Argbash Documentation

Release 2.5.0

Matěj Týč

Sep 17, 2017

Contents

CHAPTER 1

Argbash

`Argbash` (https://argbash.io) is a `bash` code generator that can assist you in writing scripts that accept arguments. You declare arguments that your script should use in a few lines and then, you run `Argbash` on those declarations to get a parsing code that can be used on all platforms that have `bash` (Linux, macOS, MS Windows, ...).

You *can have* your parsing code in the script, you can have `Argbash` to help you to use it as a `bash` library, or you can generate the library yourself and include it yourself too, it's up to you. A basic template generator `argbash-init` is part of the package, and you can *get started with it* in a couple of seconds.

`Argbash` is free and open source software, you are free to use it, share it, modify it and share the modifications with the world, since it is published under the 3-clause BSD linense.

Version 2.5.0

Authors Matěj Týč

Website https://argbash.io

Requirements

Both you and your users need:

- `bash>=3.0`

Only you need those on the top:

- `autoconf>=2.64` (`Argbash` makes use of the `autom4te` utility)
- `grep`, `sed`, etc. (if you have `autoconf`, you probably have those already)

CHAPTER 2

Quickstart

In a nutshell, using `Argbash` consists of these simple steps:

1. You write (or generate) a simple template of your script based on arguments your script is supposed to accept.
2. You run the `argbash` script (located in the package's `bin` directory) on it to get the fully functional script.

Eventually, you may want to add/remove/rename arguments your script accepts. In that case, you just need to edit the script — you don't need to repeat the two steps listed above! Why? It is so because the script retains the template section, so if you need to make adjustments to the template, you just edit the template section of the script and run `argbash` on top of the script to get it updated.

How to use `Argbash`? You can either

- *download and install it* locally,
- use the online generator, or
- use the Docker container.

Generating a template

`Argbash` features the `argbash-init` script that you can use *to generate* a template in one step. Assume that you want a script that accepts one (mandatory) positional argument `positional-arg` and two optional ones `--option` and `--print`, where the latter is a boolean argument.

In other words, we want to support these arguments:

- `--option` that accepts one value,
- `--print` or `--no-print` that doesn't accept any value, and
- an argument named `positional-arg` that we are going to refer to as positional that must be passed and that is not preceeded by *options* (such as `--foo`, `-f`).

We call `argbash-init` and as the desired result is a script, we directly pipe the output of `argbash-init` to `argbash`:

```
bin/argbash-init --pos positional-arg --opt option --opt-bool print - | ../bin/
↪argbash -o script.sh -
```

Let's see what the auto-generated script can do!

```
./script.sh -h

<The general help message of my script>
Usage: ./script.sh [--option <arg>] [--(no-)print] [-h|--help] <positional-arg>
        <positional-arg>: <positional-arg's help message goes here>
        --option: <option's help message goes here> (no default)
        --print,--no-print: <print's help message goes here> (off by default)
        -h,--help: Prints help
```

```
./script.sh --print --option opt-value pos-value

Value of --option: opt-value
print is on
Value of positional-arg: pos-value
```

We didn't have to do much, yet the script is pretty capable.

Writing a template

Now, let's explore more advanced argument types on a trivial script that accepts some arguments and then prints their values. So, let's say that we would like a script that produces the following help message:

```
This is a minimal demo of Argbash potential
Usage: ../resources/examples/minimal.sh [-o|--option <arg>] [--(no-)print] [-h|--
↪help] [-v|--version] <positional-arg>
        <positional-arg>: Positional arg description
        -o,--option: A option with short and long flags and default (default: 'boo')
        --print,--no-print: A boolean option with long flag (and implicit default:␣
↪off) (off by default)
        -h,--help: Prints help
        -v,--version: Prints version
```

Then, it means that we need following arguments:

- One mandatory positional argument. (In other words, an argument that must be passed and that is not preceded by options)
- Four optional arguments:
  - `--option` that accepts one value,
  - `--print` or `--no-print` that doesn't accept any value — it either is or isn't specified,
  - `--version` that also doesn't accept any value and the program is supposed just to print its version and quit afterwards, and finally
  - `--help` that prints a help message and quits afterwards.

Therefore, we call `argbash-init` like we did before:

```
bin/argbash-init --pos positional-arg --opt option --opt-bool print minimal.m4
```

Next, we edit the template so it looks like this:

```
#!/bin/bash

# m4_ignore(
echo "This is just a script template, not the script (yet) - pass it to 'argbash' to␣
↪fix this." >&2
exit 11  #)Created by argbash-init v2.5.0
# ARG_OPTIONAL_SINGLE([option], o, [A option with short and long flags and default],␣
↪[boo])
# ARG_OPTIONAL_BOOLEAN([print], , [A boolean option with long flag (and implicit␣
↪default: off)])
# ARG_POSITIONAL_SINGLE([positional-arg], [Positional arg description], )
# ARG_HELP([This is a minimal demo of Argbash potential])
# ARG_VERSION([echo $0 v0.1])
# ARGBASH_SET_INDENT([  ])
# ARGBASH_GO
```

The body of the script (i.e. lines past the template) is trivial, but note that it is enclosed in a pair of square brackets. They are "hidden" in comments and not seen by the shell, but still, they have to be there for the "use the script as a template" feature to function.

```
# [ <-- needed because of Argbash

if [ "$_arg_print" = on ]
then
  echo "Positional arg value: '$_arg_positional_arg'"
  echo "Optional arg '--option' value: '$_arg_option'"
else
  echo "Not telling anything, print not requested"
fi

# ] <-- needed because of Argbash
```

We generate the script from the template:

```
bin/argbash script.m4 -o script.sh
```

Now we launch it and the output is good!

```
./script.sh posi-tional -o opt-ional --print

Positional arg value: posi-tional
Optional arg --option value: opt-ional
```

Note: If something still isn't totally clear, take look at the *Examples* section.

Limitations

Warning: Please read this carefully.

1. The square brackets in your script have to match (i.e. every opening square bracket `[` has to be followed at some point by a closing square bracket `]`).

   There is a workaround — if you need constructs s.a. `red=$'\e[0;91m'`, you can put the matching square bracket behind a comment, i.e. `red=$'\e[0;91m' # match square bracket: ]`.

   This limitation does apply only to files that are processed by `argbash` — you are fine if you have the argument parsing code in a separate file and you *don't use* the `INCLUDE_PARSING_CODE` macro. You are also OK if you use *argbash-init* in the *decoupled mode*.

2. The generated code generally contains bashisms as it relies heavily on `bash` arrays to process any kind of positional arguments and multi-valued optional arguments. That said, if you stick with optional arguments only, a POSIX shell s.a. `dash` should be able to process the `Argbash`-generated parsing code.

CHAPTER 3

Index

Contents:

Installation

User installation

If you want to use Argbash locally, you have to download the software package and run the installation script.

1. Go to the release section of the GitHub project, choose the version to download, and download the source code package.
2. Unpack the contents of the archive. You can use the `bin/argbash` script without any installation (as it is described in the *Quickstart*), but you can proceed to the installation in order to be able to use `argbash` system-wide.
3. Go to the `resources` folder. There is a `Makefile`.
4. According to whether you have your `$HOME/.local/bin` folder in the `PATH`:
   - If so, run `make install PREFIX=$HOME/.local`,
   - else, run `sudo make install PREFIX=/usr`.

   Note: If you want multiple `Argbash` versions installed in parallel, install them using `make altinstall` (and uninstall using `make uninstall`) commands. This will create `argbash-X.Y.Z` script under the `bin` directory, with `argbash-X.Y`, `argbash-X` and `argbash` symlinks pointing transitively to it. If you altinstall another version of `Argbash`, the common symlinks will be overwritten (i.e. at least `argbash`).

   This way of installation won't install the `argbash-xtoy` *migration scripts*.

   ---

5. Optional: Run some checks by executing: `make check` (still in the `resources` folder). You should get a message `All is OK` at the bottom.

Argbash has this audience:

- Users — people that use scripts that make use of Argbash.
- Developers — people that use Argbash to write scripts.
- Tinkerers — people that come in contact with Argbash internals, typically curious Developers.
- bash >= 3.0 — this is obvious, everybody needs bash. There is only one exception — in cases of simple scripts, a POSIX shell s.a. dash will be enough for Users.
- autoconf >= 2.63 — Argbash is written in a m4 language extension called m4sugar, which is contained in autoconf. Developers and Tinkerers need this. autoconf is available on Linux, macOS, BSDs and can be installed on MS Windows.
- grep, sed, coreutils — The argbash script uses grep, sed, cat, and test. If you have autoconf, you probably have those already.
- GNU Make >= 4.0 — the project uses Makefiles to perform a wide variety of tasks, although it is more of interest to Tinkerers.

Building Argbash

If you identify yourself as a tinkerer (i.e. you want to play with internals of Argbash), you may use a different set of steps:

1. Clone the Git repository: git clone https://github.com/matejak/argbash.git
2. Go to the resources directory consider running a develop install there, e.g. make develop PREFIX=$HOME/.local,

   This type of installation ensures that whenever you make a change to the bin/argbash script in the repository, the argbash command always calls that bin/argbash script.
3. After you make modifications the source files (.m4 files in the src directory), you regenerate bin/argbash by running make ../bin/argbash in the resources directory.

   If you let a bug through that prevents the argbash script to regenerate itself, run make bootstrap to regenerate it in a more robust way.
4. Remember to run make check in the resources directory often to catch bugs as soon as possible.

Argbash components

The Argbash package consists of these scripts:

- argbash, the main part of Argbash. It is basically a wrapper around the autom4te utility that uses the Argbash "source code" located in the src directory. In course of an installation, both the script and the source are copied under the prefix — script goes to $PREFIX/bin and source to $PREFIX/lib/argbash.

  The argbash script itself is generated using Argbash. It can be (re)generated using a Makefile that can be found in the resources folder.
- argbash-xtoy scripts (x, y are major version numbers) that assist users in modifying their scripts in case that Argbash *changes its API*. For example, Argbash 2.1.4 (we say Argbash of major version 2) has argbash-1to2 script and Argbash of major version 3 will have scripts argbash-1to3 and argbash-2to3.
- argbash-init is a quickstart script — it enables you to create a basic *template* for your script. Then, you just have to make some slight modifications, *feed it to argbash* and you are done.

The main Makefile

The `Makefile` in the `resources` folder can do many things:

Installation

- `make install [PREFIX=foo]` runs the installation into the prefix you can specify (default is `$(HOME)/.local`). This will install the `argbash` script (notice the missing `.sh` extension) into `$PREFIX/bin` (and some support files into `$PREFIX/lib/argbash`).
- `make develop [PREFIX=foo]` is similar to `make install`, but it installs a wrapper around the local `bin/argbash`, so any change to the file will be immediately reflected for everybody who uses the system-wide one. This is inspired by Python's `python setup.py develop` pattern.
- `make uninstall [PREFIX=foo]` inverse of the above.

Running argbash

- `make ../bin/argbash`, `make bootstrap` makes (or updates) the `argbash` script (the script basically overwrites itself). Use the latter if previous update broke the current `../bin/argbash` so it is not able to regenerate itself.
- `make examples` compiles examples from `.m4` files to `.sh` files in the `examples` folder.
- `make foo/bar.sh` generates a script provided that there is a `foo/bar.m4` file.
- `make foo/bar2.sh` generates a script provided that there is a `foo/bar.sh` file.

Releasing

- `make check` runs the tests.
- `make version VERSION=1.0.0` sets the project's version to all corners of the project where it should go.
- `make release [VERSION=1.0.0]` refreshes date in the `ChangeLog` and regenerates all of the stuff (and runs tests).
- `make tag` tags the version.

Template writing guide

This section tells you how to write templates, the next one is about `argbash` script invocation.

Definitions

Positional and optional arguments

There are two major types of arguments — take an example:

```
ls -l --sort time /home
```

- Optional arguments are `-l` and `--sort`, while we have only one
- positional argument — `/home`.

Here, the argument `-l` is optional of a boolean type (it is either on or off), `--sort` is also optional, taking exactly one value (in this case `time`). `-l` and `--sort` are called options, hence the name *optional* arguments. The common pattern is that optional arguments are not required, being there just in the case you need them.

The `/home` argument is a positional one. In case of `ls`, the positional argument has a default — running `ls` without parameters is the same as running `ls "."`. `ls` itself accepts an arbitrary number of positional arguments and it treats them all in the same way.

On the other hand, the `grep` command requires at least one positional argument. The first one is supposed to be the regular expression you want to match against, and the other ones correspond to filenames, so they are not treated the same. The first positional argument `grep` accepts (i.e. the regular expression), doesn't have a default, whereas the second one normally defaults to `-`, which means `grep` will try to read input from `stdin`.

Option, Value, and others

We have positional and optional arguments sorted out, so let's define some other terms now keeping the example of `ls -l --sort time /home`:

- Option (also `flag` or `switch`): The string that identifies optional arguments on the command-line, can have a short (dash and a character, e.g. `-l`, `-?`) or long (double dash and string, e.g. `--sort`) form. POSIX conventions mention only short options, whereas the GNU conventions mention long options.
- Value: In connection with optional arguments, value of an argument is the string that follows it (provided that the argument expects a value to be given). Concerning positional arguments, it is simply the string on the command-line (whose location matches the location in which we expect the given positional argument). So in our example, the values are `time` and `home`.
- Name: Both positional and optional arguments have a name. In case of optional argument, the name is what appears after the long option's the double dash, e.g. name of `--project-path` is `project-path`. The argument's name is used in help and later in your script when you access argument's value. Names of positional arguments are much less visible to the script's user — one can see them only in the help message.
- Argument: An argument is the high-level concept. On command-line, arguments are identified by options (which themselves may or may be not followed by values). Although this is confusing, it is a common way of putting it. In our example, we have
  - `-l` — this argument has only the option, but never accepts values.
  - `--sort` — this argument accepts exactly one value (in this case, the string `time`). If you don't provide a value, you will get an error.

  `Argbash` exposes values of passed arguments as environmental variables.
- Default: In case of positional and boolean arguments, you may specify their default values.

> **Note:** General notice: There is no way of how to find out whether an argument was passed or not just by looking at the value of the corresponding environmental variable in the script. `bash` doesn't distinguish between empty variables and variables containing an empty string. Also note that it is perfectly possible to pass an empty string as an argument value.

So let's get back to argument types. Below, is a list of argument types and macros that you have to write to support those (e.g. `ARGBASH_GO` is a macro and `ARG_OPTIONAL_BOOLEAN([verbose], [Verbose mode])` is a macro called wit two arguments — `verbose` and `Verbose mode`). Place those macros in your files as `bash` comments.

Your script

You have to decide what arguments should your script support. As of this version, `Argbash` lets you choose from:

- Single-value positional arguments (with optional defaults),
- single-value optional arguments,
- boolean optional arguments,
- action optional arguments (i.e. the `--version` and `--help` type of args) and
- incremental arguments that "remember" how many times they have been repeated (e.g. `--verbose`) and
- repeatable arguments that sequentially store their values into an array (e.g. `-I`).

Plus, there are convenience macros that don't relate to argument parsing, but they might help you to write better scripts and a helper that enables you to easily wrap other `Argbash`-aware scripts without fuss.

Take a look at the API and place the declarations either to your script or in a separate file. Let yourself be inspired by the `resources/examples/simple.m4` example (`bash` syntax highlighting is recommended, despite the extension).

Then, run the following command to your file:

```
bin/argbash myfile.m4 -o myfile.sh
```

to either get a script that should just work, or a file that you include in your script.

Argbash API

Put macro parameters in square brackets. Parameters marked as optional can be left out blank.

The following code leaves second and last parameters blank. Values of first and third parameters are `verbose` and `Turn on verbose mode` respectively.

```
ARG_OPTIONAL_BOOLEAN([verbose], , [Turn on verbose mode], )
```

Positional arguments

- Single-value positional argument (with optional default):

  ```
  ARG_POSITIONAL_SINGLE([argument-name], [help message], [default (optional)])
  ```

  The argument is mandatory, unless you specify a default.

  If you leave the default blank, it is understood that you don't want one (and that the argument is mandatory). If you really want to have an explicit default of empty string, pass a quoted empty string (i.e. `""` or `''`).

- Multi-value positional argument (with optional defaults):

  ```
  ARG_POSITIONAL_MULTI([argument-name], [help message], [number of arguments], ...,␣
  ↪[default for the second-to-last (i.e. penultimate) argument (optional)],␣
  ↪[default for the last argument (optional)])
  ```

  Given that your argument accepts n values, you can specify m defaults, $(m \leq n)$ for last m values.

  For example, consider that your script makes use of only one multi-value argument, which accepts 3 values with two defaults `bar` and `baz`. Then, it is imperative that at least one value is specified on the command-line. So If you pass a value `val1` on the command-line, you will be able to retrieve `val1`, `bar` and `baz` inside the script.

If you pass `val1` and `val2`, you will be able to retrieve `val1`, `val2` and `baz`. If you pass nothing, or more than three values, an error will occur.

Arguments are available as a `bash` array (first element has index of 0).

- Infinitely many-valued positional argument (with optional defaults):

```
ARG_POSITIONAL_INF([argument-name], [help message], [minimal number of arguments␣
↪(optional, default=0)], [default for the first non-required argument␣
↪(optional)], ...)
```

`Argbash` supports arguments with arbitrary number of values. However, you can require a minimal amount of values the caller has to provide and you can also assign defaults for the values that are not required. Given that your argument accepts at least n values, you can specify defaults for $(n+1)^{\text{th}}$ argument (and so on).

For example, consider that your script makes use of infinitely many-valued argument, which accepts at least 1 value and also has two defaults `bar` and `baz`. Then, it is imperative that at least one value is specified on the command-line. So If you pass a value `val1` on the command-line, you will be able to retrieve `val1`, `bar` and `baz` inside the script. If you pass `val1`, `val2`, `val3` and `val4`, you will be able to retrieve `val1`, `val2` `val3` and `val4`.

Arguments are available as a `bash` array (first element has index of 0).

Note: The main difference between `ARG_POSITIONAL_MULTI` and `ARG_POSITIONAL_INF` is in handling of defaults. In `ARG_POSITIONAL_MULTI`, defaults determine the number of values that are required to be supplied. In `ARG_POSITIONAL_INF`, you determine the number of required values and defaults follow.

- End of optional arguments and beginning of positional ones (the double-dash `--`):

```
ARG_POSITIONAL_DOUBLEDASH()
```

You are encouraged to add this to your script if you use both positional and optional arguments.

This pattern is known for example from the `grep` command. The idea is that you specify optional arguments first and then, whatever argument follows it, it is considered to be a positional one no matter how it looks. For example, if your script accepts a `--help` optional argument and you want it to be recognized as positional, using the double-dash is the only way.

Optional arguments

- Single-value optional arguments:

```
ARG_OPTIONAL_SINGLE([argument-name-long], [argument-name-short (optional)], [help␣
↪message], [default (optional)])
```

The default default is an empty string.

- Boolean optional arguments:

```
ARG_OPTIONAL_BOOLEAN([argument-name-long], [argument-name-short (optional)],␣
↪[help message], [default (optional)])
```

The default default is `off` (the only alternative is `on`).

- Incremental optional arguments:

```
ARG_OPTIONAL_INCREMENTAL([argument-name-long], [argument-name-short (optional)],␣
↪[help message], [default (optional)])
```

The default default is 0. The argument accepts no values on command-line, but it tracks a numerical value internally. That one increases with every argument occurrence.

- Repeated optional arguments:

```
ARG_OPTIONAL_REPEATED([argument-name-long], [argument-name-short (optional)],␣
↪[help message], [default (optional)])
```

The default default is an empty array. The argument can be repeated multiple times, but instead of the later specifications overriding earlier ones (s.a. `ARG_OPTIONAL_SINGLE` does), arguments are gradually appended to an array. The form of the default is what you normally put between the brackets when you create `bash` arrays, so put whitespace-separated values in there, for example:

```
ARG_OPTIONAL_REPEATED([include], [I], [Directories where to look for include␣
↪files], ['/usr/include' '/usr/local/include'])
```

The specified values are appended to defaults, so if you consider a script that accepts the `--include` argument due to the directive above, if you pass it `-I src/include`, the argument-holding array will have three elements — `/usr/include`, `/usr/local/include` and `src/include`.

- Action optional arguments (i.e. the `--version` and `--help` type of comments):

```
ARG_OPTIONAL_ACTION([argument-name-long], [argument-name-short (optional)], [help␣
↪message], [code to execute when specified])
```

The scripts exits after the argument is encountered. You can specify a name of a function, `echo "my-script: v0.5"` and whatever else. This is simply a shell code that will be executed as-is (including `"` and `'` quotes) when the argument is passed. It can be multi-line, but if you need something sophisticated, it is recommended to define a shell function in your script template and call that one instead.

Special arguments

- Help argument (a special case of an optional action argument):

```
ARG_HELP([short program description (optional)], [long program description␣
↪(optional)])
```

This will generate the `--help` and `-h` action arguments that will print the usage information. Notice that the usage information is generated even if this macro is not used — we print it when we think that there is something wrong with arguments that were passed.

The long program description is a string quoted in double quotation marks (so you may use environmental variables in it) and additionally, occurrences of `\n` will be translated to a line break with indentation (use `\\n` to have the actual `\n` in the help description). If you want to have environmental variables and newlines, you have to make sure that the env variable contains literal newlines/tabs — you can either use the `foo=$'broken\nline'` pattern, or you can use quotes to define the variable so it contains real literal newlines / tabs.

- Version argument (a special case of an action argument):

```
ARG_VERSION([code to execute when specified])
```

- Verbose argument (a special case of a repeated argument):

```
ARG_VERBOSE([short arg name])
```

Default default is 0, so you can use a `test $_arg_verbose -ge 1` pattern in your script.

- Collect leftovers:

```
ARG_LEFTOVERS([help text (optional)])
```

This macro allows your script to accept more arguments and collect them consequently in the `_arg_leftovers` array.

A use case for this is wrapping of scripts that are completely `Argbash`-agnostic. Therefore, your script can take its own arguments and the rest that is not recognized can go to the wrapped script.

Typing macros

Warning: Features described in this section are experimental. Macros in the type-related section below are not an official part of the API yet — their names and/or signature may change.

The documentation here is just a peek into the `Argbash` future. Please raise an issue if you feel you can provide helpful feedback!

`Argbash` supports typed argument values. For example, you can declare that a certain argument requires an integer value, and if its value by the time of conclusion of the parsing part of the script is not of an integer type, an error is raised. The validator sometimes returns the value in a canonical form (e.g. it may trim leading and trailing whitespaces).

Note: Users of your script have to have a working `grep` in order to use this.

Generally, macros accept these parameters:

- Type code. In some cases, you make it up and in other cases, you have to know the right one. End-users of your script won't even see it.
- Type string. This is used in the script's help.
- List of arguments whose values are of the given type. Typically, `[arg1, arg2]` is OK*[0].

You have these possibilities:

- Built-in types:

```
ARG_TYPE_GROUP([type code], [type string], [list of arguments of that type])
```

Type code is a code of one of the types that are supported, type string is used in help.

Type code	Description
int	integer
pint	positive integer
nnint	non-negative integer
float	floating-point number (e.g. 4.2e1)
decimal	float without the exponential stuff (e.g. 42.0)
string	anything†[0]

[0] Passing `arg1, arg2` won't work (of course — this represents two arguments, not one that is a list), `[arg1, arg2]` will work in most cases (when neither `arg1` or `arg2` have been defined as a macro), whereas `[[arg1],[arg2]]` will work no matter what.

† As an example, if you have an argument `--iterations` that accepts a value representing how many times to repeat something, you use

```
ARG_TYPE_GROUP([nnint], [COUNT], [iterations])
```

- One-of values (i.e. values are restricted to be members of a set).

```
ARG_TYPE_GROUP_SET([type code], [type string], [list of arguments of that type],␣
↪[list of values of that type], [suffix of the index variable (optional)])
```

If the suffix of the index variable is provided, each argument of the type will have a variable `_arg_<stem>_<suffix>` that contains the 0-based index of the argument value in the allowed values list. You will typically want to use it as described in the next example:

Remarks:

  - Pass the list of values without shell-quoting. Double quotes will be applied later.

```
ARG_TYPE_GROUP_SET([operations], [OPERATION], [start-with,stop-with], [configure,
↪make,install], [index])
```

and later in the code, you can use a construct like

```
# fail e.g. when we start-with make and stop-with configure.
# It would work if it was the other way.
test "$_arg_stop_with_index" -gt "$_arg_start_with_index" \
   || die "The last operation has to be a successor of the first one, which is␣
↪not the case."
```

- Filenames

```
DEFINE_VALUE_TYPE_FILE([type], [mode], [type string], [list of arguments of that␣
↪type])
```

  - The `type` string is either `in` or `out`. Input files have to exist, output files have to have their parent directory writable.
  - `mode` string is a `rwx`-type of string.

Convenience macros

Plus, there are convenience macros:

- Set the indentation in the parsing part of the script:

```
ARGBASH_SET_INDENT([indentation character(s)])
```

The default indentation is one tab per level. If you wish to use two spaces as the Google style recommends, simply pass two spaces (in square brackets!) as an argument to the macro.

- Set the delimiter between option and value:

```
ARGBASH_SET_DELIM([option-value delimiter character(s)])
```

The default delimiter is either space or equal sign. You can either restrict delimiter to only space or only equal sign, or you can keep both. Assuming you have an option accepting value (can be either single-valued or repeated) `--option` with short option `-o`, the following works with these arguments to the macro:

[0] The type `string` is used as a means to modify the help message, no validation or conversion takes place.

  - `ARGBASH_SET_DELIM([ ])`: Either of `--option value`, `--o value` assigns value to the `option` argument. `--option=value` will be considered as a single positional argument.
  - `ARGBASH_SET_DELIM([=])`: Either of `--option=value`, `--o value` assigns value to the `option` argument. `--option value` will result in both `--option` and `value` to be considered as two positional arguments. `-o=value` will also be considered as a positional argument.
  - `ARGBASH_SET_DELIM([= ])` (or `[ =]`): Either of `--option=value`, `--o value`, `--option value` assigns value to the `option` argument; they are treated the same way. This is the default behavior.

- Add a line where the directory where the script is running is stored in an environmental variable:

```
DEFINE_SCRIPT_DIR([variable name (optional, default is script_dir)])
```

  You can use this variable to e.g. source `bash` snippets that are in a known location relative to the script's parent directory.

- Include a file (let's say a `parse.sh` file) that is in the same directory during runtime. If you use this in your script, `Argbash` finds out and attempts to regenerate `parse.sh` using `parse.sh` or `parse.m4` if the former is not available. Thanks to this, managing a script with body and parsing logic in separate files is really easy.

```
INCLUDE_PARSING_CODE([filename], [SCRIPT_DIR variable name (optional, default is␣
↪script_dir)])
```

  In order to make use of `INCLUDE_PARSING_CODE`, you have to use `DEFINE_SCRIPT_DIR` on preceding lines, but you will be told so if you don't.

  See also:

  Check out the example: *Separating the parsing code*

- Point to a script that uses `Argbash` (or to its template), and your script will inherit its arguments (unless you exclude some of them).

```
ARGBASH_WRAP(filename stem, [list of long options to exclude], [flags to exclude␣
↪certain arg types, default is HV for (h)elp and (v)ersion])
```

  Given that you have a script `process_single.sh` and you write its wrapper `process_file.sh` Imagine that one reads a file and passes data from every line to `process_single.sh` along with some options that `process_file.sh` accepts.

  In this case, you write `ARGBASH_WRAP([process_single], [operation])` to your `process_file.m4` template.

  - Filename stem is a filename without a directory component or an extension. Stems are searched for in search paths (current directory, directory of the template) and extensions `.m4` and `.sh` are tried out.
  - The list of long options is a list of first arguments to functions such as `ARG_POSITIONAL_SINGLE`, `ARG_OPTIONAL_SINGLE`, `ARG_OPTIONAL_BOOLEAN`, etc. Therefore, don't include leading double dash to any of the list items that represent blacklisted optional arguments. To blacklist the double dash positional argument feature, add the `--` symbol to the list.
  - Flags is a string that may contain some characters. If a flag is set, a class of arguments is excluded from the file. The default `HVIS` should be enough in most scenarios — you want your own help, version info, indentation and option–value separator, not ones from the wrapped script, right?

    Following flags are supported:

Character	Meaning
H	Don't include help.
V	Don't include version info.
I	Don't use wrapped script's indentation
S	Don't use wrapped script's option–value separator

– As a convenience feature, if you wrap a script with stem `process_single`, all options that come from the wrapped script (both arguments and values) are stored in an array `_args_process_single`. In the case where there may be issues with positional arguments (they are order-dependent and the wrapping script may want to inject its own to the wrapped script), you can use `_args_process_single_opt`, or `_args_process_single_pos`, where only optional/positional arguments are stored. Therefore, when you finally decide to call `process-single.sh` in your script with all wrapped arguments (e.g. `--some-opt foo --bar`), all you have to do is to write

```
./process-single.sh "${_args_process_single_opt[@]}"
```

which is exactly the same as

```
MAYBE_BAR=
test $_arg_bar = on && MAYBE_BAR='--bar'
./process-single.sh --some-opt "$_arg_some_opt" $MAYBE_BAR
```

The stem to array name conversion is the same as with *argument names* except the prefix `_args_` is prepended.

Note: The wrapping functionality actually only makes your script to inherit (all or some of the) the wrapped script's arguments. If you really wish to call the wrapped script, it is your responsibility to know its location, `Argbash` essentially can't and won't help you with that.

However, if you know the relative location of the wrapped script to the wrapper, you can use the *DEFINE_SCRIPT_DIR* macro.

See also:

Check out the example: *Wrapping scripts*

Warning: Features described at the rest of this section are experimental. Convenience macros below are not an official part of the API yet — their names and/or signature may change.

The documentation here is just a peek into the `Argbash` future. Please raise an issue if you feel you can provide helpful feedback!

- Declare that your script uses an environment variable, set a default for it if it is blank upon the script's invocation and optionally mention it in the script's help:

```
ARG_USE_ENV([variable name], [default if empty (optional)], [help message␣
↪(optional)])
```

For instance, if you declare `ARG_USE_ENV([ENVIRONMENT], [production], [The default environment])`, the value of the `ENVIRONMENT` environmental variable won't be empty — if the user doesn't do anything, it will be `production` and if the user overrides it, it will stay that way. It is undefined whether the user can override it so it has a blank value in the script due to the user override (i.e. it is not possible now, but it may become possible in a later release.).

- Declare that your script calls a program and enable the caller to set it using an environmental variable.

```
ARG_USE_PROG([variable name], [default if empty (optional)], [help message␣
↪(optional)], [args (optional)])
```

For instance, if you declare `ARG_USE_PROG([PYTHON], [python], [The preferred Python executable])` in your script, you can use constructs s.a. `"$PYTHON" script.py` later. This macro operates in two modes:

- `args` are not given: The program name is searched for using the `which` utility and if it isn't a executable, the script will terminate with an error. `ARG_USE_PROG([PYTHON], [python], ,)`
- `args` are given: The program is called with `args` and if the return code is non-zero, the script will terminate with an error. If you want to call the program with no arguments, leave the last argument blank — the following usage is 100% legal: `ARG_USE_PROG([PYTHON], [python], ,)` and it means "accept `PYTHON` with default value `python`, but don't bother with help message and pass no arguments when evaluating whether a program is valid".

  Notice that this approach is wrong, calling `python` without arguments won't work (since it starts the interactive Python interpreter) and you should use `ARG_USE_PROG([PYTHON], [python], , [--version])` instead.

In either case, the value of `"$PYTHON"` will be either `python` (if the user doesn't override it), or it can be whatever else what the caller sets.

- Declare every variable related to every positional argument:

```
ARG_DEFAULTS_POS()
```

By default, only variables with defaults are declared. Since values are assigned using `eval`, static analysis tools s.a. shellcheck may complain about referencing undeclared variables. This macro helps to ensure that there are not these false positives.

- Activate Argbash-powered scripts strict mode:

```
ARG_RESTRICT_VALUES([mode code])
```

The mode code restricts allowed values for all arguments.

Mode code	What is restricted
none	nothing is restricted (default behavior)
no-any-options	anything that looks like as an option (be it long or short)
no-local-options	option (long or short) of any optional argument this script supports

You may want to restrict argument values in order to prevent these possible confusions:

- The user forgets to supply value to an optional argument, so the next argument is mistaken for it. For example, when we leave `time` from `ls --sort time --long /home/me/*`, we get a syntactically valid command-line `ls --sort --long /home/me/*`, where `--long` is identified as value of the argument `--sort` instead an argument on its own.
- The user intends to pass an optional argument on the command-line (e.g. `--sort`), but makes a typo, (e.g. `--srot`), or the script actually doesn't support that argument. As an unwanted consequence, it is interpreted as a positional argument.

- Make Argbash-powered scripts getopts-compatible:

```
ARG_OPTION_STACKING([mode code])
```

The mode code either enables getopt-like grouping (a.k.a. stacking) of short arguments according to Guideline 5, or disables it.

Mode code	What is restricted
none	no grouping support
getopts	support full getopts-like functionality (default behavior)

Action macro

Finally, you have to express your desire to generate the parsing code, help message etc. You do it by specifying an "action macro" past all arguments definitions.

You can either let the parsing code to be executed (carefree mode), or you can just generate parsing functions and call them yourself (DIY mode).

- Carefree mode: Use action macro `ARGBASH_GO`. The macro doesn't take any parameters.

```
ARGBASH_GO
```

- DIY mode: Use action macro `ARGBASH_PREPARE`. The macro doesn't take any parameters.

  If you are not familiar with the DIY mode, generate the script with *embedded helpful comments* that tell you what functions you have to call in your code to fully use the Argbash potential.

```
ARGBASH_PREPARE
```

> **Warning:** This feature is under development and not part of the stable API.

Available shell stuff

- Variable `script_dir` that is available if the *DEFINE_SCRIPT_DIR* is used.
- Function `die`.

  Accepts two parameters — string that is printed to `stderr` and exit status number (optional, default is 1). If an environmental variable `_PRINT_HELP` is set to `yes`, it prints help before the error message.

Using parsing results

The key is that parsing results are saved in shell variables that relate to argument (long) names. The argument name is transliterated like this:

1. All letters are made lower-case
2. Dashes are transliterated to underscores (`include-batteries` becomes `include_batteries`)
3. `_arg_` is prepended to the string. So given that you have an argument `--include-batteries` that expects a value, you can access it via shell variable `_arg_include_batteries`.

- Boolean arguments have values either `on` or `off`. If (a boolean argument) `--quiet` is passed, value of `_arg_quiet` is set to `on`. Conversely, if `--no-quiet` is passed, value of `_arg_quiet` is set to `off`.
- Repeated arguments collect values to a bash array.
- Incremental arguments have a default value (0 by default) and their value in the script corresponds to the default plus the number of times the argument was specified.

Argbash tools

Argbash is a code generator, so what it does, it gives you code that has the ability to parse command-line arguments. The question is — what to do with the generated code? You have three options here, they are sorted by the estimated preference:

1. One file with both parsing code and script body — batteries are included!

   This is a both simple and functional approach, but the argument parsing code will pollute your script.

2. Two files — one for the parsing code and one for the script body, both taken care of by Argbash — separation of code, but you get things managed by Argbash..

   This is more suitable for people that prefer to keep things tidy, you can have the parsing code separate and included in the script at run-time. However, Argbash can assist you with that.

3. Same as the above, just without Argbash assistance — the parsing code is decoupled from the script.

   You have to take this path if your script has a non-matching square brackets problem (see *Limitations*). This approach is similar to the approach of bash argument parsing libraries with one difference — here, the library is generated by Argbash, so it may be significantly less complex than those generic libraries such as *EasyOptions*. This is very unlikely.

Note: We assume that you have installed (see *Installation*) the argbash script, so it is available in your terminal as a command argbash. If it is not the case, you just have to substitute argbash by direct invocation of bin/argbash.

Template generator

It is not advisable to write a template from scratch, since Argbash contains a tool for that. The argbash-init *can generate* a good starting template for you, so you can get started within minutes.

General usage

The most efficient way of using Argbash is probably this one (also covered in an *example*):

1. Get an idea of what arguments your script should accept.
2. Execute argbash-init with the right arguments to get a basic template.
3. Replace placeholders in the template with meaningful values.
4. Expand the template with another directives (if necessary) based on *argbash API*.
5. Run argbash over the template.

argbash-init supports generating templates with these types of arguments:

- Single-valued positional arguments (--pos argument).
- Single-valued optional arguments (--opt argument).
- Boolean optional arguments (--opt-bool argument).

Generally, you specify argument name and you add help etc. by editing the template file.

Next, argbash-init supports *wrapping* of another argbash-aware scripts. The help macro is always included.

Modes of operation

`argbash-init` allows you to select the way how the parsing code is handled (via the `-s`, `--standalone` option):

- Batteries-included mode:

  If you don't specify it, you get the case 1 from above — the parsing code is embedded in the script.

- Managed mode:

  If you specify it exactly once, you get the case 2 from above — parsing code is in a separate file, but both files contain `Argbash` directives.

- Decoupled mode:

  If you specify twice, you get the case 3 from above — parsing code is in a separate file, the script includes it without any magic involved. This also means that the *brackets matching limitation* doesn't apply to you.

There is also a `--mode` option you can use to tune the balance between parsing features and complexity of the generated code.

- `default`: Assume the standard `Argbash` behavior. Check the documentation out to find out what that means.
- `full`: Maximize script features. * The long option and the corresponding value may be separated by whitespace or by the equal sign. * Variables corresponding to every positional argument is declared (.. seealso::*_declare_pos*).
- `minimal`: Make the code as simple as possible, which means: * The long option and the corresponding value may be separated only by whitespace.

Argbash

So, you have a template and now it is time to (re)generate a shell script from it!

Parsing code and script body together

Assuming that you have created a template file `my-template.m4`, you simply run `argbash` over the script*[0]:

```
argbash my-template.m4 -o my-script.sh
```

If you want to regenerate a new version of `my-script.sh` after you have modified its template section, you can run

```
argbash my-script.sh -o my-script.sh
```

as the script can deal with input and output being the same file.

Separate file for parsing with assistance

You have two files, let's say it is a `my-parsing.m4` and `my-script.sh`. The `my-parsing.m4` file contains just the template section of `my-script.sh`. Then, you add a very small template code to `my-script.sh` at the beginning:

```
# DEFINE_SCRIPT_DIR
# INCLUDE_PARSING_CODE([my-parsing.sh])
# ARGBASH_GO
```

[0] `m4` is the file extension used for the `M4` language, but we use the `m4sugar` language extension built on top of it.

```
# [ <-- needed because of Argbash

# HERE GOES THE SCRIPT BODY

# ] <-- needed because of Argbash
```

i.e. you add those three lines with definitions and you enclose the script in square brackets.

Finally, you just make sure that `my-script.sh` and `my-parsing.m4` are next to each other and run

```
argbash my-script.sh -o my-script.sh
```

which finds `my-parsing.m4` (it would find `my-parsing.sh` too) and generates new `my-parsing.sh` and `my-script.sh` that you can use right away. If both `my-parsing.m4` and `my-parsing.sh` are found, the more recent one is used to generate the `my-parsing.sh`.

Separate file for parsing

If you want/have to take care of including the parsing code yourself, just make sure you do it in the script — for example:

```
source $(dirname $0)/my-parsing.sh

# HERE GOES THE SCRIPT BODY
```

Then, you just generate `my-parsing.sh` using `--library` option:

```
argbash my-parsing.m4 -o my-parsing.sh --library
```

Commented output

You can call `argbash` in commented mode, when the generated code is commented, so you can run through the generated code and understand the big picture fast.

To generate code with those comments, just call `argbash` with the according switch:

```
argbash my-parsing.m4 -c -o my-parsing.sh
```

API changes support

The API of the `Argbash` project may change. This typically means that

- names, parameters or effect of macros change, or
- parsed arguments are exposed differently

in a way that is not compatible with the previous API.

In case that you regenerate a script, `argbash` is able to deduce that it has been created with another version of `Argbash` and warns you. In that case, you can use a `argbash-xtoy` script, where `x` is the version of `Argbash` your script is written for and `y` is version of `Argbash` you use now.

To upgrade your script from `Argbash` version 1 to 2, you simply invoke:

```
argbash-1to2 my-script.sh -o my-script.sh
```

You can use the utility to convert scripts as well as `.m4` templates.

Warning: Always back your scripts up and perform diff between the output and the original after using `argbash-xtoy`.

API 2

Parsed arguments were exposed as lowercase (`_ARG_LONG_OPTION` became `_arg_long_option`). The change was motivated by effort to comply to bash standard variable naming convention[1],[2].

Examples

Templates

Minimal example

Let's call minimal example a script that accepts some arguments and prints their values. Let's consider a positional, optional, optional boolean, `--version` and `--help` arguments with parsing code embedded in the script. First of all, we can generate the template using `argbash-init`. Then, we will edit it and add the script body.

First of all, we go examine `argbash-init` help — either by running `argbash-init -h` or *looking into the documentation*. We find out that we can have `argbash-init` generate the positional, optional arguments and help, so we go ahead:

```
bin/argbash-init --pos positional-arg --opt option --opt-bool print minimal.m4
```

The output of `argbash-init` looks like this:

```
#!/bin/bash

# m4_ignore(
echo "This is just a script template, not the script (yet) - pass it to 'argbash' to␣
↪fix this." >&2
exit 11  #)Created by argbash-init v2.5.0
# ARG_OPTIONAL_SINGLE([option], , [<option's help message goes here>])
# ARG_OPTIONAL_BOOLEAN([print], , [<print's help message goes here>])
# ARG_POSITIONAL_SINGLE([positional-arg], [<positional-arg's help message goes here>],
↪ )
# ARG_HELP([<The general help message of my script>])
# ARGBASH_GO

# [ <-- needed because of Argbash

echo "Value of --option: $_arg_option"
echo "print is $_arg_print"
echo "Value of positional-arg: $_arg_positional_arg"
```

[1] Unix StackExchange
[2] Google bash styleguide

```
# ] <-- needed because of Argbash
```

We add useful information and the line with the `--version` macro (by looking it up in the API docs) and the template finally looks better. Plus, we append the actual script body to the template:

```
#!/bin/bash

# m4_ignore(
echo "This is just a script template, not the script (yet) - pass it to 'argbash' to␣
↪fix this." >&2
exit 11  #)Created by argbash-init v2.5.0
# ARG_OPTIONAL_SINGLE([option], o, [A option with short and long flags and default],␣
↪[boo])
# ARG_OPTIONAL_BOOLEAN([print], , [A boolean option with long flag (and implicit␣
↪default: off)])
# ARG_POSITIONAL_SINGLE([positional-arg], [Positional arg description], )
# ARG_HELP([This is a minimal demo of Argbash potential])
# ARG_VERSION([echo $0 v0.1])
# ARGBASH_SET_INDENT([  ])
# ARGBASH_GO

# [ <-- needed because of Argbash

if [ "$_arg_print" = on ]
then
  echo "Positional arg value: '$_arg_positional_arg'"
  echo "Optional arg '--option' value: '$_arg_option'"
else
  echo "Not telling anything, print not requested"
fi

# ] <-- needed because of Argbash
```

Here, we can notice multiple notable things:

1. `argbash-init` has produced code that warn us if we treat the template as a script (i.e. if we execute it). This code will not be in the final script — it will disappear as we pass the template to `argbash`.
2. Definitions of arguments are placed before the script body. From `bash` point of view, they are commented out, so the "template" can be a syntactically valid script.
3. You access the values of argument `foo-bar` as `$_arg_foo_bar` etc. (this is covered more in-depth in *Using parsing results*).

So let's try the script in action! We have to generate it first by passing the template to `argbash`:

```
./../bin/argbash -o ../resources/examples/minimal.sh ../resources/examples/minimal.m4
```

This has produced the code *we can observe below* (notice that the leading "this is not a script error" lines have disappeared). Let's see what happens when we pass the `-h` option:

```
resources/examples/minimal.sh -h

This is a minimal demo of Argbash potential
Usage: ../resources/examples/minimal.sh [-o|--option <arg>] [--(no-)print] [-h|--
↪help] [-v|--version] <positional-arg>
        <positional-arg>: Positional arg description
        -o,--option: A option with short and long flags and default (default: 'boo')
```

```
        --print,--no-print: A boolean option with long flag (and implicit default:␣
↪off) (off by default)
        -h,--help: Prints help
        -v,--version: Prints version
```

OK, so it seems that passing it one (mandatory) positional arg will do the trick:

```
resources/examples/minimal.sh foo -o bar

Not telling anything, print not requested
```

Oops, we have forgot to turn print on! Let's fix that...

```
resources/examples/minimal.sh foo -o bar --print

Positional arg value: 'foo'
Optional arg '--option' value: 'bar'
```

Separating the parsing code

Let's take a look at a script that takes filename as the only positional argument and prints size of the corresponding file. The caller can influence the unit of display using optional argument `--unit`. This script is a bit artificial, but hang on — we will try to use it from within a wrapping script.

This time, we will *separate the parsing code and the script itself*. The parsing code will be in the `simple-parsing.sh` file and the script then in `simple.sh`.

Note: This is the manual approach. A simpler way would be calling `argbash-init` in the *managed or decoupled mode* — it will create the basic templates as in the previous example.

The template for the script's parsing section is really simple. Below are the sole contents of `simple-parsing.m4` file:

```
#!/bin/bash

# ARG_POSITIONAL_SINGLE([filename])
# ARG_OPTIONAL_SINGLE([unit], u, [What unit we accept (b for bytes, k for kibibytes,␣
↪M for mebibytes)], b)
# ARG_VERSION([echo $0 v0.1])
# ARG_OPTIONAL_BOOLEAN(verbose)
# ARG_HELP([This program tells you size of file that you pass to it in chosen units.])
# ARGBASH_SET_INDENT([  ])
# ARGBASH_GO
```

Then, let's take a look at the script's template body (i.e. the `simple.m4` file):

```
#!/bin/bash

# DEFINE_SCRIPT_DIR()
# INCLUDE_PARSING_CODE([simple-parsing.sh])
# ARGBASH_GO

# [ <-- needed because of Argbash

# Now we take the parsed data and assign them no nice-looking variable names,
```

```
# sometimes after a basic validation
verbose=$_arg_verbose
unit=$_arg_unit

test -f $_arg_filename || { echo "Filename $_arg_filename doesn't seem to belong to a␣
↪file"; exit 1; }
filename="$_arg_filename"

if [ $verbose = on ]
then
  _b="bytes (B)"
  _kb="kibibytes (kiB)"
  _mb="mebibytes (MiB)"
else
  _b="B"
  _kb="kiB"
  _mb="MiB"
fi

size_bytes=$(wc -c "$filename" | cut -f 1 -d ' ')

test "$unit" = b && echo $size_bytes $_b && exit 0

size_kibibytes=$(($size_bytes / 1024))
test "$unit" = k && echo $size_kibibytes $_kb && exit 0

size_mebibytes=$(($size_kibibytes / 1024))
test "$unit" = M && echo $size_mebibytes $_mb && exit 0

test "$verbose" = on && echo "The unit '$unit' is not supported"
exit 1

# ] <-- needed because of Argbash
```

We obtain the script from the template by running `argbash` over it — it detects the parsing template and interconnects those two.

```
argbash simple.m4 -o simple.sh
```

In other words, it will examine the `simple.m4` template, finding out that there is the *INCLUDE_PARSING_CODE* macro. If the parsing template (in our case `simple-parsing.m4` or `simple-parsing.sh`) is found, a parsing script is produced out of it (otherwise, an error occurs). Finally, the `simple.sh` script is (re)generated — basically only the source directive is added, see those few lines:

```
#!/bin/bash

# DEFINE_SCRIPT_DIR([])
# INCLUDE_PARSING_CODE([simple-parsing.sh])
# ARGBASH_GO()
# needed because of Argbash --> m4_ignore([
### START OF CODE GENERATED BY Argbash v2.5.0 one line above ###
# Argbash is a bash code generator used to get arguments parsing right.
# Argbash is FREE SOFTWARE, see https://argbash.io for more info

# OTHER STUFF GENERATED BY Argbash
script_dir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)" || die "Couldn't determine␣
↪the script's running directory, which probably matters, bailing out" 2
. "$script_dir/simple-parsing.sh"  # '.' means 'source'
```

```
### END OF CODE GENERATED BY Argbash (sortof) ### ])
```

When invoked with the help option, we get:

```
resources/examples/simple.sh -h

This program tells you size of file that you pass to it in chosen units.
Usage: ../resources/examples/simple.sh [-u|--unit <arg>] [-v|--version] [--(no-
↪)verbose] [-h|--help] <filename>
        -u,--unit: What unit we accept (b for bytes, k for kibibytes, M for␣
↪mebibytes) (default: 'b')
        -v,--version: Prints version
        -h,--help: Prints help
```

It will work as long as the parsing code's location (next to the script itself) doesn't change:

Wrapping scripts

We will show how to write a script that accepts a list of directories and a glob pattern, combines them together, and displays size of files using the previous script. In order to do this, we will introduce positional argument that can accept an arbitrary amount of values and we will also use the wrapping functionality that `Argbash` possesses.

We want to wrap the `simple.m4` (or `simple.sh`). However, since the script doesn't include any command definitions, we have to wrap the parsing component `simple-parsing.`. The script's template is still quite simple:

```
#!/bin/bash

# DEFINE_SCRIPT_DIR
# ARG_POSITIONAL_INF([directory], [Directories to go through], 1)
# ARG_OPTIONAL_SINGLE([glob], , [What files to match in the directory], [*])
# ARGBASH_WRAP([simple-parsing], [filename])
# ARG_HELP([This program tells you size of specified files in given directories in␣
↪units you choose.])
# ARGBASH_SET_INDENT([  ])
# ARGBASH_GO

# [ <-- needed because of Argbash

script="$script_dir/simple.sh"
test -f "$script" || { echo "Missing the wrapped script, was expecting it next to me,␣
↪in '$script_dir'."; exit 1; }

for directory in "${_arg_directory[@]}"
do
  test -d "$directory" || die "We expected a directory, got '$directory', bailing out.
↪"
  printf "Contents of '%s' matching '%s':\n" "$directory" "$_arg_glob"
  for file in "$directory"/$_arg_glob
  do
    test -f "$file" && printf "\t%s: %s\n" "$(basename "$file")" "$("$script" "${_
↪args_simple_parsing_opt[@]}" "$file")"
  done
done

# ] <-- needed because of Argbash
```

The `simple-parsing` in *ARGBASH_WRAP* argument refers to the parsing part of the script from the previous section. Remember, we say that we are wrapping a script, but in fact, we just inherit a subset of its arguments and the actual wrapping (i.e. calling the wrapped script) is still up to us, although it is made easy by a great deal. The `filename` argument means that our wrapping script won't "inherit" the `filename` argument — that's correct, it is the wrapping script that decides what arguments make it to the wrapped one.

When invoked with the help option, we get:

```
resources/examples/simple-wrapper.sh -h

This program tells you size of specified files in given directories in units you␣
↪choose.
Usage: ../resources/examples/simple-wrapper.sh [--glob <arg>] [-u|--unit <arg>] [--
↪(no-)verbose] [-h|--help] <directory-1> [<directory-2>] ... [<directory-n>] ...
        <directory>: Directories to go through
        --glob: What files to match in the directory (default: '*')
        -u,--unit: What unit we accept (b for bytes, k for kibibytes, M for␣
↪mebibytes) (default: 'b')
        -h,--help: Prints help
```

So let's try it!

```
resources/examples/simple-wrapper.sh --glob '*.m4' ../src ../resources/examples -u k

Contents of '../src' matching '*.m4':
        argbash-1to2.m4: 1 kiB
        argbash-init.m4: 4 kiB
        argbash-lib.m4: 0 kiB
        argbash.m4: 7 kiB
        argument_value_types.m4: 4 kiB
        constants.m4: 0 kiB
        default_settings.m4: 0 kiB
        env_vars.m4: 1 kiB
        function_generators.m4: 5 kiB
        list.m4: 5 kiB
        output.m4: 0 kiB
        output-standalone.m4: 0 kiB
        progs.m4: 2 kiB
        stuff.m4: 54 kiB
        utilities.m4: 5 kiB
        value_validators.m4: 5 kiB
Contents of '../resources/examples' matching '*.m4':
        minimal.m4: 0 kiB
        minimal-raw.m4: 0 kiB
        simple.m4: 0 kiB
        simple-parsing.m4: 0 kiB
        simple-standalone.m4: 0 kiB
        simple-wrapper.m4: 0 kiB
```

Source

Minimal example

Let's examine the generated *minimal example script* (the contents are displayed below).

We can see that the header still contains the `Argbash` definitions. They are not there for reference only, you can actually change them and re-run `Argbash` on the *script* again to get an updated version! Yes, you don't need the `.m4`

template, the `.sh` file serves as a template that is equally good!

Others

Utilities

You want to write a script that does argument parsing and don't like `Argbash` or you just want to find out how awesome it really is? Then read through this list of influences and/or alternatives!

- Python `argparse`: The main inspiration: https://docs.python.org/3/library/argparse.html
  - Pros:
    - Works really well.
    - Has more features.
  - Cons:
    - It is Python, we are `bash`.
    - It has a strict value restriction turned on (analogous to having `ARG_RESTRICT_VALUES([no-option-all])` in every script) that can't be switched off.
  - Argbash says:
    - We handle the boolean options better.
    - We have the *awesome wrapping functionality*.
- `bash` — `shflags`: The `bash` framework for argument parsing: https://github.com/kward/shflags
  - Pros:
    - It works great on Linux.
  - Cons:
    - Doesn't work with Windows `bash`
    - Doesn't support long options on OSX.
  - Argbash says:
    - We work the same on all platforms that have `bash`.
- `getopt`: Eternal utility for parsing command-line. This is what powers `shflags`.
  - Pros:
    - The GNU version can work with long and short optional arguments.
  - Cons:
    - Its use is discouraged — it seems to have some issues, you still need to deal with positional arguments by other means.
- `getopts`: `bash` builtin for parsing command-line.
  - Pros:
    - Being included with `bash`, it behaves the same on all platforms.
  - Cons:

        * Supports only short optional arguments.

- `EasyOptions`: `Ruby` utility with a `bash` interface as well as its pure `bash` implementation: https://github.com/renatosilva/easyoptions
  - Pros:
    * Very simple to use.
    * Very elegant.
  - Cons:
    * You have to distribute `easyoptions.sh` with your script.
    * The library itself provides only very basic functionality.
    * The project does not seem to have any tests (as of 07/2016).
  - Argbash says:
    * We have more of nice features.
    * We offer an option to produce battery-included scripts.
    * Our declarations as not so elegant, but they are not bad either.
    * The parsing part of the script generated by `Argbash` is only as complex as necessary.
- `bash-modules` - module `argument`: `bash-modules` aims to become a standard `bash` framework and comes with an argument-parsing treat: https://github.com/vlisivka/bash-modules
  - Pros:
    * Seems to have nice features.
  - Cons:
    * `bash-modules` have to be available at run-time to run the script.
    * The documentation is poor (as of 07/2016).
  - Argbash says:
    * We have good documentation with examples.
    * We offer an option to produce battery-included scripts.

Learning resources

Do you want to write the argument-parsing part of your script yourself or you want to improve `Argbash`? Then read through this list of high-quality learning resources!

- Best practices in argument parsing: http://www.shelldorado.com/goodcoding/cmdargs.html Don't miss the list of short options and their common meaning!
- StackOverflow thread about argument parsing: http://stackoverflow.com/questions/192249/how-do-i-parse-command-line-arguments-in-bash It was mainly this thread which inspired `Argbash`. There are plenty of recipes and suggestions available. If you are a happy user of `Argbash`, consider upvoting the answer that promotes it since more people know `Argbash`, the better for them.
- Argument parsing for dummies: http://wiki.bash-hackers.org/scripting/posparams A short and nice-looking introduction with all basics covered (wiki).
- The POSIX conventions — the reason why `getopts` are still mentioned in bash learning resources.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search