DOCOPT

1. Command-line interface description language

docopt helps you:

- define the interface for your command-line app, and
- automatically generate a parser for it.

docopt is based on conventions that have been used for decades in help messages and man pages for describing a program's interface. An interface description in **docopt** *is* such a help message, but formalized. Here is an example:

```
Naval Fate.

Usage:
  naval_fate ship new <name>...
  naval_fate ship <name> move <x> <y> [--speed=<kn>]
  naval_fate ship shoot <x> <y>
  naval_fate mine (set|remove) <x> <y> [--moored|--drifting]
  naval_fate -h | --help
  naval_fate --version

Options:
  -h --help     Show this screen.
  --version     Show version.
  --speed=<kn>  Speed in knots [default: 10].
  --moored      Moored (anchored) mine.
  --drifting    Drifting mine.
```

The example describes the interface of executable **naval_fate**, which can be invoked with different combinations of *commands* (**ship**, **new**, **move**, etc.), *options* (**-h**, **--help**, **--speed=<kn>**, etc.) and positional arguments (**<name>**, **<x>**, **<y>**).

The example uses brackets "**[ ]**", parens "**( )**", pipes "**|**" and ellipsis "**...**" to describe *optional*, *required*, *mutually exclusive*, and *repeating* elements. Together, these elements form valid *usage patterns*, each starting with program's name **naval_fate**.

Below the usage patterns, there is a list of options with descriptions. They describe whether an option has short/long forms (**-h**, **--help**), whether an option has an argument (**--speed=<kn>**), and whether that argument has a default value (**[default: 10]**).

A **docopt** implementation will extract all that information and generate a command-line arguments parser, with the text of the interface description as the help message shown when the program is invoked with the **-h** or **--help** options.

2. Usage patterns

Text occuring between keyword **usage:** (case-*in*sensitive) and a *visibly* empty line is interpreted as list of usage patterns. The first word after **usage:** is interpreted as the program's name. Here is a minimal example for program that takes no command-line arguments:

```
Usage: my_program
```

Program can have several patterns listed with various elements used to describe the pattern:

```
Usage:
  my_program command --option <argument>
  my_program [<optional-argument>]
  my_program --another-option=<with-argument>
  my_program (--either-that-option | <or-this-argument>)
  my_program <repeating-argument> <repeating-argument>...
```

Each of the elements and constructs is described below. We will use the word "*word*" to describe a sequence of characters delimited by either whitespace, one of "**[]()|**" characters, or "**...**".

2.1. <argument> ARGUMENT

Words starting with "**<**", ending with "**>**" and upper-case words are interpreted as positional arguments.

```
Usage: my_program <host> <port>
```

2.2. -o --option

Words starting with one or two dashes (with exception of "**-**", "**--**" by themselves) are interpreted as short (one-letter) or long options, respectively.

- Short options can be "stacked" meaning that **`-abc`** is equivalent to **`-a -b -c`**.
- Long options can have arguments specified after space or equal "**`=`**" sign:
  `--input=ARG` is equivalent to **`--input ARG`**.
- Short options can have arguments specified after *optional* space:
  `-f FILE` is equivalent to **`-fFILE`**.

Note, writing **`--input ARG`** (as opposed to **`--input=ARG`**) is ambiguous, meaning it is not possible to tell whether **`ARG`** is option's argument or a positional argument. In usage patterns this will be interpreted as an option with argument *only* if a description (covered below) for that option is provided. Otherwise it will be interpreted as an option and separate positional argument.

There is the same ambiguity with the **`-f FILE`** and **`-fFILE`** notation. In the latter case it is not possible to tell whether it is a number of stacked short options, or an option with an argument. These notations will be interpreted as an option with argument *only* if a description for the option is provided.

2.3. command

All other words that do *not* follow the above conventions of **`--options`** or **`<arguments>`** are interpreted as (sub)commands.

2.4. [optional elements]

Elements (options, arguments, commands) enclosed with square brackets "**`[ ]`**" are marked to be *optional*. It does not matter if elements are enclosed in the same or different pairs of brackets, for example:

```
Usage: my_program [command --option <argument>]
```

is equivalent to:

```
Usage: my_program [command] [--option] [<argument>]
```

2.5. (required elements)

All elements are required by default, if not included in brackets "**`[ ]`**". However, sometimes it is necessary to mark elements as required explicitly with parens "**`( )`**". For example, when you need to group mutually-exclusive elements (see next section):

```
Usage: my_program (--either-this <and-that> | <or-this>)
```

Another use case is when you need to specify that *if one element is present, then another one is required*, which you can achieve as:

```
Usage: my_program [(<one-argument> <another-argument>)]
```

In this case, a valid program invocation could be with either no arguments, or with 2 arguments.

2.6. element|another

Mutually-exclusive elements can be separated with a pipe "**`|`**" as follows:

```
Usage: my_program go (--up | --down | --left | --right)
```

Use parens "**`( )`**" to group elements when *one* of the mutually exclusive cases is required. Use brackets "**`[ ]`**" to group elements when *none* of the mutually exclusive cases is required:

```
Usage: my_program go [--up | --down | --left | --right]
```

Note, that specifying several patterns works exactly like pipe "**`|`**", that is:

```
Usage: my_program run [--fast]
       my_program jump [--high]
```

is equivalent to:

```
Usage: my_program (run [--fast] | jump [--high])
```

2.7. element...

Use ellipsis "**`...`**" to specify that the argument (or group of arguments) to the left could be repeated one or more times:

```
Usage: my_program open <file>...
       my_program move (<from> <to>)...
```

You can flexibly specify the number of arguments that are required. Here are 3 (redundant) ways of requiring zero or more arguments:

```
Usage: my_program [<file>...]
       my_program [<file>]...
       my_program [<file> [<file> ...]]
```

One or more arguments:

```
Usage: my_program <file>...
```

Two or more arguments (and so on):

```
Usage: my_program <file> <file>...
```

2.8. [options]

"**`[options]`**" is a shortcut that allows to avoid listing all options (from list of options with descriptions) in a pattern. For example:

```
Usage: my_program [options] <path>

--all             List everything.
--long            Long output.
--human-readable  Display in human-readable format.
```

is equivalent to:

```
Usage: my_program [--all --long --human-readable] <path>

--all             List everything.
--long            Long output.
--human-readable  Display in human-readable format.
```

This can be useful if you have many options and all of them are applicable to one of patterns. Alternatively, if you have both short and long versions of options (specified in option description part), you can list either of them in a pattern:

```
Usage: my_program [-alh] <path>

-a, --all             List everything.
-l, --long            Long output.
-h, --human-readable  Display in human-readable format.
```

More details on how to write options' descriptions will follow below.

2.9. [--]

A double dash "**`--`**", when not part of an option, is often used as a convention to separate options and positional arguments, in order to handle cases when, e.g., file names could be mistaken for options. In order to support this convention, add "**`[--]`**" into your patterns before positional arguments.

```
Usage: my_program [options] [--] <file>...
```

Apart from this special meaning, "**`--`**" is just a normal command, so you can apply any previously-described operations, for example, make it required (by dropping brackets "**`[ ]`**")

2.10. [-]

A single dash "**`-`**", when not part of an option, is often used by convention to signify that a program should process **`stdin`**, as opposed to a file. If you want to follow this convention add "**`[-]`**" to your pattern. "**`-`**" by itself is just a normal command, which you can use with any meaning.

3. Option descriptions

Option descriptions consist of a list of options that you put below your usage patterns. It is optional to specify them if there is no ambiguity in usage patterns (described in the **`--option`** section above).

An option's description allows to specify:

- that some short and long options are synonymous,
- that an option has an argument,
- and the default value for an option's argument.

The rules are as follows:

Every line that starts with "`-`" or "`--`" (not counting spaces) is treated as an option description, e.g.:

```
Options:
  --verbose   # GOOD
  -o FILE     # GOOD
Other: --bad  # BAD, line does not start with dash "-"
```

To specify that an option has an argument, put a word describing that argument after a space (or equals "`=`" sign) as shown below. Follow either **`<angular-brackets>`** or **`UPPER-CASE`** convention for options' arguments. You can use a comma if you want to separate options. In the example below, both lines are valid, however it is recommended to stick to a single style.

```
-o FILE --output=FILE       # without comma, with "=" sign
-i <file>, --input <file>   # with comma, without "=" sign
```

Use two spaces to separate options with their informal description.

```
--verbose MORE text.    # BAD, will be treated as if verbose
                        # option had an argument MORE, so use
                        # 2 spaces instead
-q        Quit.         # GOOD
-o FILE   Output file.  # GOOD
--stdout  Use stdout.   # GOOD, 2 spaces
```

If you want to set a default value for an option with an argument, put it into the option's description, in the form **`[default: <the-default-value>]`**.

```
--coefficient=K  The K coefficient [default: 2.95]
--output=FILE    Output file [default: test.txt]
--directory=DIR  Some directory [default: ./]
```

4.

5. Implementations

`docopt` is available in numerous programming languages. Official implementations are listed under the .