ArgParse.jl Documentation

Release 0.3

Carlo Baldassi

October 06, 2015

This module allows the creation of user-friendly command-line interfaces to Julia programs: the program defines which arguments, options and sub-commands it accepts, and the `ArgParse` module does the actual parsing, issues errors when the input is invalid, and automatically generates help and usage messages.

Users familiar with Python's argparse module will find many similarities, but some important differences as well.

This package can be installed using the Julia package manager from the Julia command line:

```
julia> Pkg.add("ArgParse")
```

Quick overview and a simple example

First of all, the module needs to be loaded and imported:

```
using ArgParse
```

There are two main steps for defining a command-line interface: creating an `ArgParseSettings` object, and populating it with allowed arguments and options using either the macro `@add_arg_table` or the function `add_arg_table` (see *this section* (page 7) for the difference between the two):

```
s = ArgParseSettings()
@add_arg_table s begin
    "--opt1"
        help = "an option with an argument"
    "--opt2", "-o"
        help = "another option with an argument"
        arg_type = Int
        default = 0
    "--flag1"
        help = "an option without argument, i.e. a flag"
        action = :store_true
    "arg1"
        help = "a positional argument"
        required = true
end
```

In the macro, options and positional arguments are specified within a `begin...end` block, by one or more names in a line, optionally followed by a list of settings. So, in the above example, there are three options:

- the first one, `"--opt1"` takes an argument, but doesn't check for its type, and it doesn't have a default value
- the second one can be invoked in two different forms (`"--opt2"` and `"-o"`); it also takes an argument, but it must be of `Int` type (or convertible to it) and its default value is `0`
- the third one, `--flag1`, is a flag, i.e. it doesn't take any argument.

There is also only one positional argument, `"arg1"`, which is declared mandatory.

When the settings are in place, the actual argument parsing is performed via the `parse_args` function:

```
parsed_args = parse_args(ARGS, s)
```

The parameter `ARGS` can be omitted. In case no errors are found, the result will be a `Dict{String,Any}` object. In the above example, it will contain the keys `"opt1"`, `"opt2"`, `"flag1"` and `"arg1"`, so that e.g. `parsed_args["arg1"]` will yield the value associated with the positional argument.

(The `parse_args` function also accepts an optional `as_symbols` keyword argument: when set to `true`, the result of the parsing will be a `Dict{Symbol,Any}`, which can be useful e.g. for passing it as the keywords to a Julia function.)

Putting all this together in a file, we can see how a basic command-line interface is created:

```
using ArgParse

function parse_commandline()
    s = ArgParseSettings()

    @add_arg_table s begin
        "--opt1"
            help = "an option with an argument"
        "--opt2", "-o"
            help = "another option with an argument"
            arg_type = Int
            default = 0
        "--flag1"
            help = "an option without argument, i.e. a flag"
            action = :store_true
        "arg1"
            help = "a positional argument"
            required = true
    end

    return parse_args(s)
end

function main()
    parsed_args = parse_commandline()
    println("Parsed args:")
    for (arg,val) in parsed_args
        println("  $arg  =>  $val")
    end
end

main()
```

If we save this as a file called `myprog1.jl`, we can see how a `--help` option is added by default, and a help message is automatically generated and formatted:

```
$ julia myprog1.jl --help
usage: myprog1.jl [--opt1 OPT1] [-o OPT2] [--flag1] [-h] arg1

positional arguments:
  arg1             a positional argument

optional arguments:
  --opt1 OPT1      an option with an argument
  -o, --opt2 OPT2  another option with an argument (type: Int64,
                   default: 0)
  --flag1          an option without argument, i.e. a flag
  -h, --help       show this help message and exit
```

Also, we can see how invoking it with the wrong arguments produces errors:

```
$ julia myprog1.jl
required argument arg1 was not provided
usage: myprog1.jl [--opt1 OPT1] [-o OPT2] [--flag1] [-h] arg1
```

```
$ julia myprog1.jl somearg anotherarg
too many arguments
usage: myprog1.jl [--opt1 OPT1] [-o OPT2] [--flag1] [-h] arg1

$ julia myprog1.jl --opt2 1.5 somearg
invalid argument: 1.5 (conversion to type Int64 failed; you may need to overload ArgParse.parse_item;
                  the error was: ArgumentError("invalid base 10 digit '.' in \"1.5\""))
usage: myprog1.jl [--opt1 OPT1] [-o OPT2] [--flag1] arg1
```

When everything goes fine instead, our program will print the resulting `Dict`:

```
$ julia myprog1.jl somearg
Parsed args:
  arg1  =>  somearg
  opt2  =>  0
  opt1  =>  nothing
  flag1  =>  false

$ julia myprog1.jl --opt1 "2+2" --opt2 "4" somearg --flag
Parsed args:
  arg1  =>  somearg
  opt2  =>  4
  opt1  =>  2+2
  flag1  =>  true
```

From these examples, a number of things can be noticed:

- `opt1` defaults to `nothing`, since no `default` setting was used for it in `@add_arg_table`
- `opt1` argument type, begin unspecified, defaults to `Any`, but in practice it's parsed as a string (e.g. `"2+2"`)
- `opt2` instead has `Int` argument type, so `"4"` will be parsed and converted to an integer, an error is emitted if the conversion fails
- positional arguments can be passed in between options
- long options can be passed in abbreviated form (e.g. `--flag` instead of `--flag1`) as long as there's no ambiguity

CHAPTER 2

The `parse_args` function

`parse_args`(*[args], settings; as_symbols::Bool = false*)

This is the central function of the `ArgParse` module. It takes a `Vector` of arguments and an `ArgParseSettings` objects (see *this section* (page 9)), and returns a `Dict{String,Any}`. If `args` is not provided, the global variable `ARGS` will be used.

When the keyword argument `as_symbols` is `true`, the function will return a `Dict{Symbol,Any}` instead.

The returned `Dict` keys are defined (possibly implicitly) in `settings`, and their associated values are parsed from `args`. Special keys are used for more advanced purposes; at the moment, one such key exists: `%COMMAND%` (`_COMMAND_` when using `as_symbols=true`; see *this section* (page 19)).

Arguments are parsed in sequence and matched against the argument table in `settings` to determine whether they are long options, short options, option arguments or positional arguments:

- long options begin with a double dash `"--"`; if a `'='` character is found, the remainder is the option argument; therefore, `["--opt=arg"]` and `["--opt", "arg"]` are equivalent if `--opt` takes at least one argument. Long options can be abbreviated (e.g. `--opt` instead of `--option`) as long as there is no ambiguity.
- short options begin with a single dash `"-"` and their name consists of a single character; they can be grouped togheter (e.g. `["-x", "-y"]` can become `["-xy"]`), but in that case only the last option in the group can take an argument (which can also be grouped, e.g. `["-a", "-f", "file.txt"]` can be passed as `["-affile.txt"]` if `-a` does not take an argument and `-f` does). The `'='` character can be used to separate option names from option arguments as well (e.g. `-af=file.txt`).
- positional arguments are anything else; they can appear anywhere.

The special string `"--"` can be used to signal the end of all options; after that, everything is considered as a positional argument (e.g. if `args = ["--opt1", "--", "--opt2"]`, the parser will recognize `--opt1` as a long option without argument, and `--opt2` as a positional argument).

The special string `"-"` is always parsed as a positional argument.

The parsing can stop early if a `:show_help` or `:show_version` action is triggered, or if a parsing error is found.

Some ambiguities can arise in parsing, see *this section* (page 29) for a detailed description of how they're solved.

CHAPTER 3

Settings overview

The `ArgParseSettings` object contains all the settings to be used during argument parsing. Settings are divided in two groups: general settings and argument-table-related settings. While the argument table requires specialized functions such as `add_arg_table` to be defined and manipulated, general settings are simply object fields (most of them are `Bool` or `String`) and can be passed to the constructor as keyword arguments, or directly set at any time.

CHAPTER 4

General settings

This is the list of general settings currently available:

- `prog` (default = `""`): the name of the program, as displayed in the auto-generated help and usage screens. If left empty, the source file name will be used.
- `description` (default = `""`): a description of what the program does, to be displayed in the auto-generated help-screen, between the usage lines and the arguments description. It will be automatically formatted, but you can force newlines by using two consecutive newlines in the string, and manually control spaces by using non-breakable spaces (the character `'\ua0'`).
- `epilog` (default = `""`): like `description`, but will be displayed at the end of the help-screen, after the arguments description. The same formatting rules also apply.
- `usage` (default = `""`): the usage line(s) to be displayed in the help screen and when an error is found during parsing. If left empty, it will be auto-generated.
- `version` (default = `"Unknown version"`): version information. It's used by the `:show_version` action.
- `add_help` (default = `true`): if `true`, a `--help, -h` option (triggering the `:show_help` action) is added to the argument table.
- `add_version` (default = `false`): if `true`, a `--version` option (triggering the `:show_version` action) is added to the argument table.
- `autofix_names` (default = `false`): if `true`, will try to automatically fix the uses of dashes (`-`) and underscores (`_`) in option names and destinations: all underscores will be converted to dashes in long option names; also, associated destination names, if auto-generated (see *this_section* (page 15)), will have dashes replaced with underscores, both for long options and for positional arguments. For example, an option declared as `"--my-opt"` will be associated with the key `"my_opt"` by default. It is especially advisable to turn this option on then parsing with the `as_symbols=true` argument to `parse_args`.
- `error_on_conflict` (default = `true`): if `true`, throw an error in case conflicting entries are added to the argument table; if `false`, later entries will silently take precedence. See *this section* (page 25) for a detailed description of what conflicts are and what is the exact behavior when this setting is `false`.
- `suppress_warnings` (default = `false`): is `true`, all warnings will be suppressed.
- `allow_ambiguous_opts` (default = `false`): if `true`, ambiguous options such as `-1` will be accepted.
- `commands_are_required` (default = `true`): if `true`, commands will be mandatory. See *this section* (page 19) for more information on commands.
- `exc_handler` (default = `ArgParse.default_handler`): this is a function which is invoked when an error is detected during parsing (e.g. an option is not recognized, a required argument is not passed etc.). It takes

two arguments: the `settings::ArgParseSettings` object and the `err::ArgParseError` exception. The default handler prints the error text and the usage screen on standard error and exits with error code 1:

```
function default_handler(settings::ArgParseSettings, err, err_code::Int = 1)
    println(STDERR, err.text)
    println(STDERR, usage_string(settings))
    exit(err_code)
end
```

The module also provides a function `ArgParse.debug_handler` (not exported) which will just rethrow the error.

Here is a usage example:

```
settings = ArgParseSettings(description = "This program does something",
                            commands_are_required = false,
                            version = "1.0",
                            add_version = true)
```

which is also equivalent to:

```
settings = ArgParseSettings()
settings.description = "This program does something."
settings.commands_are_required = false
settings.version = "1.0"
settings.add_version = true
```

As a shorthand, the `description` field can be passed without keyword, which makes this equivalent to the above:

```
settings = ArgParseSettings("This program does something",
                            commands_are_required = false,
                            version = "1.0",
                            add_version = true)
```

Most settings won't take effect until `parse_args` is invoked, but a few will have immediate effects: `autofix_names`, `error_on_conflict`, `suppress_warnings`, `allow_ambiguous_opts`.

CHAPTER 5

Argument table basics

The argument table is used to store allowed arguments and options in an `ArgParseSettings` object. There are two very similar methods to populate it:

`@add_arg_table` (*settings*, *table...*)

This macro adds a table of arguments and options to the given `settings`. It can be invoked multiple times. The arguments groups are determined automatically, or the current default group is used if specified (see *this section* (page 20) for more details).

The `table` is a list in which each element can be either `String`, or a tuple or a vector of `String`, or an assignment expression, or a block:

- a `String`, a tuple or a vector introduces a new positional argument or option. Tuples and vectors are only allowed for options and provide alternative names (e.g. `["--opt", "-o"]`)
- assignment expressions (i.e. expressions using `=`, `:=` or `=>`) describe the previous argument behavior (e.g. `help = "an option"` or `required => false`). See *this section* (page 16) for a complete description
- blocks (`begin...end` or lists of expressions in parentheses separated by semicolons) are useful to group entries and span multiple lines.

These rules allow for a variety usage styles, which are discussed in *this section* (page 31). In the rest of this document, we will mostly use this style:

```
@add_arg_table settings begin
    "--opt1", "-o"
        help = "an option with an argument"
    "--opt2"
    "arg1"
        help = "a positional argument"
        required = true
end
```

In the above example, the `table` is put in a single `begin...end` block and the line `"--opt1", "-o"` is parsed as a tuple; indentation is used to help readability.

`add_arg_table` (*settings, [arg_name [,arg_options]]...*)

This function is very similar to the macro version. Its syntax is stricter: tuples and blocks are not allowed and argument options are explicitly specified as `Dict` objects. However, since it doesn't involve macros, it offers more flexibility in other respects, e.g. the `arg_name` entries need not be explicit, they can be anything which evaluates to a `String` or a `Vector{String}`.

Example:

```
add_arg_table(settings,
    ["--opt1", "-o"],
    Dict(
        :help => "an option with an argument"
    ),
    "--opt2",
    "arg1",
    Dict(
        :help => "a positional argument"
        :required => true
    ))
```

Note that the ‘‘Dict‘‘s are constructed using Julia 0.4 syntax in the above example.

CHAPTER 6

Argument table entries

Argument table entries consist of an argument name and a list of argument settings, e.g.:

```
"--verbose"
    help = "verbose output"
    action = :store_true
```

6.1 Argument names

Argument names are strings or, in the case of options, lists of strings. An argument is an option if it begins with a `'-'` character, otherwise it'a positional argument. A single `'-'` introduces a short option, which must consist of a single character; long options begin with `"--"` instead.

Positional argument names can be any string, except all-uppercase strings between `'%'` characters, which are reserved (e.g. `"%COMMAND%"`). Option names can contain any character except `'='`, whitespaces and non-breakable spaces. Depending on the value of the `add_help` and `add_version` settings, options `--help`, `-h` and `--version` may be reserved. If the `allow_ambiguous_opts` setting is `false`, some characters are not allowed as short options: all digits, the dot, the underscore and the opening parethesis (e.g. `-1`, `-.`, `-_`, `-(`).

For positional arguments, the argument name will be used as the key in the `Dict` object returned by the `parse_args` function. For options, it will be used to produce a default key in case a `dest_name` is not explicitly specified in the table entry, using either the first long option name in the list or the first short option name if no long options are present. For example:

argument name	default `dest_name`
`"--long"`	`"long"`
`"--long", "-s"`	`"long"`
`"-s", "--long1", "--long2"`	`"long1"`
`"-s", "-x"`	`"s"`

In case the `autofix_names` setting is `true` (it is `false` by default), dashes in the names of arguments and long options will be converted to underscores: for example, `"--my-opt"` will yield `"my_opt"` as the default `dest_name`.

The argument name is also used to generate a default metavar in case `metavar` is not explicitly set in the table entry. The rules are the same used to determine the default `dest_name`, but for options the result will be uppercased (e.g. `"--long"` will become `LONG`). Note that this poses additional constraints on the positional argument names (e.g. whitespace is not allowed in metavars).

6.2 Argument entry settings

Argument entry settings determine all aspects of an argument's behavior. Some settings combinations are contradictory and will produce an error (e.g. using both `action = :store_true` and `nargs = 1`, or using `action = :store_true` with a positional argument). Also, some settings are only meaningful under some conditions (e.g. passing a `metavar` to a flag-like option does not make sense) and will be ignored with a warning (unless the `suppress_warnings` general setting is `true`).

This is the list of all available settings:

- `nargs` (default = `'A'`): the number of extra command-line tokens parsed with the entry. See *this section* (page 17) for a complete desctiption.
- `action`: the action performed when the argument is parsed. It can be passed as a `String` or as a `Symbol` (e.g. both `:store_arg` and `"store_arg"` are accepted). The default action is `:store_arg` unless `nargs` is `0`, in which case the default is `:store_true`. See *this section* (page 17) for a list of all available actions and a detailed explanation.
- `arg_type` (default = `Any`): the type of the argument. Only makes sense with non-flag arguments. Only works out-of-the-box with string and number types, but see *this section* (page 27) for details on how to make it work for general types (including user-defined ones).
- `default` (default = `nothing`): the default value if the option or positional argument is not parsed. Only makes sense with non-flag arguments, or when the action is `:store_const` or `:append_const`. Unless it's `nothing`, it must be consistent with `arg_type` and `range_tester`.
- `constant` (default = `nothing`): this value is used by the `:store_const` and `:append_const` actions, or when `nargs = '?'` and the option argument is not provided.
- `required` (default = `false`): determines if an argument is required (this setting is ignored by flags, which are always optional, and in general should be avoided for options if possible).
- `range_tester` (default = `x->true`): a function returning a `Bool` value which tests whether an argument is allowed (e.g. you could use `arg_type = Integer` and `range_tester = isodd` to allow only odd integer values)
- `dest_name` (default = auto-generated): the key which will be associated with the argument in the `Dict` object returned by `parse_args`. The auto-generation rules are explained in *this section* (page 15). Multiple arguments can share the same destination, provided their actions and types are compatible.
- `help` (default = `""`): the help string which will be shown in the auto-generated help screen. It's a `String` which will be automaticaly formatted; also, `arg_type` and `default` will be automatically appended to it if provided.
- `metavar` (default = auto-generated): a token which will be used in usage and help screens to describe the argument syntax. For positional arguments, it will also be used as an identifier in all other messages (e.g. in reporting errors), therefore it must be unique. The auto-generations rules are explained in *this section* (page 15).
- `force_override`: if `true`, conflicts are ignored when adding this entry in the argument table (see also *this section* (page 25)). By default, it follows the general `error_on_conflict` settings.
- `group`: the option group to which the argument will be assigned to (see *this section* (page 20)). By default, the current default group is used if specified, otherwise the assignment is automatic.
- `eval_arg` (default: `false`): if `true`, the argument will be parsed as a Julia expression and evaluated, which means that for example `"2+2"` will yield the integer `4` rather than a string. Note that this is a security risk for outside-facing programs and should generally be avoided: overload *ArgParse.parse_item* instead (see *this section* (page 27)). Only makes sense for non-flag arguments.

6.3 Available actions and nargs values

The `nargs` and `action` argument entry settings are used together to determine how many tokens will be parsed from the command line and what action will be performed on them.

The `nargs` setting can be a number or a character; the possible values are:

- `'A'`: automatic, i.e. inferred from the action (this is the default). In practice, it means `0` for flag-like options and `1` for non-flag-like options (but it's different from using an explicit `1` because the result is not stored in a `Vector`).
- `0`: this is the only possibility (besides `'A'`) for flag-like actions (see below), and it means no extra tokens will be parsed from the command line. If `action` is not specified, setting `nargs` to `0` will make `action` default to `:store_true`.
- a positive integer number `N`: exactly `N` tokens will be parsed from the command-line, and the result stored into a `Vector` of length `N` (even for `N=1`).
- `'?'`: optional, i.e. a token will only be parsed if it does not look like an option (see *this section* (page 29) for a discussion of how exactly this is established), otherwise the `constant` argument entry setting will be used instead. This only makes sense with options.
- `'*'`: any number, i.e. all subsequent tokens are stored into a `Vector`, up until a token which looks like an option is encountered, or all tokens are consumed.
- `'+'`: like `'*'`, but at least one token is required.
- `'R'`: all remainder tokens, i.e. like `'*'` but it does not stop at options.

Actions can be categorized in many ways; one prominent distinction is flag vs. non-flag: some actions are for options which take no argument (i.e. flags), all others (except `command`, which is special) are for other options and positional arguments:

- flag actions are only compatible with `nargs = 0` or `nargs = 'A'`
- non-flag actions are not compatible with `nargs = 0`.

This is the list of all available actions (in each example, suppose we defined `settings = ArgParseSettings()`):

- `store_arg` (non-flag): store the argument. This is the default unless `nargs` is `0`. Example:

```
julia> @add_arg_table(settings, "arg", action => :store_arg);

julia> parse_args(["x"], settings)
{"arg"=>"x"}
```

  The result is a vector if `nargs` is a non-zero number, or one of `'*'`, `'+'`, `'R'`:

```
julia> @add_arg_table(settings, "arg", action => :store_arg, nargs => 2);

julia> parse_args(["x", "y"], settings)
{"arg"=>{"x", "y"}}
```

- `store_true` (flag): store `true` if given, otherwise `false`. Example:

```
julia> @add_arg_table(settings, "-v", action => :store_true);

julia> parse_args([], settings)
{"v"=>false}
```

```
julia> parse_args(["-v"], settings)
{"v"=>true}
```

- `store_false` (flag): store `false` if given, otherwise `true`. Example:

```
julia> @add_arg_table(settings, "-v", action => :store_false);

julia> parse_args([], settings)
{"v"=>true}

julia> parse_args(["-v"], settings)
{"v"=>false}
```

- `store_const` (flag): store the value passed as `constant` in the entry settings if given, otherwise `default`. Example:

```
julia> @add_arg_table(settings, "-v", action => :store_const, constant => 1, default => 0);

julia> parse_args([], settings)
{"v"=>0}

julia> parse_args(["-v"], settings)
{"v"=>1}
```

- `append_arg` (non-flag): append the argument to the result. Example:

```
julia> @add_arg_table(settings, "-x", action => :append_arg);

julia> parse_args(["-x", "1", "-x", "2"], settings)
{"x"=>{"1", "2"}}
```

  The result will be a `Vector{Vector}` if `nargs` is a non-zero number, or one of `'*'`, `'+'`, `'R'`:

```
julia> @add_arg_table(settings, "-x", action => :append_arg, nargs => '*');

julia> parse_args(["-x", "1", "2", "-x", "3"], settings)
{"x"=>{{"1", "2"}, {"3"}}
```

- `append_const` (flag): append the value passed as `constant` in the entry settings. Example:

```
julia> @add_arg_table(settings, "-x", action => :append_const, constant => 1);

julia> parse_args(["-x", "-x", "-x"], settings)
{"x"=>{1, 1, 1}}
```

- `count_invocations` (flag): increase a counter; the final result will be the number of times the option was invoked. Example:

```
julia> @add_arg_table(settings, "-x", action => :count_invocations);

julia> parse_args(["-x", "-x", "-x"], settings)
{"x"=>3}
```

- `show_help` (flag): show the help screen and exit. This is useful if the `add_help` general setting is `false`. Example:

```
julia> settings.add_help = false;

julia> @add_arg_table(settings, "-x", action => :show_help);
```

```
    julia> parse_args(["-x"], settings)
    usage: <command> [-x]

    optional arguments:
      -x
```

- `show_version` (flag): show the version information and exit. This is useful if the `add_version` general setting is `false`. Example:

```
    julia> settings.version = "1.0";

    julia> @add_arg_table(settings, "-x", action => :show_version);

    julia> parse_args(["-v"], settings)
    1.0
```

- `command` (special): the argument or option is a command, i.e. it starts a sub-parsing session (see *this section* (page 19))

6.4 Commands

Commands are a special kind of arguments which introduce sub-parsing sessions as soon as they are encountered by `parse_args` (and are therefore mutually exclusive). The `ArgParse` module allows commands to look both as positional arguments or as flags, with minor differences between the two.

Commands are introduced by the `action = :command` setting in the argument table. Suppose we save the following script in a file called `cmd_example.jl`:

```
require("argparse")
using ArgParse

function parse_commandline()
    s = ArgParseSettings()

    @add_arg_table s begin
        "cmd1"
            help = "first command"
            action = :command
        "cmd2"
            help = "second command"
            action = :command
    end

    return parse_args(s)
end

parsed_args = parse_commandline()
println(parsed_args)
```

Invoking the script from the command line, we would get the following help screen:

```
$ julia cmd_example.jl --help
usage: cmd_example.jl [-h] {cmd1|cmd2}

commands:
  cmd1         first command
  cmd2         second command
```

```
optional arguments:
  -h, --help  show this help message and exit
```

If commands are present in the argument table, `parse_args` will set the special key `"%COMMAND%"` in the returned `Dict` and fill it with the invoked command (or `nothing` if no command was given):

```
$ julia cmd_example.jl cmd1
{"%COMMAND%"=>"cmd1", "cmd1"=>{}}
```

This is unless `parse_args` is invoked with `as_symbols=true`, in which case the special key becomes `:_COMMAND_`. (In that case, no other argument is allowed to use `_COMMAND_` as its `dest_name`, or an error will be raised.)

Since commands introduce sub-parsing sessions, an additional key will be added for the called command (`"cmd1"` in this case) whose associated value is another `Dict{String, Any}` containing the result of the sub-parsing (in the above case it's empty). In fact, with the default settings, commands have their own help screens:

```
$ julia cmd_example.jl cmd1 --help
usage: cmd_example.jl cmd1 [-h]

optional arguments:
  -h, --help  show this help message and exit
```

The argument settings and tables for commands can be accessed by using a dict-like notation, i.e. `settings["cmd1"]` is an `ArgParseSettings` object specific to the `"cmd1"` command. Therefore, to populate a command sub-argument-table, simply use `@add_arg_table(settings["cmd1"], table...)` and similar.

These sub-settings are created when a command is added to the argument table, and by default they inherit their parent general settings except for the `prog` setting (which is auto-generated, as can be seen in the above example) and the `description`, `epilog` and `usage` settings (which are left empty).

Commands can also have sub-commands.

By default, if commands exist, they are required; this can be avoided by setting the `commands_are_required = false` general setting.

The only meaningful settings for commands in an argument entry besides `action` are `help`, `force_override`, `group` and (for flags only) `dest_name`.

The only differences between positional-arguments-like and flag-like commands are in the way they are parsed, the fact that flags accept a `dest_name` setting, and that flags can have multiple names (e.g. a long and short form).

Note that short-form flag-like commands will be still be recognized in the middle of a short options group and trigger a sub-parsing session: for example, if a flag `-c` is associated to a command, then `-xch` will parse option `-x` according to the parent settings, and option `-h` according to the command sub-settings.

6.5 Argument groups

By default, the auto-generated help screen divides arguments into three groups: commands, positional arguments and optional arguments, displayed in that order. Example:

```
julia> settings = ArgParseSettings();

julia> @add_arg_table settings begin
          "--opt"
          "arg"
```

```
            required = true
          "cmd1"
            action = :command
          "cmd2"
            action = :command
       end;

julia> parse_args(["--help"], settings)
usage: <command> [--opt OPT] [-h] arg {cmd1|cmd2}

commands:
  cmd1
  cmd2

positional arguments:
  arg

optional arguments:
  --opt OPT
  -h, --help  show this help message and exit
```

It is possible to partition the arguments differently by defining and using customized argument groups.

add_arg_group (*settings*, *description*[, *name*[, *set_as_default*]])

This function adds an argument group to the argument table in `settings`. The `description` is a `String` used in the help screen as a title for that group. The `name` is a unique name which can be provided to refer to that group at a later time.

After invoking this function, all subsequent invocations of the `@add_arg_table` macro and `add_arg_table` function will use the new group as the default, unless `set_as_default` is set to `false` (the default is `true`, and the option can only be set if providing a `name`). Therefore, the most obvious usage pattern is: for each group, add it and populate the argument table of that group. Example:

```
julia> settings = ArgParseSettings();

julia> add_arg_group(settings, "custom group");

julia> @add_arg_table settings begin
          "--opt"
          "arg"
       end;

julia> parse_args(["--help"], settings)
usage: <command> [--opt OPT] [-h] [arg]

optional arguments:
  -h, --help  show this help message and exit

custom group:
  --opt OPT
  arg
```

As seen from the example, new groups are always added at the end of existing ones.

The `name` can also be passed as a `Symbol`. Forbidden names are the standard groups names (`"command"`, `"positional"` and `"optional"`) and those beginning with a hash character `'#'`.

set_default_arg_group (*settings*[, *name*])

Set the default group for subsequent invocations of the `@add_arg_table` macro and `add_arg_table` function. `name` is a `String`, and must be one of the standard group names (`"command"`, `"positional"` or `"optional"`) or one of the user-defined names given in `add_arg_group` (groups with no assigned name cannot be used with this function).

If `name` is not provided or is the empty string `""`, then the default behavior is reset (i.e. arguments will be automatically assigned to the standard groups). The `name` can also be passed as a `Symbol`.

Besides setting a default group with `add_arg_group` and `set_default_group`, it's also possible to assign individual arguments to a group by using the `group` setting in the argument table entry, which follows the same rules as `set_default_group`.

Note that if the `add_help` or `add_version` general settings are `true`, the `--help, -h` and `--version` options will always be added to the `optional` group.

CHAPTER 7

Importing settings

It may be useful in some cases to import an argument table into the one which is to be used, for example to create specialized versions of a common interface.

`import_settings` (*settings*, *other_settings*[, *args_only*])

Imports `other_settings` into `settings`, where both are `ArgParseSettings` objects. If `args_only` is `true` (this is the default), only the argument table will be imported; otherwise, the default argument group will also be imported, and all general settings except `prog`, `description`, `epilog` and `usage`.

Sub-settings associated with commands will also be imported recursively; the `args_only` setting applies to those as well. If there are common commands, their sub-settings will be merged.

While importing, conflicts may arise: if `settings.error_on_conflict` is `true`, this will result in an error, otherwise conflicts will be resolved in favor of `other_settings` (see *this section* (page 25) for a detailed discussion of how conflicts are handled).

Argument groups will also be imported; if two groups in `settings` and `other_settings` match, they are merged (groups match either by name, or, if unnamed, by their description).

Note that the import will have effect immediately: any subsequent modification of `other_settings` will not have any effect on `settings`.

This function can be used at any time.

CHAPTER 8

Conflicts and overrides

Conflicts between arguments, be them options, positional arguments or commands, can arise for a variety of reasons:

- Two options have the same name (either long or short)
- Two arguments have the same destination key, but different types (e.g. one is `Any` and the other `String`)
- Two arguments have the same destination key, but incompatible actions (e.g. one does `:store_arg` and the other `:append_arg`)
- Two positional arguments have the same metavar (and are therefore indistinguishable in the usage and help screens and in error messages)
- An argument and a command, or two commands, have the same destination key.

When the general setting `error_on_conflict` is `true`, or any time the specific `force_override` table entry setting is `false`, any of the above conditions leads to an error.

On the other hand, setting `error_on_conflict` to `false`, or `force_override` to `true`, will try to force the resolution of most of the conflicts in favor of the newest added entry. The general rules are the following:

- In case of duplicate options, all conflicting forms of the older options are removed; if all forms of an option are removed, the option is deleted entirely
- In case of duplicate destination key and incompatible types or actions, the older argument is deleted
- In case of duplicate positional arguments metavars, the older argument is deleted
- A command can override an argument with the same destination key
- However, an argument can never override a command if they have the same destination key; neither can a command override another command when added with `@add_arg_table` (compatible commands are merged by `import_settings` though)

CHAPTER 9

Parsing to custom types

If you specify an `arg_type` setting (see *this section* (page 16)) for an option or an argument, `parse_args` will try to parse it, i.e. to convert the string to the specified type. This only works for a limited number of types, which can either be directly constructed from strings or be parsed via the Julia's built-in *parse* function. In order to extend this functionality to other types, including user-defined custom types, you need to overload the *ArgParse.parse_item* function. Example:

```
type CustomType
    val::Int
end

function ArgParse.parse_item(::Type{CustomType}, x::AbstractString)
    return CustomType(parse(Int, x))
end
```

Note that the second argument needs to be of type *AbstractString* (or *String* in Julia 0.3) to avoid ambiguity warnings. Also note that if your type is parametric (e.g. `CustomType{T}`), you need to overload the function like this: `function ArgParse.parse_item{T}(::Type{CustomType{T}, x::AbstractString)`.

CHAPTER 10

Parsing details

During parsing, `parse_args` must determine whether an argument is an option, an option argument, a positional argument, or a command. The general rules are explained in *this section* (page 7), but ambiguities may arise under particular circumstances. In particular, negative numbers like `-1` or `-.1e5` may look like options. Under the default settings, such options are forbidden, and therefore those tokens are always recognized as non-options. However, if the `allow_ambiguous_opts` general setting is `true`, existing options in the argument table will take precedence: for example, if the option `-1` is added, and it takes an argument, then `-123` will be parsed as that option, and `23` will be its argument.

Some ambiguities still remains though, because the `ArgParse` module can actually accept and parse expressions, not only numbers (although this is not the default), and therefore one may try to pass arguments like `-e` or `-pi`; in that case, these will always be at risk of being recognized as options. The easiest workaround is to put them in parentheses, e.g. `(-e)`.

When an option is declared to accept a fixed positive number of arguments or the remainder of the command line (i.e. if `nargs` is a non-zero number, or `'A'`, or `'R'`), `parse_args` will not try to check if the argument(s) looks like an option.

If `nargs` is one of `'?'` or `'*'` or `'+'`, then `parse_args` will take in only arguments which do not look like options.

When `nargs` is `'+'` or `'*'` and an option is being parsed, then using the `'='` character will mark what follows as an argument (i.e. not an option); all which follows goes under the rules explained above. The same is true when short option groups are being parsed. For example, if the option in question is `-x`, then both `-y -x=-2 4 -y` and `-yx-2 4 -y` will parse `"-2"` and `"4"` as the arguments of `-x`.

Finally, note that with the *eval_arg* setting expressions are evaluated during parsing, which means that there is no safeguard against passing things like `run(`rm -fr someimportantthing`)` and seeing your data evaporate (don't try that!). Be careful and generally try to avoid using the *eval_arg* setting.

CHAPTER 11

Argument table styles

Here are some examples of styles for the `@add_arg_table` marco and `add_arg_table` function invocation:

```
@add_arg_table settings begin
    "--opt", "-o"
        help = "an option"
    "arg"
        help = "a positional argument"
end

@add_arg_table(settings
    , ["--opt", "-o"]
    ,    help => "an option"
    , "arg"
    ,    help => "a positional argument"
    )

@add_arg_table settings begin
    (["--opt", "-o"]; help = an option)
    ("arg"; help = "a positional argument")
end

@add_arg_table(settings,
    ["-opt", "-o"],
    begin
        help = "an option"
    end,
    "arg",
    begin
        help = "a positional argument"
    end)

add_arg_table(settings,
    ["-opt", "-o"], Dict(:help => "an option"),
    "arg"         , Dict(:help => "a positional argument")
    )
```

The restrictions are:

- groups introduced by `begin...end` blocks or semicolon-separated list between parentheses cannot introduce argument names unless the first item in the block is an argument name.

Index