



arara

The cool \TeX automation tool

Reference manual

The Island of TeX



Version 7.0.2

License

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 2012–2022, Island of TeX
All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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.

Contents

● Introduction	8
● What is this tool?	8
● Core concepts	10
● Operating system remarks	12
● Support	13

● Important concepts	14
● Rules	14
● Directives	23
● Important changes in version 7	28

● Command line	30
● User interface design	30
● Options	32
● File name lookup	40

● Configuration file	42
● File lookup	42
● Basic structure	43

● Logging	51
● System information	51
● Directive extraction	52
● Directive normalization	53
● Rule interpretation	54

● Projects	57
● Rationale	57
● File structure	58

● Methods	60
● Files	61

● Conditional flow	70
● Strings	73
● Operating systems	75
● Type checking	77
● Classes and objects	78
● Dialog boxes	80
● Commands	86
● Others	89

● The official rule pack	94
● animate	94
● asymptote	95
● authorindex	97
● bib2gls	98
● biber	99
● bibtex	100
● bibtex8	100
● bibtexu	101
● clean	101
● context	102
● convert	104
● copy	104
● csplain	105
● datatooltk	106
● detex	108
● dvipdfm	109
● dvipdfmx	109
● dvips	110
● dvipspdf	110
● dvisvgm	111
● etex	112
● fig2dev	112
● frontespizio	113
● ghostscript	115
● gnuplot	115
● halt	116
● indent	116
● knitr	118

● latex	119
● latexmk	120
● llmk	121
● ltx2any	122
● luahtex	124
● lualatex	125
● luatex	126
● make	127
● makeglossaries	127
● makeglossarieslite	128
● makeindex	128
● metapost	129
● move	130
● nomencl	130
● pbibtex	131
● pdfcrop	131
● pdfcsplain	133
● pdflatex	134
● pdftex	135
● pdftk	136
● perltex	136
● platex	138
● ps2pdf	139
● pythontex	140
● qpdf	141
● sage	141
● sketch	142
● songidx	143
● spix	143
● tex	144
● texcount	145
● texindy	146
● tikzmake	147
● upbibtex	148
● uplatex	148
● uptex	149
● xdvipdfmx	150

● xelatex	151
● xetex	152
● xindex	153
● xindy	154
<hr/>	
● Building arara	156
● Requirements	156
● Compiling the tool	158
● Fetching the sources	158
● Building the executable	159
● Executing the fresh build	160
<hr/>	
● Deploying the tool	162
● Directory structure	162
● Defining a location	163
● Tool wrapping	165
<hr/>	
● Introduction to the YAML format	169
● Collections	169
● Scalars	170
● Tags	172
● Further reading	172
<hr/>	
● MVEL	173
● Basic usage	173
● Inline lists, maps and arrays	175
● Property navigation	176
● Flow control	176
● Projections and folds	178
● Assignments	178
● Basic templating	179
● Further documentation	180

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.

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:

```
\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:

```
$ latexmk -pdf hello.tex
$ rubber --pdf hello.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:


```
$ arara hello.tex
```

```
-- -- -- -- -- -- -- -- --  
/ _` | '___/ _` | '___/ _` |  
| ( _| | | | ( _| | | | ( _| |  
\\_ _|_|_| \\_ _|_|_| \\_ _|_|
```

Processing "hello.tex" (size: 70 B, last modified: 12/28/2020 07:03:16), please wait.

ERROR

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

Total: 0.04 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):

```
% 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 **IMPORTANT CONCEPTS** . 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:

```
$ arara hello.tex
```

```
-- -- -- -- -- -- -- -- --
```

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

```
Processing "hello.tex" (size: 88 B, last modified: 12/28/2020  
07:05:05), please wait.
```

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

```
Total: 0.56 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.

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 much easier.

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 re-visit these concepts later on in this manual for a comprehensive explanation (more precisely, in **IMPORTANT CONCEPTS**).

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:

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

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

Once arara finds a directive, it will look for the associated *rule* . In our example, it will look for a rule named `pdf \LaTeX` which will evidently run the `pdf \LaTeX` 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 [IMPORTANT CONCEPTS](#) .

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.

In order to decide whether a command execution is successful, arara relies on exit status checking. Typically, a command is successful if, and only if, its resulting exit status is 0 and no other value. However, 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 [IMPORTANT CONCEPTS](#) . 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.

Operating system remarks

The application is written using the Kotlin 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 Java 8 to 18, from any vendor.

Outdated Java virtual machines

Dear reader, beware of outdated software, mainly Java virtual machines! Although arara offers support for older virtual machines, try your best to keep your software updated as frequently as possible. The legacy support exists only for historical reasons, and also due to the sheer fact that we know some

people that still runs arara on very old hardware. If you are not in this particular scenario, get the latest virtual machine.

In [BUILDING ARARA](#) , we provide instructions on how to build arara from sources using Gradle. 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.

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 [MATRIX](#) and [GITTER](#) chat rooms, in which we occasionally hang out. Also, if you think the report is worthy of an issue, open one in our [GITLAB REPOSITORY](#) .

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.

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.

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:

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

```
Processing "doc1.tex" (size: 31 B, last modified: 12/28/2020  
07:37:37), please wait.
```

```
ERROR
```

```
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: 0.03 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. 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).

A note about rules

From version 6.0 on, rules included in the core distribution have been re-named to have a unique prefix in the texmf tree. File names should not be relied upon.

In short, a rule is a plain text file written in the YAML format, described in [YAML](#). 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 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. Observe, however, that rules in the core distribution have a different naming scheme than the ones located in the user space.

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. The tool requires it, so make sure to start all rules with a `!config` keyword.
- **[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 `yaml` is known to be a valid YAML extension as well, arara only considers files ending with the `yaml` extension. This is a deliberate decision.

```
identifier: pdflatex
```

- **[M] name** : This key holds the name of the *task* (a rule instantiated through a directive) as a plain string. When running arara, this value will be displayed in the output enclosed in parentheses.

```
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.

```
authors:  
- Marco Daniel  
- Paulo Cereda
```

- **M commands** : This key 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.

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. The tool offers two types of returned values:
 - A `Command` object: arara features an approach for handling system commands based on a high level structure with explicit argument parsing named `Command` . 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.

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

- A boolean value: it is also possible to exploit the expressive power of the underlying scripting language available in the rule context (see [MVEL](#) 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.

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

It is also worth mentioning that `arara` also supports lists of commands represented as `Command` 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.

```
command: "@{ return [ getCommand('ls'), getCommand('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.

There is at least one variable available in the `command` context and is described as follows (note that `MVEL` variables and `orb` tags are discussed in [MVEL](#)). 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.

- `♦ reference` : This variable holds the canonical, absolute path representation of the file name 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 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.

From version 6.0 on, the `<arara>` shorthand is not supported anymore. We encourage the use of a YAML feature named *folded style* when writing such values. The idea here is to use the scalar content in folded style. The new code will look like this:

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

Mind the indentation, as YAML requires it to properly identify blocks. If your code still relies on the `<arara>` shorthand, please update it accordingly to use YAML's folded style instead.

- **[O]** `commands` → `exit` : This key holds a special purpose, 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:

```
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.

```
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:

```
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.

```
identifier: shell
```

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

Reserved names for rule arguments

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

- **[O] 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 list.

Return type

From version 6.0 on, the return value for `flag` is now transformed into a proper `List<String>` type instead of a plain,

generic Object reference, as seen in previous versions. The following rules apply:

- If a list is returned, it will be flattened and all values will be turned into strings.

```
[ 'a', 1, [ 2, 'b' ] ] → [ 'a', '1', '2', 'b' ]
```

- If a string is returned, a single list with only that string will be returned.

```
'hello world' → [ 'hello world' ]
```

- If another type is returned, it will be turned into string.

```
3.1415 → [ '3.1415' ]
```

Other return types than string or lists are not encouraged. However, if such types are used, they will be transformed into a list of strings, as previously seen. If you need interoperability of complex command code with older versions, use the following trick to get the value of previously non-list values:

```
isList(variable) ? variable[0] : variable
```

In this way, one can keep a compatibility layer for older versions. However, it is highly recommended to use the latest version of arara whenever possible.

```
flag: >
  @{
    isTrue(parameters.shell, '--shell-escape',
            '--no-shell-escape')
  }
```

There are two variables available in the `flag` context, described as follows. Note that there 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 [METHODS](#).

- ❖ `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.

```
check = parameters.contains("foo");
```

- ❖ `reference` : This variable holds the canonical, absolute path representation of the file name as a `File` object. This is useful if it is necessary to know the hierarchical structure of a project. Since the reference is a Java object, we can use methods available in the `File` class.

```
parent = reference.getParent();
```

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.

- **[O]** `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 and always expects a string value, as if it were provided by the user in the directive context.

No more evaluation and variables

In earlier versions, arara used to evaluate the `default` key and return a plain, generic `Object` reference, which was then forwarded directly to the corresponding command context. The workflow changed for version 6.0 on.

From now on, `default` always expects a string value, as if it were provided by the user in the directive context. No variables are available and no more evaluation is expected from this key. Consider the following example:

```
default: "@{ 1 == 1 }"
```

There is an `orb` tag expression in this string, which should resolve to `true` in previous versions of `arara`. However, from now on, it will not be evaluated at all and the literal string will be assigned to the `default` key.

The `default` key, whenever available and in the scenario in which the user does not provide an explicit value for the current argument in the directive context, is forwarded to the `flag` context for proper evaluation. Then the workflow proceeds as usual.

Return type

The `default` key, whenever available, returns a string to be evaluated in the corresponding `flag` context. However, if the target evaluation context does not exist (i.e, there is no corresponding `flag` key), the value is transformed into a list of strings and then forwarded directly to the `command` context. For instance:

```
- identifier: foo
  default: 'bar'
```

This scenario will directly forward [`'bar'`] (a list of strings containing the specified value as single element) as the value for the `foo` variable in the corresponding `command` context.

```
default: 'stable'
```

- **[O]** `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.

```
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.

```
arguments:
- identifier: shell
  flag: >
    @{
      assertTrue(parameters.shell,
        '--shell-escape',
        '--no-shell-escape')
    }
  required: false
  default: 'false'
```

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 [COMMAND LINE](#)).

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.

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 [INTRODUCTION](#) , 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 lists, except the ones with default values, mapped to lists containing single elements.

```
% 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.

```
% 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 [YAML](#).

```
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:

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

```
Processing "hello.tex" (size: 102 B, last modified: 12/28/2020  
10:28:00), please wait.
```

```
ERROR
```

```
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.

Sometimes, directives can span several columns of a line, particularly the ones with several parameters. 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:

```
% 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.

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` 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.

```
% 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.

```
% 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.

```
% 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.

```
% arara: pdflatex while missing('pdf') ||  
% 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.

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.

All directives, regardless of their type, are internally mapped alongside with the `reference` parameter, discussed earlier on, as a special variable 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). 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.

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

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

```
Directive: { identifier: pdflatex, parameters:
  {shell=yes, 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 `reference` value (resolved to a proper absolute, canonical path of the file name). See, for instance, the report of the directive extraction and normalization process performed by arara when inspecting `doc3.tex`, available in the log file.

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

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

```
Directive: { identifier: pdflatex, parameters:
  {reference=/home/paulo/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.

Orb tag expansion in parameter values

From version 6.0 on, arara is able to expand orb tags within a special `options` parameter in the directive context. For instance:

```
% arara: lualatex: {
% arara: --> options: [ '--output-directory=@{getSession().
% arara: -->                                     get("arg:builddir")}'
% arara: --> ]
% arara: --> }
```

This feature supports the following methods with their documented meanings, as seen in `METHODS`: ♦ `getBaseline`, ♦ `getSession` and ♦ `getOriginalReference`.

Keep in mind that this feature is disabled when arara is running in safe mode, as seen in [COMMAND LINE](#) .

Important changes in version 7

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.

Enabling header mode by default

The header mode (parse only the first commented lines of a file) is now enabled by default. You may return to the old behavior disabling header mode in the configuration file or using the `-w / --whole-file` command line flag.

Using an own I/O API instead of Java's File objects

In previous versions, arara's rules relied on Java's `File` API. That was bad for several reasons. Most importantly, we have switched to Java's `Path` API quite a while ago. Hence, what was used internally and what users accessed diverged.

With our general refactoring, there has been a change of strategies: we now avoid exposing any Java-specific API. The new API which you have access to when using the `toFile("some file.txt")` method exposes the following properties and methods:

- The properties `isAbsolute` , `fileName` , `fileSize` , `lastModified` , `parent` , `exists` , `isDirectory` , and `isRegularFile` do what their names indicate.
- The method `startsWith(File)` checks if the string representation of the one file is prefix of the other one's.
- `normalize()` turns a path into an absolute path and normalizes it.
- `resolve(String | File)` resolves a child.
- `resolveSibling(String | File)` resolves a sibling.
- `readLines()` reads the file's content into a `List<String>` .
- `readText()` reads the file's content into a continuous `String` .

- `writeText(String, append? = false)` writes the argument to the file; the optional argument allows appending instead of overwriting.

If you use the `toFile` method in your rules, you do not need to change anything. All the arara-internal methods like `exists(File)` have been adjusted to accept objects of the new format. In the end, the only need to change anything is in rules where you have accessed Java's `File` API yourself.

Add projects

arara now supports projects. See [PROJECTS](#) for further information on this new feature.

This section pretty much covered the basics of the changes to this version. Of course, it is highly advisable to make use of the new features available in arara 7.0 for achieving better results. If you need any help, please do not hesitate to contact us. See [INTRODUCTION](#) for more details on how to get help.

If you are upgrading you may also be interested in reading our [CHANGELOG](#) or the announcement blog post of this release in the [NEWS SECTION ON OUR WEBSITE](#) .

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.

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. We redesigned the interface in order to look more pleasant to the eye, after all, we work with TeX and friends:

```
  _ _ _ _ _ _ _ _ _ _  
 / _ \ | ' _ / _ \ | ' _ / _ \ |  
 | ( _ | | | | ( _ | | | | ( _ |  
 \ _ _ / _ \ | \ _ _ / _ \ |
```

```
Processing 'doc5.tex' (size: 285 B, last modified: 03/01/2020  
19:25:40), please wait.
```

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

```
Total: 1.14 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:

```
Processing 'doc5.tex' (size: 285 B, last modified: 03/01/2020
19:25:40), 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:

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

As previously mentioned in [IMPORTANT CONCEPTS](#) , 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:

```
(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.

```
Total: 1.14 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:

```
(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:

```
-----
(BibTeX) The BibTeX reference management software
-----
```

```
This is BibTeX, Version 0.99d (TeX Live 2020)
The top-level auxiliary file: doc5.aux
```

```
The style file: plain.bst
Database file #1: mybib.bib
```

```
----- SUCCESS
```

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.

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:

```
$ arara doc6.tex
```

From version 5.0 on, arara may receive more than one file as parameter. It will compile them sequentially (starting with the leftmost). The process fails on the first failure of these executions. For the files to be flawlessly compiled by TeX, they should be in the same working directory. If you process your files with other tools, this requirement could be lifted.

```
$ arara doc20.tex doc21.tex
```

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

```
Processing 'doc20.tex' (size: 28 B, last modified: 02/28/2020
07:15:02), please wait.
```

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

```
Processing 'doc21.tex' (size: 28 B, last modified: 02/28/2020
07:15:10), please wait.
```

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

```
Total: 1.20 seconds
```


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:

```
$ arara
Usage: arara [OPTIONS] file...
```

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

The cool TeX automation tool.

arara executes the TeX workflow you tell it to execute. Simply specify your needs within your TeX file and let arara do the work. These directives feature conditional execution and parameter expansion.

Options:

-l, --log	Generate a log output
-v, --verbose / -s, --silent	Print the command output
-n, --dry-run	Go through all the motions of running a command, but with no actual calls
-S, --safe-run	Run in safe mode and disable potentially harmful features. Make sure your projects uses only allowed features.
-w, --whole-file	Extract directives in the file, not only in the header
-p, --preamble TEXT	Set the file preamble based on the configuration file
-t, --timeout INT	Set the execution timeout (in milliseconds)
-L, --language TEXT	Set the application language
-m, --max-loops INT	Set the maximum number of loops (> 0)
-d, --working-directory PATH	Set the working directory for all tools
-P, --call-property VALUE	Pass parameters to the application to be used within the session.
--generate-completion [bash zsh fish]	Generate a completion script for arara. Add 'source <(arara --generate-completion <shell>)' to your shell's init file.
-V, --version	Show the version and exit
-h, --help	Show this message and exit

Arguments:

file The file(s) to evaluate and process

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. Addi-

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

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

% arara: pdftex
```

When running `arara` without this option, one directive will be extracted (line 1). However, with `-w` enabled, the directives in lines 1 and 5 will be extracted. This option can also be disabled by default in the configuration file.

- `-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. Refer to [LOGGING](#) for more details on the logging feature.
- `-L / --language` : 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 IETF BCP 47 norm, standardized nomenclature used to classify languages. For example, this is our tool speaking Dutch:

```
$ arara -L nl doc5.tex

-- -- -- -- --
/ _\ | '___/ _\ | '___/ _\ |
| (| | | | (| | | | (| |
\_,_|_| \_,_|_| \_,_|_|

Verwerken van 'doc5.tex' (grootte: 285 B, laatst gewijzigd:
03/01/2020 19:25:40), een ogenblik geduld.

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

```
(PDFLaTeX) PDFLaTeX engine ..... SUCCESVOL
```

```
Totaal: 1,07 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.

Available languages: `en` for English, `de` for German, `en-QN` for Broad Norfolk, `it` for Italian, `nl` for Dutch, and `pt-BR` for Portuguese (BR).

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. However, one can always override this setting by running the tool with an explicit `-L` option.

Invalid language codes

From version 6.0 on, if you pass an invalid language code, arara will now run in English and issue a log warning but not fail anymore. Failing due to the wrong language in the output was considered inappropriate.

- `-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. For instance, consider the following 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:

```
$ arara -m 2 doc8.tex
-- -- -- -- --
```

```

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

```

Processing 'doc8.tex' (size: 45 B, 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. 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:

```
$ arara -n doc5.tex
```

```

-- -- -- -- --
/ _\ | ' _/ _\ | ' _/ _\ |
| (| | | | (| | | | (| |
\_ _ , | | \_ _ , | | \_ _ , |

```

Processing "doc5.tex" (size: 360 B, last modified: 12/28/2020 13:03:32), please wait.

```
[DR] (PDFLaTeX) PDFLaTeX engine
```

```

-----
Author: Island of TeX
About to run: [pdflatex, doc5.tex] @ /home/paulo/Downloads

```

```
[DR] (BibTeX) The BibTeX reference management software
```

```

-----
Author: Island of TeX
About to run: [bibtex, doc5] @ /home/paulo/Downloads

```

```
[DR] (PDFLaTeX) PDFLaTeX engine
```

```

-----
Author: Island of TeX
About to run: [pdflatex, doc5.tex] @ /home/paulo/Downloads

```

```
[DR] (PDFLaTeX) PDFLaTeX engine
```

```

-----
Author: Island of TeX
About to run: [pdflatex, doc5.tex] @ /home/paulo/Downloads

```

```
Total: 0.18 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:

```
twopdftex: |
% arara: pdftex
% arara: pdftex
```

```
Hello world.
\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:

```
$ arara -p twopdftex doc9.tex
```

```
-----
/ _\ | ' _/ _\ | ' _/ _\ |
| C_ | | | C_ | | | C_ |
\_,_|_| \_,_|_| \_,_|
```

```
Processing 'doc9.tex' (size: 18 B, last modified: 05/29/2018
14:39:21), please wait.
```

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

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

```
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.

- `-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:

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

```
$ arara --timeout 3000 doc9.tex
```

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

```
Processing 'doc10.tex' (size: 63 B, 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.

- `d / --working-directory <path>` This option allows you to change the working directory to `<path>` . That is, the commands will run from a different directory than the directory you launched arara in. This is especially useful when calling a TeX engine as they resolve files against the working directory. For that reason, arara will also resolve each file you pass to it that has no absolute path against the working directory. The working directory is fixed for the whole call; passing multiple files to arara will resolve all of them against and execute all actions within that one working directory.
- `-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. 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.
 - `-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 [CONFIGURATION FILE](#)). It is important to note that this command line option has higher priority over the `--verbose` counterpart.
 - `-S / --safe-run` : This option enables the safe mode of arara, protecting the system by disallowing certain user actions. Currently, the following features are restricted:
 - File lookup will only perform explicit file resolution. Wildcard filters are disabled.
 - `❖ unsafelyExecuteSystemCommand` will raise an exception and abort the run. Keep in mind that rules are still allowed to construct arbitrary commands using `Command` objects, so this restriction only disallows arbitrary system commands that would not get logged and are thus invisible to the user.
 - The `--options` parameter does not expand orb tags in any directive.
 - `-P / --call-property <entry>` : This option forwards the provided `<entry>` , in the `key=value` format, to the session map. For instance, consider the following call:

```
$ arara -P foo=bar hello.tex
```

In a rule, you may now retrieve the value associated to the `foo` key, which is `bar` , set at runtime, by calling the following method in your code:

```
getSession().get('arg:foo')
```

This option may be called multiple times, as a means to provide as many data pairs as needed. Please refer to ❖ `getSession` in **METHODS** for more details.

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.

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: `tex` → `dtex` → `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. 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.

```
$ arara doc11.tex

-----
/ _\ | '___/ _\ | '___/ _\ |
| C_ | | | C_ | | | C_ |
\__,-|-| \__,-|-| \__,-|-|

Processing 'doc11.tex' (size: 34 B, last modified: 05/29/2018
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.

```
$ arara doc11

-----
/ _\ | '___/ _\ | '___/ _\ |
| C_ | | | C_ | | | C_ |
```



```

\_,_|_| \_,_|_| \_,_|_|

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

(PDFTeX) PDFTeX engine ..... SUCCESS

Total: 0.69 seconds

```

- Many shells complete file names that have multiple extensions in the same directory, so that they end with a period. We try to resolve against them as well!

```

$ arara doc11.

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

Processing 'doc11.tex' (size: 34 B, 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, and 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 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.

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: `.araraconfig.yaml` → `araraconfig.yaml` → `.arararc.yaml` → `arararc.yaml` .

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 later on.

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 [LOGGING](#).

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.
- **[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.

```
paths:
- '/home/paulo/rules'
- '/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 [MVEL](#)). 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.

```
paths:
- '♦{user.home}/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.

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

- ◊ `application.workingDirectory` : This variable, as the name implies, holds the value of the absolute, canonical path of the working directory associated with the current execution, as plain string.

```
paths:
- '@{application.workingDirectory}/rules'
```

Observe that the ◊ `user` and ◊ `application` variables actually holds maps. However, for didactic purposes, it is easier to use the property navigation feature of MVEL and consider the map references as three independent variables. You can use property navigation styles interchangeably.

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.

- **[O]** (*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.

```
language: nl
```

- **[O]** (*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.

```
loops: 30
```

- **[O]** (*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.

```
verbose: true
```

- **[O]** (*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.

```
logging: true
```

- **[O]** (*boolean*) `header` (default: `true`): 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 `--whole-file` command line option can override this setting if, and only if, this variable holds `true` as the value.

New default

Prior to version 7.0 the default has been `false`.

```
header: false
```

- **[O]** (*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.

Log paths

From version 6.0 on, the log file may now be specified as path *anywhere* on the file system. However, keep in mind that this behavior may be altered for future updates in safe mode.

```
logname: mylog
```

[O] (*string*) `dbname` (default: `arara`): This key modifies the default YAML database file name, according to the associated plain string value, plus the `yaml` 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.

```
dbname: mydb
```

[O] (*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 [METHODS](#). Note that this value is used by the underlying Java runtime environment, so a full qualified class name is expected.

```
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, 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, and `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.

[O] (*string map*) `preambles`: This key holds a string map containing predefined preambles for later use with the `--preamble` option (see [COMMAND LINE](#)). Note that each map key must be unique. Additionally, 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.

```
preambles:
  twopdfTeX: |
    % arara: pdfTeX
    % 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]** (*string*) `defaultPreamble` : This key allows to specify a preamble for arara to use even if there are no directives in the file nor preambles specified on the command-line. Preambles are resolved at execution time, which means that preambles from local configurations will take precedence over global preambles.

```
defaultPreamble: twopdfTeX
```

- **[O]** (*boolean*) `prependPreambleIfDirectivesGiven` : This key allows you to specify a boolean value indicating whether preambles should be applied to all files or only those without directives. It defaults to `true` to avoid breaking existing workflows.

```
prependPreambleIfDirectivesGiven: false
```

- **[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.

```
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 (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.

```
^\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.

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

For instance, let us reverse the default file name lookup order presented in **COMMAND LINE**. 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.

```
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:

```
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 provid-

ed 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.

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

```
filetypes:
- extension: c
  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.

```
filetypes:
- extension: tex
- extension: dtx
- extension: ltx
- extension: drv
- extension: ins
- extension: c
  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 the underlying regular expression in some scenarios (i.e, a lit-

eral 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 [LOGGING](#).

Logging

The logging feature of arara, as discussed earlier on, is activated through either the `--log` command line option or the equivalent key in the configuration file. 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:

```
% 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 deliberately removed from the log entries in order to declutter the output, and line breaks were included in order to easily spot each entry.

System information

The very first entry to appear in the log file is the current version of arara.

```
Welcome to arara 6.0.0!
```

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.

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

A privacy note

We 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. From version 6.0 on, if you pass an invalid language code (for instance, `-L foo` in the command line), arara will default to English and issue a log warning:

```
Language foo not available; defaulting to English.
```

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:

```
Processing 'doc12.tex' (size: 74 B, last modified:
06/02/2018 05:36:40), please wait.
```

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.

```
I found a potential pattern in line 1: pdftex
I found a potential pattern in line 2: clean: { extensions: [ log ] }
```

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*.

```
I found a potential directive: Directive: { identifier: pdftex,
parameters: {}, conditional: { NONE }, lines: [1] }
I found a potential directive: Directive: { identifier: clean,
parameters: {extensions=[log]}, conditional: { NONE }, lines: [2] }
```

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.

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.

```
All directives were validated. We are good to go.
```

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.

```
----- DIRECTIVES -----
Directive: { identifier: pdftex, parameters:
```

```
{reference=/home/paulo/Documents/doc12.tex},
conditional: { NONE }, lines: [1] }
Directive: { identifier: clean, parameters: {extensions=[log],
reference=/home/paulo/Documents/doc12.tex},
conditional: { NONE }, lines: [2] }
-----
```

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.

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:

```
I am ready to interpret rule 'pdftex'.
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 [IMPORTANT CONCEPTS](#). In this specific scenario, there is just one task associated with the `pdftex` rule. Both task name and system command are shown:

```
I am ready to interpret task 'PDFTeX engine' from rule 'PDFTeX'.
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.

```
----- 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/pdftex
```

```
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 pdf-tex 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:

```
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.

```
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.

```
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.

Log paths

From version 6.0 on, the log file may now be specified as path *anywhere* on the file system. However, keep in mind that this behavior may be altered for future updates in safe mode. Please refer to [CONFIGURATION](#) for more details.

Projects

Compatibility information

The project feature has been introduced in version 7 of arara and is still experimental. It will not vanish in the next versions but it might receive major changes based on user feedback. If you have certain use cases or ideas for projects, please open an issue in our [REPO](#).

Instead of passing one or multiple files with directives to the arara command-line you may pass one `.lua` file (the exact extension is important) with a project specification. arara will process the Lua script and then execute all the projects as specified in the Lua script.

Important remark: If arara encounters a single Lua file as its argument, it tries to parse it as a project. If it fails for any reason (e.g. malformed Lua), it will treat the Lua file as regular input file and scan it for directives which will fail if you did not define a Lua input file type. However, that allows to continue using Lua files as input files to arara.

Rationale

Projects have been identified as the sweet spot between arara's already make-like directive system and actual external specification of directives and dependencies. Though currently unable to express dependencies between single files, arara is able to handle dependencies between projects.

We have built projects with the following use cases in mind:

- You have dependencies between files and want to avoid abusing the `files` parameter of arara's directives.
- You want to compile multiple documents whenever you compile and want to avoid specifying all of them on the command-line.
- You want to use specific settings for some file but not for others (e.g. another pattern for the same file type).

However, the basic idea is to be explicit about what belongs to a project and what does not. In that sense, a declarative notion of a project spanning multiple files is what should be covered by this feature.

File structure

A project specification file is a Lua file which contains a script returning a list/table of project specifications or a project specification itself (if it is too technical at this point, stay with us, examples will follow). A single project minimally requires a list of files, so a minimal `project.lua` could look like

```
return {  
  files = { ["file.tex"] = { } }  
}
```

There are a number of optional parameters:

- `name` : The project's name which is used to identify the project e.g. when depending on it. Otherwise, it is used for display purposes on the command-line only. If not specified, there will be a placeholder name for untitled projects.
- `workingDirectory` : The working directory to execute the project in. Specifying a working directory with this option overrides any working directory given using the command-line option `-d / --working-directory` .
- `dependencies` : Other projects a project depends on. Make sure this project is actually specified. Project specification files with invalid dependencies will be treated as invalid.

A more complete example showcasing the list of project specifications approach would be

```
return {  
  {  
    name = "My awesome book",  
    workingDirectory = "book",  
    files = {  
      ["a.mp"] = {  
        directives = { "% arara: metapost" },  
        priority = 1  
      },  
      ["file.tex"] = { }  
    }  
  },  
  {  
    name = "My awesome book vol. 2",  
    workingDirectory = "bookv2",  
  }  
}
```

```
files = { ["file.tex"] = { } },  
dependencies = { "My awesome book" }  
}  
}
```

Note how the second project depends on the first one and the `a.mp` file has additional parameters. For files, there are the following additional options:

- `priority` : An integer value that specifies a priority within the project. The default priority is 0. If a file *A* has higher priority than file *B* it means that *A* will be executed before *B*.
- `directives` : Specify a list of directives that will be used instead of the directives specified in the file. They have to correspond to the file type's directives, i.e. they will be accepted if they would be accepted by the patterns of the file type.
- `fileType` : Specify the file type of the file. The same rules as for specifying a file type in the configuration file apply. The `extension` is mandatory. If the extension is not among the default set of extensions, a `pattern` is mandatory as well.

Apart from the return expression you are free to use Lua code to perform computations or other actions within the project specification file. Please note that for technical reasons arara's Lua interpreter only accepts Lua 5.2.

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 [MVEL](#), 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.

A file by any other name

As of version 7, we now avoid exposing any Java-specific API. When you see a mention to `File`, keep in mind that this is actually a wrapper to our new internal representation.

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. At last, an entry marked by **[E]** denotes that the corresponding method is available in the orb tag expansion within a special `options` parameter in the directive context.

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 [IMPORTANT CONCEPTS](#).

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

- **[C|E|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 [IMPORTANT CONCEPTS](#).

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

- **[C|R]** `currentFile(): File` This method returns the file reference, as a `File` object, for the current directive. It is important to observe that arara replicates the directive when the special `files` parameter is detected amongst the parameters, so each instance will have a different reference.

```
% 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.

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

- **[C|E|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. Also, this method will throw an exception if the provided reference is not a proper file.

```
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.

```
basename = getBasename('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.

```
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.

```
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.

```
% 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.

```
% 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.

```
% 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.

```
% 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 YAML database file named `arara.yaml` located in the same directory as the current file (the database name can be overridden in the configuration file. The method semantics (including the return values) is presented as follows.

FILE EXISTS?	ENTRY EXISTS?	HAS CHANGED?	DB ACTION	RESULT
Yes	Yes	Yes	Update	Yes
Yes	Yes	No	—	No
Yes	No	—	Insert	Yes
No	No	—	—	No
No	Yes	—	Remove	Yes

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.

```
% 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 YAML database file named `arara.yaml` located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in [CONFIGURATION](#)). The method semantics (including the return values) is presented as follows.

FILE EXISTS?	ENTRY EXISTS?	HAS CHANGED?	DB ACTION	RESULT
Yes	Yes	Yes	Update	Yes
Yes	Yes	No	—	No
Yes	No	—	Insert	Yes
No	No	—	—	No
No	Yes	—	Remove	Yes

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.

```
% 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 YAML database file named `arara.yaml` located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in [CONFIGURATION](#) . The method semantics (including the return values) is presented as follows.

FILE EXISTS?	ENTRY EXISTS?	HAS CHANGED?	DB ACTION	RESULT
Yes	Yes	Yes	Update	No
Yes	Yes	No	—	Yes
Yes	No	—	Insert	No
No	No	—	—	Yes
No	Yes	—	Remove	No

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.

```
% 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 YAML database file named `arara.yaml` located in the same directory as the current file (the database name can be overridden in the configuration file, as discussed in [CONFIGURATION](#) . The method semantics (including the return values) is presented as follows.

FILE EXISTS?	ENTRY EXISTS?	HAS CHANGED?	DB ACTION	RESULT
Yes	Yes	Yes	Update	No
Yes	Yes	No	—	Yes
Yes	No	—	Insert	No
No	No	—	—	Yes
No	Yes	—	Remove	No

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.

```
% 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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
files = listFilesByExtensions(toFile('/home/paulo/Documents'),  
    [ '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.

```
files = listFilesByExtensions('/home/paulo/Documents',  
    [ '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.

```
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.

```
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.

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:

- **True:** yes , true , 1 , on
- **False:** 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 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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

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.

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

- **[R] isEmpty(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.

```
result = isEmpty('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.

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

- **[R] isEmpty(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.

```
result = isEmpty('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.

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

- **[R] isEmpty(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.

```
result = isEmpty('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.

```
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.

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

- **[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.

```
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.

```
% arara: pdflatex while found(toFile('article.log'),  
% 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 ❖ current-

File 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.

```
% 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.

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.

```
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.

```
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.

```
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.

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

[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.

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

Cygwin paths

It is worth mentioning that Cygwin has its own path handling which is not reliable when (JVM) applications need to invoke system commands. For instance, the following invocation does not work:

```
% arara: pdflatex
```

Since `pdflatex` is a symbolic link to `pdftex` with the proper format, we encourage Cygwin users to rely on the following trick:

```
% arara: pdftex: { options: [ '--fmt=pdflatex' ] }
```

The same trick can be employed by other TeX engines with appropriate `--fmt` flags as well. We are still investigating this issue and looking for potential alternatives.

- **[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.

```
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.

```
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.

```
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.

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

- **[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.

```
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.

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.

```
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.

```
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.

```
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.

```
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 sub-type of the provided `Class class`. It is interesting to note that all methods presented in this section internally rely on ❖ `checkClass` for type checking.

```
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.

Classes and objects

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 property accessors:

- ❖ `first: A` : This property accessor, as the name implies, returns the first entry of the ordered pair, as an `A` object.

- ❖ second: B : This property accessor, as the name implies, returns the second entry of the ordered pair, as a B object.

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
- 1 : File does not exist
- 2 : File URL is incorrect
- 3 : Class was not found
- 4 : Access policy violation
- 5 : Instantiation exception

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<ClassLoading.ClassLoadingStatus, 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 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.

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

- **[C|R]** `loadClass(String ref, String name):`
`Pair<ClassLoading.ClassLoadingStatus, 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 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.

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

- **[C|R]** `loadObject(File file, String name):`
`Pair<ClassLoading.ClassLoadingStatus, 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.

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

- **[C|R]** `loadObject(String ref, String name):`
`Pair<ClassLoading.ClassLoadingStatus, 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.

```
result = loadObject('mymath.jar',  
                  '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.

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 [CONFIGURATION](#). 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 icon types, illustrated below, to be used with dialog boxes. Observe that each icon type 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.

- 1 : Error
- 2 : Information
- 3 : Attention
- 4 : Question
- 5 : Plain

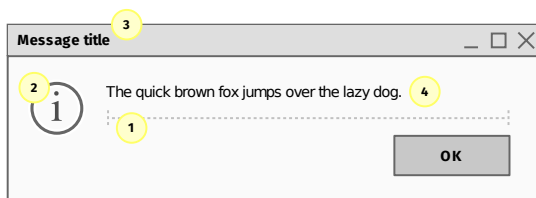
As good practice, make sure to provide descriptive messages to be placed in dialog boxes in order to ease and enhance the user experience. It is also highly advisable to always provide an associated icon, so avoid the plain option whenever possible.

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.

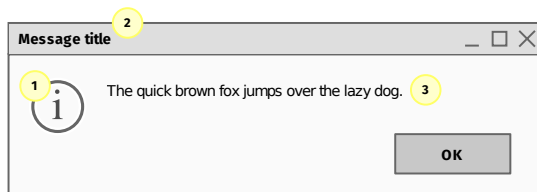
- **[R]** showMessage(int width, int icon, String title, String text): void



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.

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

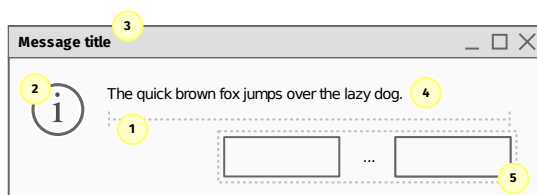
• [R] showMessage(int icon, String title, String text): void



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.

```
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 ob-

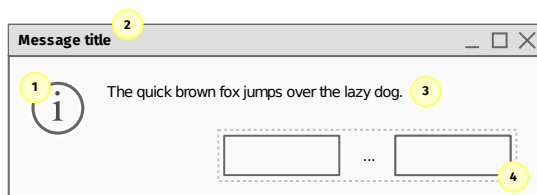
serve that arara temporarily interrupts the execution and waits for the dialog box disposal. Also note that the total time includes the idle period as well.

```
% 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.

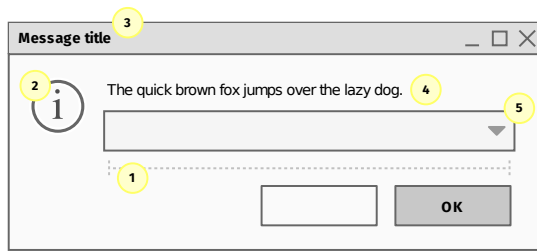
- **[C|R]** `showOptions(int icon, String title, String text, Object... options): int`



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.

```
% 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`



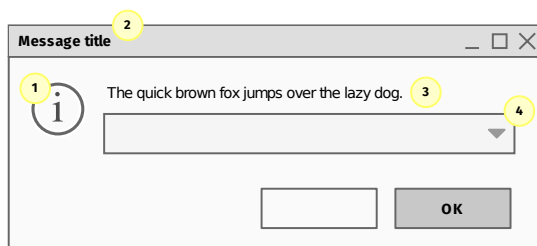
This method shows a dialog box according to the provided parameters, including options represented as an array of `Object` objects. This array is portrayed in the dialog box as a dropdown list. The first element from the array is automatically selected. The dialog box is disposed when the user either 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.

```
% 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`



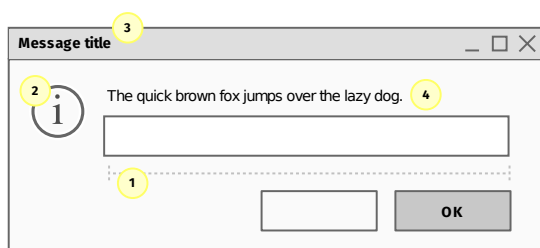
This method shows a dialog box according to the provided parameters, including options represented as an array of `Object` objects. This array is portrayed in the dialog box as a dropdown list. The first element from the array is automatically selected. The dialog box is disposed when the user either 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.

```
% 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.

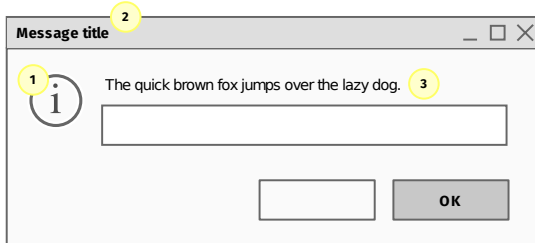
• **[C|R]** `showInput(int width, int icon, String title, String text):`
String



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.

```
% arara: pdflatex if showInput(250, 4, 'Important!',
% 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.

```
% 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.

Commands

`arara` features the `Command` object, a new approach for handling system commands based on a high level structure with explicit argument parsing.

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, given `['pdflatex', '--shell-escape', '--synctex=1', 'thesis.tex']`, we have:

• Head: `'pdflatex'`

- Tail: ['--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:

```
[ 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.

```
[ 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.

- **[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.

```
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.

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

- [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.

```
return getCommandWithWorkingDirectory(toFile('/home/paulo'),  
    [ '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.

```
return getCommandWithWorkingDirectory('/home/paulo',  
    [ '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.

```
return getCommandWithWorkingDirectory(toFile('/home/paulo'),  
    '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.


```
return getCommandWithWorkingDirectory('/home/paulo',  
    '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.

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 of a session are described as follows:

- ❖ `put(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.
- ❖ `contains(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.
- ❖ `get(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.

- **[C|E|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.

```
name = getSession().get('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.

```
options = 'not a list';  
if (!isList(options)) {  
    throwError('I was expecting a list.');
```

- **[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 configuration file. Note that the logical negation of such method indicates whether the tool is being executed in silent mode.

```
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 com-

mon patterns. This behaviour is expected and by design. However, be mindful that the search is case sensitive.

```
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.

```
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.

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

• **[R]** `getOrNull(List<String> list, int index): String` This method attempts to retrieve a list element based on an integer index. If the index is out of bounds, a `null` value is returned instead.

```
list = [ 'a', 'b', 'c' ];  
third = getOrNull(list, 2);
```

• **[R]** `getOrNull(List<String> list): String` This method attempts to retrieve the first element of the provided list. If the list is empty, a `null` value is returned instead.

```
list = [ 'a', 'b', 'c' ];  
first = getOrNull(list);
```

Flags and reserved storage in a session

From version 6.0 on, there are three reserved namespaces within a session. They are described as follows:

- **environment** This namespace is quite intuitive: arara will store the current state of the systems environment variables in its session. You may alter these values in the session storage but they will not be written back to the system configuration. To access an environment variable, you can use its usual name prefixed by `environment:` :

```
path = getSession().get('environment:PATH');
```

- **arara** This namespace provides flags that control the underlying behaviour of arara. Flags are used in rules and may be manipulated by the user. Be aware that every change in this namespace will result in the tool acting like you know what you did. Use this power with care. Currently, there is only one relevant flag: `arara:FILENAME:halt` . This will stop the currently run command execution on the file with the specified file name. The value of this map entry is the exit status you want arara to have.

```
getSession().put('arara:myfile.tex:halt', 42);
```

- **arg** This namespace acts as a bridge between contexts and the command line by providing access to key/value pairs defined at runtime by `--call-property` command line flags.

```
key = getSession().get('arg:key');
```

Please refer to [COMMAND LINE](#) for more details on the `--call-property` command line flag.

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]** before 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 **IMPORTANT CONCEPTS**, 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

This rule creates an animated gif file from the corresponding base name of the `❖ current-File` 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.

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

asymptote

This rule executes the `asy` command line program, referring to Asymptote, a powerful descriptive vector graphics language for technical drawings, inspired by Metapost but with an improved syntax. Please note that you will have to make the `.asy` extension known to arara in order to compile Asymptote files. Furthermore, it is advised to use this in your regular TeX document specifying the `files` parameter to include all graphics you want to compile for inclusion in your document.

- `color` : This option, as the name suggests, provides the underlying color model to be used in the current execution. Possible values are:
 - `bw` : This option value, as the name suggests, converts all colors to a black and white model.
 - `cmyk` : This option value converts the RGB (red, green and blue) color model to the CMYK (cyan, magenta, yellow and black) counterpart.
 - `rgb` : This option value converts the CMYK (cyan, magenta, yellow and black) color model to the RGB (red, green and blue) counterpart.

- `gray` : This option value, as the name suggests, converts all colors to a grayscale model.
- `engine` (default: `latex`) : This option, as the name indicates, sets the underlying TeX engine to be used for the current compilation. Make sure to take a look at the Asymptote manual for further details on this option. Possible values are:
 - `latex` : This value, as the name suggests, sets the underlying TeX engine to `latex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `pdflatex` : This value, as the name indicates, sets the underlying TeX engine to `pdflatex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `xelatex` : This value, as the name suggests, sets the underlying TeX engine to `xelatex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `lualatex` : This value, as the name indicates, sets the underlying TeX engine to `lualatex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `tex` : This value, as the name suggests, sets the underlying TeX engine to `tex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `pdftex` : This value, as the name indicates, sets the underlying TeX engine to `pdftex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `luatex` : This value, as the name suggests, sets the underlying TeX engine to `luatex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `context` : This value, as the name indicates, sets the underlying TeX engine to `context` for the current compilation. Note that the engine might play a major role in the generated code.
 - `none` : This value, as the name suggests, sets the underlying TeX engine to `none` for the current compilation. In this case, there will be no associated engine.
- `format` : This option, as the name suggests, converts each output file to a specified format. Make sure to take a look at the Asymptote manual for further details.
- `output` : This option, as the name suggests, sets an alternative output directory or file name. Make sure to take a look at the Asymptote manual for further 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.

```
% arara: asymptote: { files: [ mydrawing.asy ] }
```

authorindex

This rule calls the `authorindex` wrapper, a Perl script that processes auxiliary files generated by the package of the same name, producing author index files, with the `ain` extension.

- **[S] draft** : This option sets whether the script should write additional information to the produced file. For each author, the labels of all references and the page numbers where they are cited are included as comments. This detail may help if you manually edit the generated author index.
- **[S] index** : This option sets whether the script should create a file suitable for further processing with `makeindex` or the like. For example, you could use that to make a common author and subject index. Note the extension of the generated file still will be the default one.
- **[S] keep** : This option sets whether the script should retain the temporarily generated `bst` file after the run finishes. This information will give you a good starting point for advanced customization of the author index.
- **[S] print** : This option, as the name indicates, sets whether the script should print the result to standard output instead of writing it to the output file.
- **[S] recurse** : This option sets whether the script should automatically process auxiliary files produced by included files. This behaviour is enabled by default.
- `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.

```
% arara: authorindex: { draft: yes }
```

bib2gls

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.
- **[S]** `group` : 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`.
- **[S]** `interpret` : 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`.
- **[S]** `breakspace` : 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.
- **[S]** `trimfields` : This option sets whether `bib2gls` will trim leading and trailing spaces from field values. The default behaviour does not trim spaces.
- **[S]** `recordcount` : 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`.
- **[S]** `recordcountunit` : 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`.
- **[S]** `cite` : 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`.

- **[S] verbose** : 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.
- **[S] merge** : 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.
- **[S] uniscript** : 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.

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

biber

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.

- **[S] tool** : This option sets whether the bibliography processor should be executed in *tool mode*, intended for transformations and modifications of datasources. Since this mode is oriented towards a datasource rather than a document, make sure to use it alongside the `options` option.

• `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.

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

bibtex

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.

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

bibtex8

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.

```
% arara: bibtex8: { options: [ '--trace', '--huge' ] }
```

bibtexu

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.

```
% arara: bibtexu: { options: [ '--language', 'fr' ] }
```

clean

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 **COMMAND LINE**, 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.

```
% arara: clean: { extensions: [ aux, log ] }
```

context

This rule runs the `context` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document Format. Please refer to the user manual for further details.

- **[S] make** : This option, as the name indicates, sets whether the engine should create context formats. Please refer to the user manual for further details on this option.
- `ctx` : This option, as the name indicates, sets the `ctx` file for process management specification. Please refer to the user manual for further details on this option.
- `interface` : This option, as the name indicates, sets the specified user interface. Please refer to the user manual for further details on this option.
- **[S] autopdf** : This option, as the name indicates, closes the corresponding pdf file in viewer and then reopens it afterwards.
- `purge` : This option, as the name indicates, purges files according to the provided rule. Possible values are:
 - `partial` : This value, as the name suggests, purges files either or not after a proper run, defined through a pattern. Please refer to the user manual for further details.
 - `all` : This value, as the name suggests, purges all files either or not after a proper run, defined through a pattern. Please refer to the user manual for further details.
 - `result` : This value, as the name suggests, purges the resulting file before the actual run. Please refer to the user manual for further details.
- `modules` : This option, as the name indicates, sets a list of modules and styles to be loaded, normally part of the distribution.
- `environments` : This option, as the name indicates, sets a list of environment files (document styles) to load first.
- `mode` : This option, as the name indicates, enables modes according to the provided list (conditional processing in styles).
- `path` : This option, as the name indicates, consults the given paths in the provided list during a file lookup.

- `arguments` : This option, as the name indicates, sets variables that can be consulted during a run. Such variables are defined as key/value pairs.
- `trackers` : This option, as the name indicates, sets the list of tracker variables. Please refer to the user manual for further details.
- `directives` : This option, as the name indicates, sets the list of directive variables. Please refer to the user manual for further details.
- `silent` : This option, as the name indicates, disables the log categories based on the provided list. Please refer to the user manual for further details.
- `errors` : This option, as the name indicates, shows errors at the end of a run and quits when it the provided list. Please refer to the user manual for further details.
- **[S]** `synctex` : 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.
- `interaction` : This option alters the underlying engine behaviour. If this option is omitted, the engine 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.
- `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.

```
% arara: context
```

convert

This rule runs the `convert` program, a member of the ImageMagick suite of tools. This program is used to convert between image formats as well as resize an image, blur, crop, despeckle, dither, draw on, flip, join, resample, and more.

- `program` (default: `convert`): This option specifies the command utility path as a means to avoid potential clashes with underlying operating system commands.
- **[R]** `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.

```
% arara: convert: { options: [ 'photo1.jpg', '--resize',  
% arara: --> '50%', 'photo2.jpg' ] }
```

copy

This rule copies the `File` reference to the provided `target` using the underlying operating system copy operation. The target is *always* overwritten. However, be mindful that an error will be thrown if you try to overwrite the file referenced in ❖ `getOriginalReference`.

- **[R]** `target`: This option, as the name implies, specifies the target for the copy operation. Keep in mind that this option is required.

```
% arara: copy: { target: 'backup/thesis.tex' }
```


csplain

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.
- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.
- **[S]** `draft` : 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.

```
% arara: csplain: { interaction: batchmode, shell: yes }
```

datatooltk

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.

- **[R] output** : 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.
- **[S]** `sortcase` : 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.
- **[S]** `shuffle` : This option sets whether the database will be properly shuffled. Shuffle is always performed after sort, regardless of the option order.
- **[S]** `csvheader` : This option sets whether the `csv` file has a header row. The spreadsheet import functions also use this setting.
- **[S]** `debug` : This option, as the name indicates, sets whether the debug mode of `datatooltk` is activated. The debug mode is disabled by default.
- **[S]** `owneronly` : 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.
- **[S]** `maptex` : 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.

```
% arara: datatooltk: {
% arara: --> output: books.dbtex,
% arara: --> csv: booklist.csv }
```

detex

This rule runs `detex`, a command line application that, as the name indicates, reads the provided ❖ `currentFile` reference, removes all comments and TeX control sequences and writes the remainder to the standard output or file.

- **[S]** `references` : This option defines whether the tool should echo values from page counters, references and citations.
- **[S]** `follow` : This option defines whether the tool should follow files referenced through typical input mechanisms.
- **[S]** `math` : This option defines whether the tool should replace math with nouns and verbs in order to preserve grammar.
- **[S]** `spaces` : This option, as the name indicates, define whether the tool should replace control sequences with spaces.
- **[S]** `words` : This option, as the name indicates, define whether the tool should output only a list of words.
- `environments` : This option takes a list of environments to be ignored by the tool during the text transformation.
- `mode` : This option, as the name indicates, defines the operation mode for the tool. Possible values are:
 - `latex` : This value, as the name suggests, enables and forces the LaTeX mode for the tool.

- `tex` : This value, as the name suggests, enables and forces the plain TeX mode for the tool.
- `output` : This value, as the name indicates, sets the output file in which the contents will be redirected.

```
% arara: detex: { output: thesis.txt }
```

dvipdfm

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.

```
% arara: dvipdfm: { output: thesis }
```

dvipdfmx

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.

```
% arara: dvipdfmx: { options: [ '-K', '40' ] }
```

dvips

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.

```
% arara: dvips: { output: thesis }
```

dvipspdf

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.

```
% arara: dvipspdf: { output: article }
```

dvisvgm

This rule runs `dvisvgm` in order to obtain a corresponding `svg` file, a vector graphics format based on XML, from the initial 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.

- `entry` : This option sets the extension to be used for the initial reference (i.e, the current file name) as input to the command line tool. The following values are available for this option:
 - `dvi` : This value sets the extension to be used for the initial reference as a device independent format. This is the default value when no value is provided.
 - `xdv` : This value sets the extension to be used for the initial reference as an extended device independent format.
 - `eps` : This value sets the extension to be used for the initial reference as an encapsulated PostScript graphics format.
 - `pdf` : This value sets the extension to be used for the initial reference as a Portable Document Format.
- `pages` : This value, as the name implies, takes a list of integers indicating the pages to be processed by the command line tool.
- `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.

```
% arara: dvisvgm
```

etex

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.
- **[S]** `shell` : 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.

```
% arara: etex: {  
% arara: --> shell: yes,  
% arara: --> interaction: batchmode  
% arara: --> }
```

fig2dev

This rule runs `fig2dev` , a command line application that translates `fig` code in the corresponding base name of the ❖ `currentFile` reference (i.e, the name without the associated

extension) as a string concatenated with the `fig` suffix into the specified graphics language and puts them in the specified output file.

- **[R] language** : This option, as the name indicates, sets the output graphics language for proper transformation. Possible values are listed as follows (you can infer their types as well, based on the values): `box` , `cgm` , `epic` , `eepic` , `eepicemu` , `emf` , `eps` , `gif` , `ibmgl` , `jpeg` , `latex` , `map` , `mf` , `mp` , `mmp` , `pcx` , `pdf` , `pdftex` , `pdftex_t` , `pic` , `pictex` , `png` , `ppm` , `ps` , `pstex` , `pstex_t` , `ptk` , `shape` , `sld` , `svg` , `textyl` , `tiff` , `tk` , `tpic` , `xbm` , and `xpm` .
- **[R] output** : This option, as the name indicates, sets the output file which will contain the translated code based on the provided file and language.
- **magnification** : This option, as the name suggests, sets the magnification level at which the figure is rendered.
- **font** : This option sets the default font used for text objects to the provided value. Keep in mind that the format of this option depends on the graphics language in use.
- **size** : This option, as the name suggests, set the default font size (in points) for text objects to the provided 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.

```
% arara: fig2dev: { language: mp, output: drawing.mp }
```

frontespizio

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).
- **[S]** `shell` : 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.

```
% arara: frontespizio: { engine: xelatex,
% arara: --> shell: yes, interaction: nonstopmode }
```

ghostscript

This rule runs `ghostscript` , an interpreter for PostScript and Portable Document Format files, according to the provided parameters.

- `program` (default: `gs`): This option specifies the command utility path as a means to avoid potential clashes with underlying operating system commands or specific Windows naming schemes.
- **[R]** `options` : This option, as the name indicates, takes a list of raw command line options and appends it to the actual `ghostscript` call. An error is thrown if any data structure other than a proper list is provided as the value. This option is required.
- `device` : This option specifies which output device the tool should use. If this option is not given, the default device (usually a display device) is used.
- `output` : This option, as the name indicates, specifies a file in which the tool should send the output. Please refer to the documentation for more details.

```
% arara: ghostscript: { options: [ '-dCompatibilityLevel=1.4',  
% arara: --> '-dPDFSETTINGS=/printer', '-dNOPAUSE', '-dQUIET',  
% arara: --> '-dBATCH', 'input.pdf' ],  
% arara: --> output: output.pdf,  
% arara: --> device: pdfwrite }
```

gnuplot

This rule runs `gnuplot` , a command-driven plotting program that can generate plots of functions, data and data fits. The program also provides scripting capabilities, looping, functions, text processing, variables, macros, arbitrary pre-processing of input data (usually across columns), as well as the ability to perform non-linear multi-dimensional multi-set weighted data fitting.

- **[S]** `persist` : This option, as the name implies, sets whether the program should let plot windows survive after the main execution exits.
- **[S]** `default` : this option, as the name suggests, sets whether the program should read the default settings from either `gnuplotrc` or `~/.gnuplot` on entry.
- `commands` : This option, as the name implies, executes the requested commands before loading the next input file. Please refer to the user manual for further details.

[R] `input` : This required option, as the name indicates, sets the list of input file names to be processed by the program. An error is thrown if any data structure other than a proper list is provided as the value.

```
% arara: gnuplot: { input: [ myplot.gnuplot ], default: yes }
```

halt

This rule, as the name suggests, sets a `halt` flag, which stops the current interpretation workflow, such that subsequent directives are ignored. This rule contains no associated options. Please refer to [METHODS](#) for more information on flags.

```
% arara: halt
```

indent

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.

- **[S]** `silent` : This option, as the name indicates, sets whether the script will operate in silent mode, in which no output is given to the terminal.
- **[S]** `overwrite` : 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.

- **[S] screenlog** : 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.
- **[S] modifylinebreaks** : 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.
- `check` : This option, as the name indicates, checks whether the text after indentation matches that given in the original file. Two possible values are available:
 - `standard` : This value, as the name indicates, refers to the standard behaviour. Please refer to the documentation for further details.
 - `verbose` : This value, as the name indicates, refers to including more details to the check. The tool will print the result in the standard output, as well as in its own log file. Please refer to the documentation for further details.
- `lines` : This option instructs the tool to operate only on specific line ranges within the file being inspected. Please refer to the documentation for further details.

```
% arara: indent: { overwrite: yes }
```

knitr

This rule calls the `knitr` package, a transparent engine for dynamic report generation with R. It takes an `.Rnw` file as input, extracts the R code in it according to a list of patterns, evaluates the code and writes the output in another file. It can also tangle R source code from the input document.

- `output` (default: `NULL`): This option sets the output file. when absent, `knitr` will try to guess a default, which will be under the current working directory.
- **[S]** `tangle` : This option sets whether to tangle the R code from the input file. Note that, when used, this option requires `output` to be specified as well, otherwise an error is thrown.
- **[S]** `quiet` : This option, as the name indicates, sets whether the tool should suppress both progress bar and messages.
- `envir` (default: `parent.frame()`): This option sets the environment in which code chunks are to be evaluated. Please refer to the documentation for further details.
- `encoding` (default: `getOption("encoding")`): This option, as the name indicates, sets the encoding of the input file. Please refer to the documentation for further details.

```
% arara: knitr: { quiet: yes }
```

latex

This rule runs the `latex` TeX engine on the provided `currentFile` reference, generating a corresponding file in a device independent format.

- `branch` (default: `stable`): This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - `stable`: This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - `developer`: For experienced users, this value enables the experimental, `developer` engine format branch.
- `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.
- **[S]** `shell`: This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex`: 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.
- **[S]** `draft`: 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.

```
% arara: latex: { interaction: scrollmode, draft: yes }
```

latexmk

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 Format. 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 Format.
- `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.
 - `latex-dev` : This value, as the name indicates, sets the underlying TeX engine of the script to `latex-dev` (the development branch) 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.
 - `pdflatex-dev` : This value, as the name indicates, sets the underlying TeX engine of the script to `pdflatex-dev` (the development branch) 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.

- `xelatex-dev` : This value, as the name indicates, sets the underlying TeX engine of the script to `xelatex-dev` (the development branch) 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.
- `lualatex-dev` : This value, as the name indicates, sets the underlying TeX engine of the script to `lualatex-dev` (the development branch) 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.

```
% arara: latexmk: { engine: pdflatex }
```

l_lmk

This rule runs `llmk`, a command line tool specific for building LaTeX documents. The tool's aim is to provide a simple way to specify a workflow of processing documents and encourage people to always explicitly show the right workflow for each document.

- `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 Format. 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 Format.
- `debug` : This option activates the specified debug category, so debugging messages related to the activated category will be shown. Please refer to the documentation for more details.

- **[S] dry** : This option sets whether the tool should display a list of commands to be executed without actually invoking them.
- **mode** : This option sets the verbosity level of messages to be displayed during a run. Three possible values are available:
 - **quiet** : This value, as the name indicates, suppresses most of the messages from the program during execution.
 - **silent** : This value, as the name indicates, silences messages from invoked programs by redirecting both standard output and standard error streams to the null device.
 - **verbose** : This value, as the name indicates, displays additional information such as invoked commands with options and arguments by the program.

```
% arara: llmk: { mode: verbose }
```

ltx2any

This rule runs `ltx2any`, a command line tool written in Ruby that acts as a LaTeX build wrapper, on the provided `currentFile` reference.

- **[S] clean** : This option, as the name indicates, sets whether all intermediate results generated during the compilation to be deleted.
- **engine** : This option, as the name indicates, sets the engine to be using during the current execution.
- **parameters** : This option, as the name indicates, takes a list of parameters to be passed to the engine. An error is thrown if any data structure other than a proper list is provided as the value.
- **tikzimages** : This option takes a list of externalised *TikZ* images to rebuild. An error is thrown if any data structure other than a proper list is provided as the value.
- **jobname** : This option, as the name indicates, sets the job name to be used in the resulting file.
- **logname** : This option, as the name indicates, sets the log file name to be used during the current execution.

- `logformat` : This option, as the name indicates, sets the log format to be used during the current execution. Three possible values are available:
 - `raw` : This value, as the name indicates, sets the log format to be raw, i.e, as generated by the underlying engines.
 - `markdown` : This value, as the name indicates, sets the log format to be displayed in Markdown.
 - `pdf` : This value, as the name indicates, sets the log format to be displayed in the Portable Document Format.
- `loglevel` : This option, as the name indicates, sets the log level to be used during the current execution. Three possible values are available:
 - `error` : This value, as the name indicates, sets the base log level to report errors only. No other information is appended.
 - `warning` : This value, as the name indicates, sets the base log level to report warnings and errors. No other information is appended.
 - `info` : This value, as the name indicates, set the base log level to report all information available, regardless of message categories.
- `frequency` : This option, as the name indicates, sets how often the engine runs. Values smaller than one will cause it to run until the resulting file no longer changes.
- `directory` : This option, as the name indicates, sets the directory to hold intermediate files during the compilation.
- **[S]** `synctex` : 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.

```
% arara: ltx2any: { synctex: yes }
```

luahtex

This rule runs the `luahtex` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document 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.
- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.
- **[S]** `draft` : 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.

```
% arara: luahtex: { interaction: batchmode,  
% arara: --> shell: yes, draft: yes }
```

lualatex

This rule runs the new `lualatex` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document Format, as expected.

- `branch` (default: `stable`): This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - `stable`: This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - `developer`: For experienced users, this value enables the experimental, developer engine format branch.
- `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.
- **[S]** `shell`: This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex`: 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.
- **[S]** `draft`: 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.

```
% arara: luatex: { interaction: errorstopmode,  
% arara: --> synctex: yes }
```

luatex

This rule runs the `luatex` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document 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.
- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.
- **[S]** `draft` : 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.

```
% arara: luatex: { interaction: batchmode,  
% arara: --> shell: yes, draft: yes }
```

make

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.

```
% arara: make: { targets: [ compile, package ] }
```

makeglossaries

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.

```
% arara: makeglossaries if found('aux', '@istfilename')
```

makeglossarieslite

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.

```
% arara: makeglossarieslite if found('aux', '@istfilename')
```

makeindex

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.
- **[S]** `german` : 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.

```
% arara: makeindex: { style: book.ist }
```

metapost

This rule runs `metapost`, a tool to compile the Metapost graphics programming language. Please note that you will have to make the `.mp` extension known to arara in order to compile Metapost files. Furthermore, it is advised to use this in your regular TeX document specifying the `files` parameter to include all graphics you want to compile for inclusion in your document.

- `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.

- `numbersystem` : This option sets the number system Metapost will use for calculations.
 - `scaled` : In this mode, 32-bit fixed-point arithmetics is used.
 - `double` : In this mode, IEEE floating-point arithmetics with 64 bits is used.
 - `binary` : This mode is similar to `double` but without a fixed-length mantissa.
 - `decimal` : In this mode, arbitrary precision arithmetics is used and numbers are internally represented in base 10.
- `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.

```
% arara: metapost: { files: [ graphics.mp ] }
```

move

This rule moves the `File` reference to the provided target using the underlying operating system move operation. The target is *always* overwritten. However, be mindful that an error will be thrown if you try to move or overwrite the file referenced in ❖ `getOriginalReference` .

- **[R]** `target` : This option, as the name implies, specifies the target for the move operation. Keep in mind that this option is required.

```
% arara: move: { files: [ 'thesis.pdf' ],
% arara: --> target: 'backup/thesis.pdf' }
```

nomencℓ

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` ref-

erence (i.e, the name without the associated extension) as a string concatenated with the `nls` suffix and a special style file in order to generate the nomenclature list as a special `nls` file.

- `style` (default: `nomenc1.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 `nomenc1` 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.

```
% arara: nomenc1
```

pbibtex

This rule runs the `pbibtex` 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.

```
% arara: pbibtex
% arara: --> if exists(toFile('references.bib'))
```

pdfcrop

This rule runs `pdfcrop`, a command line utility to calculate and remove empty margins, on corresponding base name of the ❖ `currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `pdf` suffix.

- `output`: This option, as the name indicates, sets the output file. When omitted, the tool uses the input base name with the `-crop.pdf` suffix.
- **[S]** `verbose`: This option, as the name indicates, sets whether the command line tool will be executed in verbose mode.

- **[S] debug** : This option, as the name indicates, sets whether the command line tool will be executed in debug mode.
- **engine** : This option, as the name indicates, sets the underlying TeX engine to be used during the run. Three possible values are available:
 - **pdftex** : This value, as the name indicates, sets `pdftex` as the underlying TeX engine to be used during the run.
 - **xetex** : This value, as the name indicates, sets `xetex` as the underlying TeX engine to be used during the run.
 - **luatex** : This value, as the name indicates, sets `luatex` as the underlying TeX engine to be used during the run.
- **margins** : This option, as the name indicates, takes a list of four elements denoting left, top, right and bottom margins, respectively. An error will be thrown if no list is provided or if the list does not contain exactly four elements.
- **[S] clip** : This option, as the name indicates, sets whether the command line tool should include clipping support, if margins are set.
- **[S] hires** : This option, as the name indicates, sets whether the command line tool should use a high resolution bounding box feature.
- **[S] ini** : This option, as the name indicates, sets whether the `initex` variant of the underlying TeX engine is used.
- **[S] restricted** : This option, as the name indicates, sets whether the command line tool should run on restricted mode.
- **papersize** : This option, as the name indicates, sets the paper size. According to the documentation, this option should only be used with older versions of `ghostscript` .
- **resolution** : This option, as the name indicates, sets the resolution by forwarding the value to the underlying `ghostscript` call.
- **bbox** : This option, as the name indicates, takes a list of four elements denoting left, bottom, right and top margins, respectively, to override bounding box values found by `ghostscript` . An error will be thrown if no list is provided or if the list does not contain exactly four elements.
- **[S] uncompress** : This option, as the name indicates, sets whether the tool should generate an uncompressed Portable Document Format file, useful for debugging.
- **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.

```
% arara: pdfcrop
```

pdfcsplain

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.
- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.
- **[S]** `draft` : 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.

```
% arara: pdfcsplain: { shell: yes, synctex: yes }
```

pdf_lat_ex

This rule runs the pdf_lat_ex TeX engine on the provided ♦ `currentFile` reference, generating a corresponding file in the Portable Document Format, as expected.

- `branch` (default: `stable`): This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - `stable`: This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - `developer`: For experienced users, this value enables the experimental, developer engine format branch.
- `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.
- **[S]** `shell`: This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex`: 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.
- **[S]** `draft`: 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.

```
% arara: pdflatex: { interaction: batchmode }  
% arara: --> if missing('pdf') || changed('tex')
```

pdftex

This rule runs the `pdftex` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document 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.
- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.
- **[S]** `draft` : 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.

```
% arara: pdftex: { draft: yes }
```

pdftk

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.

```
% arara: pdftk: { options: [ burst ] }
```

perltex

This rule runs `perltex`, a wrapper that enables a symbiosis between Perl, a popular general purpose programming language, and a TeX engine, on the provided `currentFile` reference.

- `engine` (default: `latex`) : This option, as the name indicates, sets the underlying TeX engine to be used for the current compilation. Make sure to take a look at the manual for further details on this option. Possible values are:
 - `latex` : This value, as the name suggests, sets the underlying TeX engine to `latex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `pdflatex` : This value, as the name indicates, sets the underlying TeX engine to `pdflatex` for the current compilation. Note that the engine might play a major role in the generated code.
 - `xelatex` : This value, as the name suggests, sets the underlying TeX engine to `xelatex` for the current compilation. Note that the engine might play a major role in the generated code.

- `luatex` : This value, as the name indicates, sets the underlying TeX engine to `luatex` for the current compilation. Note that the engine might play a major role in the generated code.
- `tex` : This value, as the name suggests, sets the underlying TeX engine to `tex` for the current compilation. Note that the engine might play a major role in the generated code.
- `pdftex` : This value, as the name indicates, sets the underlying TeX engine to `pdftex` for the current compilation. Note that the engine might play a major role in the generated code.
- `luatex` : This value, as the name suggests, sets the underlying TeX engine to `luatex` for the current compilation. Note that the engine might play a major role in the generated code.
- `context` : This value, as the name indicates, sets the underlying TeX engine to `context` for the current compilation. Note that the engine might play a major role in the generated code.
- **[S]** `safe` : This option sets whether the wrapper should enable sandboxing. When explicitly disabled, the wrapper might execute any arbitrary Perl code, including that which can harm files.
- `permit` : This option takes a list of values in which indicate particular Perl operations to be performed, enabling finer-grained control over the wrapper sandbox.
- **[S]** `standalone` : This option generates a specific style file to make the document suitable for distribution to users who do not have the wrapper installed. Please refer to the manual for further details on this option.
- `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.

- **[S] shell** : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S] synctex** : 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.

```
% arara: perl\textex: { safe: no, standalone: yes }
```

platex

This rule runs the `platex` TeX engine on the provided `currentFile` reference, generating a corresponding file in a device independent format.

- **branch** (default: `stable`): This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - **stable** : This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - **developer** : For experienced users, this value enables the experimental, developer engine format branch.
- **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.

- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.

```
% arara: platex: { interaction: scrollmode, shell: yes }
```

ps2pdf

This rule runs `ps2pdf` , a tool that converts PostScript to Portable Document Format, 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.

```
% arara: ps2pdf: { output: article }
```

pythontex

This rule runs `pythontex`, a wrapper that provides access to Python from within typical TeX documents, on the provided `currentFile` reference. Make sure to take a look at the documentation for further details.

- `encoding` : This option sets the encoding of the underlying TeX document and all related files. If an encoding is not specified, Unicode is assumed.
- **[S]** `errorcode` : This option determines whether an exit code of 1 is returned if there were errors. On by default, but can be turned off since it is undesirable when working with some editors.
- **[S]** `runall` : This option sets whether all code to be executed, regardless of modification. It is useful when code has not been modified, but a dependency such as a library or external data has changed.
- `rerun` : This option, as the name indicates, sets the underlying threshold for reexecuting code. By default, the wrapper will rerun code that has been modified or that produced errors on the last run. Possible values are:
 - `never` : When this value is used, the wrapper never executes code. In this scenario, a warning is issued if there is modified code. Please refer to the documentation for further details.
 - `modified` : When this value is used, as the name indicates, the wrapper only executes code that has been modified or that has modified dependencies.
 - `errors` : When this value is used, as the name indicates, the wrapper executes code that has been modified as well as code that produced errors on the last run.
 - `warnings` : When this value is used, as the name indicates, the wrapper executes code that has been modified as well as code that produced errors or warnings on the last run.
 - `always` : When this value is used, as the name indicates, the wrapper executes all code, regardless of modification or errors and warnings. It is useful when code has not been modified, but a dependency such as a library or external data has changed.
- **[S]** `hashdependencies` : This option, as the name suggests, determines whether dependencies are checked for changes via their hashes or modification times.

- `jobs` : This option, as the name suggests, takes an integer value denoting the maximum number of concurrent processes. By default, the wrapper relies on the number of CPUs in the system.
- **[S]** `verbose` : This option sets whether the wrapper should be executed in verbose mode, providing more output information, including a list of all processes that are launched.
- `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.

```
% arara: pythontex: { jobs: 2, verbose: yes }
```

qpdf

This rule runs `qpdf` , a command line application that does structural, content-preserving transformations of Portable Document Format files, as well as providing capabilities to developers.

- **[R]** `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.

```
% arara: qpdf: { options: [ '--linearize', 'input.pdf',  
% arara: --> 'output.pdf' ] }
```

sage

This rule runs `sage` , a free open source mathematics software system, on the corresponding base name of the `currentFile` reference (i.e, the name without the associated extension) as a string concatenated with the `sage` extension (which can be overridden).

- `program` (default: `sage`): This option, as the name indicates, sets the program name. If the tool is not directly available in your system path, make sure to use the full path to the installed `sage` binary.

- `extension` (default: `sage`): This option, as the name indicates, sets the default extension to the input file to be processed by `sage`. Three possible values are available:
 - `sage`: This value, as the name indicates, sets the extension to refer to the Sage format, the default one used by the software system.
 - `py`: This value, as the name indicates, sets the extension to refer to a typical Python source code.
 - `spyx`: This value, as the name indicates, sets the extension to refer to the SPYX format, a specific Sage compiled source code.
- `command`: This option, as the name indicates, forwards the provided value to the command line utility to be evaluated as a Sage code.
- **[S]** `dotsage`: This option, as the name indicates, sets whether the tool should consider using the `.sage` directory in the user home directory or a temporary one.
- `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.

```
% arara: sage
```

sketch

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 [CONFIGURATION](#).

- `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.

```
% arara: sketch
```

songidx

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.

- **[R] input** : 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.

```
% arara: songidx: { input: songs }
```

spix

This rule runs `spix`, an interesting command line TeX automation tool written in Python, on the provided `currentFile` reference.

- **dry** : This option sets whether the tool should display a list of commands to be executed without actually invoking them.
- **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.

```
% arara: spix
```

tex

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.
- **[S]** `shell` : 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.

```
% arara: tex: { shell: yes }
```


texcount

This rule runs `texcount`, a Perl script designed to count words in TeX and LaTeX files, ignoring macros, tables, formulae and more on the provided `currentFile` reference. The script is highly configurable, so make sure to check the manual for further information.

- `rules` : This option, as the name suggests, sets the rules which dictate how the script should work regarding word counting and option handling. Possible values are:
 - `relaxed` : This value, as the name indicates, sets a relaxed set of rules, allowing more general cases to be counted as either words and macros.
 - `restricted` : This value, as the name indicates, sets a more restricted set of rules for word counting and option handling.
- `verbosity` : This option, as the name suggests, sets the verbosity level of the script according to the provided integer value. Possible values are:
 - `0` : This value sets the lowest verbosity level of all, such that the script does not present parsing details.
 - `1` : This value raises the details a bit and sets the verbosity level to include parsed words and marked formulae.
 - `2` : This value adds more details from the previous verbosity level by including ignored text as well.
 - `3` : This value adds more details from the previous verbosity level by including comments and options.
 - `4` : This value sets the highest verbosity level of all, such that the script includes parsed worded, marked formulae, ignored text, comments, options and internal states.
- `[S] strict` : This option sets whether the tool should enable strict mode, so certain groups for which rules are not defined raise warnings.
- `[S] html` : This option, as the name suggests, defines whether the tool should output the report in the HTML format.
- `[S] total` : This option, as the name suggests, defines whether the tool should provide a total sum instead of partial sums (per file).
- `[S] unicode` : This option, as the name indicates, defines whether the tool should select Unicode as encoding for both input and output.

- `output` : This option, as the name suggests, sets the output file name in which the report will be written.
- `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.

```
% arara: texcount: { output: report.txt }
```

texindy

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.

- **[S]** `quiet` : 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.

```
% arara: texindy: { markup: latex }
```

tikzmake

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.

- **[S]** `force` : 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.

```
% arara: tikzmake: { force: yes, jobs: 2 }
```

upbibtex

This rule runs the `upbibtex` 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.

```
% arara: pbibtex
% arara: --> if exists(toFile('references.bib'))
```

uplatex

This rule runs the `uplatex` TeX engine on the provided ❖ `currentFile` reference, generating a corresponding file in a device independent format.

- `branch` (default: `stable`): This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - `stable` : This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - `developer` : For experienced users, this value enables the experimental, developer engine format branch.
- `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.
- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.

```
% arara: uplatex: { interaction: scrollmode, shell: yes }
```

uptex

This rule runs the `uptex` 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.

- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.

```
% arara: uptex
```

xdvipdfmx

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 a certain suffix, generating a Portable Document Format `pdf` file.

- `entry` : This option sets the extension to be used for the initial reference (i.e, the current file name) as input to the command line tool. The following values are available for this option:
 - `dvi` : This value sets the extension to be used for the initial reference as a device independent format. This is the default value when no value is provided.
 - `xdv` : This value sets the extension to be used for the initial reference as an extended device independent format.
- `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.

```
% arara: xdvipdfmx: { output: thesis }
```

xelatex

This rule runs the new `xelatex` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document Format, as expected.

- `branch` (default: `stable`): This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - `stable`: This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - `developer`: For experienced users, this value enables the experimental, developer engine format branch.
 - `unsafe`: This value enables the `unsafe` engine format branch. According to the documentation, at all costs, avoid using this, or any, unsafe invocation with documents off the net or that are otherwise untrusted in any way.
- `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.
- **[S]** `shell`: This option sets whether the possibility of running underlying system commands from within TeX is activated.

- **[S] synctex** : 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.

```
% arara: xelatex: { shell: yes, synctex: yes }
```

xetex

This rule runs the `xetex` TeX engine on the provided `currentFile` reference, generating a corresponding file in the Portable Document Format, as expected.

- **branch** (default: `stable`) : This option allows branching formats for the current engine, mainly focused on package development. Users of current TeX distributions might benefit from format branching in order to easily test documents and code against the upcoming releases. Possible values are:
 - **stable** : This value, as the name implies, enables the `stable` engine format branch. Note that this is the default format.
 - **unsafe** : This value enables the `unsafe` engine format branch. According to the documentation, at all costs, avoid using this, or any, unsafe invocation with documents off the net or that are otherwise untrusted in any way.
- **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.

- **[S]** `shell` : This option sets whether the possibility of running underlying system commands from within TeX is activated.
- **[S]** `synctex` : 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.

```
% arara: xetex: { interaction: scrollmode, synctex: yes }
```

xindex

This rule runs `xindex` , a flexible and powerful indexing system, on a provided `idx` input. This tool is completely with the `makeindex` program.

- **[R]** `input` : This option, as the name indicates, corresponds to the `idx` reference to be processed by the indexing system. Note that this option is required.
- `config` (default: `cfg`): This option specifies a configuration extension. Make sure to take a look at the documentation for further details.
- `language` (default: `en`): This option, as the name suggests, specifies the language. Make sure to take a look at the documentation for further 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.

```
% arara: xindex: { input: mydoc.idx }
```

xindy

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.

- **[S] quiet** : 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.

```
% 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:

```
$ 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!

Building arara

arara is a Kotlin 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 [GITLAB](#). This chapter provides detailed instructions on how to build our tool from source.

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 [GITLAB](#), 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):

```
$ git --version
git version 2.29.2
```

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 `\rbox{git}` installed out of the shelf.

- Our tool is written in the Kotlin 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):

```
$ 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 Gradle, a software project management and comprehension tool. Gradle is a build tool just like ours with a much more comprehensive build framework to provide support for the JVM ecosystem. In order to check if `gradle`, the Gradle binary, is available in your operating system, run the following command in the terminal (version numbers might vary):

```
$ gradle --version
-----
Gradle 6.0.1
-----

Build time:   2019-11-21 11:47:01 UTC
Revision:     <unknown>

Kotlin:       1.3.50
Groovy:       2.5.8
Ant:          Apache Ant(TM) version 1.10.7 compiled
              on September 1 2019
JVM:          1.8.0_232 (Oracle Corporation 25.232-b09)
OS:           Linux 5.5.0-1-MANJARO amd64
```

Please refer to the Gradle [PROJECT WEBSITE](#) in order to obtain specific installation instructions for your operating system. In general, most recent Linux distributions have the Gradle binary, as well the proper associated dependencies, available in their corresponding repositories.

- For a proper repository cloning, as well as the first Gradle build, an active Internet connection is required. In particular, Gradle dynamically downloads Java libraries and plugins 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 Gradle 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 interest-

ing 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 of Gradle.

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.

Compiling the tool

Fetching the sources

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:

```
$ mkdir git-projects
$ cd git-projects
```

Run the following command in the terminal to clone the arara project:

```
$ git clone https://gitlab.com/islandoftex/arara.git
```

Wait a couple of seconds (or minutes, depending on your Internet connection) while the previous command clones the project repository hosted at GitLab. 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:

```
$ git clone https://gitlab.com/islandoftex/arara.git --depth 1
```

This operation is way faster than the previous one, for obvious reasons.

Building the executable

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 the directory named `arara/`. Run the following command in the terminal:

```
$ cd arara
```

The previous command should take us inside the `arara/` 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:

```
$ ls build.gradle.kts
build.gradle.kts
```

Great, we are in the correct location! From the previous output, let us inspect the directory contents. The `application/` directory, as the name suggests, contains the source code of the main application organized in an established package structure, whereas `build.gradle.kts` is the corresponding Gradle build file written to efficiently compile the project. In order to build our tool, run the following command in the terminal:

```
$ gradle build
```

Gradle 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. After compiling, the code can be packaged, tested and run. The `build` target actually compiles, tests and packages our tool. Afterwards, you will have a `cli/build/libs/` directory with multiple JAR files, one containing `with-deps`. That file is ready to run as it bundles all dependencies. Subsequent builds will be significantly faster than the first build because they do not fetch dependencies and rely on a build cache. Finally, after some time, Gradle will output the following message as result (please note that the entire compilation and packaging only took 4 seconds on my development machine due to an existing local cache):

```
BUILD SUCCESSFUL in 4s
15 actionable tasks: 15 up-to-date
```

Executing the fresh build

Now, let us move the resulting Java archive file from that particular directory to our current directory. Run the following command in the terminal (please note that the Java archive file was also renamed during the move operation):

```
$ mv cli/build/libs/arara-cli-with-deps-*.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 [DEPLOYING](#). You can also test the resulting file by running the following command in the terminal:

```
$ java -jar arara.jar
Usage: arara [OPTIONS] file...
```

```

  -- -- -- -- --
 / _ \ | ' _ / _ \ | ' _ / _ \ |
 | ( _ | | | | ( _ | | | | ( _ |
 \ _ , | | | \ _ , | | | \ _ , |

```

The cool TeX automation tool.

`arara` executes the TeX workflow you tell it to execute. Simply specify your needs within your TeX file and let `arara` do the work. These directives feature conditional execution and parameter expansion.

Options:

<code>-l, --log</code>	Generate a log output
<code>-v, --verbose / -s, --silent</code>	Print the command output
<code>-n, --dry-run</code>	Go through all the motions of running a command, but with no actual calls
<code>-S, --safe-run</code>	Run in safe mode and disable potentially harmful features. Make sure your projects uses only allowed features.
<code>-w, --whole-file</code>	Extract directives in the file, not only in the header
<code>-p, --preamble TEXT</code>	Set the file preamble based on the <code>configuration</code> file
<code>-t, --timeout INT</code>	Set the execution timeout (in milliseconds)
<code>-L, --language TEXT</code>	Set the application language
<code>-m, --max-loops INT</code>	Set the maximum number of loops (<code>> 0</code>)
<code>-d, --working-directory PATH</code>	Set the working directory for all tools
<code>-P, --call-property VALUE</code>	Pass parameters to the application to be used within the session.
<code>--generate-completion [bash zsh fish]</code>	Generate a completion script for <code>arara</code> . Add <code>'source <(arara --generate-completion</code>


```
<shell>)' to your shell's init file.  
-V, --version      Show the version and exit  
-h, --help         Show this message and exit  
  
Arguments:  
  file  The file(s) to evaluate and process
```

The following optional Gradle phase is used to handle the project cleaning, including the complete removal of the `build/` 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:

```
$ gradle clean
```

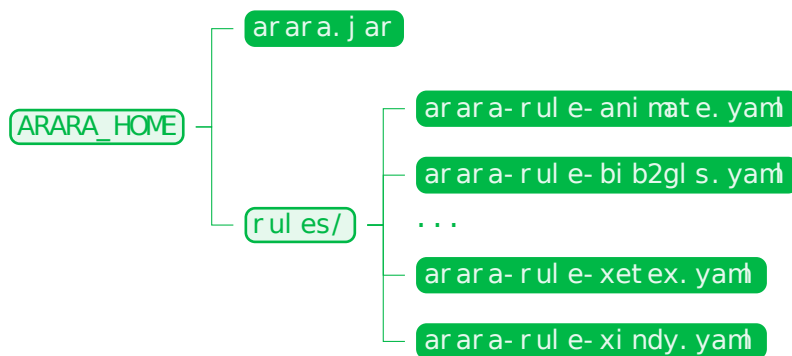
This section covered the compilation details for building arara from source. The aforementioned steps are straightforward and can be automated in order to generate snapshots and daily builds. If you run into any issue, please let us know. Happy compilation!

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 [BUILDING](#)).

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. 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 CTAN package is available in the [ARTIFACTS SECTION](#) of our project repository. Once the package is properly downloaded, we simply need to extract it into a proper `ARARA_HOME` location.

Defining a location

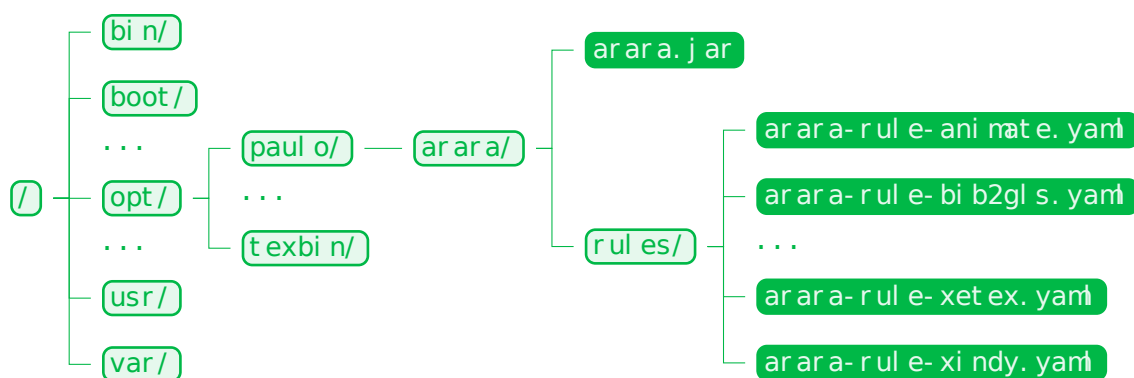
First and foremost, we need to obtain `master-ctan.zip` from our project repository at Git-Lab, which is our CTAN package artifact. As the name indicates, this is a compressed file format, so we need to extract it into a proper location. Run the following commands in the terminal:

```
$ unzip master-ctan.zip
$ unzip arara.tds.zip
$ mv scripts/arara .
```

As a result of the previous commands, we obtained a directory named `arara` with the exact structure presented in the previous section in our working directory (amongst other files and directories that can be safely discarded). 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):

```
$ 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 **BUILDING**), 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):

```
$ java -jar /opt/paulo/arara/arara.jar
Usage: arara [OPTIONS] file...
```

```

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

```

The cool TeX automation tool.

`arara` executes the TeX workflow you tell it to execute. Simply specify your needs within your TeX file and let arara do the work. These directives feature conditional execution and parameter expansion.

Options:

<code>-l, --log</code>	Generate a log output
<code>-v, --verbose / -s, --silent</code>	Print the command output
<code>-n, --dry-run</code>	Go through all the motions of running a command, but with no actual calls
<code>-S, --safe-run</code>	Run in safe mode and disable potentially harmful features. Make sure your projects uses only allowed features.
<code>-w, --whole-file</code>	Extract directives in the file, not only in the header
<code>-p, --preamble TEXT</code>	Set the file preamble based on the configuration file
<code>-t, --timeout INT</code>	Set the execution timeout (in milliseconds)
<code>-L, --language TEXT</code>	Set the application language
<code>-m, --max-loops INT</code>	Set the maximum number of loops (> 0)
<code>-d, --working-directory PATH</code>	Set the working directory for all tools
<code>-P, --call-property VALUE</code>	Pass parameters to the application to be used within the session.
<code>--generate-completion [bash zsh fish]</code>	Generate a completion script for arara. Add <code>'source <(arara --generate-completion <shell>)'</code> to your shell's init file.
<code>-V, --version</code>	Show the version and exit
<code>-h, --help</code>	Show this message and exit

Arguments:

`file` The file(s) to evaluate and process

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.

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:

```
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:

```
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. 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):

```
#!/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:

```
$ 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):

```
$ 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):

```
$ 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:

```
$ arara
Usage: arara [OPTIONS] file...
```

```

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

The cool TeX automation tool.

`arara` executes the TeX workflow you tell it to execute. Simply specify your needs within your TeX file and let `arara` do the work. These directives feature conditional execution and parameter expansion.

Options:

<code>-l, --log</code>	Generate a log output
<code>-v, --verbose / -s, --silent</code>	Print the command output
<code>-n, --dry-run</code>	Go through all the motions of running a command, but with no actual calls
<code>-S, --safe-run</code>	Run in safe mode and disable potentially harmful features. Make sure your projects uses only allowed features.
<code>-w, --whole-file</code>	Extract directives in the file, not only in the header
<code>-p, --preamble TEXT</code>	Set the file preamble based on the configuration file
<code>-t, --timeout INT</code>	Set the execution timeout (in milliseconds)
<code>-L, --language TEXT</code>	Set the application language
<code>-m, --max-loops INT</code>	Set the maximum number of loops (> 0)
<code>-d, --working-directory PATH</code>	Set the working directory for all tools
<code>-P, --call-property VALUE</code>	Pass parameters to the application to be used within the session.
<code>--generate-completion [bash zsh fish]</code>	Generate a completion script for <code>arara</code> . Add <code>'source <(arara --generate-completion <shell>)'</code> to your shell's init file.
<code>-V, --version</code>	Show the version and exit
<code>-h, --help</code>	Show this message and exit

Arguments:

`file` The file(s) to evaluate and process

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:

```
$ 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.

Introduction to the YAML format

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 [IMPORTANT CONCEPTS](#) .
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 [IMPORTANT CONCEPTS](#) .
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 [CONFIGURATION](#) .

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.

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 of scalars:

```
team:
- Paulo Cereda
- Marco Daniel
```

- Brent Longborough
- Nicola Talbot
- Ben Frank

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:

```
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:

```
developer:  
  name: Paulo  
  country: Brazil
```

Similarly, the flow mapping uses curly braces. Observe that this is the form adopted by a parametrized directive (see syntax in [IMPORTANT CONCEPTS](#)):

```
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:

```
developers:  
- name: Paulo  
  country: Brazil  
- name: Marco  
  country: Germany
```

The previous code can be easily represented in flow style by using square and curly brackets to represent sequences and attribute maps.

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, de-

noted 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 important to note that arara intensively uses both styles (as seen in Section~\ref{sec:rule}, on page~\pageref{sec:rule}). Let us see an example:

```
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 (see [MVEL](#) for reference), it might be advisable to quote scalars when starting with forbidden symbols in YAML.

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:

```
!config
```

In particular, rules and configuration files of arara are properly covered in [IMPORTANT CONCEPTS](#) and [CONFIGURATION](#). For now, it suffices to say that the `!config` global tag is necessary to provide the correct mapping of values inside our tool.

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 [IMPORTANT CONCEPTS](#) .
2. *Conditionals* , as logical expressions must be evaluated in order to decide whether and how a directive should be interpreted. Conditionals are detailed in [IMPORTANT CONCEPTS](#) .

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.

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:

```
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 [IMPORTANT CONCEPTS](#) . The scripting language can also be used for evaluating a boolean expression:

```
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:

```
(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:

```
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:

```
foo = 10;  
bar = (foo = foo * 2) + 10;  
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:

```
foo = 10;  
bar = (foo = foo * 2) + 10;  
return foo;
```

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:

```
"123" == 123;
```

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.

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:

```
[ "Jim", "Bob", "Smith" ]
```

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:

```
[ "Foo" : "Bar", "Bar" : "Foo" ]
```

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:

```
{ "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:

```
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.

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):

```
user[5]
```

```
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):

```
user["foobar"]  
user.foobar
```

```
user.get("foobar")
```

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.

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:

```
if (var > 0) {  
    r = "greater than zero";  
}  
else if (var == 0) {  
    r = "exactly zero";  
}  
else {
```



```
r = "less than zero";  
}
```

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:

```
answer == true ? "yes" : "no";
```

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:

```
foreach (name : people) {  
    System.out.println(name);  
}
```

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.

```
for (int i = 0; i < 100; i++) {  
    System.out.println(i);  
}
```

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):

```
do {  
    x = something();  
} while (x != null);
```

```
do {  
    x = something();  
} until (x == null);
```

Finally, MVEL also implements the standard `while`, with the significant addition of an `until` counterpart (for inverted logic):

```
while (isTrue()) {  
    doSomething();  
}
```

```
until (isFalse()) {  
    doSomething();  
}
```

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.

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:

```
names = (user.name in users);
```

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.

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.

```
str = "My string";  
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:

```
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:

```
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.

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; we highly recommend the latter, detailed in [IMPORTANT CONCEPTS](#). 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:

```
My favourite team is @{ person.name == 'Enrico' ? '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).

Further documentation

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.