The Clon User Manual

The Command-Line Options Nuker, Version 1.0 beta 24 "Michael Brecker"

Didier Verna <didier@didierverna.net>

Cover art by Alexis Angelidis.

Table of Contents

Copying

1 Introduction

`Clon` is a library for managing command-line options in standalone Common Lisp applications. It provides a unified option syntax with both short and long names, automatic completion of partial names and automatic retrieval/conversion of option arguments from the command-line, associated environment variables, fallback or default values. `Clon` comes with a set of extensible option types (switches, paths, strings *etc.*). `Clon` also provides automatic generation and formatting of help strings, with support for highlighting on `tty`'s through ISO/IEC 6429 SGR. This formatting is customizable through *themes*.

Depending on the target audience, `Clon` stands for either "The Command-Line Options Nuker" or "The Common Lisp Options Nuker". `Clon` also has a recursive acronym: "`Clon` Likes Options Nuking", and a reverse one: "Never Omit to Link with `Clon`". Other possible expansions of the acronym are still being investigated.

This manual is for the `Clon` *user*, that is, the developer of a Common Lisp application who wants to use `Clon` for command-line option management[1]. As opposed to the user of the *library*, the user of an *application* powered by `Clon` is called an *end-user*. `Clon` provides both a user manual (this one) and an end-user manual (see *The Clon End-User Manual*). Everybody should read the end-user manual first.

Chapter 3 [Quick Start], page 7, provides a very short example in order to give an overview of what's coming next. Chapter 4 [Using Clon], page 11, explains in detail how to clonify your application, and Chapter 5 [Extending Clon], page 21, describe how to extend `Clon` with your own option types.

[1] An application using `Clon` for its command-line option management is said to be *clonified*. It is also possible to say *clonfiscated*. However, we advise against using *clonistified*. The term *clonificated* is also considered bad style, and the use of *clonificationated* is strictly prohibited.

2 Installation

See Section A.3 [Supported Platforms], page 36, for information on portability or requirements. See `Clon`'s homepage for tarballs, Git repository and online documentation. `Clon` is also available via Quicklisp.

In order to install and load the bare Lisp library, unpack it somewhere in the ASDF 3 source registry and type this at the REPL:

```
(asdf:load-system :net.didierverna.clon)
```

Alternatively, if you just want to use the core library without all the platform-dependent bells and whistles (see Appendix A [Technical Notes], page 35), you can also just load the '`net.didierverna.clon.core`' system.

In the general case, and if you are using SBCL, you may want to make sure that the `CC` environment variable is set to your favorite C compiler in your init file. Otherwise, `Clon` will run in restricted mode (see Appendix A [Technical Notes], page 35). For instance, put this in your `.sbclrc`:

```
(require :sb-posix)
(sb-posix:setenv "CC" "gcc" 1)
```

In addition to the library itself, the `Clon` distribution offers documentation in the form of 3 different manuals, some data files like sample themes (see Section "Theme Creation" in *The Clon End-User Manual*), a couple of demonstration programs and an Emacs Lisp library for supporting the development. If you want to benefit from all those wonders, some bits of manual installation are needed. After unpacking somewhere in the ASDF 3 source registry, please perform the following steps, in order.

1. Edit `make/config.make` to your specific needs.
2. Type `make` to compile the documentation and the demo programs (end-user manual, user manual and possibly reference manual). By default, the documentation is built in info, PDF and HTML formats. If you want other formats (DVI and PostScript are available), type `make all-formats`. You can also type individually `make dvi` and/or `make ps` in order to get the corresponding format.
3. As documented in `make/config.make`, the reference manual is only generated if you have SBCL and the Declt library at hand (see `http://www.lrde.epita.fr/~didier/software/lisp/misc.php#declt`).
4. Type `make install` to install both the documentation and the data files. If you have compiled the documentation in DVI and PostScript format, those will be installed as well. The same goes for the reference manual. The demo programs are not installed anywhere.

Type `make uninstall` to uninstall the library.

3 Quick Start

In this chapter, we assume that you have properly installed `Clon` (see Appendix A [Technical Notes], page 35), and we build a very short program to get you started. Let's call it `quickstart.lisp`.

3.1 Full Source

For cut'n paste convenience, the complete source code is given below. For a slightly longer example, have a look at the demonstration program called `simple` in the distribution.

```
(in-package :cl-user)

(require "asdf")
(asdf:load-system :net.didierverna.clon)
(use-package :net.didierverna.clon)

(defsynopsis (:postfix "FILES...")
  (text :contents "A very short program.")
  (group (:header "Immediate exit options:")
    (flag :short-name "h" :long-name "help"
          :description "Print this help and exit.")
    (flag :short-name "v" :long-name "version"
          :description "Print version number and exit.")))

(defun main ()
  "Entry point for our standalone application."
  (make-context)
  (when (getopt :short-name "h")
    (help)
    (exit))
  (do-cmdline-options (option name value source)
    (print (list option name value source)))
  (terpri)
  (exit))

(dump "quickstart" main)
```

3.2 Explanation

Let's examine this program step-by-step now.

First, we put ourselves in the Common Lisp user package, and load `Clon` from its ASDF system '`net.didierverna.clon`'. Next, we use the `Clon` package, also named '`net.didierverna.clon`'.

```
(in-package :cl-user)

(require "asdf")
(asdf:load-system :net.didierverna.clon)
(use-package :net.didierverna.clon)
```

In fact, using the `Clon` package directly is done here for simplicity, but is not recommended. In case you find the package name too long to prefix every symbol with, `Clon` provides a utility function that allows you to add (and use) a shorter nickname instead (the demonstration programs in the distribution show how to use it):

`nickname-package` **&optional** *NICKNAME* [Function]
Add *NICKNAME* (`:clon` by default) to the `:NET.DIDIERVERNA.CLON` package.

The next thing you want to do is to create a set of options, groups or texts that your application will recognize and/or display in its help string. The structure of your command-line is known to `Clon` as the *synopsis*. In order to create it, use the macro `defsynopsis`.

```
(defsynopsis (:postfix "FILES...")
  (text :contents "A very short program.")
  (group (:header "Immediate exit options:")
    (flag :short-name "h" :long-name "help"
          :description "Print this help and exit.")
    (flag :short-name "v" :long-name "version"
          :description "Print version number and exit.")))
```

Note that the synopsis has a double role: it not only serves to define the options recognized by your application, but also the order in which every element appears in the help string. In that particular example, we define a line of text and a group with a header and two flags.

Now, we are going to define a function `main` for our standalone application. The first thing that we need to do is create a *context*. A context is an object that `Clon` uses to store various things, including some state related to the parsing of the command-line. You create a context with the function `make-context`.

```
(defun main ()
  "Entry point for our standalone application."
  (make-context)
```

At that point, `Clon` is ready to retrieve the options you want from the command-line. Let's first see if the user has called the option `-h`, and give him the help string. Option retrieval is done with the function `getopt`, and automatic help string output with the function `help`. Note that what we do here is *not* process the command-line in sequence, but look directly for a specific option by name (this retrieval method is said to be *explicit*).

```
  (when (getopt :short-name "h")
    (help)
    (exit))
```

The `exit` function is a wrapper around an implementation-dependent way to exit (shall I say quit?) the Common Lisp environment, hence the program. It takes an optional argument that stands for the exit status.

And now, we are going to retrieve the other options and simply print them. This time however, we process the command-line sequentially (so this retrieval method is said to be *sequential*). This is done with the `do-cmdline-options` macro. We also close the `main` function.

```
(do-cmdline-options (option name value source)
  (print (list option name value source)))
(terpri)
(exit))
```

Finally, time to save the Lisp image.

```
(dump "quickstart" main)
```

The first argument to `dump` is the name of the executable to produce, and the second argument is the name of the function to call automatically at startup. In fact, `dump` also accepts a `&rest` argument that just be passed on to the underlying (implementation-specific) dumping facility.

In order to get a standalone executable from this program, all you need to do now is to type '`CC=gcc sbcl --script quickstart.lisp`'. Note that the actual way of dumping executables is compiler-dependent. For more information on the proper way to do it and on the compilers currently supported, see Section A.4 [Dumping Executables], page 36.

Note also that it is possible to use `Clon` without actually dumping a standalone executable (see Section A.5 [Not Dumping Executables], page 38).

4 Using Clon

Using `Clon` in your application is a two stages process. In phase 1, you create a *synopsis*, which is a description of your application's command-line and a *context*, which describes this specific execution of the program. In phase 2, you retrieve the option values and possibly display help strings. Phase 1 is called the *initialization phase*, while phase 2 is called the *runtime phase*.

4.1 Synopsis Definition

Step one of the `Clon` initialization phase consists in defining a *synopsis*. A synopsis is essentially a description of your application's command-line: it describes what are the available options, whether your application has a postfix *etc.* The synopsis, however, plays a second role: it also describes the contents of your application's help string. When you create a synopsis, you describe the command-line and the help string at the same time.

4.1.1 Synopsis Items

Look again at the synopsis definition provided in Chapter 3 [Quick Start], page 7.

```
(defsynopsis (:postfix "FILES...")
  (text :contents "A very short program.")
  (group (:header "Immediate exit options:")
    (flag :short-name "h" :long-name "help"
          :description "Print this help and exit.")
    (flag :short-name "v" :long-name "version"
          :description "Print version number and exit.")))
```

You define a synopsis with the `defsynopsis` macro.

`defsynopsis` ([*OPTIONS...*]) *ITEMS...* [Macro]

Define a new synopsis and return it. *OPTIONS* are key/value pairs. *ITEMS* are text, group or option descriptions.

The following *OPTIONS* are currently available.

`:postfix` A string which will appear at the end of the synopsis line in the help string. When you provide a postfix, you also implicitely tell `Clon` that your application accepts non-option arguments at the end of the command-line (this is called the *remainder* of the command-line). See Section "Option Separator" in *The Clon End-User Manual* for more information on the behavior of `Clon` with respect to postfixes. Also, see Section 4.2.2.2 [Command-Line Remainder], page 16, on how to access the command-line remainder.

We now examine the syntax for each possible *ITEM*.

4.1.1.1 Text

In order to add arbitrary text to your help string, use the following form:

`(text` [*OPTIONS...*])

OPTIONS are key/value pairs. The following *OPTIONS* are currently available.

`:contents`

The actual text string. Try to make proper sentences when adding arbitrary text. You can use explicit newline characters in your text if you really want to go next line, but in general, you should not worry about the formatting because the themes are here to do so. In particular, don't finish your text with a newline. This would break potential theme specifications.

`:hidden` When non-`nil`, the text won't appear in the help string. Hidden texts can still be displayed individually though (see Section 4.5 [Help], page 19).

4.1.1.2 Options

In order to add an option to your help string, you must provide a list beginning with the option type and followed by key/value pairs describing the option's properties. For instance, to add a flag with a short name and a description, you could do this:

```
(flag :short-name "h" :description "Print this help and exit.")
```

Option properties vary depending on the option type. The exact list of available option types, and the corresponding properties are described in Section 4.1.2 [Built-In Option Types], page 12.

4.1.1.3 Groups

In order to add a group to your help string, use the following form:

`(group` ([*OPTIONS...*]) *ITEMS...*`)`

OPTIONS are key/value pairs. *ITEMS* simply are arbitrary text, option or sub-group descriptions as we've just seen.

The following *OPTIONS* are currently available.

`:header` A string which will be displayed above the group's contents in the help string. The same formatting recommendations as for arbitrary text apply (see Section 4.1.1.1 [Text], page 11).

`:hidden` When non-`nil`, the group won't appear in the help string. Hidden groups can still be displayed individually though (see Section 4.5 [Help], page 19). For instance, the `Clon` built-in group is hidden in the regular help string, but the `--clon-help` option still displays it individually.

4.1.2 Built-In Option Types

In this section, we will review all the built-in option types that `Clon` provides, along with their corresponding properties. You can use them directly in your synopsis description. For adding personal option types to `Clon`, see Section 5.1 [New Option Types], page 21.

4.1.2.1 Common Properties

All option types in `Clon`, including those you define yourself (see Section 5.1 [New Option Types], page 21), have a set of basic, common properties. Here is a list of them.

`:short-name`
: The option's short name. A string or `nil`.

`:long-name`
: The option's long name. A string or `nil`.

`:description`
: The option's descriptive text. A string or `nil`. The same formatting recommendations as for arbitrary text apply (see Section 4.1.1.1 [Text], page 11).

`:env-var` The option's associated environment variable. A string or `nil`.

`:hidden` When non-`nil`, the option won't appear in the help string. Hidden options can still be displayed individually though (see Section 4.5 [Help], page 19).

Note that an option is required to have at least one name (either short or long). Non-`nil` but empty names are also prohibited, and of course, a short name cannot begin with a dash (otherwise, it would be mistaken for a long name, but did I really need to mention this?).

4.1.2.2 Flags

In `Clon`, options that don't take any argument are of type `flag`. These options don't provide additional properties on top of the common set described in Section 4.1.2.1 [Common Properties], page 12. All properties default to `nil`.

4.1.2.3 Common Valued Option Properties

All non-flag options in `Clon` are said to be *valued*. All valued options, including those you define yourself (see Section 5.1 [New Option Types], page 21), share a set of additional properties. Here is a list of them.

`:argument-name`
: The name of the option's argument, as it appears in the help string. It defaults to `"ARG"`, so that for instance, a '`name`' option would appear like this: '`--name=ARG`'.

`:argument-type`
: The status of the argument. Possible values are `:required` (the default) and `:mandatory` which are synonyms, or `:optional`.

`:fallback-value`
`:default-value`
: The option's fallback and default values. Remember that a fallback value only makes sense when the argument is optional. Besides, also when the argument is optional, you need to provide at least a fallback or a default value (or both of course; see Section "Value Sources" in *The Clon End-User Manual*).

4.1.2.4 Built-In Valued Options

`Clon` currently defines 6 built-in valued option types. These option types may change the default value for some common properties, and / or provide additional properties of their own. All of this is described below.

`stropt`
: This option type is for options taking strings as their argument. String options don't provide any additional properties, but their default argument name is changed from `"ARG"` to `"STR"`.

`lispobj`
: This option type is for options taking any kind of Lisp object as their argument. `lispobj` options change their default argument name from `"ARG"` to `"OBJ"`. Also, they provide an additional property called `:typespec` which must be a Common Lisp type specifier that the argument must satisfy. It defaults to `t`. Look at the `--clon-line-width` built-in option for an example.

`enum`
: This option type is for options taking values from an enumerated set of keywords. `enum` options change their default argument name from `"ARG"` to `"TYPE"`. Also, they provide an additional property called `:enum` to store the list of Common Lisp keywords enumerating the possible values. The end-user does not use a colon when providing an argument to an `enum` option. Only the keyword's name. The end-user also has the ability to abbreviate the possible values. An empty argument is considered as an abbreviation for the first element in the set. Look at the `--clon-version` built-in option for an example.

`path`
: This option type is for options taking a colon-separated list of pathnames as argument. `path` options change their default argument name from `"ARG"` to `"PATH"`. Also, they provide an additional property called `:type` which specifies the kind of path which is expected. Possible values are: `:file`, `:directory`, `:file-list`, `:directory-list` or `nil` (meaning that anything is allowed). Null paths are allowed, and may be provided by an empty argument. Look at the `--clon-search-path` and `--clon-theme` built-in options for examples.

`switch` This option type is for Boolean options. `switch` options change their default argument type from required to optional and provide a fallback value of `t` automatically for optional arguments. `switch` options provide a new property called `:argument-style`. Possible values are `:yes/no` (the default), `:on/off`, `:true/false`, `:yup/nope`, `:yeah/nah`. This property affects the way the argument name and true or false values are advertized in help strings. However, all possible arguments forms (see Section "Switches" in *The Clon End-User Manual*) are always available to all switches.

`xswitch` This option type stands for *extended* switch. Extended switches result from the mating of a male switch and a female enumeration, or the other way around (elementary decency prevents me from describing this mating process in detail): their possible values are either Boolean or from an `:enum` property as in the case of `enum` options. As simple switches, `xswitch` options change their default argument type from required to optional and provide a fallback value of `t` automatically for optional arguments. They also provide the `:argument-style` property. Contrary to switches, however, this property does not affect the argument name. It only affects the way true or false values are displayed in help strings. Look at the `--clon-highlight` built-in option for an example.

4.1.3 Advanced Synopsis Creation

The fact that `defsynopsis` lets you define things in a *declarative* way has not escaped you. Declarative is nice but sometimes it gets in the way, so it is time to see how things work under the hood. Every item in a synopsis is in fact implemented as an object (an instance of some class), so it turns out that `defsynopsis` simply is a convenience wrapper around the corresponding constructor functions for all such objects. Instead of using `defsynopsis`, you can then use those constructor functions explicitely.

4.1.3.1 Constructors

Let's have a look at the expansion of `defsynopsis` from the quick start example (see Chapter 3 [Quick Start], page 7).

The original code is like this:

```
(defsynopsis (:postfix "FILES...")
  (text :contents "A very short program.")
  (group (:header "Immediate exit options:")
    (flag :short-name "h" :long-name "help"
          :description "Print this help and exit.")
    (flag :short-name "v" :long-name "version"
          :description "Print version number and exit.")))
```

And once the macro is expanded, it will look like this:

```
(make-synopsis :postfix "FILES..."
  :item (make-text :contents "A very short program.")
  :item (make-group :header "Immediate exit options:"
          :item (make-flag :short-name "h"
                           :long-name "help"
                           :description "Print this help and exit.")
          :item (make-flag :short-name "v"
                           :long-name "version"
                           :description "Print version number and exit.")))
```

As you can see, every synopsis element has a corresponding `make-`*`SOMETHING`* constructor, and the keywords used here and there in `defsynopsis` are in fact initargs to those constructors. We now examine those constructors in greater detail.

`make-text` [`:hidden` *BOOL*] `:contents` *STRING* [Function]
Create an arbitrary text object, possibly hidden, whose contents is *STRING*.

`make-`*`OPTION`* *:INITARG INITVAL...* [Function]
Create a new *OPTION* object, *OPTION* being any built-in option type (`flag`, `stropt` *etc.*, see Section 4.1.2.4 [Built-In Valued Options], page 13) or user-defined one (see Section 5.1 [New Option Types], page 21). For a list of available initialization arguments (depending on the option type), see Section 4.1.2 [Built-In Option Types], page 12.

`make-group` [`:header` *STRING* `:hidden` *BOOL*] `:item` *ITEM1* `:item` *ITEM2...* [Function]
Create a group object, possibly hidden, whose header is *STRING*. Every *ITEM* is an arbitrary text object, option object or group object. The order is important as it determines the display of the help string.

`make-synopsis` [`:postfix` *STRING*] `:item` *ITEM1* `:item` *ITEM2...* [Function]
Create a synopsis object whose postfix is *STRING*. Every *ITEM* is an arbitrary text object, option object or group object. The order is important as it determines the display of the help string.

In fact, the `defsynopsis` macro allows you to freely mix declarative forms, constructor calls or whatever Lisp code you may want to use. The way this works is as follows: if a synopsis *ITEM* (see Section 4.1.1 [Synopsis Items], page 11) is a list whose `car` is `text`, `group`, or any option type (inluding those you define yourselves; see Section 5.1 [New Option Types], page 21), then the *ITEM* is expanded as explained earlier. Otherwise, it is just left *as-is*.

To sum up, here's a example of things you can do in `defsynopsis`.

```
(net.didierverna.clon:defsynopsis ()
  (flag #| ... |#)
  *my-great-option*
  (setq *another-option* (net.didierverna.clon:make-switch #| ... |#)))
```

4.1.3.2 Advantages

So, why would you want to use constructors directly or mix declarative and imperative forms in `defsynopsis`? There are several reasons for doing so.

1. Some people prefer to declare (or should I say, create) their arbitrary texts, options and groups locally, in files, modules or ASDF components where they belong. In such a case, you need to keep references to the corresponding objects in order to compute the synopsis in the end.

2. Since using constructors explicitly allows you to keep references to the created objects, these objects can be *reused*. For instance, you can use the same text at different places, you can also use a single option several times, or even a single group several times so that its items appear in different places *etc.* Note that `Clon` knows its way around multiple occurrences of the same object: even if you use the same option object several times in a synopsis, `Clon` only maintains a single option definition.

4.1.3.3 Group Definition

There is one last point we need to address in order to complete this section. There might be times when you need to manipulate an explicit group object, but the object itself can still be created in a declarative (or mixed) way because you don't need to keep references on its items. For this, `Clon` provides a macro called `defgroup`.

`defgroup` (*OPTIONS...*) *ITEMS...* [Macro]

Define a new group and return it. This macro behaves exactly like the `group` form in a call to `defsynopsis` (see Section 4.1.1.3 [Groups], page 12). In fact, an intermediate step in the expansion of the `defsynopsis` macro is to transform `group` forms into `defgroup` macro calls. As for `defsynopsis`, `defgroup` allows you to mix declarative forms, constructor calls or any kind of Lisp code.

4.2 Context Creation

Step two of the `Clon` initialization phase consists in creating a *context*. A context is an object representing a particular instance of your program, for example (and most notably) with an actual command-line as the user typed it.

4.2.1 Making A Context

You create a context with the `make-context` function.

`make-context` [Function]

Create a new context. That's it.

4.2.2 Contextual Information

Once a context object is created, you have access to some interesting contextual information.

4.2.2.1 Program Name

The application's program name, as it appears on the command-line, may be accessed from `Clon`. You may find it easier to do this way, as `Clon` wraps around implementation-dependent access methods to `argv[0]`.

In order to retrieve `argv[0]`, use the `progname` function like this: `(progname)`.

4.2.2.2 Command-Line Remainder

In the case your command-line has a remainder (that is, a non-options part; see Section "Option Separator" in *The Clon End-User Manual* and Section 4.1.1 [Synopsis Items], page 11), you may need to access it in order to process it independently from `Clon`. Since `Clon` is responsible for parsing the command-line, it is also in charge of figuring out where the remainder of the command-line begins.

The command-line remainder is known to `Clon` as soon as a context is created. You can retrieve it by using the `remainder` function like this: `(remainder)`. The remainder is provided as a list of strings.

4.2.2.3 Command-Line Polling

`Clon` provides two utility functions to inquire on the current status of the command-line.

The function `cmdline-options-p` returns true if there are still unprocessed options on the command-line. The function `cmdline-p` returns true if there's *anything* left on the command-line, that is, either unprocessed options, or a remainder. A potential use of `cmdline-p` at a very early stage of your application could be to automatically display a help string if the command-line is effectively empty.

4.3 Integrity Checks

At this point, you know about the two necessary steps to initialize `Clon`: defining a synopsis and creating a context. If you paid attention to the quick start application (see Chapter 3 [Quick Start], page 7), you may have noticed that `defsynopsis` was called as a top-level form whereas `make-context` was called from the function `main`. So why the difference?

First, I hope that you see why a context cannot be created as a toplevel form. If you do that, you will end-up creating a context relevant to the Lisp environment from which the application is created, not run.

The synopsis, on the other hand, could be defined either as a toplevel form, as done in the quick start and the demo programs, or in the function `main`, just before making a context. There is a very good reason to prefer a toplevel form however: that reason is called "integrity checks".

When you define a synopsis (or any synopsis item, for that matter), `Clon` performs a number of checks to make sure that you're making a sensible use of the library. In fact, the number of semantic mistakes that you can make is quite puzzling. You could for instance define several options with identical names, forget to provide a fallback or default value when it is required, provide invalid fallback or default values, and the list goes on and on. These are just a few examples but there are many more, and `Clon` checks all of those (I think).

Since those mistakes relate to the definition of the application itself, they do not depend on a particular execution of it. Consequently, the sooner `Clon` catches them, the better. If you define your application's synopsis as a toplevel form, `Clon` will be able to perform its integrity checks when the application is created, not only when it is used. In other words, you won't be able to get a working application until your use of `Clon` is semantically correct.

This is why it is strongly recommended to create synopsis from toplevel forms, and this also explains why `Clon` chooses *not* to provide an `initialize` function that would wrap around `defsynopsis` and `make-context` together.

4.4 Option Retrieval

During the runtime phase of `Clon`, your main activity will be to retrieve options and their values. `Clon` provides two techniques for retrieving options: you can request the value for a specific option directly, or you can process the command-line sequentially, which is the more traditional approach.

Both of these techniques can be freely combined together at any time, because `Clon` keeps track of the current status of the command-line. In fact, `Clon` never works on the original command-line, but uses a mutable *copy* of it after parsing. If you want to access the real command-line of your application (except for the compiler-specific options, see Section A.5 [Not Dumping Executables], page 38), you may use the `cmdline` function, which is a wrapper around an implementation-dependent way to access it.

Finally, remember that the command-line is scanned from left to right during option retrieval (see Section "Option Retrieval" in *The Clon End-User Manual*).

4.4.1 Explicit Retrieval

Since `Clon` lets you retrieve options on demand (at any time), it makes sense to be able to request the value of a specific option explicitely. For instance, you might want to try the '`--help`' option first, without looking at the rest of the command-line because the application will in fact quit immediately after having displayed the help string.

`getopt` *:KEY VALUE...* [Function]

Retrieve the value of a specific option. The following *:KEY*s are currently available.

`:short-name`
`:long-name`
: Use one of these 2 keys to specify the name of the option you wish to retrieve.

`:option` Alternatively, you can use a reference to an option object (see Section 4.1.3.1 [Constructors], page 14).

This function return two values: the option's value and the value's source (see Section "Value Sources" in *The Clon End-User Manual*).

The value's source may have the following forms:

`(:cmdline` *NAME*`)`
: This is for options found on the command-line. *NAME* is the name used on the command-line. It can be the option's long or short name, or a completed long name if the option's name was abbreviated. A completed name displays the omitted parts in parentheses (`"he(lp)"` for instance).

`(:fallback` *NAME*`)`
: The same but when the fallback value is used, that is, when an option is not provided with its (optional) argument.

`(:default` *NAME*`)`
: The same, but when the default value is used (because there is no fallback).

`(:environment` *VAR*`)`
: This is for options not found on the command-line but for which there is an associated environment variable set in the application's environment. *VAR* is the name of the corresponding environment variable.

`:default` This is for options not found anywhere, but for which a default value was provided.

Note that because flags don't take any argument, `getopt` returns a virtual value of `t` when they are found or a corresponding environment variable exists in the environment. For the same reason, a flag's value source may ony be `(:cmdline` *NAME*`)` or `(:environment` *VAR*`)`.

When an option is not found anywhere and there is no default value, `getopt` just returns nil (no second value). Also, note that when your option accepts `nil` as a value, you *need* to handle the second retrun value to make the difference between an option not found, and an actual value of `nil`.

4.4.2 Sequential Retrieval

The more traditional approach to option retrieval is to scan the command-line for options in their order of appearance. `Clon` supports this by providing you with one function and two macros, as explained below.

`getopt-cmdline` [Function]

Get the *next* command-line option, that is, the first option on the command-line that has not been previously retrieved, either explicitely or sequentially.

When there are no more options on the command-line, this function returns `nil`. Otherwise, four values are returned: the corresponding option object from the synopsis definition (see Section 4.1.3.1 [Constructors], page 14), the name used on the command-line, the option's value and the value source (`:cmdline`, `:fallback` or `:default`). As in the case of explicit retrieval (see Section 4.4.1 [Explicit Retrieval], page 18), the option's name may be completed in case of abbreviation.

Unless you keep references to all your option objects (and thus can compare them directly to the one returned by this function), you can still identify the retrieved option by using the `short-name` and `long-name` readers on it: simply use `(long-name` *`OPTION`*`)` or `(short-name` *`OPTION`*`)` and you will get the corresponding strings.

`multiple-value-getopt-cmdline` (*OPTION NAME VALUE SOURCE*) *BODY* [Macro]

Evaluate *BODY* with *OPTION*, *NAME*, *VALUE* and *SOURCE* bound to the values returned by the `getopt-cmdline` function above. Note that *BODY* is not evaluated if there was no remaining option on the command-line, so you don't need to conditionalize on *OPTION* being `null` yourself.

`do-cmdline-options` (*OPTION NAME VALUE SOURCE*) *BODY* [Macro]

As above, but loop over all command-line options.

4.5 Help

One of the first motivations in the design of `Clon` was to automate the generation of the help string, which is a very boring maintenance task to do by hand. The application's synopsis contains all the necessary information to do so. In order to print your application's help string, use the `help` function.

`help` [`:item` *ITEM*] [Function]

Print the application's help string. Printing honors the search path, theme, line width and highlight settings provided by the corresponding built-in options (see Section "Theme Mechanism" in *The Clon End-User Manual*).

By default, `help` prints the whole application help string, excluding hidden items. However, if you have kept a reference to any synopsis item (option, text, group), you can pass it as the value of the `:item` key, and `Clon` will only print the help string for that particular item. In this case however, it will be displayed even if was declared as hidden (this exception does not extend to its sub-items, though, and they will *not* be displayed if themselves declared hidden).

For instance, the `Clon` built-in group is normally hidden, so it doesn't show up in the global help string, but the `--clon-help` option uses the `help` function on it explicitely, so it discards its hidden state.

Here is a potentially useful application of hidden groups in conjunction with the `:item` key. Look at ImageMagick's `convert` program's help string for instance: it is 276 lines long. Gosh. The help string is decomposed into several categories: image settings, image operators, misc options *etc.*. If I were to implement this program, I would rather have the `--help` option display an overview of the program, advertise `--version` and a couple of others, and I would then implement `--help` as an enumeration for listing every option category individually (they would normally be stored in hidden groups). The user could then use `--help=settings`, `--help=operators` and so on to display only the category she's interested in.

Finally, here is potentially useful application of hidden options that are never ever displayed in any help string whatsoever, and I mean, like, ever. This is the perfect tool for backdoor'ing a program. For instance, if you ever need to implement a `--discard-all-security-measures-and-blow-the-nuke` option, then you'd better have it hidden...

5 Extending Clon

As you know, `Clon` already provides seven built-in option types: flags and six other types for valued options (see Section 4.1.2.4 [Built-In Valued Options], page 13). After using `Clon` for a while, you may find that however brilliant and perfectly designed it is, none of the provided built-in types fullfill your requirements exactly. There are two ways around this: the right way and the wrong way (hint).

The wrong, although perhaps quicker way would be to use the `stropt` option type to simply retrieve unprocessed string values, and then do whatever tweaking required on them. In doing so, you risk reinventing some of `Clon`'s wheels.

The right way is to define a new option type. Properly defined option types are a good thing because they allow for reusability and also extensibility, since new option types can always be defined on top of others. In this chapter we explain how to extend `Clon` by providing new option types. We illustrate the process with the example of the built-in `enum` one.

Oh, and I almost forgot. I hope it is obvious to everyone that new option types are always *valued*. There's no point in extending `Clon` with options that don't take any arguments, since we already have flags.

5.1 New Option Types

From a software engineering point of view, it is better to implement new option types in a file of their own, preferably named after the option type itself, and to put this file in the `Clon` package, like this:

```
(in-package :net.didierverna.clon)
```

Creating your own option type involves 5 steps: providing a class for them, implementing three protocols related to argument/value tweaking, and providing a constructor function. We now review those 5 steps in order.

5.1.1 New Option Classes

`Clon` maintains a class hierarchy for all option types. The mother of all option types is the `option` abstract class. It handles the options'short and long names, description and associated environment variable (see Section 4.1.2.1 [Common Properties], page 12). Valued options inherit from an abstract subclass of `option` called `valued-option`. This class handles the option's argument name and status (optional or mandatory), fallback and default values (see Section 4.1.2.3 [Common Valued Option Properties], page 13).

In order to create a new option type, use the `defoption` macro.

`defoption` *CLASS SUPERCLASSES SLOTS* **&rest** *OPTIONS* [Macro]
Create a new option CLASS and register it with `Clon`. Syntactically, thismacro behaves like `defclass`. Option types created like this implicitely inherit from `valued-option` and in turn `option`, so you don't need to put them explicitely in the *SUPERCLASSES* list.

Let's look at the enumeration example now.

```
(defoption enum (enum-base)
  ((argument-name ;; inherited from the VALUED-OPTION class
    :initform "TYPE"))
  (:documentation "The ENUM class.
This class implements options whose values belong to a set of keywords."))
```

As you can see, this class inherits from `enum-base`, which is the class handling the `:enum` property. The reason for this split is that there are currently two option types providing enumeration-like facility: `enum` and `xswitch`, so `xswitch` also inherits from `enum-base`.

There are no new slots in this class, but the `argument-name` slot provided by the `valued-option` class has its initform changed from `"ARG"` to `"TYPE"`.

5.1.2 Value Check Protocol

Now that we have our new option class, we need to implement the so-called *value check* protocol. This protocol is used to make sure that values provided for options of your new type actually comply with the type in question. Values going through this protocol are fallback values, default values, and values provided from a debugger restart (see Section "Error Management" in *The Clon End-User Manual*). In the case of fallback and default values (which, by the way, are provided by *you*, the `Clon` user), the check is performed only once, when the option object is created. Values provided from a debugger restart come from the application end-user, and hence are checked every time.

The value check protocol is implemented through a `check` generic function for which you must provide a method.

`check` *OPTION VALUE* [Generic Function]
Check that *VALUE* is valid for *OPTION*. If *VALUE* is valid, return it. Otherwise, raise an `invalid-value` error.

As you can see, you need to provide a method with the first argument specialized to your new option type. This method must return *VALUE* if it is okay, and raise an `invalid-value` error otherwise.

`Clon` maintains a hierarchy of error conditions. The `invalid-value` error condition is defined like this:

```
(define-condition invalid-value (option-error)
  ((value :documentation "The invalid value."
          :initarg :value
          :reader value)
   (comment :documentation "An additional comment about the error."
            :type string
            :initarg :comment
            :reader comment))
  (:report (lambda (error stream)
             (format stream "Option ~A: invalid value ~S.~@[~%~A~]"
               (option error) (value error) (comment error))))
  (:documentation "An invalid value error."))
```

When the error is raised, you must fill in the `value` and `comment` slots appropriately. The super-condition `option-error` provides an additional `option` slot that you must also fill in when the error is raised.

Let's look at the enumeration example now.

```
(defmethod check ((enum enum) value)
  "Check that VALUE is a valid ENUM."
  (unless (member value (enum enum))
    (error 'invalid-value
           :option enum
           :value value
           :comment (format nil "Valid values are: ~A."
                      (list-to-string (enum enum)
                                      :key #'prin1-to-string))))
  value)
```

This code should be self-explanatory. We check that the value we got belongs to the enumeration. `list-to-string` is a utility function that will separate every element with comas in the resulting string.

5.1.3 Argument Conversion Protocol

The next protocol we need to implement is the so-called *argument conversion* protocol. This protocol is used to convert option arguments (that is, strings) to an actual value of the proper type. Arguments going through this protocol come from the command-line, the value of an environment variable or a debugger restart (see Section "Error Management" in *The Clon End-User Manual*). Also, note that `Clon` assumes that you implement this protocol correctly, so no value check is performed on values coming from the conversion of an argument.

The conversion protocol is implemented through a `convert` generic function for which you must provide a method.

`convert` *OPTION ARGUMENT* [Generic Function]
Convert *ARGUMENT* to *OPTION*'s value. If *ARGUMENT* is invalid, raise an `invalid-argument` error.

As you can see, you need to provide a method with the first argument specialized to your new option type. This method must return the conversion of *ARGUMENT* to the appropriate type if it is valid, and raise an `invalid-argument` error otherwise.

The `invalid-argument` error condition is defined like this:

```
(define-condition invalid-argument (option-error)
  ((argument :documentation "The invalid argument."
             :type string
             :initarg :argument
             :reader argument)
   (comment :documentation "An additional comment about the error."
            :type string
            :initarg :comment
            :reader comment))
  (:report (lambda (error stream)
             (format stream "Option ~A: invalid argument ~S.~@[~%~A~]"
               (option error) (argument error) (comment error))))
  (:documentation "An invalid argument error."))
```

When the error is raised, you must fill in the `argument` and `comment` slots appropriately. As before, the super-condition `option-error` provides an additional `option` slot that you must also fill in when the error is raised.

Let's look at the enumeration example now.

```
(defmethod convert ((enum enum) argument)
  "Convert ARGUMENT to an ENUM value."
  (or (closest-match argument (enum enum) :ignore-case t :key #'symbol-name)
      (error 'invalid-argument
             :option enum
             :argument argument
             :comment (format nil "Valid arguments are: ~A."
                        (list-to-string (enum enum)
                                        :key (lambda (value)
                                               (stringify enum value)))))))
```

Since enumerations allow their arguments to be abbreviated, a utility function named `closest-match` is used to find the closest match between an argument and the possible values. Otherwise, an `invalid-argument` error is raised. For an explanation of `stringify`, See Section 5.1.5 [Value Stringification Protocol], page 24.

5.1.4 Error Management

Let's take a short break in our `Clon` extension process. We have seen that `Clon` may throw errors in different situations, including invalid arguments or values. The end-user manual advertises that the interactive error handler offers a set of "options to fix problems" (if you don't know what I'm talking about, please read Section "Error Management" in *The Clon End-User Manual* and you will know). Well, you may have guessed that these options are simply "restarts" (of course, we don't use that term there, as we wouldn't want to frighten the casual user, or would we?).

When you ship your application, you are encouraged to disable the debugger in the standalone executable because that also might scare the casual user a little bit (the end-user manual only mentions the `none` error-handler to explain that people shouldn't use it). The way error handling is done in Common Lisp standalone executables is implementation-dependent, so please refer to your favorite compiler's documentation. More generally, the behavior of Common Lisp standalone executables may depend on:

- the state of the Lisp environment when the application was dumped, which may in turn depend on command-line options passed to the Lisp itself (see Section A.4 [Dumping Executables], page 36),
- the arguments passed to the dumping function (look at `Clon`'s `dump` macro to see what it does),
- worse: *both*.

No, really, Common Lisp is no fun at all.

5.1.5 Value Stringification Protocol

Okay, back to implementing our new option type.

The third and last protocol we need to implement is called the *value stringification* protocol. This protocol can be seen as the reverse protocol for argument conversion (see Section 5.1.3 [Argument Conversion Protocol], page 23): its purpose is to transform an option's value into a corresponding argument that the end-user could have provided in order to get that value.

The main use for this protocol is to advertise the fallback and default values correctly in help strings: the end-user does not want to see those *values*, but rather the *argument* that would lead to them. However, you are free to use it wherever you like (see the `convert` method for `enum` options for instance).

The value stringification protocol is implemented through a `stringify` generic function for which you must provide a method.

`stringify` *OPTION VALUE* [Generic Function]
Transform OPTION's *VALUE* into an argument.

I admit that this function could also have been called `argumentize` or even `deconvertify`. As you can see, you need to provide a method with the first argument specialized to your new option type. You can assume that *VALUE* is a valid value for your option, so no checking is necessary and no error needs to be raised.

Let's look at the enumeration example now.

```
(defmethod stringify ((enum enum) value)
  "Transform ENUM's VALUE into an argument."
  (string-downcase (symbol-name value)))
```

Pretty straightforward, right?

5.1.6 Constructor Functions

The last required step to complete our new option type extension is to provide a *constructor* function that wraps around `make-instance` on the corresponding option class. I won't insult you by explaining how to write a constructor. Let me just give four good reasons why providing constructors is important.

Providing a constructor for every new option type is important because:

1. it is important,
2. it is a good software engineering practice,
3. it is important,
4. and above all, it makes your new option type automatically available in calls to `defsynopsis` and `defgroup` (see Section 4.1.1 [Synopsis Items], page 11, and Section 4.1.3.3 [Group Definition], page 16).

Let's look at the enumeration example now.

```
(defun make-enum (&rest keys
                  &key short-name long-name description
                       argument-name argument-type
                       enum env-var fallback-value default-value
                       hidden)
  "Make a new enum option.
- SHORT-NAME is the option's short name (without the dash).
  It defaults to nil.
- LONG-NAME is the option's long name (without the double-dash).
  It defaults to nil.
- DESCRIPTION is the option's description appearing in help strings.
  It defaults to nil.
- ARGUMENT-NAME is the option's argument name appearing in help strings.
- ARGUMENT-TYPE is one of :required, :mandatory or :optional (:required and
  :mandatory are synonyms).
  It defaults to :optional.
- ENUM is the set of possible values.
- ENV-VAR is the option's associated environment variable.
  It defaults to nil.
- FALLBACK-VALUE is the option's fallback value (for missing optional
  arguments), if any.
- DEFAULT-VALUE is the option's default value, if any.
- When HIDDEN, the option doesn't appear in help strings."
  (declare (ignore short-name long-name description
                   argument-name argument-type
                   enum env-var fallback-value default-value
                   hidden))
  (apply #'make-instance 'enum keys))
```

Woah, what a mouthful for a single line of code. . . Yeah, I'm a maniac and I like redundancy. I always restate all the available keys explicitely, and everything again in the docstring so that all the interesting information is directly available (I might change my mind as I grow older though).

5.2 Extension Tips

So that's it. Now you know how to extend `Clon` with your own option types. Here is some piece of advice that you might find useful in the process.

5.2.1 Incremental Option Types

If one of the built-in options is almost what you need, you may be tempted to subclass it directly instead of using `defoption`, and only change what's needed, After all, it's Lisp. Lisp is a world of mess^D^D^D^Dfreedom.

Wrong.

`defoption` is not *only* a convenience wrapper around `defclass`. It also arranges for `defsynopsis` and `defgroup` to recognize your new option type. So please, do use it systematically.

5.2.2 Lisp Option Abuse

Along with the same lines, you may find that the `lispobj` type is all you need in many situations. Let's take an example. Suppose you want to implement a `--stars` option to assign a rank to a movie, from 0 to 5. The lazy approach is to simply create a `lispobj` option with a `:typespec` (type specifier) of `(integer 0 5)` and you're done.

But now, remember that the end-user of your application is probably not a Lisper (in fact, I would hope that `Clon` contributes to increasing the number of standalone Common Lisp applications out there...). What do you think would be her reaction, if, after providing a bogus value to the `--stars` option, she get the following error message:

```
 Option 'stars': invalid argument "6".
Argument "6" must evaluate to (integer 0 5).
```

or worse, a `"Cannot parse argument"` error message because of a typo?

Not very friendly, right? In other words, you need to think in terms of what the end-user of your application will expect. In that particular situation, you might want to subclass `lispobj` (with `defoption`!) only to provide friendlier error messages.

6 Advanced Usage

This chapter contains information about different features that are present in `Clon` because of design decisions, but that I expect to be used only rarely, if at all.

6.1 Multiple `Clon` Instances

It is possible to use different instances of `Clon` in parallel in a single application, by using a virtual command-line instead of the real one, different synopsis and multiple contexts simultaneously.

6.1.1 Using Different Synopsis

Did you notice that after defining a synopsis, there is actually never any explicit reference to it anymore? So where is the magick? In fact, there's no magick at all involved here.

`Clon` has a global variable named `*synopsis*` which holds the current synopsis. When you define/create a synopsis with either `defsynopsis` or `make-synopsis`, it is automatically made the default one, unless you use the `:make-default` option/initarg with a value of `nil`, like this:

```
(defsynopsis (:make-default nil) ...)
```

or this:

```
(make-synopsis :make-default nil ...)
```

When you create a context with `make-context`, the default synopsis is used implicitely, but you have two ways to avoid this.

1. At any time in your program, you may change the value of `*synopsis*`. All subsequent calls to `make-context` will hence use this other synopsis.
2. If you prefer to use another synopsis only temporarily, you can use the `:synopsis` initarg to `make-context` instead.

6.1.2 Using Different Command-Lines

In Section 4.2 [Context Creation], page 16, we saw that a context object describes a particular instance of your application, most notably depending on the actual command-line the end-user provided. It turns out, however that the command-line doesn't need to be the actual program's command-line, as the user typed it. Any list of strings can act as a command-line.

The function `make-context` has a `:cmdline` key that allows you to provide any list of strings that will act as the command-line. Of course, the default is to use the actual program's one.

6.1.3 Using Multiple Contexts

Did you also notice that after creating a context, there is actually never any explicit reference to it anymore? So again, where is the magick? In fact, there's no magick at all involved here either.

`Clon` has a global variable named `*context*` which holds the current context. When you create a context with `make-context`, it is automatically made current, unless you use the `:make-current` initarg with a value of `nil`.

The whole runtime phase API of `Clon` uses a context implicitely. This involves `progname`, `remainder`, `cmdline-options-p`, `cmdline-p`, `getopt`, `getopt-cmdline`, `multiple-value-getopt-cmdline`, `do-cmdline-options` and `help`. As a consequence, it is possible to use `Clon` with multiple contexts at the same time. There are in fact three ways to achieve this.

1. At any time in your program, you may change the value of `*context*`. All subsequent calls to the runtime phase API will hence use this other context.
2. `Clon` also provides a macro which changes the current context for you.

`with-context` *CONTEXT* **&body** *BODY* [Function]
Execute *BODY* with `*context*` bound to *CONTEXT*.

3. If you prefer to use another context only once, you can use the `:context` key instead. The whole runtime phase API of `Clon` understands it. For the functions `getopt`, `getopt-cmdline` and `help`, it's just another key in addition to those we've already seen. For the macros `multiple-value-getopt-cmdline` and `do-cmdline-options`, the key must appear at the end of the first (list) argument, like this:

```
(multiple-value-getopt-cmdline (option name value :context ctx) ...)
(do-cmdline-options (option name value :context ctx) ...)
```

6.1.4 Potential Uses

By combining `Clon`'s ability to use a virtual command-line, different synopsis and multiple contexts, you can achieve very neat (read: totally useless) things. For instance, you could write an application that takes an option providing command-line arguments for an external program to be forked. Some revision control systems do that for controlling external `diff` programs for instance, so no big deal. The big deal is that you can completely control the validity of the external program's command-line, before it is forked, from your original one.

Here is another idea, again related to revision control systems. Some of them feature a command-line syntax like the following:

```
prog [global options] command [command-specific options]
```

You can achieve this with `Clon` quite easily. In fact, the demonstration program called `advanced` in the distribution shows you how to do it. First, define a synopsis which only handles the global options, and provide a postfix of `"command [command-specific option]"` or something like that. This will authorize a command-line remainder which will start with the command name.

Now, for every command in your program, define a specific synopsis with only the command-specific options. Get the remainder of the original command-line (see Section 4.2.2.2 [Command-Line Remainder], page 16) and figure out which command was used. Depending on it, create a new context with the appropriate synopsis and the original command-line's remainder as the new, virtual, command-line. You're done: retrieve global options from the first context, and command-specific ones from the second one.

What's even cooler is that you can display the command-specific options on demand quite easily as well (like what `git` does when you call it like this: `git commit --help` for instance): calling the `help` function on the original context gives you the global options's help string while calling it on the command-specific one will display the command-specific usage.

One thing to remember here is that every context/synopsis duet you create gets its own set of built-in `Clon` options. As a consequence, there is currently no simple way to have a single set of built-in options apply to the whole application, for instance, to both a global and a command-specific context. Let me make this clearer: if your end-user calls `prog --clon-theme=foo command -h`, then the theme option will have no effect because it would only affect the global help option. In order to actually use the expected theme, your end-user would need to use `prog command --clon-theme=foo -h`. Depending on which cerebral emisphere (s)he prefers to use, this may seem logical or not.

Finally, note that you can use the virtual command-line / specific synopsis technique recursively to manage complicated command-line syntax, for instance alternating options and non-options parts several times.

In the future, `Clon` may provide better ways to achieve this kind of things (a notion of "sub-context" may be in order).

6.2 Programmatic Help Strings

So far, we've seen how to use the `help` function to implement a typical `--help` option. This is mostly intended for the end-user. There are also times when this function could be useful to *you*, the application developer. For instance, one could imagine that part of the compilation phase would involve generating the help string in order to include it in the manual. Another idea would be that '`make install`' creates a `REFCARD` file in `/usr/local/share/doc/my-app/` which contains the help string formatted with the `refcard` theme, *etc.*.

In such situations, calling the `help` function might not be directly associated with an end-user level option, or at least not `--help`, and you might not want to honor the end-user level settings for theme, search path, line-width, or highlighting either (remember that these settings might come from the environment variables associated with `--clon-theme`, `--clon-search-path`, `--clon-line-width` and `--clon-highlight`).

Because of this, the `help` function provides additional keys that allow you to override those settings (they are in fact stored in the context object). The keys in question are: `:theme`, `:search-path`, `:line-width` and `:highlight`.

In addition to that, there is an `:output-stream` key which defaults to `*standard-output*` which you could use for instance to write directly to a file. Note that there is no end-user level access to this parameter.

6.3 Version Numbering

As `Clon` evolves over time, you might one day feel the need for conditionalizing your code on the version of the library. While the end-user of your application has limited means to access the current version number of `Clon` (see Section "Clonification" in *The Clon End-User Manual* and the built-in option `--clon-version`), you, the application programmer and `Clon` user, have a finer grained access to it.

The first thing you can do to access the current version number of `Clon` is use the `version` function (this is in fact the function bound to the `--clon-version` option).

`version` **&optional** (*TYPE* **:number**) [Function]

Return the current version number of `Clon`. *TYPE* can be one of `:number`, `:short` or `:long`. For `:number`, the returned value is a fixnum. Otherwise, it is a string.

A `Clon` version is characterized by 4 elements as described below.

- A major version number stored in the parameter `*release-major-level*`.
- A minor version number, stored in the parameter `*release-minor-level*`.
- A release status stored in the parameter `*release-status*`. The status of a release can be `:alpha`, `:beta`, `:rc` (standing for "release candidate") or `:patchlevel`. These are in effect 4 levels of expected stability.
- A status-specific version number stored in the parameter `*release-status-level*`. Status levels start at 1 (alpha 1, beta 1 and release candidate 1) except for stable versions, in which case patch levels start at 0 (*e.g.* 2.4.0).

In addition to that, each version of `Clon` (in the sense *major.minor*, regardless of the status) has a name, stored in the parameter `*release-name*`. The general naming theme for `Clon` is "Great Jazz Musicians". Anyone daring to mention Kenny G at that point will be shot on sight.

Here is how the `version` function computes its value.

- A version `:number` is computed as *major* `*` *10000* `+` *minor* `*` *100* `+` *patchlevel*, effectively leaving two digits for each level. Note that alpha, beta and release candidate status are ignored in version numbers (this is as if the corresponding status level was considered to be always 0). Only stable releases have their level taken into account.

- A `:short` version will appear like this for unstable releases: 1.3a4, 2.5b8 or 4.2rc1. Remember that alpha, beta or release candidate levels start at 1. Patchlevels for stable releases start at 0 but 0 is ignored in the output. So for instance, version 4.3.2 will appear as-is, while version 1.3.0 will appear as just 1.3.
- A `:long` version is expanded from the short one, and includes the release name. For instance, 1.3 alpha 4 "Bill Evans", 2.5 beta 8 "Scott Henderson", 4.2 release candidate 1 "Herbie Hancock" or 4.3.2 "Chick Corea". As for the short version, a patchlevel of 0 is ignored in the output: 1.3 "Bill Evans".

7 Conclusion

So that's it I guess. You know all about `Clon` now. The next step is to actually use it to clonify your favorite application, write new applications using it and contaminate the world with standalone Common Lisp programs, featuring unprecedented command-line power and thrill-a-minute option hacking.

Now, go Luke. The Attack of the `Clon` is ready to begin.

Appendix A Technical Notes

This chapter contains important information about the library's configuration, supported platforms, non-ANSI features and portability concerns.

A.1 Configuration

Some aspects of `Clon`'s behavior can be configured *before* the library is actually loaded. `Clon` stores its user-level configuration (along with some other setup parameters) in another ASDF system called '`net.didierverna.clon.setup`' (and the eponym package). In order to configure the library (I repeat, prior to loading it), you will typically do something like this:

```
(require "asdf")
(asdf:load-system :net.didierverna.clon.setup)
(net.didierverna.clon.setup:configure <option> <value>)
```

`configure` *KEY VALUE* [Function]
Set *KEY* to *VALUE* in the current `Clon` configuration.

Out of curiosity, you can also inquire the current configuration for specific options with the following function.

`configuration` *KEY* [Function]
Return *KEY*'s value in the current `Clon` configuration.

Currently, the following options are provided.

`:swank-eval-in-emacs`
This option is only useful if you use Slime, and mostly if you plan on hacking `Clon` itself. The library provides indentation information for some of its functions directly embedded in the code. This information can be automatically transmitted to (X)Emacs when the ASDF system is loaded if you set this option to `t`. For this to work, there are two things to do first in your (X)Emacs session:

1. set the Slime variable `slime-enable-evaluate-in-emacs` to `t`,
2. and load the `clon` Emacs Lisp library. This library is part of the distribution and installed in a standard Emacs Lisp directory by '`make install`'.

If you're interested to know how this process works, I have described it in the following blog entry: `http://www.didierverna.com/sciblog/index.php?post/2011/07/20/One-more-indentation-hack`.

`:restricted`
Some non-ANSI features of `Clon` require external functionality that may not be available in all contexts. Normally, `Clon` should autodetect this and switch to so-called *restricted mode* at build-time (see Section A.2 [Non-ANSI Features], page 36). If `Clon` has failed to autodetect the problem (in which case I would like to know), or if for some reason, you explicitly want to disable those features, you may set the `:restricted` configuration option to `t`. Another way to do it, without even bothering with configuration is to just use the '`net.didierverna.clon.core`' system instead of the regular one.

`:dump` This option is only used by the ABCL port. Section A.4 [Dumping Executables], page 36, provides more information on its use.

A.2 Non-ANSI Features

One feature of `Clon` that is beyond the ANSI standard is terminal autodetection (it requires an `ioctl` call and hence a foreign function interface). Terminal autodetection is used in several situations, for turning on highlighting automatically and for detecting a terminal line width.

If, for some reason, terminal autodetection is not available, `Clon` will work in so-called *restricted mode*. This means that `--clon-highlight=auto` won't work (highlighting will *not* be turned on automatically on a `tty`). For the same reason, unless otherwise specified via either the `COLUMNS` environment variable or the `--clon-line-width` option, terminal output will be formatted for 80 columns regardless of the actual terminal width (see Section "Global Control" in *The Clon End-User Manual*).

A.3 Supported Platforms

`Clon` is an ASDF 3 library. It currently works on Unix (including MacOS X) and Windows (Cygwin or MinGW) and has been ported to 8 Common Lisp implementations. It requires `editor-hints.named-readtables`. The following table lists the supported platforms and some additional, platform-dependent dependencies.

Compiler	Minimum Version	Dependencies
SBCL	1.0.56	
CMU-CL	20b	
CCL		
ECL	11.1.2[1]	
CLISP		`cffi` (optional)
ABCL	1.1.0[2]	
Allegro[3]		`cffi` (optional)
LispWorks		`cffi` (optional)

CLISP, Allegro and LispWorks specificities

As mentioned in the above table, CLISP, Allegro and LispWorks dependency on `cffi` is optional. They need `cffi` in order to implement terminal autodetection only (note that many other implementations come with their own foreign function interface). If `cffi` cannot be found when the ASDF system is loaded (or in the case of CLISP, if it has been compiled without `ffi` support), you get a big red blinking light and a warning but that's all. `Clon` will still work, although in restricted mode.

ABCL specificities

`Clon`'s ABCL port currently has two limitations:

- It only works in restricted mode (see Section A.2 [Non-ANSI Features], page 36).
- Since Java doesn't have a `putenv` or `setenv` function (!!), the `modify-environment` restart, normally proposed when an environment variable is set to a bogus value, is unavailable (see Section "Error Management" in *The Clon End-User Manual*).

A.4 Dumping Executables

Creating standalone executables is orthogonal to `Clon`. `Clon` is just a library that you might want to use, and in fact, it is also possible to use it without dumping executables at all (see Section A.5 [Not Dumping Executables], page 38).

[1] more precisely, git revision 3e2e5f9dc3c5176ef6ef8d7794bfa43f1af8f8db

[2] more precisely, svn trunk revision 140640

[3] both standard and modern images are supported

Unfortunately, there is no standard way to dump executables from Lisp. We're entering the portability mine field here. `Clon`, however, wants to help. We've already seen that it provides utility wrappers like `exit` and `dump` to make your life easier (I mean, more portable) in Chapter 3 [Quick Start], page 7. If you're not interested in portability or if you prefer to do it your own way, you don't *need* to use those wrappers. If you do, however, please read on.

Continuing on the quickstart example, the table below provides ready-to-use command-line samples for dumping the program with the compilers currently supported. Remember that they work in conjunction with the somewhat rudimentary `dump` macro mentionned above.

SBCL `CC=gcc sbcl --script quickstart.lisp`
The `CC=gcc` bit is needed for the `sb-grovel` contrib module that `Clon` uses (unless restricted mode is requested).

CMUCL `lisp -noinit -nositeinit -load quickstart.lisp`

CCL `ccl --no-init --load quickstart.lisp`

ECL ECL doesn't work like the other Lisp compilers. In particular, creating an executable does not involve dumping a Lisp image, but compiling the source into separate object files and then linking them. The consequence is that there is no simple command-line recipe to show here. Instead, you might want to look at the file `demos/dump.lisp` in the distribution for an example. Note that this file doesn't really attempt to dump the whole application. Only the actual example file is dumped.

Because the dumping scheme is different with ECL, it is also less straightforward to write portable code for standalone applications. The demo programs in the distribution both contain comments that are worth reading at the top of them.

One specific difference between ECL and the other Lisp compilers is that it doesn't require you to specify a particular function as the entry point for your executable, but instead relies on having code to execute directly at the toplevel of some file. `Clon`'s `dump` macro helps you with this: it normally expands to a compiler-specific function which saves the current Lisp image, but for ECL, it simply expands to the "main" function call, hence consituting the entry point to the code that ECL needs.

CLISP `CC=gcc clisp -norc -i quickstart.lisp`
The `CC=gcc` bit lets you choose your preferred C compiler for `cffi`. Please note that executables dumped (the `Clon` way) by CLISP still understand lisp-specific command-line options when they are prefixed with `--clisp-`, so you should obviously avoid defining options by those names.

ABCL ABCL is a totally different beast because it's a Java-based Common Lisp implementation. Most notably, you don't create standalone executables in Java, so the normal way of running ABCL is by telling it to load some lisp code. If you want to run the quickstart program in the usual way, you will typically type this:

```
abcl --noinform --noinit --nosystem --load quickstart.lisp
```

Note that for this to work, the `dump` macro expands to the "main" function call for ABCL, just as for ECL.

There's more to it than that though. It is possible to get closer to standalone executables with ABCL by 1/ providing your whole application in a `jar` file directly, and 2/ overriding ABCL's entry point by providing your own interpreter which runs your main function automatically. I have described this process in more details in the following blog entry: `http://www.didierverna.com/sciblog/index.php?post/2011/01/22/Towards-ABCL-Standalone-Executables`.

`Clon` helps you to do this to some extend. If the `:dump` configuration option is set (see Section A.1 [Configuration], page 35), then the `dump` macro doesn't expand to

a call to the "main" function, but instead dumps a Java file containing an alternate ABCL interpreter that you can compile and add to the original ABCL `jar` file, along with `quickstart.lisp`. If you want more details on this, you will need to look at the `Makefile` in the `demo` directory and also at the file `make/config.make`.

Allegro Dumping a complete application with Allegro is complicated, but more importantly only possible in the Enterprise edition. Because of that, `Clon`'s dumping facility only makes use of the `dumplisp` feature which just dumps a Lisp image. You can perform this operation with a command-line such as '`alisp -qq -L quickstart.lisp`'. The Makefile infrastructure in the `demo` directory also creates shell script wrappers around this image to simplify their use.

LispWorks

`lispworks -init - -siteinit - -load quickstart.lisp`
Please note that dumping a complete application with LispWorks is only possible in the Professional or Enterprise editions.

A.5 Not Dumping Executables

Although command-line processing really makes sense for standalone executables, you may want to preserve interactivity with your application during the development phase, for instance for testing or debugging purposes.

It is possible to use `Clon` interactively, that is, within a regular Lisp REPL, without dumping anything. First of all, we have already seen that it is possible to use a virtual command-line (see Section 6.1.2 [Using Different Command-Lines], page 29). You can also use `Clon` interactively with the actual Lisp invocation's command-line, although a word of advice is in order here.

As you might expect, the problem in that situation lies in the way the different Lisp implementations treat their own command-line. Guess what, it's a mess. When you dump an executable the `Clon` way, the `cmdline` function will always contain user options only (the `dump` macro instructs the compilers to *not* process the command-line at all). When you use `Clon` interactively and mix compiler-specific options with application ones when invoking your Lisp environment, the situation is as follows:

SBCL SBCL processes its own options and leaves the others unprocessed on the command-line. This means that as long as there is no overlap between SBCL's options and yours, you can just put them all together on the command-line. In case of overlap however, you need to separate SBCL's options from yours with a call to `--end-toplevel-options` (that's an SBCL specific option separator).

CMUCL CMUCL processes its own options, issues a warning about the options it doesn't know about, but in any case, it eats the whole command-line. Consequently, if you want to provide options to your application, you need to put them after a `--` separator. Note that you can still use a second such separator to provide `Clon` with both some options and a remainder (see Section "Option Separator" in *The Clon End-User Manual*).

CCL
ECL
CLISP CCL, ECL and CLISP all process their own options but will abort on unknown ones. Consequently, if you want to provide options to your application, you need to put them after a `--` separator. Note that you can still use a second such separator to provide `Clon` with both some options and a remainder (see Section "Option Separator" in *The Clon End-User Manual*).

ABCL ABCL processes its own options and leaves the others unprocessed on the command-line. This means that as long as there is no overlap between ABCL's options and

yours, you can just put them all together on the command-line. In case of overlap however, you need to separate ABCL's options from yours with a `--` separator. Note that you can still use a second such separator to provide `Clon` with both some options and a remainder (see Section "Option Separator" in *The Clon End-User Manual*).

Allegro Allegro processes its own options, issues only a warning about options it doesn't know of, and leaves anything after a `--` alone. Consequently, if you want to provide options to your application, you need to put them after a `--` separator. Note that you can still use a second such separator to provide `Clon` with both some options and a remainder (see Section "Option Separator" in *The Clon End-User Manual*).

LispWorks

LispWorks processes its own options and ignores the other ones, but always leaves everything on the command-line. It does not currently support `--` as an option separator either. Consequently, if you want to provide options to your application, you need to put them after a `--` separator, although you *will* get into trouble if any option there is recognized by LispWorks itself. Note that you can still use a second such separator to provide `Clon` with both some options and a remainder (see Section "Option Separator" in *The Clon End-User Manual*).

Appendix B API Quick Reference

B.1 Setup

`configure` *KEY VALUE* [Function]
See Section A.1 [Configuration], page 35.

`configuration` *KEY* [Function]
See Section A.1 [Configuration], page 35.

B.2 Utilities

`dump` *NAME FUNCTION* **&rest** *ARGS* [Macro]
See Chapter 3 [Quick Start], page 7.

`exit` **&optional** (*STATUS* **0**) [Function]
See Chapter 3 [Quick Start], page 7.

`cmdline` [Function]
See Section 4.4 [Option Retrieval], page 17.

`nickname-package` **&optional** *NICKNAME* [Function]
See Chapter 3 [Quick Start], page 7.

B.3 Initialization Phase API

`make-text` **&key** *CONTENTS HIDDEN* [Function]
`make-`*`OPTION`* *:INITARG INITVAL...* [Function]
`make-group` **&key** *HEADER HIDDEN ITEM* [Function]
`make-synopsis` **&key** *POSTFIX ITEM* (*MAKE-DEFAULT* **t**) [Function]
See Section 4.1.3.1 [Constructors], page 14, and Section 6.1.1 [Using Different Synopsis], page 29.

`defgroup` (**&key** *HEADER HIDDEN*) **&body** *FORMS* [Macro]
See Section 4.1.3.3 [Group Definition], page 16.

`defsynopsis` (**&key** *POSTFIX MAKE-DEFAULT*) **&body** *FORMS* [Macro]
See Section 4.1 [Synopsis Definition], page 11, and Section 6.1.1 [Using Different Synopsis], page 29.

`*synopsis*` [User Option]
See Section 6.1.1 [Using Different Synopsis], page 29.

`make-context` **&key** (*SYNOPSIS* ***synopsis***) *CMDLINE* (*MAKE-CURRENT* **t**) [Function]
See Section 4.2 [Context Creation], page 16, Section 6.1.1 [Using Different Synopsis], page 29, and Section 6.1.2 [Using Different Command-Lines], page 29.

B.4 Runtime Phase API

`*context*` [User Option]
See Section 6.1.3 [Using Multiple Contexts], page 29.

`with-context` *CONTEXT* **&body** *BODY* [Function]
See Section 6.1.3 [Using Multiple Contexts], page 29.

`progname` **&key** (*CONTEXT* ***context***) [Function]
See Section 4.2.2 [Contextual Information], page 16, and Section 6.1.3 [Using Multiple Contexts], page 29.

`remainder` **&key** (*CONTEXT* ***context***) [Function]
See Section 4.2.2 [Contextual Information], page 16, and Section 6.1.3 [Using Multiple Contexts], page 29.

`cmdline-options-p` **&key** (*CONTEXT* ***context***) [Function]
See Section 4.2.2.3 [Command-Line Polling], page 17, and Section 6.1.3 [Using Multiple Contexts], page 29.

`cmdline-p` **&key** (*CONTEXT* ***context***) [Function]
See Section 4.2.2.3 [Command-Line Polling], page 17, and Section 6.1.3 [Using Multiple Contexts], page 29.

`getopt` **&key** (*CONTEXT* ***context***) *SHORT-NAME LONG-NAME OPTION* [Function]
See Section 4.4.1 [Explicit Retrieval], page 18, and Section 6.1.3 [Using Multiple Contexts], page 29.

`getopt-cmdline` **&key** (*CONTEXT* ***context***) [Function]
See Section 4.4.2 [Sequential Retrieval], page 18, and Section 6.1.3 [Using Multiple Contexts], page 29.

`multiple-value-getopt-cmdline` (*OPTION NAME VALUE SOURCE* **&key** *CONTEXT*) **&body** *BODY* [Macro]
`do-cmdline-options` (*OPTION NAME VALUE SOURCE* **&key** *CONTEXT*) **&body** *BODY* [Macro]
See Section 4.4.2 [Sequential Retrieval], page 18, and Section 6.1.3 [Using Multiple Contexts], page 29.

`short-name` *OPTION* [Reader]
`long-name` *OPTION* [Reader]
See Section 4.4.2 [Sequential Retrieval], page 18.

`help` **&key** (*CONTEXT* ***context***) (*ITEM* **(synopsis context)**) (*OUTPUT-STREAM* ***standard-output***) (*SEARCH-PATH* **(search-path context)**) (*THEME* **(theme context)**) (*LINE-WIDTH* **(line-width context)**) (*HIGHLIGHT* **(highlight context)**)) [Function]
See Section 4.5 [Help], page 19, Section 6.1.3 [Using Multiple Contexts], page 29, and Section 6.2 [Programmatic Help Strings], page 31.

B.5 Extension API

`defoption` *CLASS SUPERCLASSES SLOTS* **&rest** *OPTIONS* [Macro]
see Section 5.1.1 [New Option Classes], page 21,

`check` *OPTION VALUE* [Generic Function]
See Section 5.1.2 [Value Check Protocol], page 22.

`option-error` *OPTION* [Error Condition]
`invalid-value` *VALUE COMMENT* [Error Condition]
See Section 5.1.2 [Value Check Protocol], page 22.

`convert` *OPTION ARGUMENT* [Generic Function]
See Section 5.1.3 [Argument Conversion Protocol], page 23.

`invalid-argument` *ARGUMENT COMMENT* [Error Condition]
See Section 5.1.3 [Argument Conversion Protocol], page 23.

`stringify` *OPTION VALUE* [Generic Function]
See Section 5.1.5 [Value Stringification Protocol], page 24.

B.6 Versioning API

`version` **&optional** (*TYPE* **:number**) [Function]
See Section 6.3 [Version Numbering], page 31.

`*release-major-level*` [Parameter]
`*release-minor-level*` [Parameter]
`*release-status*` [Parameter]
`*release-status-level*` [Parameter]
`*release-name*` [Parameter]
See Section 6.3 [Version Numbering], page 31.

Appendix C Indexes

C.1 Concepts

O

P

R

S

T

U

V

C.2 Functions

C

D

E

G

H

L

M

N

P

R

S

V

W

C.3 Variables

*

C

E

P

S

C.4 Data Types

C

E

F

I

L

N

O

P

S

V

X

Appendix D Acknowledgments

The following people have contributed bug reports or fixes, suggestions, compiler support or any other kind of help. You have my gratitude!

Alessio Stalla

Antony *something*

Erik Huelsmann

Erik Winkels

François-René Rideau

Jan Moringen

James Wright

Martin Simmons

Nikodemus Siivola

Sam Steingold