Arkage Parser Convention (EN)

Parsing and interpreting command lines: standards and guidelines.

Version 1.0, August 19, 2024 - By Téo Conan

From https://devolution.studio/blog/en/arkage-parser-convention/#

Edit me on GitHub : https://github.com/Devolution-Studio/devolution.studio/tree/master/public/articles

Introduction

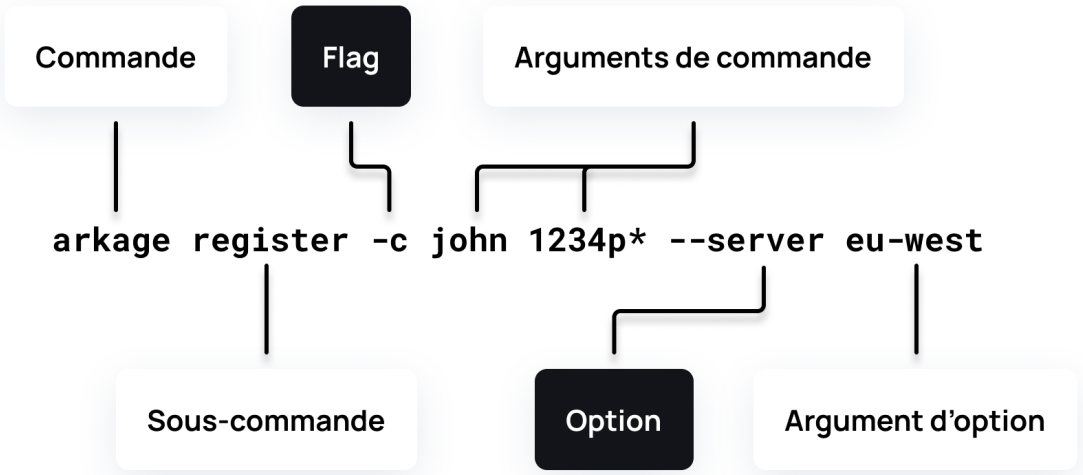
This document aims to establish a rigorous framework for the design and interpretation of command-line interfaces (CLIs). By defining a strict set of rules regarding syntax, data types, and command structures, this convention seeks to:

- **Improve the readability and consistency** of CLI interfaces, thereby facilitating their use by end-users.
- **Reduce interpretation errors** and ambiguities, ensuring reliable command processing.
- **Promote interoperability** between different CLI tools by adopting common conventions.
- **Serve as a reference** for developers looking to create new CLI tools.

This convention will delve into the constituent elements of a command line and how to perform proper syntactic analysis. It will also define supported data types and interpretation mechanisms. Practical examples and diagrams will illustrate the application of these rules in real-world scenarios.

This convention can be particularly useful for developers who wish to create their own CLI tools.

Foundations of Interpretation



Process a string of characters into a specific action is the challenge of command-line interpretation. Behind the apparent simplicity of a command lie complex mechanisms of syntactic analysis, raising numerous questions:

- How do you associate the correct arguments with options/commands?
- How do you verify the validity of the data provided by the user?
- How do you handle errors?

This chapter explores the various interpretation techniques, pitfalls to avoid, and best practices to adopt for creating reliable and robust CLI tools.

Naming convention

Within this technical convention, we will adopt the `kebab-case` **naming convention** for options, commands, and subcommands. This choice ensures maximum consistency and readability in defining our commands and options.

Kebab-case consists of writing words in lowercase, separated by hyphens (`-`). For example: `create-user`, `list-files`, `set-config`.

This choice is motivated by its improved **readability** due to the clear separation of words and its compatibility with the syntax of long options on the command line (e.g., `--create-user`). This convention is also **widely adopted** in many areas of web development, making it familiar to most developers.

Types

Many CLI tools use auto-completion to suggest possible options and values. Argument typing is crucial to provide relevant suggestions and reduce input errors.

While basic types like `strings`, `numbers`, and `booleans` are well-known, it can be beneficial to delve deeper into expected types to offer a better user experience. Here is a non-exhaustive list of interesting types to consider:

Nom	type	Description
String	`string`	Careful, char `"` is reserved to wrap the `string`, but you can escape it with `"\""`
Char	`char`	Any single char, char `'` is reserved to wrap the `string`, but you can escape it with `'\''`
Boolean	`bool`	Able to parse a `TrUe` as `true` for example
File	`file`	Can check the existence of the file and its extension when the command is launched (optional)
Folder	`folder`	Can check the existence of the directory when the command is launched (optional)
Number	`number`	`double` type, can be signed
Url	`url`	Able to recognize the format of a `URL`
Enumeration	`enum`	A precise list of possible values
Array	`Array<T>`	Special case, can only be used for command arguments as the last argument

Command

The first word of a command specifies the program to be executed. This program, often referred to as an **executable** or **binary**, is a file containing the instructions to be carried out. The operating system searches for executables in specific directories, defined in your configuration (e.g., in the `.bashrc` file). The command is the entry point of your program.

```
~ which git
/usr/bin/git
```

Sub-command

Subcommands are specific commands associated with a main program. They allow for organizing a program's functionalities into finer modules.

For example, `git` uses subcommands to perform operations such as `checkout`, `merge`, or `commit`. These subcommands are actually full-fledged commands that are executed within the context of the command.

Note: Not all programs use subcommands. Some programs perform a single task and take their arguments directly.

```
# For example, git has sub-commands :
git checkout
git merge
git commit

# But ping command don't have any, it directly takes an argum
ping www.google.com
```

Argument overloading

Subcommands, much like in Java, can offer multiple possibilities and combinations of arguments for a single subcommand. This is referred to as having **multiple signatures for a subcommand**.The rules are similar to those in Java: signatures are declared for a single subcommand, and signatures must have a different number of arguments or different types. We will explore a practical example later in this document.

Options

What is an option ?

An option is a special argument that modifies a command's behavior. It is preceded by two hyphens (`--`) to distinguish it from other arguments. Its purpose is to provide additional information to the command so that it executes in a specific way.

An option always requires a **typed value** to function. For example:

```
--output ./my_file.txt
# Here, --output is the option and ./my_file.txt is
# the value indicating where to save the output
```

```
--output 4
# The option expected a file path, a number doesn't make sens
```

Global options

Some options are defined at the main program level and are therefore **available for all subcommands**. These are called **global options**. Other options are specific to a particular subcommand and are therefore called **local options**.

For example, the `--verbose` option could be **global** and available throughout the program, while the `--output` option would be specific to the subcommand being executed.

Boolean Options: A Special Case

Although an option generally requires a value to function, there's a scenario where specifying a value isn't necessary. This scenario arises when an option expects a Boolean value:

- **Default Value**: By default, a Boolean option is considered `false`. If you don't specify it, the default value will be assumed.
- **Activation**: To activate a Boolean option, simply include it in the command, and its value will be set to `true`.

Explicitly Disabling a Boolean Option

In certain cases, you might want to explicitly indicate that you wish to disable a Boolean option, for better user understanding. To achieve this, many commands offer a specific syntax, often by adding the prefix `no-` before the option name:

```
--verbose # Option "verbose" will be equals to true
--no-logs # Option "no-logs" will be equals to true
--logs false # Unable to set an arguments to an boolean optio
```

Naming convention reminder

As explained at the beginning of this document, to improve the clarity and maintainability of scripts, it is recommended to use `kebab-case` notation for

naming options. This convention consists of separating words with hyphens, using only lowercase letters.

Example:

```
--output-file --set-credentials --no-logs
--outputFile # Bad format
```

Rules to remember :

- **Uniqueness:** Each option must have a unique name within a command.
- **Data type:** Options can have different data types (string, number, boolean, ...).
- **Default value:** Boolean options have a default value (`false`).
- **Kebab-case:** Options must use the kebab-case convention.

Flags

To simplify command input, many commands offer short options, often called flags. A flag is represented by a single character preceded by a single hyphen (`-`).

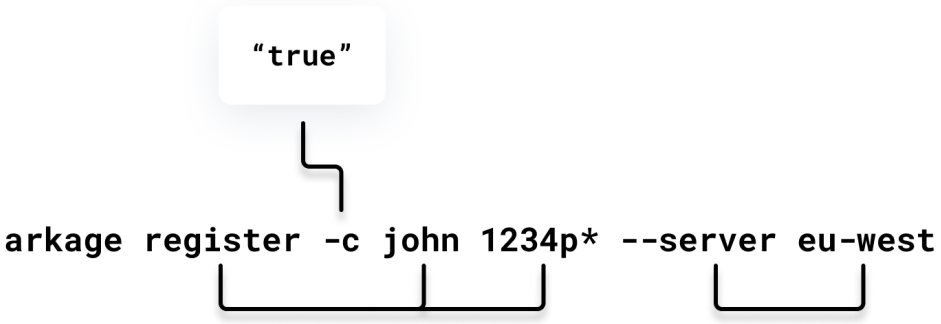
Flags have the particularity of being able o accumulate. It is common to be able to combine multiple flags into a single argument. For example, `ls -al` is equivalent to `ls -a -l`.

Flags are often shortcuts for long options, which are generally more explicit.

Rules to remember:

- **Uniqueness:** Each flag must be unique within a command.
- **Optionality:** Each option does not necessarily require a flag; it is simply a "shortcut".
- **Explicitness:** A flag should have a letter that is explicit about the option to which it is attached, for example, `-o` for `--output`.

Arguments



Arguments are data provided to a command or subcommand to specify the elements on which it should operate. They are closely related to options and flags but are **distinguished by their type and their position** in the command line.

To ensure the consistency and robustness of a command, each argument must correspond to a referenced data type. This typing allows for the validation of user inputs and prevents runtime errors. If necessary, it will allow for the return of detailed errors.

Arguments array

Some commands may accept a variable number of arguments. To represent this concept, the `Array<T>` type must be used. An argument of array type can contain zero, one or more elements of type `T`.

Because of their variable nature, array type arguments **must be placed in the last position** in the list of arguments of a command. This convention allows for command line parsing and enables the correct interpretation of values provided by the user.

Here is an example:

```
# Acceptable arguments are : number, Array<string>
arkage ping 4 us uk eu-west eu-est
```

Ce processus est très proche des `variadic` en PHP, cependant il n'est possible d'avoir un type `Array<T>` uniquement **pour les arguments de commandes / sous-commande**. Ce type n'est pas utilisable pour une option pour des raisons de lisibilité.

Examples and Practical Cases

Complete Command Declaration

Global Options

For our practical case, let's consider a fictitious command that we will name `arkage`. As a first step, we can declare our command's global options:

Option	Flag	Type	Default	Description
`--verbose`	`-v`	Boolean	`true`	Enable verbose mode
`--lang`	`-l`	`string`	`en`	Change language
`--en`		Boolean	`true`	Change language for `en`
`--fr`		Boolean	`true`	Change language for `fr`

By deduction, we understand that it would be redundant to put both the `--en` and `--fr` options on the same command line. However, from a syntactic point of view, **nothing prevents us from doing so**. It will be up to the program to decide which language to choose or if it wants to raise an error.

```
arkage help --fr --en # Valid syntax
```

Sub-commands

Once the global options have been declared, we need subcommands to have something to execute. To declare a subcommand, it is important to declare its name (always in `kebab-case`), its options, and its possible arguments.

Sub-commands	Options	Flag	Type	Sub-command arguments
`help`				No arguments
`ping`				`int`, `Array<string>`
`register`	`--credentials`	`-c`	Boolean	`string`, `string` or `filepath`
	`--key`	`-k`	Boolea	
	`--server`	`-s`	`string`	

Available sub-commands are :

- `help`
- `ping`
- `register`
- `start`
- `exit`

`help` command don't takes any arguments but can takes these options :

- `--lang` or `-l` with a string arguments behind
- `--fr` is a boolean
- `--en` is a boolean

For the sub-command `ping`, this one don't takes any option but can take an infinity of `string` arguments

```
arkage ping 4 us uk eu-west eu-est
```

The first argument specifies the number of attempts, and the rest specify the servers to reach. The arguments are converted back into an array for processing. Here is the declaration of the arguments for this subcommand:

- Attempts : `int`
- Servers : `Array<string>`

An `Array` must always be the last argument because it can contain an infinite number of values.

Global options

```
arkage register -c teo@arkage.com 1234aze*
arkage register -k ./id_rsa
arkage register -s eu-west -k ./id_rsa -v --lang en
# Error, but on the command processing side because -c and -k
```