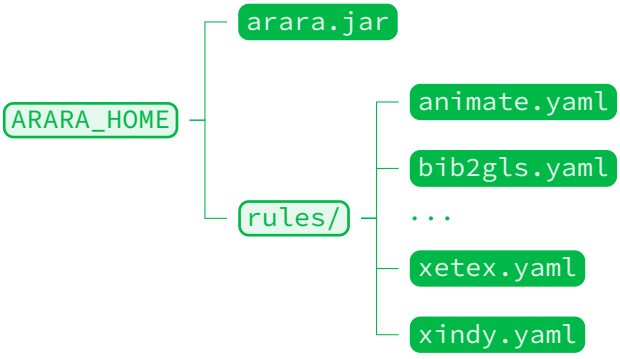


No more installers

Be mindful that, from version 4.0 on, the team decided to not release cross-platform installers any more. Our tool is available off the shelf on all major TEX distributions, including TEX Live and MiKTEX, which makes manual installation unnecessary given the significant coverage of such distributions. Chances are you already have arara in your system!

9.1 Directory structure

From the early development stages, our tool employs a very straightforward directory structure. In short, we provide the `ARARA_HOME` alias to the directory path in which the `arara.jar` Java archive file is located. This particular file is the heart and soul of our tool and dictates the default rule search path, which is a special directory named `rules/` available from the same level. This directory contains all rules specified in the YAML format, as seen in Section 2.1, on page 8. The structure overview is presented as follows.



Provided that this specific directory structure is honoured, the tool is ready for use off the shelf. In fact, the official arara package available in the release section of our project repository, as well as the Bintray software distribution service, exactly mirrors this structure. Once the package is properly downloaded, we simply need to extract it into a proper `ARARA_HOME` location.

9.2 Defining a location

First and foremost, we need to obtain `arara-4.0.zip` from either our project repository at GitHub or at the Bintray service mirror. As the name indicates, this is a compressed file format, so we need to extract it into a proper location. Run the following command in the terminal:

Terminal

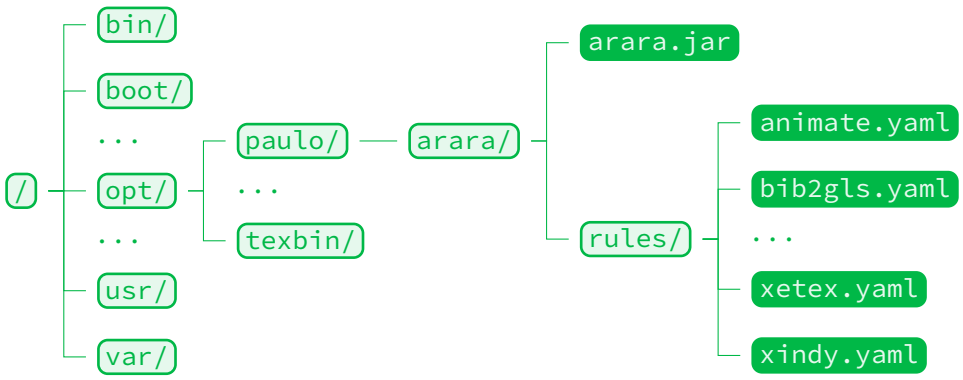
```
$ unzip arara-4.0.zip
```

As a result of the previous command, we obtained a directory named `arara` with the exact structure presented in Section 9.1 in our working directory. Now we need to decide where arara should reside in our system. For example, I usually deploy my tools inside the `/opt/paulo` path, so I need to run the following command in the terminal (please note that my personal directory already has the proper permissions, so I do not need superuser privileges):

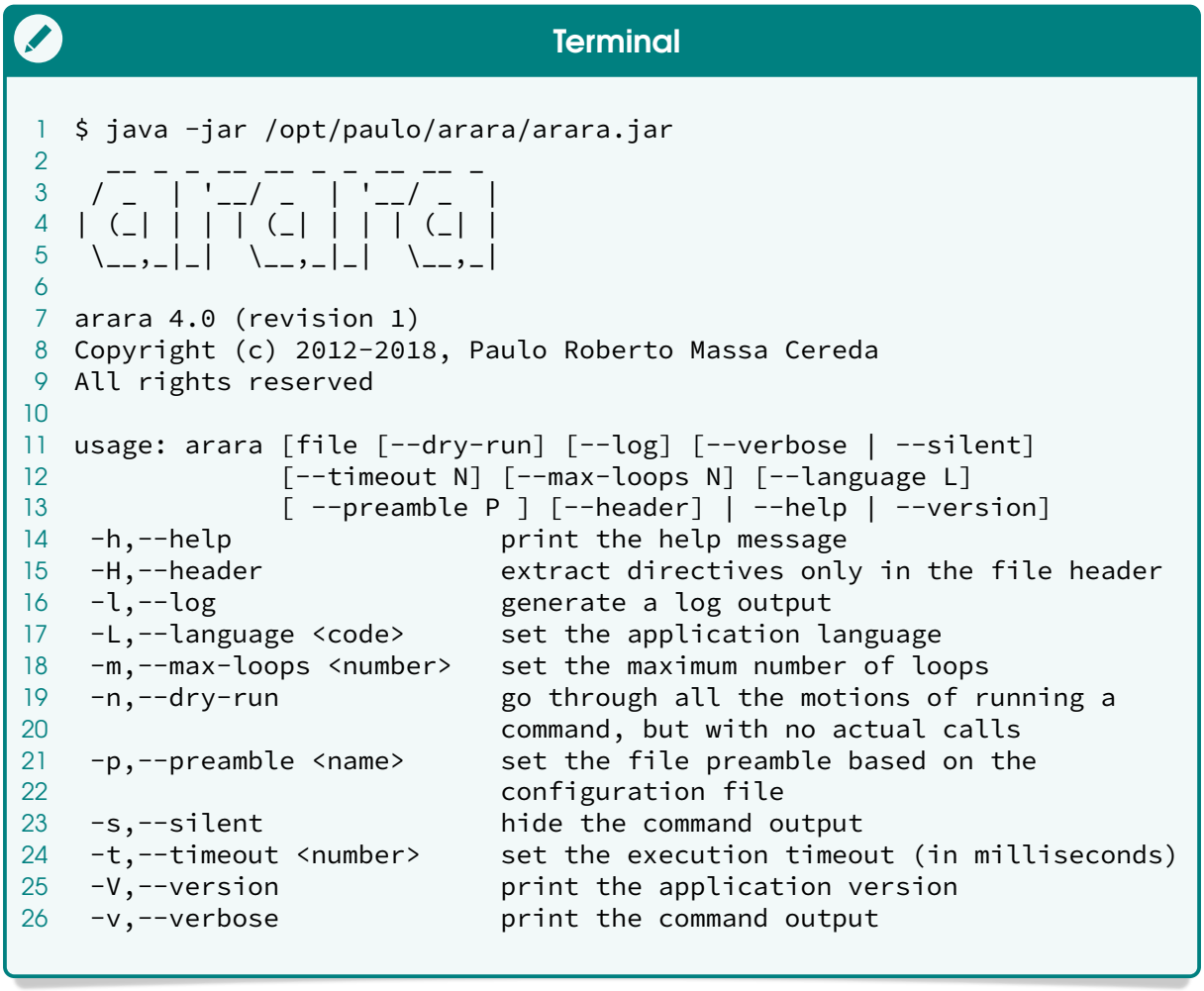
Terminal

```
$ mv arara /opt/paulo/
```

The tool has found a comfortable home inside my system! Observe that the full path of the `ARARA_HOME` reference points out to `/opt/paulo/arara` since this is my deployment location of choice. The resulting structure overview, from the root directory, is presented as follows:



If the tool was built from source (as indicated in Section 8.2, on page 138), make sure to construct the provided directory structure previously presented. We can test the deployment by running the following command in the terminal (please note the full path):



Please observe that, provided that the underlying operating system has an appropriate Java virtual machine installed, arara can be used as a portable, standalone application. Portable applications can be stored on any data storage device, including external devices such as USB drives and floppy disks.

9.3 Tool wrapping

arara is now properly deployed in our system, but we still need to provide the full path of `arara.jar` to the Java virtual machine in order to make our tool work. This section provides three approaches regarding the creation of a *wrapper*, a shell feature that embeds a system command or utility, that accepts and passes a set of parameters to that command.

shell alias

An *alias* is a command available in various shells which enables a replacement of a word by another string. It is mainly used for abbreviating a system command, or for adding default arguments to a regularly used

command. In order to create a shell alias for our tool, open `.bashrc` (a script that is executed whenever a new terminal session starts in interactive mode) in your favourite editor and add the following line, preferably at the end:



Source code

```
1 alias arara='java -jar /opt/paulo/arara/arara.jar'
```

Save the file and restart your terminal. It is important to observe that the given full path must be properly quoted if it contains spaces. There is no need to provide explicit parameters, as an alias simply acts as an inline string replacement.

shell function

A *shell function* is, as the name suggests, a subroutine, a code block that implements a set of operations as a means to performs a specified task. In order to create a shell function for our tool, open `.bashrc` (a script that is executed whenever a new terminal session starts in interactive mode) in your favourite editor and add the following line, preferably at the end:



Source code

```
1 arara() {
2     java -jar /opt/paulo/arara/arara.jar "$@"
3 }
```

Save the file and restart your terminal. It is important to observe that the given full path must be properly quoted if it contains spaces. Note that the `$@` symbol used in the function body represents a special shell variable that stores all the actual parameters in a list of strings.



Alias or function?

In general, an alias should effectively not do more than change the default options of a command, as it constitutes a mere string replacement. A function should be used when you need to do something more complex than an alias. In our particular case, as the underlying logic is pretty straightforward, both approaches are valid.

script file

A *script* is a computer program designed to be run by an interpreter. In our context, the script merely sets up the environment and runs a system command. In order to provide a script for our tool, open your favourite editor and create the following file called `arara` (no extension):

Source file — arara

```
#!/bin/bash
jarpath=/opt/paulo/arara/arara.jar
java -jar "$jarpath" "$@"
```

It is important to observe that the given full path must be properly quoted if it contains spaces. Note that the `$@` symbol used in the script body represents a special shell variable that stores all the actual parameters in a list of strings. This script file will act as the entry point for our tool. Now, we need to make it executable (i.e, set the corresponding execute permission) by running the following command in the terminal:

Source code

```
$ chmod +x arara
```

Now we need to move this newly executable script file to one of the directories set forth in the `PATH` environment variable, where executable commands are located. For illustrative purposes only, let us move the script file to the `/usr/local/bin/` directory, a location originally designed for programs that a normal user may run. Run the following command in the terminal (note the need for superuser privileges):

Source code

```
$ sudo mv arara /usr/local/bin/
```

Alternatively, the script can be placed inside a special directory named `bin/` from the home directory of the current user, which is usually added by default to the system path. Observe that, in this particular case, superuser privileges are not required, as the operation is kept at the current user level. Run the following command in the terminal instead (please note that the `~` symbol is a shell feature called tilde expansion and refers to the home directory of the current user):

Source code

```
$ mv arara ~/bin/
```

There is no need to restart your terminal, as the reference becomes available as soon as it is moved to the new location. Note that a shell script

can provide a convenient variation of a system command where special environment settings, command options, or post-processing apply automatically, but in a way that allows the new script to still act as a fully normal Unix command.

Regardless of the adopted approach, there should be an `arara` wrapper available as an actual Unix command in your shell session. In order to test the wrapper, run the following command in the terminal:



Terminal

```
$ arara
  __ _ _ __ __ _ _ __ __ _
 / _  | '__/ _  | '__/ _  |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

arara 4.0 (revision 1)
Copyright (c) 2012-2018, Paulo Roberto Massa Cereda
All rights reserved

usage: arara [file [--dry-run] [--log] [--verbose | --silent]
             [--timeout N] [--max-loops N] [--language L]
             [ --preamble P ] [--header] | --help | --version]
 -h,--help                  print the help message
 -H,--header                extract directives only in the file header
 -l,--log                   generate a log output
 -L,--language <code>       set the application language
 -m,--max-loops <number>    set the maximum number of loops
 -n,--dry-run               go through all the motions of running a
                            command, but with no actual calls
 -p,--preamble <name>       set the file preamble based on the
                            configuration file
 -s,--silent                hide the command output
 -t,--timeout <number>      set the execution timeout (in milliseconds)
 -V,--version               print the application version
 -v,--verbose               print the command output
```

It is important to observe that the wrapper initiative presented in this section might cause a potential name clash with existing TEX Live or MiKTEX binaries and symbolic links. In this particular scenario, make sure to inspect the command location as a means to ensure a correct execution. To this end, run the following command in the terminal:



Terminal

```
$ which arara
/usr/local/bin/arara
```

The `which` command shows the full path of the executable name provided

as parameter. This particular utility does this by searching for an executable or script in the directories listed in the `PATH` environment variable. Be mindful that aliases and shell functions are listed as well.



part three

A primer on formats and scripting



YAML

According to the specification, YAML (a recursive acronym for *YAML Ain't Markup Language*) is a human-friendly, cross language, Unicode-based data serialization language designed around the common native data type of programming languages. arara uses this format in three circumstances:

1. *Parametrized directives*, as the set of attribute/value pairs (namely, argument name and corresponding value) is represented by a map. This particular type of directive is formally introduced in Section 2.2, on page 18.
2. *Rules*, as their entire structure is represented by a set of specific keys and their corresponding values (a proper YAML document). A rule follows a very strict model, detailed in Section 2.1, on page 8.
3. *Configuration files*, as the general settings are represented by a set of specific keys and their corresponding values (a proper YAML document). Configuration files are covered in Chapter 4, on page 42.

This chapter only covers the relevant parts of the YAML format for a consistent use with arara. For advanced topics, I highly recommend the complete format specification, available online.

10.1 Collections

According to the specification, YAML's block collections use indentation for scope and begin each entry on its own line. Block sequences indicate each entry with a dash and space. Mappings use a colon and space to mark each *key: value* pair. Comments begin with an octothorpe `#`. arara relies solely on mappings and a few scalars to sequences at some point. Let us see an example of a sequence:



A sequence of scalars in YAML

```
team:
- Paulo Cereda
- Marco Daniel
- Brent Longborough
- Nicola Talbot
```

It is quite straightforward: `team` holds a sequence of four scalars. YAML also has flow styles, using explicit indicators rather than indentation to denote scope. The flow sequence is written as a comma-separated list within square brackets:



A sequence of scalars in YAML

```
1 primes: [ 2, 3, 5, 7, 11 ]
```

Attribute maps are easily represented by nesting entries, respecting indentation. For instance, consider a map `developer` containing two keys, `name` and `country`. The YAML representation is presented as follows:



An attribute map in YAML

```
1 developer:
2  name: Paulo
3  country: Brazil
```

Similarly, the flow mapping uses curly braces. Observe that this is the form adopted by a parametrized directive (see syntax in Section 2.2, on page 18):



An attribute map in YAML (flow mapping)

```
1 developer: { name: Paulo, country: Brazil }
```

An attribute map can contain sequences as well. Consider the following code where `developers` holds a list of two developers containing their names and countries:



An attribute map with sequences in YAML

```
1 developers:
2 - name: Paulo
3   country: Brazil
4 - name: Marco
5   country: Germany
```

The previous code can be easily represented in flow style by using square and curly brackets to represent sequences and attribute maps.

10.2 Scalars

Scalar content can be written in block notation, using a literal style, indicated by a vertical bar, where *all line breaks are significant*. Alternatively, they can be written with the folded style, denoted by a greater-than sign, where *each line break is folded to a space* unless it ends an empty or a more-indented line. It is mportant to note that arara intensively uses both styles (as seen in Section 2.1, on page 8). Let us see an example:



Scalar content in literal and folded styles

```
logo: |
  This is the arara logo
  in its ASCII glory!
    __ _ _ __ __ _ _ __ __ _
   / _  | '__/ _  | '__/ _  |
  | (_| | | | (_| | | | (_| |
   \__,_|_|  \__,_|_|  \__,_|
slogan: >
  The cool TeX
  automation tool
```

As seen in the previous code, `logo` holds the ASCII logo of our tool, respecting line breaks. Similarly, observe that the `slogan` key holds the text with line breaks replaced by spaces (in the same fashion TEX does with consecutive, non-empty lines).



Block indentation indicator

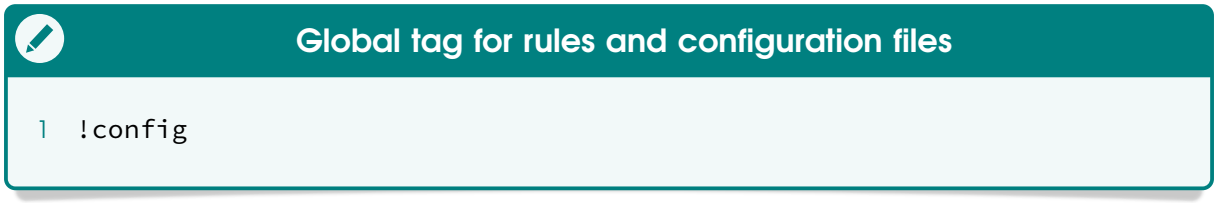
According to the YAML specification, the indentation level of a block scalar is typically detected from its first non-empty line. It is an error for any of the leading empty lines to contain more spaces than the first non-empty line, hence the ASCII logo could not be represented, as it starts with a space.

When detection would fail, YAML requires that the indentation level for the content be given using an explicit indentation indicator. This level is specified as the integer number of the additional indentation spaces used for the content, relative to its parent node. It would be the case if we want to represent our logo without the preceding text.

YAML's flow scalars include the plain style and two quoted styles. The double-quoted style provides escape sequences. The single-quoted style is useful when escaping is not needed. All flow scalars can span multiple lines. Note that line breaks are always folded. Since arara uses MVEL as its underlying scripting language (Chapter 11, on page 154), it might be advisable to quote scalars when starting with forbidden symbols in YAML.

10.3 Tags

According to the specification, in YAML, untagged nodes are given a type depending on the application. The examples covered in this primer use the `seq`, `map` and `str` types from the fail safe schema. Explicit typing is denoted with a tag using the exclamation point symbol. Global tags are usually uniform resource identifiers and may be specified in a tag shorthand notation using a handle. Application-specific local tags may also be used. For arara, there is a special schema used for both rules and configuration files, so in those cases, make sure to add `!config` as global tag:



In particular, rules and configuration files of arara are properly covered in Section 2.1 and Chapter 4, on pages 8 and 42, respectively. For now, it suffices to say that the `!config` global tag is necessary to provide the correct mapping of values inside our tool.

10.4 Further reading

This chapter does not cover all features of the YAML format, so further reading is advisable. I highly recommend the official YAML specification, currently covering the third version of the format.



MVEL

According to the Wikipedia entry, the MVFLEX Expression Language (hereafter referred as MVEL) is a hybrid, dynamic, statically typed, embeddable expression language and runtime for the Java platform. Originally started as a utility language for an application framework, the project is now developed completely independently. arara relies on this scripting language in two circumstances:

1. *Rules*, as nominal attributes gathered from directives are used to build complex command invocations and additional computations. A rule follows a very strict model, detailed in Section 2.1, on page 8.
2. *Conditionals*, as logical expressions must be evaluated in order to decide whether and how a directive should be interpreted. Conditionals are detailed in Section 2.2, on page 18.

This chapter only covers the relevant parts of the MVEL language for a consistent use with arara. For advanced topics, I highly recommend the official language guide for MVEL 2.0, available online.

11.1 Basic usage

The following primer is provided by the official language guide, almost verbatim, with a few modifications to make it more adherent to our needs with arara. Consider the following expression:



Simple property expression

```
1 user.name
```

In this expression, we have a single identifier `user.name`, which by itself is a property expression, in that the only purpose of such an expression is to extract a property out of a variable or context object, namely `user`. Property expressions are widely used by arara, as directive parameters are converted to a map inside the corresponding rule scope. For instance, a parameter `foo` in a directive will be mapped as `parameters.foo` inside a rule during interpretation.

This topic is detailed in Section 2.2, on page 18. The scripting language can also be used for evaluating a boolean expression:



Boolean expression evaluation

```
1 user.name == 'John Doe'
```

This expression yields a boolean result, either `true` or `false` based on a comparison operation. Like a typical programming language, MVEL supports the full gamut of operator precedence rules, including the ability to use bracketing to control execution order:



Execution order control through bracketing

```
1 (user.name == 'John Doe') && ((x * 2) - 1) > 20
```

You may write scripts with an arbitrary number of statements using a semicolon to denote the termination of a statement. This is required in all cases except in cases where there is only one statement, or for the last statement in a script:



Multiple statements

```
1 statement1; statement2; statement3
```

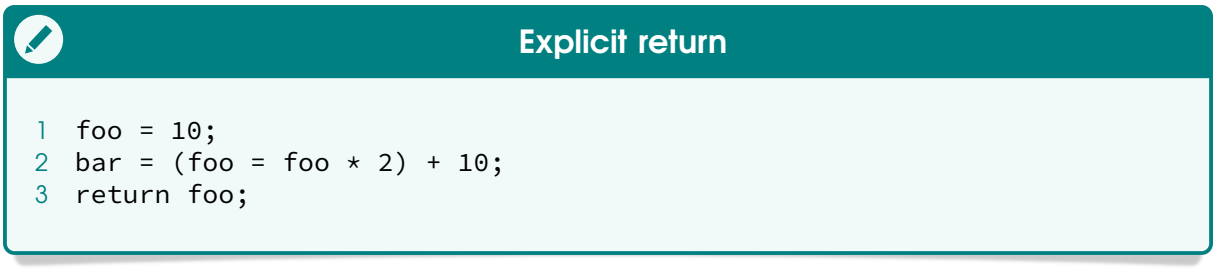
It is important to observe that MVEL expressions use a *last value out* principle. This means, that although MVEL supports the `return` keyword, it can be safely omitted. For example:



Automatic return

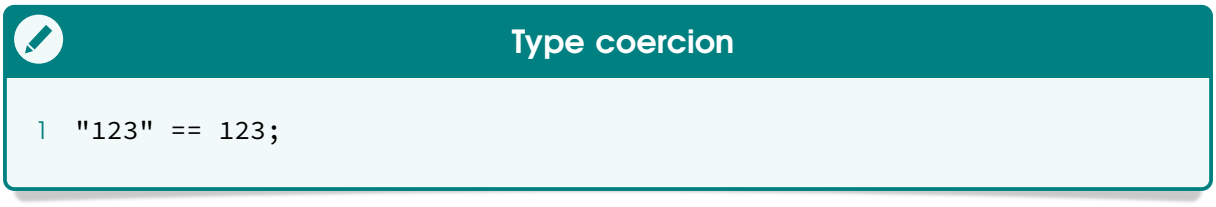
```
1 foo = 10;
2 bar = (foo = foo * 2) + 10;
3 foo;
```

In this particular example, the expression automatically returns the value of `foo` as it is the last value of the expression. It is functionally identical to:



Personally, I like to explicitly add a `return` statement, as it provides a visual indication of the expression exit point. All rules released with arara favour this writing style. However, feel free to choose any writing style you want, as long as the resulting code is consistent.

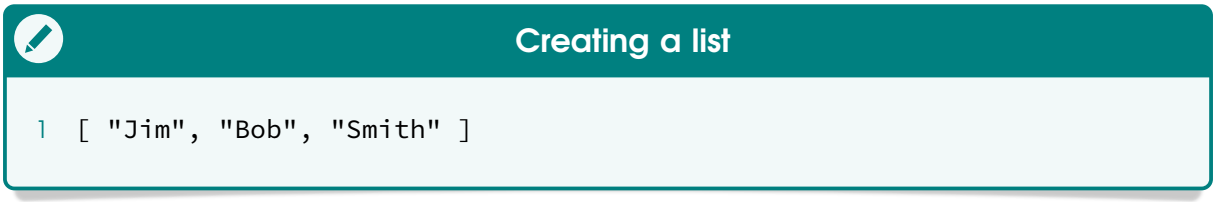
The type coercion system of MVEL is applied in cases where two incomparable types are presented by attempting to coerce the right value to that of the type of the left value, and then vice-versa. For example:



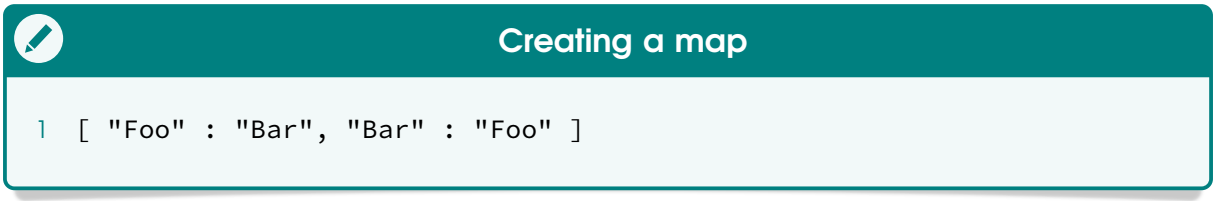
Surprisingly, the evaluation of such expression holds `true` in MVEL because the underlying type coercion system will coerce the untyped number `123` to a string `123` in order to perform the comparison.

11.2 Inline lists, maps and arrays

According to the documentation, MVEL allows you to express lists, maps and arrays using simple elegant syntax. Lists are expressed in the following format:



Note that lists are denoted by comma-separated values delimited by square brackets. Similarly, maps (sets of key/value attributes) are expressed in the following format:



Note that attributes are composed by a key, a colon and the corresponding

value. A map is denoted by comma-separated attributes delimited by square brackets. Finally, arrays are expressed in the following format:



Creating an array

```
1 { "Jim", "Bob", "Smith" }
```

One important aspect about inline arrays is their special ability to be coerced to other array types. When you declare an inline array, it is untyped at first and later coerced to the type needed in context. For instance, consider the following code, in which `sum` takes an array of integers:



Array coercion

```
1 math.sum({ 1, 2, 3, 4 });
```

In this case, the scripting language will see that the target method accepts an integer array and automatically type the provided untyped array as such. This is an important feature exploited by arara when calling methods within the rule or conditional scope.

11.3 Property navigation

MVEL provides a single, unified syntax for accessing properties, static fields, maps and other structures. Lists are accessed the same as arrays. For example, these two constructs are equivalent (MVEL and Java access styles for lists and arrays, respectively):



MVEL access style for lists and arrays

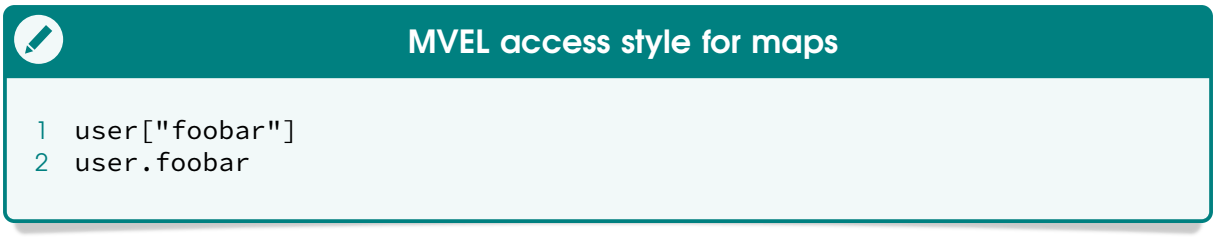
```
1 user[5]
```

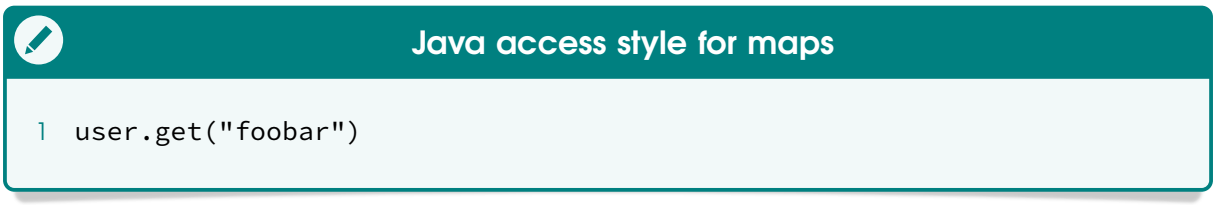


Java access style for lists and arrays

```
1 user.get(5)
```

Observe that MVEL accepts plain Java methods as well. Maps are accessed in the same way as arrays except any object can be passed as the index value. For example, these two constructs are equivalent (MVEL and Java access styles for maps, respectively):

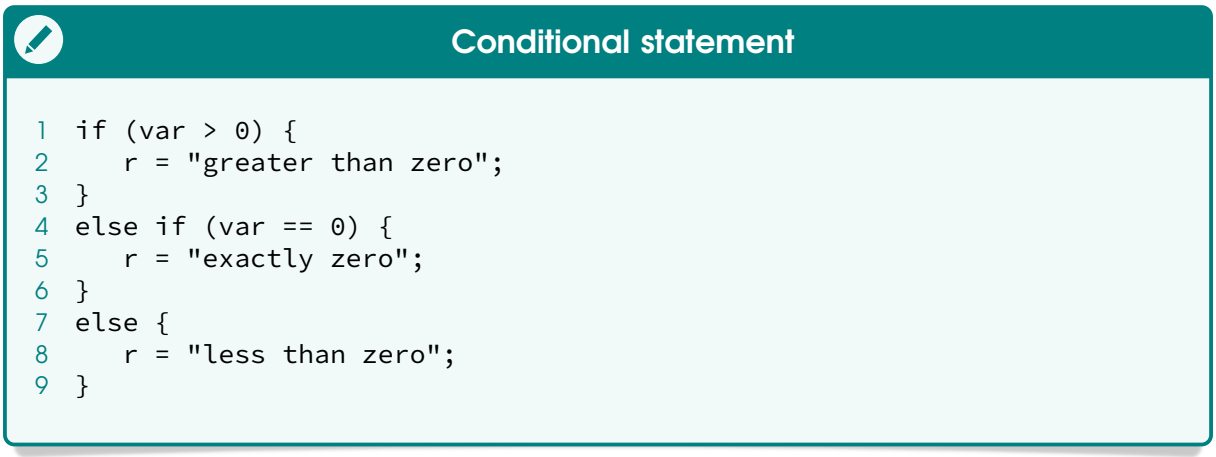




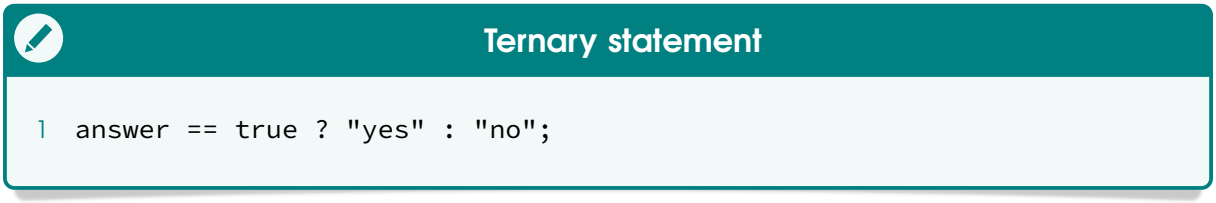
It is advisable to favour such access styles over their Java counterparts when writing rules and conditionals for arara. The clean syntax helps to ensure more readable code.

11.4 Flow control

The expression language goes beyond simple evaluations. In fact, MVEL supports an assortment of control flow operators (namely, conditionals and repetitions) which allows advanced scripting operations. Consider this conditional statement:



As seen in the previous code, the syntax is very similar to the ones found in typical programming languages. MVEL also provides a shorter version, known as a ternary statement:



The `foreach` statement accepts two parameters separated by a colon, the first being the local variable holding the current element, and the second the

collection or array to be iterated over. For example:



Iteration statement

```
1 foreach (name : people) {
2     System.out.println(name);
3 }
```

As expected, MVEL also implements the standard C `for` loop. Observe that newer versions of MVEL allow an abbreviation of `foreach` to the usual `for` statement, as syntactic sugar. In order to explicitly indicate a collection iteration, we usually use `foreach` in the default rules for arara, but both statements behave exactly the same from a semantic point of view.



Iteration statement

```
1 for (int i = 0; i < 100; i++) {
2    System.out.println(i);
3 }
```

The scripting language also provides two versions of the `do` statement: one with `while` and one with `until` (the latter being the exact inverse of the former):



Iteration statement

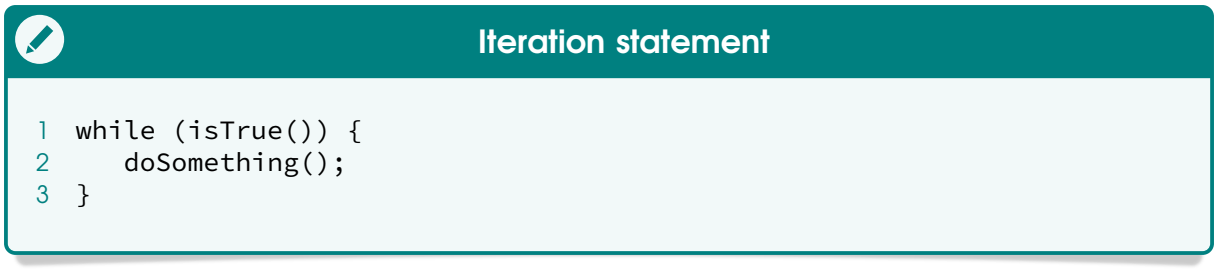
```
1 do {
2     x = something();
3 } while (x != null);
```

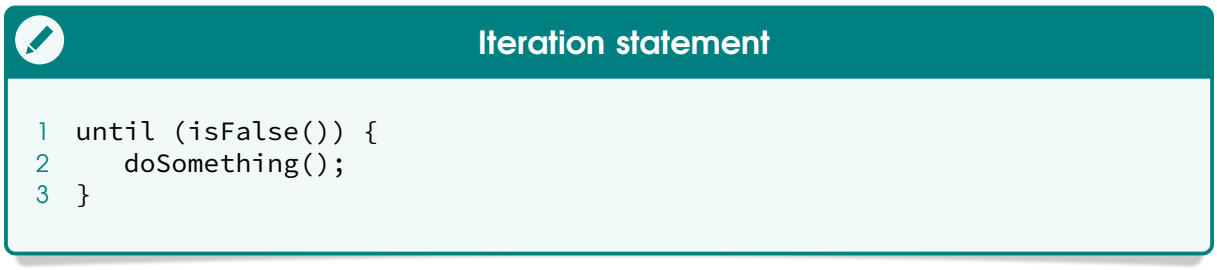


Iteration statement

```
1 do {
2    x = something();
3 } until (x == null);
```

Finally, MVEL also implements the standard `while`, with the significant addition of an `until` counterpart (for inverted logic):

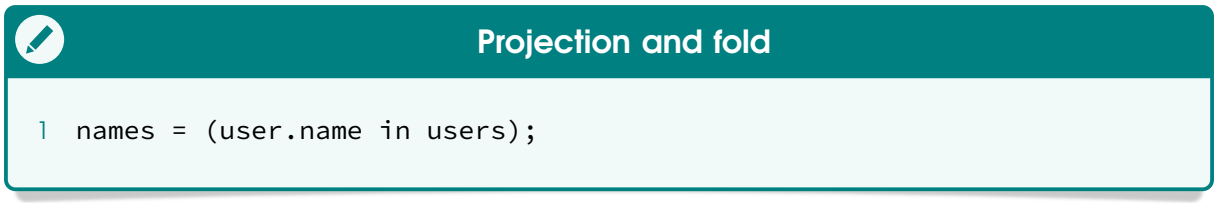




Since `while` and `until` are unbounded (i.e, the number of iterations required to solve a problem may be unpredictable), we usually tend to avoid using such statements when writing rules for arara.

11.5 Projections and folds

Projections are a way of representing collections. According to the official documentation, using a very simple syntax, one can inspect very complex object models inside collections in MVEL using the `in` operator. For example:



As seen in the above code, `names` holds all values from the `name` property of each element, represented locally by a placeholder `user`, from the collection `users` being inspected. This feature can even perform nested operations.

11.6 Assignments

According to the official documentation, the scripting language allows variable assignment in expressions, either for extraction from the runtime, or for use inside the expression. As MVEL is a dynamically typed language, there is no need to specify a type in order to declare a new variable. However, feel free to explicitly declare the type when desired.

Assignment

```
1 str = "My string";
2 String str = "My string";
```

Unlike Java, however, the scripting language provides automatic type conversion (when possible) when assigning a value to a typed variable. In the following example, an integer value is assigned to a string:

Assignment

```
1 String num = 1;
```

For dynamically typed variables, in order to perform a type conversion, it is just a matter of explicitly casting the value to the desired type. In the following example, an explicit string cast is assigned to the `num` variable:

Assignment

```
1 num = (String) 1;
```

When writing rules for arara, is advisable to keep variables to a minimum in order to avoid unnecessary assignments and a potential performance drop. However, make sure to favour readability over unmaintained code.

11.7 Basic templating

MVEL templates are comprised of *orb* tags inside a plain text document. Orb tags denote dynamic elements of the template which the engine will evaluate at runtime. arara heavily relies on this concept for runtime evaluation of conditionals and rules. For rules, we use orb tags to return either a string from a textual template or a proper command object. The former constituted the basis of command generation in previous versions of our tool; from version 4.0 on, we highly recommend the latter, detailed in Section 2.1, on page 2.1. Conditionals are in fact orb tags in disguise, such that the expression (or a sequence of expressions) is properly evaluated at runtime. Consider the following example:

Template

```
1 My favourite team is @{ person.name == 'Enrico'
2 ? 'Juventus' : 'Palmeiras' }!
```

The above code features a basic form of orb tag named *expression orb*. It contains an expression (or a sequence of expressions) which will be evaluated to a certain value, as seen earlier on, when discussing the *last value out* principle. In the example, the value to be returned will be a string containing a football team name (the result is of course based on the comparison outcome).

11.8 Further reading

This chapter does not cover all features of the MVEL expression language, so further reading is advisable. I highly recommend the MVEL language guide currently covering version 2.0 of the language.