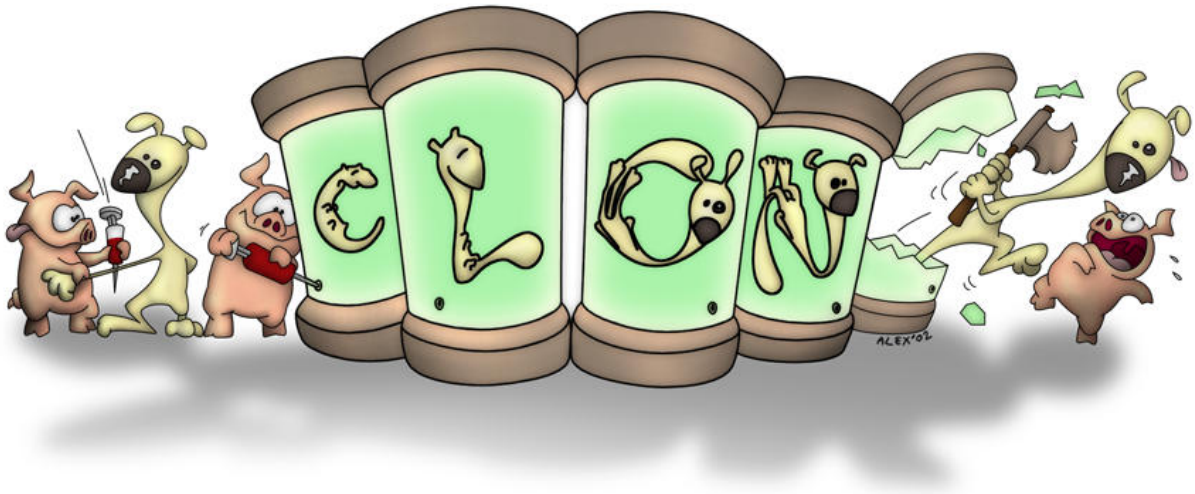
The Clon End-User Manual

The Command-Line Options Nuker, Version 1.0 beta 27 "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` *end-user*, that is, the user of an application powered by `Clon`. It describes how to use the command-line of clonified[1] applications and how to customize `Clon`'s output. Everybody should read this manual first. If you want to use `Clon` in one of your applications, then you are considered a `Clon` *user*, as opposed to a `Clon` *end-user*, and you should then read the corresponding manual (see *The Clon User Manual*).

Chapter 2 [Clonification], page 5, shows how to verify that an application is powered by `Clon`. Chapter 3 [The Command Line], page 7, explains how to use the command-line of a clonified application, and Chapter 4 [Output], page 13, describes how to customize the output of `Clon`.

[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 Clonification

Given this wonderful standalone Common Lisp application, how do I know if it uses `Clon` for handing the command-line? Perhaps the simplest way to know is to type

```
program --clon-help
```

and see what happens. If you get an error, then you are out of luck. Otherwise, you will get a list of `Clon`-specific options. Every clonified application has these options built-in, and they are all called `--clon-`*something*. We will describe most of them when appropriate in this manual, but here's already the description for a couple of them (in addition to `--clon-help` that we've just mentioned).

`--clon--banner`
: When given this option, the application outputs a whole bunch of information, including the version of `Clon` it is using, and `Clon`-specific copyright information.

`--clon-version`
: This option makes the application output the version of `Clon` it is using. By default (or by using it like this: `--clon-version=long`), this information is provided in *long* form (release number, status and name; just try it). If you ask for `--clon-version=short`, you'll get a more compact version information, and if you try `--clon-version=number`, you will end up with a purely numerical version number. For more information on `Clon` version numbering, see Section "Version Numbering" in *The Clon User Manual*.

`--clon-lisp-information`
: Finally, this option displays some information about the underlying Lisp implementation that was used to create this executable.

3 The Command Line

`Clon` provides applications with usual command-line features such as option names coming in short (*e.g.* '`-h`') or long (*e.g.* '`--help`') form. `Clon` also comes with a set of more specific features, such as *packs* or abbreviated forms, that you might want to become familiar with in order to use an application's command-line to its full extent.

3.1 Option Calls

`Clon` offers a set of precise syntactic rules that every clonified application follows implicitly, making them work in a homogeneous way. Although application programmers have the ability to extend `Clon` by defining their own option *types*, even those new options obey the same syntactic rules as the built-in ones.

3.1.1 Option Names

An option can come with either a short name, a long name, or both. It is also possible to abbreviate long names.

3.1.1.1 Full Names

To provide an option by short name, use a *short form*: a single dash followed by the option's short name (*e.g.* '`-h`'). To provide an option by long name, use a *long form*: two dashes followed by the option's long name (*e.g.* '`--help`').

Short names typically consist of a single character. They are concise but more difficult to remember. Long names can be whole words, or even word sequences (usually separated by dashes). They are easier to remember but longer to type on the command-line.

You should be aware of the fact that in a clonified application, neither short names nor long ones are restricted in length. In fact, it would be perfectly possible to have an option's short name longer than the corresponding long one, although that would make little sense. The only real difference is whether you use one or two dashes. Some other differences also come into play when an option expects an argument (see Section 3.1.2 [Option Arguments], page 7) though (it would be no fun otherwise).

3.1.1.2 Abbreviations

When calling an option by long form, it is possible to abbreviate its name, at the risk of being ambiguous. An abbreviation is simply the beginning of the option's name (for instance, '`--he`' can stand for '`--help`'). In case of ambiguity, `Clon` always chooses the option which is "closest" to the abbreviation (here, the "distance" between an abbreviation and an option's name is the number of missing characters).

3.1.2 Option Arguments

In `Clon`, there are different kinds of options: in addition to having a short and/or long name, an option may or may not take an argument. For those taking one, the argument may be either mandatory or optional. Then, there is also an additional, extended call syntax for some of them. But then again, that is not the whole story about getting an option's value (see Section 3.4 [Option Retrieval], page 10). If everything goes well, you are now wondering whether you really want to use a command-line at all.

3.1.2.1 Flags

First of all, some options don't take an argument at all. In the `Clon` jargon, these options are called *flags*. Flags just stand for themselves: either they are present on the command-line, or they are not (as a matter of fact, this is a lie, see Section 3.4 [Option Retrieval], page 10). A

typical example of a flag would be displayed like this in a standard help string (but see Chapter 4 [Output], page 13):

```
-h, --help        Print this help and exit.
```

3.1.2.2 Valued Options

Options taking an argument are said to be *valued*. We need a bit of terminology here: and option's *argument* is typically what you provide on the command-line (or elsewhere, see Section 3.4 [Option Retrieval], page 10). An option's argument is thus a string. An option's *value* is what results from the conversion of the option's argument to the proper type (the same string, a number, whatever).

In `Clon`, a valued option's argument can be either mandatory or optional. We know that valued options, just like flags, may be provided in short or long form. When using the long form, the proper way of providing an argument is to append it after an '`=`' sign, like this: '`--phone=01020304`'. When using the short form, the proper way of providing an argument is to stick it right behind the option's name, like this: '`-p01020304`'. In both long and short form, we call these arguments "sticky". All this should look familiar.

When the option's argument is mandatory, you also have the ability to provide it in the next command-line item instead of sticking it to the option's name. These arguments are said to be "separated". Back to the previous example, this means that you can also say '`--phone 01020304`' or '`-p 01020304`'. Keep in mind that this is not possible when the argument is optional (in the general case, it is not possible to decide whether the next command-line item is an option's argument, a new option or something else).

There is also another case where you can't use this alternate syntax, even when the argument is mandatory: that is when the argument itself looks like an option. `Clon` will think it is, and consequently will also think that the previous one lacks its mandatory argument.

The phone example above would look like this in a standard help string (but see Chapter 4 [Output], page 13):

```
-p, --phone=NUM    Set phone number to NUM.
```

And here is an example illustrating a typical output when the argument is optional:

```
-f, --fax[=NUM]    Set fax number to NUM, or same as phone.
```

3.1.2.3 Switches

For those of you who are not satisfied with short and long forms, `Clon` provides a specific option type for Boolean values, along with an additional call syntax. These options are called *switches*.

Just like any other kind of valued option, a switch can have a short and/or a long name, and its argument may be mandatory or optional. However, the possible arguments for a switch are restricted to true or false. In fact, you can freely use '`yes`', '`on`', '`true`', '`yup`', '`yeah`' and '`no`', '`off`', '`false`', '`nope`', '`nah`' respectively.[1]

In addition to the standard short and long forms, a switch can be provided in *negated form*, that is, by appending its short name behind a '`+`' character. This form never takes any argument and always means false (turn the switch off, if you prefer).

Although an application might do something different, the intended usage for switches is to take an optional argument that defaults to true. A typical example would look like this in a standard help string (but see Chapter 4 [Output], page 13):

```
-(+)d, --debug[=yes/no]    Whether to enable debugging.
```

Given such an option, you can turn debugging on by saying just '`-d`' or '`--debug`' (but you can also provide an explicit argument like this: '`-dyes`' or that: '`--debug=yes`'). Similarly, to turn debugging off, use '`+d`' or '`--debug=no`'.

[1] If you want more, a very modest additional fee will be charged.

One last word about switches: `Clon` lets application developers create new options based on (but not limited to) switches. These options typically accept Boolean arguments as well as some other value(s). As soon as an option is switch-based (and has a short name), the negated form becomes automatically available, and this should be advertised in the application's help string.

3.2 Option Packs

In addition to using options individually (see Section 3.1 [Option Calls], page 7), `Clon` offers the possibility to group option calls together under some circumstances. This feature is know as option *packs*. `Clon` offers two kinds of packs.

3.2.1 Short Packs

A short pack allows you to group multiple short forms in a single command-line item. For instance, instead of calling your program like this: '`program -c -u -p`', you can directly use: '`program -cup`'.

A short pack can only contain options the short names of which are exactly one character long. Also, it should be obvious that you cannot provide an argument to an option in a pack. As a consequence, only flags and options with optional (and not provided) arguments may appear in a short pack.

There is one exception however: it *is* possible to provide an argument to the *last* option in the pack. If provided, this argument must be located in the next command-line item. It cannot be sticked to the option, so this means in particular that this option's argument is mandatory (see Section 3.1.2.2 [Valued Options], page 8).

Beware that using short packs comes at the risk of ambiguity. When `Clon` parses a command-line item beginning with only one dash, it tries to detect options first, options with a sticky argument next, and then short packs. For that reason, you cannot possibly start a pack with a valued option. Indeed, suppose that the option '`-c`' takes an argument. What you *think* is a short pack ('`-cup`') will in fact be interpreted as the option '`-c`' with a sticky (and maybe invalid) argument '`up`'. On the other hand, if the option '`-u`' is only a flag (see Section 3.1.2.1 [Flags], page 7), then you can safely pack your options in a different order, like this: '`-ucp`'.

A single command-line may contain as many short packs as you like, and they can also be intermixed with regular option calls.

3.2.2 Negated Packs

In a very similar way, a negated pack allows you to group multiple negated forms in a single command-line item. As such, this feature only applies to switches or switch-based options (see Section 3.1.2.3 [Switches], page 8). For instance, instead of calling your program like this: '`program +c +u +p`', you can directly use: '`program +cup`'.

A negated pack can only contain options the short names of which are exactly one character long. Also, remember that negated forms never take any argument.

Beware that using negated packs comes at the risk of ambiguity (although much less than with short packs). When `Clon` parses a command-line item beginning with a '`+`', it tries to detect options first, and then negated packs. For that reason, the order in which you specify the options in the pack is important. Indeed, suppose that your application has a '`+cup`' switch (this would be a very bad idea, but still...). If you want to provide the same pack as above, then you need to modify the options order, like this: '`+ucp`'.

A single command-line may contain as many negated packs as you like, and they can also be intermixed with regular option calls or short packs.

3.3 Option Separator

`Clon` separates the command-line in two parts. The left part contains option calls and packs while the right part contains the rest. The right part is also called the *postfix*).

You can force this distinction by using the special construct '`--`' on the command-line. Everything that follows it will be completely ignored by `Clon` (not necessarily by the application itself though).

In the case you don't split your command-line explicitly, `Clon` does this for you automatically by noticing where the last option (or its potential argument) stands. The behavior is different from that of explicit splitting in one regard however: if the application is not expecting any postfix and there's an implicit one, then `Clon` will throw an error at your face.

One final note on command-line separation: in the case an application's postfix is supposed to contain something looking like an option (perhaps real options to pass on to another program), you **need** an explicit separator. Otherwise, `Clon` will be confused: it could for instance wrongly detect unknown options, junk on the command-line *etc.*

You don't want to confuse `Clon`. `Clon` is nasty when it is scared.

3.4 Option Retrieval

The process of getting a value for a specific option is called *retrieval*. This section explains how it works.

3.4.1 Value Sources

The command-line is not the only place where `Clon` looks for option *values*. Other sources for option values are: *fallback* values, *environment* variables and *default* values. The existence of a fallback value, environment variable or default value should be advertised in the application's help string (see Chapter 4 [Output], page 13).

- Fallback values are used when an option exists on the command-line without a corresponding argument (so this applies only to options taking *optional* arguments only).
- Applications may also associate an option with a specific environment variable which contains a value for it.
- Finally, default values are used when every other source has failed.

When `Clon` attempts to retrieve a value for a particular option, it always does so in a specific order: first, the command-line is searched. If an argument is present, it is used. Otherwise, a fallback or default value is used in that order (note that when an option's argument is optional, the option is required to provide at least a fallback or a default value). Next, an environment variable is tried (when appropriate). Finally, when everything else fails, the option's default value is used (if any).

`Clon` always scans the command-line from left to right, and stops at the first match. Please note that the match in question may be a regular option call or a pack, depending on what appears first on the command-line. There is no concept of priority amongst option forms.

Also, note that it is possible to provide several calls to the same option on a single command-line. Some applications may take advantage of this: every consecutive request for an option will use the next match on the command-line until there is none left.

Finally, note that fallback or default values don't make any sense for flags, but flags can still be associated with environment variables. In such a case, the very *existence* of the variable in the environment, regardless of its value, stands for the presence of the corresponding option on the command-line.

3.4.2 Error Management

OK, now you're completely overwhelmed by the power and flexibility of `Clon`, to the point that the fact that you didn't write it (because *I* did) even starts to upset you. So I know what you're thinking: "there's gotta be a way to break it". I don't know, like, giving a value to a flag, using an unknown option, providing an invalid value for an option, using an equal sign in a negated call *etc.*

Unfortunately for you, `Clon` is like a pitbull. Whatever you do to beat it, it *will* fight back. The behavior of `Clon` with respect to error management during option retrieval is well defined, but contrary to the traditional approach, *you*, the end-user have control over it. Not the application. Error handling may occur when the command-line is parsed, but also when environment variables are used.

The error management behavior of `Clon` is controlled by a built-in option named `--clon-error-handler`, and its accompanying environment variable `CLON_ERROR_HANDLER`. Possibles values for it are currently the following.

`quit` This is the default. It means that when `Clon` encounters an error related to option retrieval, it prints an informative message about the error and terminates the application immediately (with an exit status of 1).

`help` Along with `quit`, this one implements a behavior also frequently found in many applications out there (I mean, those not using `Clon`; believe it or not, there are some; oh boy, do I feel sorry for them). When an error occurs, the behavior is the same as above, but in addition to that, the application's help string is also printed automatically.

There are two exceptions to this however. When the problem is related to a known option, the original error message is already informative enough, so there is no point in printing the full help string on top of it. Finally, when the problem seems related to a built-in option (that is, an option starting with `clon-`, then the `Clon`-specific help string is printed out, rather than the application's one.

`interactive`
When interactive error handling is selected and an error is signaled, you are presented with a list of possible options to "fix" the problem. Such options include notably the ability to modify an option's name or value (handy in case of command-line typo), discard the call altogether and many others, depending on the exact error.

When the error implies a bad value for a particular option, you will notice that some of the choices that `Clon` proposes in order to fix the problem involve providing another *value* or another *argument*. Again, you need to remember the terminology here (see Section 3.1.2.2 [Valued Options], page 8). The argument is what you provide on the command-line, and the value is the conversion of the argument to the proper type. This means that most of the time, you will want to use the "argument" choice. If you know the Common Lisp language (see below), you can also provide a value directly, in which case what you type in is in fact Common Lisp code.

`none` *Using this option is not encouraged, unless you are the author of the application and you are debugging it.* A value of `none` literally means no particular error handler. Here, I must apologize because I need to go into some technical details about Common Lisp, the language in which applications using `Clon` are written. Common Lisp mandates the existence of a *debugger* in which you are dropped in when an unhandled error condition is thrown. However some Common Lisp implementations may disable the debugger when creating standalone programs. So the situation when the `Clon` error handler is set to `none` depends on the application.

One last note about erorr handling: we have several chicken-and-egg problems with it. The error handler must be known for parsing the command-line, but in order to get it, we'd need to retrieve and option, which implies parsing the command-line... Whoops. On top of that, if the error handler is set to `help`, it requires the use of several parameters (four, precisely) that also come from options (see Chapter 4 [Output], page 13), themselves likely to trigger errors when misused. Second whoops.

Because of this problem, the options affecting error handling are treated in a very special way.

1. First of all, a built-in default value of `quit` is used initially for error handling.
2. Next, the environment variables affecting output are polled. These are `CLON_SEARCH_PATH`, `CLON_THEME`, `CLON_LINE_WIDTH`, and `CLON_HIGHLIGHT`. They will be described in greater detail in the next chapter. Most of the time, you won't need to use the associated command-line options anyway. During all this time, only the basic error handler is available.
3. Then the environment is polled for `CLON_ERROR_HANDLER`, at which point a fully functional `help` one could be set up. Remember that all of this happens *before* even looking at the command-line itself.
4. Finally, the command-line is parsed, and during parsing, the five built-in options involved in error handling / help output are processed immediately if ever encountered, so as to affect the rest of the parsing. These are `--clon-error-handler`, `--clon-search-path`, `--clon-theme`, `--clon-line-width`, and `--clon-highlight`. Again, the last four ones will be explained in the next chapter.

Now you need to get some rest.

4 Output

In the previous chapter, we have seen how to make the best usage of a clonified application's command-line. The second aspect we need to look at is `Clon`'s output, typically what you get when you type `program --help`. From an end-user perspective, one key feature of `Clon` is that control on the formatting of the help strings is given to *you* instead of being the programmer's responsibility. What this means is that *you* get to choose the way you want help strings to be formatted, and all of a sudden, every clonified application you use will conform to your specifications.

Section 4.1 [Output Elements], page 13, surveys the different items composing a `Clon` help string by looking at examples in a default setting. Section 4.2 [Theme Mechanism], page 14, explains how to switch between predefined layouts. Section 4.3 [Global Control], page 15, describe two built-in options that give you some control over the layout on a global scale. Finally, Section 4.4 [Theme Creation], page 16, describes how to build your own layouts.

4.1 Output Elements

Let's look at the output of `program --help` in a default setting first:

```
Usage: program [-hdF] [+d] [OPTIONS] FILES...

A clonified program.

  -h, --help                    Print this help and exit.
Runtime options:
  -(+)d, --debug[=yes/no]       Turn debugging on or off.
                                Fallback: yes
                                Environment: DEBUG
                                Default: no
  --simulate=yes/no             Simulate only. Nothing will happen for real,
                                except for log messages.
User identification:
  -f, --first-name=STR          Set the user's first name to STR.
  -F, --family-name[=NAME]      Set the user's family name to NAME.
                                Fallback: unknown
```

The first line of output is what's called the *synopsis*. This synopsis indicates that the program accepts a number of options and also a postfix consisting of file names. The set of available options is not detailed in the synopsis but for convenience, `Clon` shows the available short and negated packs explicitly.

The next non-empty line is just text. A clonified application is free to put arbitrary text anywhere in its help string. This can be used to describe what the application is about for instance.

Two other lines in the help string look like arbitrary text, but in fact are not: "Runtime options:" and "User identification:". These are not arbitrary text, but *group titles*. A group is a way of putting help string items together, for instance because they are related to the same topic. A group has an optional title, and may contain options, arbitrary text or even sub-groups.

In addition to displaying the full help string, a clonified application may display a group's help string only (in such a case, you don't get to see the synopsis). This is what happens when you type `program --clon-help` for instance. `Clon`'s built-in options belong to a reserved built-in group.

Let's have a look at the options now. In the first column, you can see that depending on the option, the long, short and negated forms are advertised. Valued options also advertise their argument, enclosed in square brackets when it is optional. In this example, '`-h`' is a flag, '`-d`' and '`--simulate`' are switches with different settings, while '`-f`' and '`-F`' are standard valued options.

The second column of the help string provides each option's *description*. Descriptions can span across several lines, as in the case of '`--simulate`'. `Clon` takes care of properly aligning all the material that needs to be displayed.

Finally, you can see that potential fallback values, environment variables and default values are also advertised in that order, when appropriate.

4.2 Theme Mechanism

The output of `program --help` that we have seen in the previous section corresponds to a default setting, but is really just one possibility. As the user of a clonified application, you have the ability to customize the appearance of `Clon`'s output. The way things look is specified by so-called *themes*. `Clon` has a theme named `raw` which every clonified application has built-in.

Other themes are stored in files the standard extension of which is '`cth`' (as in **C**lon **TH**eme). In fact, theme files can be named as you like; this is just a convention.

4.2.1 Standard Themes

`Clon` comes with several standard themes in files that can be used as-is or serve as the basis for creating new ones (see Section 4.4 [Theme Creation], page 16). The list is given below.

`raw` Exactly the same as the built-in one, only copied into a theme file.

`optlist` Displays only a list of available options, one per line, without any description, group indication or arbitrary text. This can be used to quickly remember the name or the syntax of an option.

`refcard` Displays a very compact help string. The synopsis is there, so are the group titles and arbitrary texts, but the options are displayed without any description, and the lines are filled as much as possible. The intent of this theme is to produce some sort of "reference card" that could be printed.

`roomy` Displays a full help string occupying more space than the `raw` theme. Option descriptions don't start *after* the option's name, but *under* it. As a result, they occupy more horizontal space. Also, sub-groups are indented to the right.

`dvl` This is my personal theme that I personally use myself in person.

`christmas`
This one is full of bells and whistles, which makes it essentially unusable. It exists only to provide a concrete example of all the available formatting and highlighting features of `Clon`. The `christmas` theme is only available once a year, at exactly 23:59:59 on December the 24th.

4.2.2 Search Path

`Clon` maintains a search path for looking up files. Theme files are supposed to be located in a `themes` directory of every directory in the search path. By default, the search path is as follows:

`~/.clon/`

`~/Library/Application Support/Clon/` (MacOS only)

`~/share/clon/`

`/Library/Application Support/Clon/` (MacOS only)

```
/usr/local/share/clon/
/usr/share/clon/
```

The search path can be modified with the `--clon-search-path` built-in option or its accompanying `CLON_SEARCH_PATH` environment variable. It takes a list of colon-separated directories as argument. This argument is in fact optional. Not providing an argument is equivalent to providing an empty one, which basically reduces the search path to the application's current directory.

4.2.3 Theme Selection

In order to select a theme, you can use the built-in option `--clon-theme` or its accompanying environment variable `CLON_THEME`. Just provide it with the theme name, and the corresponding theme file will be looked up in the search path (see Section 4.2.2 [Search Path], page 14), stopping at the first match.

You can omit the '`cth`' extension: `Clon` looks for a file named exactly as you said first and attempts to add the theme extension next. Note that searching with or without extension has priority over the search path. For instance, suppose that the following files exist:

```
/usr/local/share/clon/themes/foobar
/usr/share/clon/themes/foobar.cth
```

Using `--clon-theme=foobar` will match the first one while `--clon-theme=foobar.cth` will match the second.

Now suppose that the following files exist:

```
/usr/local/share/clon/themes/foobar.cth
/usr/share/clon/themes/foobar
```

Using either `--clon-theme=foobar` or `--clon-theme=foobar.cth` will always match the first file. Which brings us to the next point (how clever).

It is possible to bypass the search path when looking for a theme: just provide `--clon-theme` with a theme name starting with a path component (*i.e.* '`/`', '`./`' or '`../`'). Even in that case, you can still omit the '`cth`' extension.

Finally, here is an important precision about `--clon-theme`. This option has a fallback of `nil` and a default of `raw`. This means that if you don't use the option at all, a theme named `raw` will be looked up in the search path. This theme exists in the standard distribution and you are free to modify it. On the other hand, since this option has a fallback, it means that its argument is optional. If you don't provide any argument (that is, if you just use `--clon-theme`), then the *built-in* raw theme will be used. This theme is originally identical to the one provided in the file `raw.cth` but since it is built in every clonified application, you cannot change it.

4.3 Global Control

In addition to selecting the appropriate theme, `Clon` provides two built-in options that give you additional control on the output. These options are orthogonal to themes: all themes implicitly honor them. Since their description is pretty self-explanatory, they are given below.

The first one allows you to control the output's line width:

```
--clon-line-width=WIDTH     Set Clon's output line width.
                            If not given, the value of the COLUMNS
                            environment variable, the terminal size,
                            or a default of 80 columns will be used.
                            Environment: CLON_LINE_WIDTH
```

The second one deals with *highlighting* (see Section 4.4.3 [Highlight], page 18). `Clon` has the ability to highlight the output through ISO/IEC 6429 SGR escape sequences. The built-in raw theme doesn't do highlighting but other do.

```
--clon-highlight[=ARG]      Set Clon's output highlighting to on/off/auto.
                            Auto (the default) means on for tty output and
                            off otherwise.
                            Fallback: yes
                            Default: auto
                            Environment: CLON_HIGHLIGHT
```

A word of caution is in order here. For technical reasons (in fact, the potential inability to detect a terminal properly), it is possible that the `auto` setting for `--clon-highlight`, which happens to be the default, doesn't work. In such a case, highlight is switched off, and you need to use `--clon-highlight=yes` explicitely to force it.

For the same technical reason, it may be impossible to detect a terminal line width from time to time, in which case it would fall back to 80 columns. This particular problem is much less likely to bite you because the `COLUMNS` environment variable should be set all the time.

Both of these problems may or may not occur for specific applications, depending on their underlying implementation.

4.4 Theme Creation

Perhaps the most exciting thing in Computer Science is to spend more time hacking a tool than actually using it. After reading this section, you will find yourselves spending days customizing theme files instead of using clonified applications.

4.4.1 Theme Elements

Hear hear! If you understand what `M-x all-hail-[x]emacs` means, you're going to get quite comfy here. `Clon` themes are articulated around two basic concepts: *faces* and (face) *properties*. A property describes some visual attribute for some piece of text, for instance bold, red, indented by 2 columns to the right. A face is more or less a set of properties (but there's more to it than that; please hold your post). Every piece of text in `Clon`'s output is associated with a face, which in turn defines specific values for specific properties.

There are two property types in `Clon`: *highlight properties*, which describe the visual appearance of characters (color, font *etc.*), and *layout properties*, which describe the text geometry (line width, indentation *etc.*). Please see Section 4.4.3.1 [Highlight Properties], page 18, and Section 4.4.4.1 [Layout Properties], page 20, for an exhaustive list of them.

Let's have a look at a very simple theme file now.

```
;; A very simple theme file.

:background black
;; ...

:face (synopsis :foreground red
                ;; ...
                :face (header :bold t #| ...|#)
                #| ... |#)
;; ...
```

The first line is a comment. Comments begin with a semi-colon and extend to the end of the line. There is another syntax for comments, as show later in the file: a comment can be opened

with '`#|`' and closed with '`|#`'. This form allows you to create comments that span across several lines, or just part of a single line.[1]

The next line sets the '`background`' property to '`black`'. It means that `Clon`'s output will be displayed over a black background (no kidding). Every property in `Clon` has a name, which you specify by using a *keyword* (*i.e.* the name prepended with a colon). But wait, properties are supposed to belong to faces, right? So what is that property doing, floating around like that in the file? That's right, clever. In fact, the whole output of `Clon` is wrapped in a global face called '`toplevel`'. You will never need to mention this face explicitly, though, because the contents of theme files is always enclosed in that face.

Later in the file, you find yourself contemplating a face specification. Faces are specified by using the keyword '`:face`'. What follows is a list beginning with the face name, and continuing with property specifications. In this particular example, we're specifying that the synopsis line (the one that says "Usage: blah blah") should appear in red, although as you can see, the story does not stop there.

Indeed, there is also a '`header`' face specification *within* the '`synopsis`' one. What it says it that the header part (the "Usage:" portion of the synopsis line) should additionally be displayed in bold font. So it turns out that face specifications can be nested. In fact, *all* the faces you specify are sub-faces of the '`toplevel`' face at some level. In this example, the '`header`' face is a sub-face of the '`synopsis`' one. This is important for two reasons:

1. face nesting leads to the notion of property inheritance (see Section 4.4.3.2 [Highlight Inheritance], page 19),
2. `Clon` makes use of faces with the same name in different (nesting) contexts. For instance, there are many places where the '`header`' face is used (see Section 4.4.2 [Faces], page 17), but this face can have very different specifications according to where it appears in a face tree.

4.4.2 Faces

The following figure shows the full face tree defined by `Clon`. In fact, it is extracted from the theme file `raw.cth` (property specifications have been removed). By the way, that is why it is a good idea to start from this one when you want to create a new theme: all the faces are in there. It should be pretty easy to figure out what portion of text each face applies to.

[1] By now, you have realized that a theme file is just a piece of Common Lisp code... No, wait! Don't go!

```
;; Remember that everything in here belongs to the toplevel face.
:face (synopsis :face header
                :face program
                :face short-pack
                :face negated-pack
                :face options
                :face postfix)
:face text
:face (option :face (syntax :face (short :face name
                                         :face argument)
                            :face (long  :face name
                                         :face argument))
              :face (usage :face description
                           :face (fallback :face header
                                           :face value)
                           :face (default :face header
                                          :face value)
                           :face (environment :face header
                                              :face variable)))
:face (group :face header
             :face (items #| :face text :face option :face group |#))
```

As mentioned earlier, you can see that faces of the same name may appear at different places in the tree (*e.g.* the '`header`' one) and hence may have different specifications, depending on the context (their branch in the tree).

Pushing this idea one step further, note the peculiarity of the '`group`' face: since group items may be arbitrary text, options or even sub-groups, you can specify as many levels of group nesting as you want. In fact, you can spend your whole life specifying sub-groups, although it is very unlikely that a clonified application provides more than two or three levels of group imbrication.

One last point to note: you might be surprised to find *empty* face specifications like this one: '`:face description`'. This *is* a valid specification, and this *is* different from not mentioning the face at all, or specifying some properties explicitly. See Section 4.4.4.2 [Layout Inheritance], page 22, for an explanation.

4.4.3 Highlight

The first category of properties that we are going to describe deals with *highlighting*, that is, modifying the visual aspect of characters in the output.

4.4.3.1 Highlight Properties

`Clon` supports 10 highlight properties defined by the ISO/IEC 6429 SGR standard (under the assumption that your favorite terminal program supports them). Their description is given below.

`:foreground`
`:background`
: The face's foreground and background colors. Possible values are `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`. It is also possible to use the special values `nil` or `reset` to reset the color to the terminal's default value.

`:intensity`
: The face's intensity. Possible values are `bold`, `normal` (or `nil`) and `faint`. Please note that `faint` is usually not very well supported. For convenience, a `:bold` Boolean property is also provided which will set the face's intensity to `bold` if true (`t`) and `normal` if false (`nil`).

`:italic`
: Whether to display the face's contents in italics. This is a Boolean property. Possible values are `t` or `nil`. The effect of this property depends a lot on the font you use in your terminal application.

`:underline`
: Whether to underline the face's contents. Possible values are `single` (or `on` or `t`), `double` and `none` (or `off` or `nil`). Please note that `double` is usually not very well supported.

`:blink`
: The face's blink speed. Possible values are `slow` (or `on` or `t`), `rapid` and `off` (or `nil`). Please note that `rapid` is usually not very well supported.

`:inverse`
: Whether to display the face's contents in inverse video. This is a Boolean property. Possible values are `t` or `nil`.

`:crossed-out`
: Whether to display the face's contents crossed out. This is a Boolean property. Possible values are `t` or `nil`.

`:framed`
: Whether to display the face's contents framed or encircled. This is a Boolean property. Possible values are `t` or `nil`.

`:concealed`
: Whether to conceal the face's contents. This is a Boolean property. Possible values are `t` or `nil`. In case it is not obvious, concealing means that the face's contents is not displayed, but still occupies its normal space, so this is different from hiding it (see Section 4.4.4.1 [Layout Properties], page 20). Alternately, you can also use the opposite Boolean property named `:revealed`.

4.4.3.2 Highlight Inheritance

By default (this is also the case in the '`raw`' theme), all highlight properties are turned off: text is output in whatever way is standard on the current terminal.

When you set a face's highlight property however, it is inherited by all the sub-faces. This is in fact the most natural behavior. For instance, if you set the foreground of the '`synopsis`' face to red, you probably expect the whole synopsis line to be output in red, including the header, program *etc.* parts. If not, you need to explicitly neutralize the unwanted effect, for instance by saying something like this:

```
:face (synopsis :foreground red
                :face (program :foreground reset)
                :face (postfix :foreground cyan))
```

In other words, you can consider that the way some piece of text is output is not specified exactly by its associated face, but rather by the merging of all active properties from the whole corresponding face tree. This is exactly how Emacs faces work by the way.

4.4.4 Layout

The second category of properties that we are going to describe deals with *layout*, that is, modifying the placement of text in the output.

4.4.4.1 Layout Properties

In addition to the 10 highlight properties described in the previous section, `Clon` defines 6 layout properties. Here are the first 4 of them, along with their description and default value.

`:visible` Whether the face's contents is visible. Possible values are `t` (the default), or `nil`. You can also use the opposite property named `:hidden`. The '`optlist`' and '`refcard`' themes make use of this to avoid printing the options descriptions. Since the notions of hiding, concealing or making text visible are somewhat intermixed, `Clon` also provides a "bastard" property which, depending on its value, would be considered either a layout, or a highlight one. This property is named `:display`. Possible values are: `visible` (or `revealed` or `t`), `concealed` and `hidden` (or `nil`).

`:padding-top`
The face's top padding, that is, the number of lines to skip before opening it and displaying its contents. This property can take the following forms.

- `nil` This face may be opened anywhere, including on the current line. This is the default.
- `0` This face should be opened on the next line.
- `<num>` (positive) This face should be opened after skipping `<num>` empty lines. Note that the empty lines are displayed in the enclosing face, *not* the face which specifies the top padding.

`:padding-bottom`
This face's bottom padding, that is, the number of lines to skip before opening the next face and displaying its contents. This property can take the following forms.

- `nil` The next face may be opened anywhere, including on the current line. This is the default.
- `0` The next face should be opened on the next line.
- `<num>` (positive) The next face should be opened after skipping `<num>` lines. Note that the empty lines are displayed in the enclosing face, *not* the face which specifies the bottom padding.

`:item-separator`
The face's item separator, that is, a piece of text to insert between every sub-face's contents. The default value is `#\space` (a space character).

The default values above have been chosen because they are convenient for the majority of the faces, but not necessarily all the time. Let us now review some examples of non-default settings in the '`raw`' theme.

- The '`synopsis`' face has its bottom padding set to 1, which means to leave an empty line below it. Note that this line does not belong to the synopsis. It is output in the '`toplevel`' face.
- The '`text`', '`option`' and '`group`' faces have both their top and bottom paddings set to 0. This means that all these items should start and end on lines of their own.
- Finally, the '`syntax`' face has its item separator set to `", "`, which in this context is used to separate the short and long name syntax, as in '`-h, --help`'.

In order to describe the last two layout properties, we need to introduce the notion of *frame* first. A frame is a rectangle in which `Clon` prints something. When `Clon` opens a face for printing its contents, it associates a frame with it. Usually, `Clon` doesn't know the frame's

height because it doesn't know in advance how many lines of output will be necessary to display the face's contents. However, `Clon` usually knows the frame's width: a frame starts at a certain column (the left one) and stops at another (the right one).

There is an isomorphic relation between frames and faces: when a face is a sub-face of a super-face, the corresponding frame is a sub-frame of the corresponding super-frame. Sub-frames are geometrically enclosed within their super-frames: a sub-frame can only draw in an equally large or narrower band than its super-frame. That's how `Clon` handles vertical alignment of text.

In order to give you control on the starting and ending columns of every frame (in fact, every face), the following two layout properties are defined.

`:padding-left`
: A face's left padding determines its starting column. This property can take the following forms.

  `<num>`
  : Skip `<num>` columns relatively to the enclosing face. The default value is 0, meaning to open this face at the same column number as the enclosing one.

  `(<num> :relative-to <face>)`
  : As above, but relatively to a *parent* face named `<face>` instead of just the enclosing one. This lets you go more than one level up in the face tree.

  `(<num> absolute)`
  : Open this face exactly at column `<num>`.

  `self`
  : This face may be opened anywhere, and when it is, its left column is set to the current column number.

`:padding-right`
: A face's right padding, determines its ending column. This property can take the following forms.

  `<num>`
  : Close this face `<num>` columns before the enclosing one.

  `(<num> :relative-to <face>)`
  : As above, but relatively to a *parent* face named `<face>` instead of just the enclosing one. This lets you go more than one level up in the face tree.

  `(<num> absolute)`
  : Close this face exactly at column `<num>`.

  `self`
  : This face may be closed anywhere. This is the default.

The default values above have been chosen because they are convenient for the majority of the faces, but not necessarily all the time. Let us now review some examples of non-default settings in the '`raw`' theme.

- The '`option`' face has its left padding set to 2. Since all super-faces default to 0, this essentially mean an indentation of 2 columns.
- The '`usage`' face has its left padding set to `(30 absolute)` which means that the descriptive texts for all options start aligned at column 30.

There is one important point to understand about self-ending faces (which, again, is the default setting). A self-ending face typically doesn't know at which column it stops until it stops. In fact, this is not completely true: if the face's contents needs to span across several lines, then the ending column will be known at the end of the first line, when the ending column

of the enclosing face is reached. However, for a piece of contents that fits on a single line, the point holds.

Because of that, it is impossible for a face's right padding to be relative to a self-ending face. This would be like saying "stop 2 columns before I don't know where". Here are two examples of such invalid settings (an error will be thrown if you try that in a theme):

```
:face (option :face (syntax :padding-right 2))
:face (option
        :face (syntax
                :face (short
                        :padding-right (2 :relative-to option))))
```

4.4.4.2 Layout Inheritance

This section is a misnomer. Contrary to highlight properties, there is *no* inheritance for layout properties across a face tree. In other words, every missing layout property in a face specification has the property in question set to its default value.

Again, this design decision has been adopted because it is the most natural thing to do. If you're not convinced, consider this: when you specify that the '`option`' face has a top padding of 0, you mean that every option should be described on a line of its own. However, you probably do *not* mean that every individual sub-part of the option's description (syntax part, usage part *etc.*) should also start on its own line.

Now that we know all about highlight and layout inheritance, we are able to explain the face specification shortcut mentioned earlier, when a face is specified directly by name, without any explicit property specification:

```
:face description
```

This syntactic shortcut actually lets you specify a face which gets all default values for layout properties, and inherits all current values from its super-faces for highlight properties. This is in fact a shortcut for this:

```
:face (description)
```

This, however, does not explain why you would want to issue such a specification instead of just not mentioning the face at all. See Section 4.4.5 [Implicit Faces], page 22, for an explanation.

4.4.5 Implicit Faces

In order to fully understand how themes work, we need to tackle one last aspect of their conception: the case of *implicit faces*. Although `Clon` itself needs a completely defined face tree to perform output correctly, a theme file is not required to define all of them (that is impossible by the way: because of group nesting, a complete theme file would be infinitely big). In fact, a theme file can be totally empty, in which case the output will effectively conform to the built-in '`raw`' theme. When a face is "missing" from a theme, `Clon` arranges to define it in a sensible way. Such a face is said to be *implicit*. The exact rules for defining implicit faces is what this section is all about.

4.4.5.1 Face Tree Reuse

Suppose that `Clon` needs to display an option's description, and that option belongs to a group. The corresponding face in a theme file would be the following one:

```
:face (group :face (items :face (option  #| ... |#)))
```

This face describes how options belonging to a level 1 group are to be displayed. As such, it is not applicable to toplevel options. When this face is missing from the theme file, `Clon` tries to find a *more general* one. In that particular case, the theme file might define a "toplevel" '`option`' face like this:

```
:face (option #| ... |#)
```

This face is more general because it lies at the toplevel. As such, it is considered applicable to toplevel options, but also to options belonging to groups at any level. In the same vein, specifying an '`option`' face within a '`group`' face implicitly makes it applicable at any higher group level.

This face reuse mechanism applies to *any* face in a theme; not only to the ones used in the examples above. For instance, if you decide that all headers in `Clon`'s output should be displayed in the same way, you can very well specify a '`header`' face at the toplevel of a theme file (although `Clon` never has anything to display in a '`header`' face at the toplevel), and this face will be used in synopsis headers, group headers, fallback headers *etc.*.

Now, I must confess that the face reuse mechanism described above was over-simplified. What really happens is not exactly face reuse, but *face tree* reuse. Here is a more complicated example to clarify things a little.

Suppose `Clon` needs to display the syntax part of an option that belongs to a level 1 group. The corresponding face in a theme file would be the following:

```
:face (group :face (items :face (option :face (syntax #| ... |#))))
```

If that face is missing, then `Clon` will in fact attempt to find a more general one *while preserving as much context as possible*. In other words, the following specifications will be tried, in that order:

```
:face (items :face (option :face (syntax #| ... |#)))
:face (option :face (syntax #| ... |#))
:face (syntax #| ... |#)
```

I hope this makes sense to you because again, it is the most natural thing to do. What this roughly means is that when you are looking for a way to display an option's syntax part, you should reuse a more general option's syntax face first, and only as a very last resort fall back to a '`syntax`' face specification that would be floating around on its own.

4.4.5.2 Sibling Faces

Now, what happens when the face you're looking for is not specified *at all*, not even at a more general level? In such a situation, `Clon` uses a *sibling*. A sibling face is a face that plays the same role as the one you're looking for, only it is extracted from the built-in '`raw`' theme. The built-in '`raw`' theme ensures that every face is defined at least once, so all faces will eventually be found. When a sibling face is not found in the exact context in which it is needed (for instance, the '`group`' face is empty in the '`raw`' theme), then the same process of face tree reuse as described in the previous section occurs on the sibling face tree.

5 Conclusion

So that's it I guess. Enjoy using clonified applications, don't spend too much time hacking themes, and please contribute to the Common Lisp world by developing standalone programs with `Clon`. Read *The Clon User Manual* to learn how to do that.

Hmmm. What kind of conclusion was that...

Concept Index

G

H

I

L

M

N

O

P

R

S

T

V