CLIFF: Command Line Interface Functional Framework

Daniel Jay Haskin

Contents

Overview

Do you want to have your Common Lisp app look at configuration files, the environment, and command line to get your app's options, but don't want to have to actually *code* any of that stuff? Feel bogged down by writing the same old I/O code for every app?

Then CLIFF is for you!

This document is available as a PDF document.

Tutorial

The code discussed in this tutorial can be found here.

Install

CLIFF is on OCICL and I'm trying to get it added on Quicklisp. Otherwise, just clone the source code out to your QuickLisp or ASDF local projects directory. At the moment, you'll probably want to clone NRDL out to that directory as well, since it's a dependency of CLIFF.

Up and Running

This is a CLI framework/library for the *impatient*, so let's make a CLI tool in Common Lisp *really fast*.

We'll make a CLI math calculator.

```
;;; calculator.lisp -- A CLI calculator.
;;;;
;;;; SPDX-FileCopyrightText: 2024 Daniel Jay Haskin
;;;; SPDX-License-Identifier: MIT
;;;;
```

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

(defpackage
  #:com.djhaskin.calc (:use #:cl)
  (:documentation
    "
    A CLI calculator.
    ")
  (:import-from #:com.djhaskin.cliff)
  (:local-nicknames
    (#:cliff #:com.djhaskin.cliff))
  (:export #:main))

(in-package #:com.djhaskin.calc)


(defun main ()
  (cliff:execute-program
    "calc"))
```

We just made a CLI tool.

This is what happens when we run it:

```
* (main)
Welcome to `calc`!

This is a CLIFF-powered program.

Configuration files and, optionally, environment variables can be set
using NRDL, a JSON superset language. Output will be a NRDL document.
More information about NRDL can be found here:

https://github.com/djha-skin/nrdl

Options can be given via:
  - Configuration file, as in `{ <option> <value> }`
  - Environment variable, as in `CALC_<KIND>_<OPTION>`
  - Command line, as in `--<action>-<option>`

Configuration files are consulted first, then environment variables, then
the command line, with later values overriding earlier ones.

Configuration files can be in the following locations:
```

```
  - A system-wide config file in an OS-specific location:
      - On Windows: `%PROGRAMDATA%\calc\config.nrdl`
        (by default `C:\ProgramData\
      - On Mac:   `/Library/Preferences/~A/config.nrdl`~%\config.nrdl`)

calc      - On Linux/POSIX: `/etc/calc/config.nrdl`
        (by default ~/.config/calc/config.nrdl)`
  - A home-directory config file in an OS-specific location:
      - On Windows: `%LOCALAPPDATA%\calc\config.nrdl`
        (by default `%USERPROFILE%\AppData\Local\calc\config.nrdl`)

      - On Mac:   `~/Library/Preferences/calc/config.nrdl`
      - On Linux/POSIX: `${XDG_CONFIG_HOME}/calc/config.nrdl`
        (by default ~/.config/calc/config.nrdl)`
  - A file named `.calc.nrdl` in the directory `/home/skin/Code/djha-skin/calc/`
    (or any of its parents)
Options can be set via environment variable as follows:
  - `CALC_FLAG_<OPTION>=1` to enable a flag
  - `CALC_ITEM_<OPTION>=<VALUE>` to set a string value
  - `CALC_LIST_<OPTION>=<VAL1>,<VAL2>,...` to set a list option
  - `CALC_TABLE_<OPTION>=<KEY1>,<VAL1>=<KEY2>,<VAL2>=...` to set
    a key/value table option
  - `CALC_NRDL_<OPTION>=<NRDL_STRING>` to set a value using
    a NRDL string
  - `CALC_FILE_<OPTION>=<FILE_PATH>` to set a value using the contents
    of a NRDL document from a file as if by the `--file-*` flag
  - `CALC_RAW_<OPTION>=<FILE_PATH>` to set a value using the raw bytes
    of a file as if by the `--raw-*` flag

Options can be changed or set on the command line in the following ways:
  - To enable a flag, use `--enable-<option>`.
  - To disable a flag, use `--disable-<option>`.
  - To reset any value, use `--reset-<option>`.
  - To add to a list, use `--add-<option> <value>`.
  - To set a string value, use `--set-<option> <value>`.
  - To set using a NRDL string, use `--nrdl-<option> <value>`.
  - To set using NRDL contents from a nrdl document from file,
    use `--file-<option> <url>`.
  - To set using the raw bytes of a file, use `--raw-<option> <url>`.

  - For `--file-<option>` and `--raw-<option>`, URLs are also supported:
      - `http(s)://user:password@url` for basic auth
      - `http(s)://header=val@url` for a header
      - `http(s)://token@url` for a bearer token
      - `file://location` for a local file
```

```
      - `-` for standard input
    Anything else is treated as a file name.


The following options have been detected:
{
}
Documentation not found.

No action exists for the command.

{
    status successful
}
0
```

CLIFF generated a generous help page for us. Since we didn't tell it to *do* anything, it figured it should print the help page.

First, let's compile our program into an executable image and run `main` as the entry point. There are lots of ways to do this. On SBCL, the typical tool of choice is `save-lisp-and-die`, like this:

```
(sb-ext:save-lisp-and-die
  "calc"
  :toplevel #'main
  :executable t
  :compression t
  :save-runtime-options t)
```

Then from the command line:

```
$ calc
```

We get the same output as before.

Open World Configuration

CLIFF is a different kind of CLI framework. It assumes what we'll call an "open world" configuration. It defines a correct format for options to appear in configuration files, the environment, and the command line, and gathers them as it sees them. This frees the developer from having to over-specify their command line options when just starting out. It also enables some interesting workflows where the developer can pass the options hash table down to lower libraries, which may or may not expect certain options set in the table.

Since we didn't tell it to do anything yet, it just prints the help page in addition to what it finds. This part of the output shows what options have been found:

```
The following options have been detected:
{
}
```

Looks like it didn't find anything. Let's help it find some options.

Now let's add some options:

```
$ calc --enable-floating-point
```

When we run this, CLIFF's help page prints this at the bottom:

```
{
    floating-point true
}
```

It's printing out that big nested hash table of options it finds, but in NRDL, a JSON superset data language designed to work well for configuration files and as well as command line output for programs written in lisp. It was specially built to power CLIFF.

The equivalent alist might be

```
'((:FLOATING-POINT . t))
```

Cool. Let's try to add more options.

```
export CALC_LIST_OPERANDS=1,2,3,4,5
export CALC_TABLE_NAMES=plus=+,minus=-,times=*
./calc
```

The help page now shows this:

```
{
    floating-point true
    names {
        minus "-"
        plus "+"
        times "*"
    }
    operands [
```

```
        "1"
        "2"
        "3"
        "4"
        "5"
    ]
}
```

Notice that the `operands` are added as strings. This is true of most options added to the options hash table via command line arguments or environment variables. However, there is a workaround: using `--nrdl-*` on the CLI or the `CALC_NRDL_<thing>` environment variable.

To demonstrate this, we run our command with a `--nrdl-*` option as follows:

```
export CALC_NRDL_OPERANDS=[1,2,3,4,5]
./calc --nrdl-divisors '{"one": 1, "two": 2, "three": 3}'
```

Doing so yields this output at the end of the help page:

```
{
    divisors {
        "one" 1
        "three" 3
        "two" 2
    }
}
```

The user can specify *any* option in this way, *even if the program doesn't use it.* This way, the command can take some options verbatim and pass it on, print it out, return it, or change it in arbitrary ways. This open-world assumption of options allows the program to compose better with libraries and other programs.

The Options Tower

CLIFF gathers options for the calling code tool (in this case `calc`) from configuration files, environment variables, and command-line flags. It merges what it finds from various sources into one single, nested hash table. The various sources comprise what we will call the Options Tower.

Let's add something from a configuration file now. CLIFF looks in the present working directory, as well as an OS-dependent, home-based location, which is printed out in the help page.

Let's make a file corresponding to the "home directory" option. Make a file called `~/.config/calc/config.nrdl`(on Linux) and put the following content in it (if you are on a different OS, consult the help page above):

```
{
    floating-point-size double
    scale 11
    name-of-calculation "fizzle"
    calculations [
       {
          operands [
            1
            2
            3
            4
            5
          ]
          operator +
        }
        {
          operands [
            10
            20
            30
            40
            50
          ]
          operator *
        }
      ]
}
```

Now we run `./calc` and it yields this output:

```
{
    calculations [
        {
            operands [
                1
                2
                3
                4
                5
            ]
            operator +
        }
        {
            operands [
```

```
                10
                20
                30
                40
                50
            ]
            operator *
        }
    ]
    floating-point-size double
    name-of-calculation "fizzle"
    scale 11
}
```

Now we add a file called `.calc.nrdl` in the current directory with this content:

```
{
  flaoting-point false
  sumall true
}
```

Now when we run `./calc`, we see that it sees these options:

```
{
    calculations [
        {
            operands [
                1
                2
                3
                4
                5
            ]
            operator +
        }
        {
            operands [
                10
                20
                30
                40
                50
            ]
            operator *
```

```
        }
    ]
    floating-point false
    floating-point-size double
    name-of-calculation "fizzle"
    names {
        minus "-"
        plus "+"
        times "*"
    }
    operands [
        "1"
        "2"
        "3"
        "4"
        "5"
    ]
    scale 11
    sumall true
}
```

Note the `sumall` key we added in `.calc.nrdl` and the `floating-point` key. The `floating-point` key was true in the home config file, but was overridden in the current directory config file. Now, CLIFF looks both in the present working directory *and all its parents*, and uses the file it may or may not find as an override to the home directory config file, which serves as a base.

Let's say we want to override the `scale` key with environment variables. We set `CALC_NRDL_SCALE=10`(we use NRDL to ensure that, as a number, it is parsed).

Now we see that `scale` is 10 in the output.

To override environment variables, set command line flags.

To see this, we will call `calc` with an option override for `set-name-of-calculation`:

```
./calc --set-name-of-calculation dizzle
```

We see that `name-of-calculation` is set to `"dizzle"` in the printed out help page.

Sweet. We have a bunch of options. Our app observes 12-factor goodness by default, checking for config files, environment variables, and even harvesting options for us from the command line, all without us having to write much.

Add a Default Function

Now let's do something with this information. Right now the command just prints out a help page when the command line tool is called. Change the lisp file which calls `execute-program`

and add a default function, like this:

```
;;;; calculator.lisp -- A CLI calculator.
;;;;
;;;; SPDX-FileCopyrightText: 2024 Daniel Jay Haskin
;;;; SPDX-License-Identifier: MIT
;;;;
;;;;

(in-package #:cl-user)

(defpackage
  #:com.djhaskin.calc (:use #:cl)
  (:documentation
    "
    A CLI calculator.
    ")
  (:import-from #:com.djhaskin.cliff)
  (:local-nicknames
    (#:nrdl #:com.djhaskin.nrdl)
    (#:cliff #:com.djhaskin.cliff))
  (:export #:main))

(in-package #:com.djhaskin.calc)


(defparameter operators
    `(("+" . ,#'+)
      ("-" . ,#'-)
      ("*" . ,#'*)
      ("/" . ,#'/)))

(defun calc (options)
  (let* ((result (make-hash-table :test #'equal))

         (operands (cliff:ensure-option-exists :operands options))
         (operator (cliff:ensure-option-exists :operator options))
         (func (cdr (assoc operator operators :test #'equal))))
    (setf (gethash :result result)(apply func operands))
    (setf (gethash :status result) :successful)
    result))


(defun main ()
  (sb-ext:exit
```

```
    :code
    (cliff:execute-program
      "calc"
      :default-function #'calc)))
```

We have some changes here in `calculator.lisp` from our original listing.

First, we create a new function called `calc`. It takes one argument, `options`, which will be that hash table of options we have been discussing building.

It calls `cliff:ensure-option-exists` on specific options that it expects to be present in the hash table. This function ensures a particular key exists in the hash table, and if it doesn't, it signals an error.

We then set it as the `:default-function` when we call `cliff:execute-program`.

We also add `sb-ext:exit` and set `:code` as the return value of `cliff:execute-program` to ensure the exit code is propogated to the OS.

Now we compile and run our program:

```
$ ./calc
{
    error-message
        |Abnormal exit error
        |
        ^
    missing-option operands
    status cl-usage-error
}
$ echo $?
64
```

We see that a `:cl-usage-error` error has been signaled and that the command returned a non-zero status code corresponding to the type of error signaled. The error is printed to standard output in the form of a NRDL document.

In fact, by default, everything CLIFF prints out will be in the form of a NRDL document, though as we'll see, this can be turned off. This default is to enable CLIFF to fulfill its mission: just plug a few functions into CLIFF, and it'll handle the rest. I/O is all taken care of by default, in a human-friendly machine readable format.

Let's call `./calc` again, with operands and an operator.

Because CLIFF doesn't care where it gets its options, we can mix and match where they come from. In our example, we'll say that we pretty much always want `calc` to use `+` as an operator, unless we want to override it. We'll put that in our configuration file, and specify the operands on the CLI.

We put this in our `.calc.nrdl` in the current directory:

```
{
    operator "+"
}
```

And then we call `./calc`:

```
$ ./calc --nrdl-operands '[1,2,3,4]'
{
   result 10
   status successful
}
```

String Conversion

Next, we observe that specifying operands might be more convenient if they were specified one at a time, like this:

```
./calc --add-operands 1 --add-operands 2 --add-operands 3 --add-operands 4
```

If we run that though, we get an error:

```
$ ./calc --add-operands 1 --add-operands 2 --add-operands 3 --add-operands 4
{
    error-datum "\"4\""
    error-expected-type "NUMBER"
    error-message
        |The value
        |  "4"
        |is not of type
        |  NUMBER
        ^

    status data-format-error
}
```

Calc doesn't know what "type" arguments are when they are specified on the command line, so it assumes they are a string. We can see this when we run

```
./calc help --add-operands 1 --add-operands 2
```

The help page shows what `calc` actually sees:

```
The following options have been detected:
{
    operands [
        "2"
        "1"
    ]
    operator [
        "+"
    ]
}
```

It shows our default operator, but it also shows our operands as a list of strings.

If we expect that our operands will always be specified on the command line rather than the environment or via config file, we may wish to check for strings and convert them to non-strings if possible.

To allow for this, CLIFF provides `:setup` and `:teardown` optional arguments to `execute-program`. The `:setup` function takes an options map and return a modified version. This is the version which the main logic functions will see. The `:teardown` function takes the map that the main logic functions create, changes or creates a new version based on it, and returns that. This modified map will be what CLIFF sees when it starts to try to wrap up the program.

These functions provide a lot of power in terms of what we can do or how we can interact with CLIFF.

To accomplish the string to number transformation, we add a `:setup` lambda:

```
(defun main (argv)
  (cliff:execute-program
    "calc"
    :default-function #'calc
    :cli-arguments argv
    :defaults '((:operator "+"))
    :setup (lambda (options)
             (let ((operands (gethash :operands options)))
               (setf (gethash :operands options)
                     (map 'list #'parse-integer operands))
               options))))
```

Then we recompile and run again:

```
$ ./calc --add-operands 1 --add-operands 2
{
    result 3
    status successful
}
```

The downside is that operands would need to be specified as strings if there ever were a need to put them in a configuration file, but if the target audience typically uses the command line to specify the arguments, then maybe this is a good trade-off.

Provide Command-Line Aliases

It feels bad to make the user punch in `--add-operands` for every operand. We would like to enable a single letter for that option, so we will add a CLI alias for it using `execute-program`'s optional `:cli-aliases` option:

```
(defun main (argv)
  (cliff:execute-program
    "calc"
    :default-function #'calc
    :cli-arguments argv
    :defaults '((:operator "+"))
    :cli-aliases
    '(("-h" . "help")
      ("--help" . "help")
      ("-o" . "--add-operands"))
    :setup (lambda (options)
             (let ((operands (gethash :operands options)))
               (setf (gethash :operands options)
                     (map 'list #'parse-integer operands))
               options))))
```

Note, we also added `--help` and `-h` aliases.

CLI Aliases are simple substitutions. If CLIFF sees what is specified as a key in the alist on the command line, it will replace it with the value.

CLIFF provides a ‘help‘ subcommand, but not a ‘--help‘ or ‘-h‘ option. Providing these aliases will help the user if they don't know what to do.

We also added the ‘-o‘ alias to mean ‘--add-operands‘.

Now we can recompile and run with the new, nice shorter arguments:

```
$ ./calc -o 1 -o 2 -o 3
{
    result 6
    status successful
}
```

Override Default Output

The main mission of CLIFF is to enable users to write potentially pure functions, and hook them up to the command-line, configuration files, and the environment using `execute-program` so that the function can just do what functions do best: compute. In order to fulfill its mission, it provides default output, which is simply printing the result hash table returned by the command function in NRDL format.

However, if more control over output is desired, it is easy to take back control.

We first add `:suppress-final-output t` to the call to `execute-program`:

```
(defun main (argv)
  (cliff:execute-program
    "calc"
    :default-function #'calc
    :cli-arguments argv
    :defaults '((:operator "+"))
    :cli-aliases
    '(("-h" . "help")
      ("--help" . "help")
      ("-o" . "--add-operands"))
    :setup (lambda (options)
             (let ((operands (gethash :operands options)))
               (setf (gethash :operands options)
                     (map 'list #'parse-integer operands))
               options))
    :suppress-final-output t))
```

We also add a `format` call to our `calc` function:

```
(defun calc (options)
  (let* ((result (make-hash-table :test #'equal))
         (operands (cliff:ensure-option-exists :operands options))
         (operator (cliff:ensure-option-exists :operator options))
         (func (cdr (assoc "+" operators :test #'equal)))
         (out (apply func operands)))
    (format t "~A~%" out)
    (setf (gethash :result result) out)
    (setf (gethash :status result) :successful)
    result))
```

Now our output is much simpler:

```
$ ./calc -o 1 -o 2 -o 3
6
```

Also, the CLI aliases we defined were added automatically to the help page:

```
$ ./calc help
...
The following command line aliases have been defined:

      This                                          Translates To
        -h                                                  help
    --help                                                  help
        -o                                         --add-operands
...
```

Add Subcommands

As a means of demonstration, we will add different calculation operators (multiply, divide, add, subtract) as subcommands and get rid of the default action, ensuring the command run with no subcommands will print the help page.

All we need to do is provide an alist mapping different collections of subcommands with different functions, where the functions expect an option table and return a result table.

First, we'll add some additional operators:

```
(defparameter operators
    `(("+" . ,#'+)
      ("-" . ,#'-)
      ("*" . ,#'*)
      ("/" . ,#'/)
      ("&" . ,#'logand)
      ("%" . ,(lambda (&rest args)
                (multiple-value-bind
                    (quotient remainder)
                    (apply #'truncate args)
                  remainder)))))
```

For demonstration purposes, we'll create some functions that just set the operator and then pass execution into our previously created `calc` function:

```
(defun programmer-and (options)
  (setf (gethash :operator options) "&")
  (calc options))

(defun modulus (options)
  (setf (gethash :operator options) "%")
  (calc options))
```

Then we just add these new functions as subcommands:

```
(defun main (argv)
  (cliff:execute-program
    "calc"
    :subcommand-functions
    `((("programmer" "and") . ,#'programmer-and)
      (("modulus") . ,#'modulus))
    :default-function #'calc
    :cli-arguments argv
    :defaults '((:operator "+"))
    :cli-aliases
    '(("-h" . "help")
      ("--help" . "help")
      ("-o" . "--add-operands"))
    :setup (lambda (options)
             (let ((operands (gethash :operands options)))
               (setf (gethash :operands options)
                     (map 'list #'parse-integer operands))
               options))
    :suppress-final-output t))
```

Not the `:subcommand-functions` argument above. It maps collections of subcommands to the function that should be called when that subcommand is specified.

After compiling again, we can now do this:

```
$ ./calc programmer and -o 1 -o 3
1
$ ./calc modulus -o 3 -o 5
2
```

Add More Help Documentation

CLIFF is pretty good at adding general documentation around the option tower, but not really around each individual function.

Specifying the default function's help is pretty easy, just give a string argument to the `:default-func-help` option. Specifying help strings for the different subcommands are likewise easy; just give an alist that map subcommand strings to help strings in the `:subcommand-helps` option:

```
(defun main (argv)
  (cliff:execute-program
```

```
    "calc"
    :subcommand-functions
    `((("programmer" "and") . ,#'programmer-and)
      (("modulus") . ,#'modulus))
    :subcommand-helps
    ((("programmer" "and") . "Sets the operator to )&`")
      (("modulus") . "Sets the operator to %")
    :default-function #'calc
    :default-func-help
    (format
      nil
      "~@{~@?~}"
      "Welcome to calc.~%"
      "~%"
      "This is a calculator CLI.~%"
      "~%"
      "The default action expects these options:~%"
      "  operand           Specify operand~%"
      "                    (may be specified multiple times)~%"
      "  operator (string) Specify operator~%")
    :cli-arguments argv
    :defaults '((:operator "+"))
    :cli-aliases
    '(("-h" . "help")
      ("--help" . "help")
      ("-o" . "--add-operands"))
    :setup (lambda (options)
             (let ((operands (gethash :operands options)))
               (setf (gethash :operands options)
                     (map 'list #'parse-integer operands))
               options))
    :suppress-final-output t))
```

Now running `./calc help` shows the default function help:

```
$ ./calc help
...
Documentation:

Welcome to calc.

This is a calculator CLI.

The default action expects these options:
```

```
  operand           Specify operand
                    (may be specified multiple times)
  operator (string) Specify operator
```

Likewise, running `./calc help modulus` and `./calc help programmer and` return their respective helps at the bottom of the help page:

```
Documentation for subcommand `modulus`:

Sets the operator to `%`

Documentation for subcommand `programmer and`:

Sets the operator to `&`
```

Wrap-up

We have created a fully-functional CLI tool. It is a 12 factor app that looks in config files, the environment, and the command line for its options and merges them together. We have easily been able to add commands, subcommands, and documentation for them. We have also shown how we can have simpler arguments on the command line.

API Reference

This section details the use of the actual API presented by CLIFF.

`com.djhaskin.cliff`

This package's main export is the function `execute-program`, but it also exports other convenience functions listed below, as well as re-exporting all the public symbols of the `com.djhaskin.cliff/errors` package.

`execute-program`

```
execute-program(program-name &key (cli-arguments t) (err-strm *error-output*)
(list-sep ,) (map-sep =) (setup (function identity)) (strm *standard-output*)
(teardown (function identity)) cli-aliases default-func-help default-function
defaults disable-help environment-aliases environment-variables reference-file
root-path subcommand-functions subcommand-helps suppress-final-output)
```

Overview The function `execute-program` aims to be a simple to use one stop shop for all your command line needs.

The function gathers options from the "Option Tower", or out of configuration files, environment variables, and the command line arguments into an options table. Then it calls the *action function*, which is a user-defined function based on what subcommand was specified; either the `default-function` if no subcommands were given, or the function corresponding to the subcommand as given in `subcommand-functions` will be called. Expects that function to return a results map, with at least the `:status` key set to one of the values listed in the `*exit-codes*` hash table.

The Options Tower The function first builds, in successive steps, the options table which will be passed to the function in question.

Configuration Files It starts with the options hash table given by the `defaults` parameter.

The function next examines the given `environment-variables`, which should be given as an alist with keys as variable names and values as their values. If no list is given, the current environment variables will be queried from the OS.

Then `execute-program` uses those environment variables to find an OS-specific system-wide configuration file in one of the following locations:

- **Windows**: `%PROGRAMDATA%\<program-name>\config.nrdl`, or `C:\ProgramData\<program-name>\` if that environment variable is not set.
- **Mac**: `/Library/Preferences/<program-name>/config.nrdl`
- **Linux/POSIX**: `/etc/<program-name>/config.nrdl`

It reads this file and deserializes the options from it, merging them into the options table, overriding any options when they exist both in the map and the file.

Next, it looks for options in an OS-specific, user-specific home-directory-based configuration file in one of the following locations:

- **Windows**: `%LOCALAPPDATA%\<program-name>\config.nrdl`, or `%USERPROFILE%\AppData\Local\<` if that environment variable is not set.
- **Mac**: `$HOME/Library/Preferences/<program-name>/config.nrdl`
- **Linux/POSIX**: `$XDG_CONFIG_HOME/<program-name>/config.nrdl`, or `$HOME/.config/<program-` if XDG_CONFIG_HOME is not set.

When performing this search, `execute-program` may signal an error of type `necessary-env-var-absent` if the HOME var is not set on non-Windows environments and the `USERPROFILE` variable if on Windows.

If it finds a NRDL file in this location, it deserializes the contents and merges them into the options table, overriding options when they exist both in the map and the file.

Finally, it searches for the `reference-file` in the `root-path`. If it can't find the `reference-file` in `root-path`, it searches successively in all of `root-path`'s parent directories.

If it finds such a file in one of these directories, it next looks for the file `.<program-name>.nrdl` in that exact directory where `reference-file` was found. If that file exists, `execute-program` deserializes the contents and merges them into the options table, overriding options when they exist both in the table and the file.

If `reference-file` is not given, it is simply taken to be the configuration file itself, namely `.<program-name>.nrdl`. If `root-path` is not given, it is taken to be the present working directory.

Environment Variables Next, this function examines the `environment-variables` for any options given by environment variable. Again, `environment-variables` should be given as an alist with keys as variable names and values as their values. If no list is given, the current environment variables will be queried from the OS.

For each environment variable, it examines its form.

If the variable matches the regular expression `^(<PROGRAM_NAME>)_(?P<opt>LIST|TABLE|ITEM|FLAG|NRDL` then the variable's value will be used to add to the resulting options hash table, overriding any options which are already there.

If the `opt` part of the regex is `LIST`, the value of the variable will be split using `list-sep` and the resulting list of strings will be associated with the keyword `arg` in the options.

If the `opt` is `TABLE`, the value of the variable will be split using `list-sep`, then each entry in that list will also be split using `map-sep`. The resulting key/value pair list is turned into a hash-table and this hash table is associated to the keyword `arg` in the options.

If the `opt` is `ITEM`, the value of the variable will be set to the keyword `arg` in the options.

If the `opt` is `NRDL`, the value of the variable will be parsed as a NRDL string and its resultant value set as the value of the keyword `arg` in the returned options hash table.

In addition, any environment variables whose names match any keys in the `environment-aliases` alist will be treated as if their names were actually the value of that key's entry.

Command Line Arguments Finally, `execute-program` turns its attention to the command line.

It examines each argument in turn. It looks for options of the form `--<action>-<option-key>`, though this is configurable via `*find-tag*`. It deals with options of this form according to the following rules:

- If the argument's action is `enable` or `disable` the keyword named after the option key is associated with `t` or `nil` in the resulting hash table, respectively.

- If the argument's action is `set`, the succeeding argument is taken as the string value of the key corresponding to the option key given in the argument, overriding any previously set value within the option table.
- If the argument's action is `add`, the succeeding argument is taken as a string value which must be appended to the value of the option key within the option table, assuming that the value of such is already a list.
- If the argument's action is `join`, the succeeding argument must be of the form `<key><map-sep><value>`, where `map-sep` is the value of the parameter `map-sep`. The key specified becomes a keyword, and the value a string, set as a hash table entry of the hash table found under the option key within the parent option table, assuming the value of such is already a hash table.
- If the argument's action is `nrdl`, the succeeding argument must be a valid NRDL document, specified as a string. This argument's deserialized value will be taken as the value of the option key within the option map, overriding any previously set value within the option table.
- If the argument's action is `file`, it will be assumed that the succeeding argument names a resource consumable via `data-slurp`. That resource will be slurped in via that function, then deserialized from NRDL. The resulting data will be taken as the value of the option key within the option table, overriding any previously set value within that table.
- If the argument's action is `raw`, it will be assumed that the succeeding argument names a resource consumable via `data-slurp`. That resource will be slurped in via that function, as a raw string. That string will be taken as the value of the option key within the option table, overriding any previously set value within that table.
- If the argument's action is `reset`, the option key in question is removed from the option table.

In addition, any argument of any form whose string value `equal`'s any keys in the `cli-aliases` alist will be treated as if their string value were actually the value of that key's entry.

The Setup Function Finally, if a setup function is specified via `setup`, it is called with one argument: the options table so far. This function is expected to add or remove elements from the options table and return it.

Having done all this, `execute-program` considers the option table is complete and prepares to feed it to the action function.

Determining the Action Function Any other arguments which `execute-program` finds on the command line other than those recognized either as `cli-aliases` or as options of the form matched by `*find-tag*` will be taken as subcommand terms.

`execute-program` then finds all such terms puts them in a list in the order in which they were found. It attempts to find this list (using `equal`) in the alist `subcommand-functions`. If such a list exists as a key in that alist, the value corresponding to that key is taken to

be the action function. All action functions must be a function of one hash table as an argument. This will be the options map previously constructed.

If no subcommand was given, the `default-function` is taken as the action function instead.

The Help Page By default, if `execute-program` sees the `help` subcommand on in the command line arguments, it will print a help page to `err-strm`. `err-strm` may be given as `t`, `nil`, or otherwise must be a stream, just as when calling `format`. If left unspecified, `err-strm` defaults to standard error. This behavior may be suppressed by setting the `disable-help` option to `nil`. If disabled, Users may then define their own help pages by specifying functions that print them using `subcommand-functions`.

This help page gives users the following information:

- Details to users how they may specify options using the Options Tower for the program
- Lists all defined environment variable aliases
- Lists all defined command line interface aliases
- Prints out all options found within the Options Tower in NRDL format.
- Prints out whether there is a default action (function) defined.
- Prints out all available subcommands

If there were any subcommand terms after that of the `help` term in the command line arguments, they are put in a list and `execute-program` attempts to find this list (again, using `equal`) as a key in the alist `subcommand-helps`. It then prints this help string as part of the documentation found in the help page. If it If there were no such terms after the `help` term in the command line arguments, `execute-program` prints the help string found in `default-help`, if any.

Execution If `execute-program` is able to determine an action function and what options to put in the option table, it calls that function with the found options. This function is either that which prints the default help page as described above, it comes from `subcommand-functions` and was chosen based on present subcommand terms on the command line, or comes from `default-function` if no subcommand was given on the command line. If no match was found in `subcommands-functions` matching the subcommands given on the command line, an error is printed. This function is called the action function.

It computes the result hash table by taking the return value value of the action function and passing it to the function specified in the `teardown` parameter, if it was given. If not, the result from the action function is taken as the result hash table itself.

By default, it then prints this hash table out to `strm` as a prettified NRDL document. `strm` may be given as `t`, `nil`, or otherwise must be a stream, just as when calling `format`. If left unspecified, `strm` defaults to standard output.

This return value is expected to be a hash table using `eql` semantics. That table must contain at least one value under the `:status` key. The value of this key is expected to

be one of the keys found in the `*exit-codes*` alist corresponding to what should be the exit status of the whole program. If the function was successful, the value is expected to be `:successful`. This value will be used as the key to look up a numeric exit code from c(*exit-codes*). The numeric exit code found will be taken as the desired exit code of the whole program, and will be the first value returned by the function `execute-program`. The second value will be the result hash table itself.

Error Handling During the entirety of its run, `execute-program` handles any and all `serious-condition`s. If one is signaled, it computes the exit status of the condition using `exit-status` and creates a final result vector containing the return value of that function under the `:status` key. It then populates this table with a key called `error-message` and any key/value pairs found in the alist computed by calling `exit-map-members` on the condition.

It prints this table out in indented NRDL format to `err-strm`(or standard error if that option is left unspecified) unless `suppress-output` is given as `t`.

`execute-program` then returns two values: the numeric exit code corresponding to the exit status computed as described above, and the newly constructed result map containing the error information.

Discussion Command line tools necessarily need to do a lot of I/O. `execute-program` attempts to encapsulate much of this I/O while providing clean architecture by default. Ideally, `execute-program` should enable an action function to be relatively pure, taking an options hash table and returning a result hash table with no other I/O required. This is why `execute-program` prints out the resulting hash table at the end. If a pure action function is called, hooking it up to a subcommand or as the default command action using `execute-program` should enable this function to interact with the outside world by means of its result table.

It was also written with dependency injection, testing, and the REPL in mind. Since `execute-program` doesn't actually exit the process at the end, only returning values instead, `execute-program` may simply be called at the REPL. Many arguments to the function only exist for dependency injection, which enables both testing and REPL development. Generally, many arguments won't be specified in a function call, such as the `cli-arguments`, `environment-variables`, `err-strm` and `strm` parameters (though of course they may be specified if e.g. the user needs to redirect output to a file at need.)

`ensure-option-exists`

`ensure-option-exists(key options)`

Check options hash table `options` for the key `key`.

Signal a restartable condition if the option is missing. Employs the `use-value` and `continue` restarts in that case.

This function is meant to be used by the user to ensure an option exists.

data-slurp

`data-slurp(resource &rest more-args)`

Slurp a resource, using specified options. Return the contents of the resource as a string.

If `resource` is a URL, download the contents according to the following rules:

- If it is of the form 'http(s)://user:password@url', it performs basic HTTP authentication using the provided username and password;
- If it is of the form 'http(s)://header=val@url', the provided header is set when downloading the contents;
- If it is of the form 'http(s)://<token>@url', bearer authorization is used with the provided token;
- If it is of the form `file://<location>`, it is loaded as a normal file;
- If it is of the form `-` the contents are loaded from standard input;

Otherwise, the contents are loaded from the resource as if it named a file.

parse-string

`parse-string(thing)`

Parse the NRDL string `thing`.

generate-string

`generate-string(thing &optional &key (pretty 0))`

Serialize `thing` to NRDL.

find-file

`find-file(from marker)`

Starting at the directory `from`, look for the file `marker`.

Continue looking for the file in successive parents of `from` until no more parents exist or the file is found.

Returns the pathname of the found file or nil if no file could be found.

necessary-env-var-absent

`necessary-env-var-absent`

Option	Value
Superclasses:	`(error t)`
Metaclass:	`sb-pcl::condition-class`
Default Initargs:	`nil`

Condition used by CLIFF to signal that a required environment variable is not present.

Implmements `exit-status` and `exit-map-members`.

- `env-var`
  Environment variable that should exist (but doesn't).

Option	Value
Allocation:	instance
Type:	`nil`
Initarg:	`:env-var`
Initform:	`(error "Need to give argument `:env-var`.")`
Readers:	`(env-var)`

`invalid-subcommand`

`invalid-subcommand`

Option	Value
Superclasses:	`(error t)`
Metaclass:	`sb-pcl::condition-class`
Default Initargs:	`nil`

Condition used by CLIFF to signal there were no functions given to CLIFF that correspond the subcommand given.

Implmements `exit-status` and `exit-map-members`.

- `given-subcommand`
  Subcommand terms found on the command line.

Option	Value
Allocation:	instance
Type:	`nil`
Initarg:	`:given-subcommand`
Initform:	`nil`
Readers:	`(given-subcommand)`

find-tag

`*find-tag*`

cl-ppcre regex scanner containing two capture groups, the first of which must capture the CLI verb (one of `enable`, `disable`, `reset`, `add`, `set`, `nrdl`, or `file`), and the second of which is the name of the variable. This is used ultimately by `execute-program` to recognize command line options (as opposed to subcommands). Currently its value corresponds to the regular expression `"^--([^-]+)-(.+)$"`.

com.djhaskin.cliff/errors

exit-codes

`*exit-codes*`

This parameter points to a hash table mapping keywords used by CLIFF to numeric error codes. These error codes and their names are taken from Linux's `/usr/include/sysexit.h` in an attempt to be somewhat compliant to that OS's standard.

Here are its contents:

Exit Code Name	Exit Code
:unknown-error	128
:general-error	1
:successful	0
:cl-usage-error	64
:data-format-error	65
:no-input-error	66
:no-user-error	67
:no-host-error	68
:service-unavailable	69
:internal-software-error	70
:system-error	71
:os-file-error	72
:cant-create-file	73
:input-output-error	74
:temporary-failure	75
:protocol-error	76
:permission-denied	77
:configuration-error	78

exit-status

`exit-status(condition)`

Return a keyword describing the program exit status implied by a given condition. This keyword must be in `*exit-codes*`.

Users may define their own methods to this function.

`exit-map-members`

`exit-map-members(condition)`

Return an alist of items to be added to the exit map of CLIFF in the event of the condition in question being caught by `execute-program`.

This generic function has been implemented for all standard Common Lisp `condition` types.

Users may define their own methods to this function. Arbitrary mappings between keywords and NRDL-serializable objects are allowed in the resulting alist.