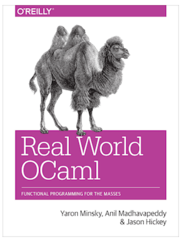


Buy in print and eBook.

Login with GitHub to view and add comments

Chapter 14. Command-Line Parsing

Many of the OCaml programs that you'll write will end up as binaries that need to be run from a command prompt. Any nontrivial command line should support a collection of basic features:

- Parsing of command-line arguments
- Generation of error messages in response to incorrect inputs
- Help for all the available options
- Interactive autocompletion

It's tedious and error-prone to code all of this manually for every program you write. Core provides the Command library, which simplifies all of this by letting you declare all your command-line options in one place and by deriving all of the above functionality from these declarations.

Command is simple to use for simple applications but also scales well as your needs grow more complex. In particular, Command provides a sophisticated subcommand mode that groups related commands together as the complexity of your user interface grows. You may already be familiar with this command-line style from the Git or Mercurial version control systems.

In this chapter, we'll:

- Learn how to use Command to construct basic and grouped command-line interfaces
- We will build simple equivalents to the cryptographic `md5` and `shasum` utilities
- Demonstrate how *functional combinators* can be used to declare complex command-line interfaces in a type-safe and elegant way

BASIC COMMAND-LINE PARSING

Let's start by working through a clone of the *md5sum* command that is present on most Linux installations (the equivalent command on Mac OS X is simply `md5`). The following function defined below reads in the contents of a file, applies the MD5 one-way cryptographic hash function to the data, and outputs an ASCII hex representation of the result:

```
open Core.Std

let do_hash file =
  In_channel.with_file file ~f:(fun ic ->
    let open Cryptokit in
    hash_channel (Hash.md5 ()) ic
    |> transform_string (Hexa.encode ())
    |> print_endline
  )
```

OCaml ∗ command-line-parsing/basic_md5.ml ∗ all code

The `do_hash` function accepts a `filename` parameter and prints the human-readable MD5 string to the console standard output. The first step toward turning this function into a command-line program is to declare all the possible command-line arguments in a *specification*. `Command.Spec` defines combinators that can be chained together to define optional flags and positional arguments, what types they should map to, and whether to take special actions (such as pausing for interactive input) if certain inputs are encountered.

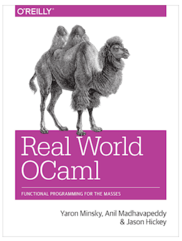
Anonymous Arguments

Let's build the specification for a single argument that is passed directly on the command line. This is known as an *anonymous* argument:

```
let spec =
  let open Command.Spec in
  empty
  +> anon ("filename" %: string)
```

OCaml ∗ command-line-parsing/basic_md5.ml , continued (part 1) ∗ all code

The `Command.Spec` module defines the tools you'll need to build up a command-line specification. We start with the `empty` value and add parameters to that using the `+>` combinator.



Buy in print and eBook.

Login with GitHub to view and add comments

(Both of these values come from `Command.Spec`.)

In this case, we defined a single anonymous argument called `filename`, which takes a value of type `string`. Anonymous parameters are created using the `%:` operator, which binds a textual name (used in the help text to identify the parameter) to an OCaml conversion function that parses the command-line string fragments into a higher-level OCaml data type. In the preceding example, this is just `Command.Spec.string`, but we'll see more complex conversion options later in the chapter.

Defining Basic Commands

Once we've defined a specification, we need to put it to work on real input. The simplest way is to directly create a command-line interface via the `Command.basic` module:

```
let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    spec
    (fun filename () -> do_hash filename)
```
OCaml ∗ command-line-parsing/basic_md5.ml , continued (part 2) ∗ all code

`Command.basic` defines a complete command-line interface that takes the following extra arguments, in addition to the ones defined in the specification:

`summary`
A required one-line description to go at the top of the command help screen.

`readme`
For longer help text when the command is called with `-help`. The `readme` argument is a function that is only evaluated when the help text is actually needed.

The specification and the callback function follow as nonlabeled arguments.

The callback function is where all the work happens after the command-line parsing is complete. This function is applied with the arguments containing the parsed command-line values, and it takes over as the main thread of the application. The callback's arguments are passed in the same order as they were bound in the specification (using the `+>` operator).

The Extra unit Argument to Callbacks

The preceding callback needs an extra `unit` argument after `filename`. This is to ensure that specifications can work even when they are empty (i.e. the `Command.Spec.empty` value).

Every OCaml function needs at least one argument, so the final `unit` guarantees that it will not be evaluated immediately as a value if there are no other arguments.

Running Basic Commands

Once we've defined the basic command, running it is just one function call away:

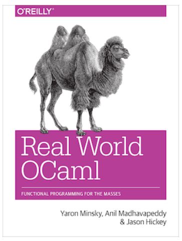
```
let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```
OCaml ∗ command-line-parsing/basic_md5.ml , continued (part 3) ∗ all code

`Command.run` takes a couple of optional arguments that are useful to identify which version of the binary you are running in production. You'll need to install Cryptokit via `opam install cryptokit` before building this example. Once that's completed, run the following to compile the binary:

```
$ corebuild -pkg cryptokit basic_md5.native
```
Terminal ∗ command-line-parsing/build_basic_md5.out ∗ all code

You can now query the version information for the binary you just compiled:

```
$ ./basic_md5.native -version
 1.0
$ ./basic_md5.native -build-info
 RWO
```


Buy in print and eBook.

Login with GitHub to view and add comments

Terminal ∗ command-line-parsing/get_basic_md5_version.out ∗ all code

The versions that you see in the output were defined via the optional arguments to `Command.run`. You can leave these blank in your own programs or get your build system to generate them directly from your version control system (e.g., by running `hg id` to generate a build revision number, in the case of Mercurial):

```
$ ./basic_md5.native
 Generate an MD5 hash of the input data

   basic_md5.native FILENAME

 More detailed information

 === flags ===

   [-build-info]  print info about this build and exit
   [-version]     print the version of this build and exit
   [-help]        print this help text and exit
                  (alias: -?)

 missing anonymous argument: FILENAME
```
Terminal ∗ command-line-parsing/get_basic_md5_help.out ∗ all code

When we invoke this binary without any arguments, it helpfully displays all of the command-line options available, along with a message to the standard error that informs you that a required argument `filename` is missing.

If you do supply the `filename` argument, then `do_hash` is called with the argument and the MD5 output is displayed to the standard output:

```
$ ./basic_md5.native ./basic_md5.native
 70542622b37dd76c09296bc86dac5dec
```
Terminal ∗ command-line-parsing/run_basic_md5.out ∗ all code

And that's all it took to build our little MD5 utility! Here's a complete version of the example we just walked through, made slightly more succinct by removing intermediate variables:

```
open Core.Std

let do_hash file () =
  In_channel.with_file file ~f:(fun ic ->
    let open Cryptokit in
    hash_channel (Hash.md5 ()) ic
    |> transform_string (Hexa.encode ())
    |> print_endline
  )

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(empty +> anon ("filename" %: string))
    do_hash

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```
OCaml ∗ command-line-parsing/basic_md5_succinct.ml ∗ all code

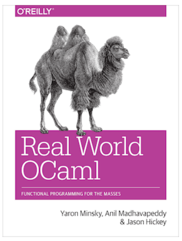
Now that we have the basics in place, the rest of the chapter will examine some of the more advanced features of Command.

ARGUMENT TYPES

You aren't just limited to parsing command lines as strings, of course. `Command.Spec` defines several other conversion functions (shown in Table 14.1, "Conversion functions defined by `Command.spec`") that validate and parse input into various types.

Table 14.1. Conversion functions defined by `Command.spec`

Argument type	OCaml type	Example
`string`	`string`	`foo`


Buy in print and eBook.

Login with GitHub to view and add comments

Argument type	OCaml type	Example
int	int	123
float	float	123.01
bool	bool	true
date	Date.t	2013-12-25
time_span	Span.t	5s
file	string	/etc/passwd

We can tighten up the specification of the command to `file` to reflect that the argument must be a valid filename, and not just any string:

```
let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(empty +> anon ("filename" %: file))
    do_hash

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```

OCaml ∗ command-line-parsing/basic_md5_as_filename.ml , continued (part 1) ∗ all code

Running this with a nonexistent filename will now output an error if the file doesn't exist. As a bonus, it also enables interactive command-line completion to work on the filename argument (explained later in the chapter):

```
$ ./basic_md5_as_filename.native nonexistent
 Uncaught exception:

   (Sys_error "nonexistent: No such file or directory")

 Raised by primitive operation at file "pervasives.ml", line 292, characters 20-46
 Called from file "lib/in_channel.ml", line 19, characters 46-65
 Called from file "lib/exn.ml", line 69, characters 6-10
```

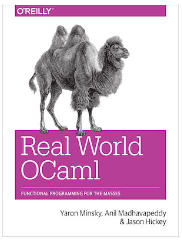
Terminal ∗ command-line-parsing/run_basic_md5_as_filename.out ∗ all code

Defining Custom Argument Types

We can also define our own argument types if the predefined ones aren't sufficient. For instance, let's make a `regular_file` argument type that ensures that the input file isn't a character device or some other odd UNIX file type that can't be fully read:

```
open Core.Std

let do_hash file () =
  In_channel.with_file file ~f:(fun ic ->
    let open Cryptokit in
    hash_channel (Hash.md5 ()) ic
    |> transform_string (Hexa.encode ())
    |> print_endline
  )

let regular_file =
  Command.Spec.Arg_type.create
    (fun filename ->
       match Sys.is_file filename with
       | `Yes -> filename
       | `No | `Unknown ->
         eprintf "'%s' is not a regular file.\n%!" filename;
         exit 1
    )

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(empty +> anon ("filename" %: regular_file))
    do_hash

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```

OCaml ∗ command-line-parsing/basic_md5_with_custom_arg.ml ∗ all code



Buy in print and eBook.

Login with GitHub to view and add comments

The `regular_file` function transforms a `filename` string parameter into the same string but first checks that the file exists and is a regular file type. When you build and run this code, you will see the new error messages if you try to open a special device such as `/dev/null`:

```
$ ./basic_md5_with_custom_arg.native /etc/passwd
 b96af7576939a17ac4b2d4b6edb50ce7
$ ./basic_md5_with_custom_arg.native /dev/null
 '/dev/null' is not a regular file.
```

Terminal ∗ command-line-parsing/run_basic_md5_with_custom_arg.out ∗ all code

Optional and Default Arguments

A more realistic MD5 binary could also read from the standard input if a `filename` isn't specified:

```
let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(empty +> anon (maybe ("filename" %: string)))
    do_hash

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```

OCaml ∗ command-line-parsing/basic_md5_with_optional_file_broken.ml , continued (part 1) ∗ all code

This just wraps the `filename` argument declaration in the `maybe` function to mark it as an optional argument. However, building this results in a compile-time error:

```
$ corebuild -pkg cryptokit basic_md5_with_optional_file_broken.native
 File "basic_md5_with_optional_file_broken.ml", line 18, characters 4-11:
 Error: This expression has type string -> unit -> unit
        but an expression was expected of type string option -> unit -> unit
        Type string is not compatible with type string option
 Command exited with code 2.
```

Terminal ∗ command-line-parsing/build_basic_md5_with_optional_file_broken.out ∗ all code

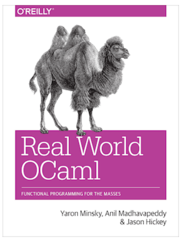
This is because changing the argument type has also changed the type of the callback function. It now wants a `string option` instead of a `string`, since the value has become optional. We can adapt our example to use the new information and read from standard input if no file is specified:

```
open Core.Std

let get_inchan = function
  | None | Some "-" ->
    In_channel.stdin
  | Some filename ->
    In_channel.create ~binary:true filename

let do_hash filename () =
  let open Cryptokit in
  get_inchan filename
  |> hash_channel (Hash.md5 ())
  |> transform_string (Hexa.encode ())
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(empty +> anon (maybe ("filename" %: file)))
    do_hash

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```

OCaml ∗ command-line-parsing/basic_md5_with_optional_file.ml ∗ all code

The `filename` parameter to `do_hash` is now a `string option` type. This is resolved into an input channel via `get_inchan` to determine whether to open the standard input or a file, and then the rest of the command is similar to our previous examples.

Another possible way to handle this would be to supply a dash as the default filename if one isn't specified. The `maybe_with_default` function can do just this, with the benefit of not having to change the callback parameter type (which may be a problem in more complex applications).



Buy in print and eBook.

Login with GitHub to view and add comments

The following example behaves exactly the same as the previous example, but replaces `maybe` with `maybe_with_default`:

```
open Core.Std

let get_inchan = function
  | "-"      -> In_channel.stdin
  | filename -> In_channel.create ~binary:true filename

let do_hash filename () =
  let open Cryptokit in
  get_inchan filename
  |> hash_channel (Hash.md5 ())
  |> transform_string (Hexa.encode ())
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(
      empty
      +> anon (maybe_with_default "-" ("filename" %: file))
    )
    do_hash

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```

OCaml ∗ command-line-parsing/basic_md5_with_default_file.ml ∗ all code

Building and running both against a system file confirms that they have the same behavior:

```
$ cat /etc/passwd | ./basic_md5_with_optional_file.native
 b96af7576939a17ac4b2d4b6edb50ce7
$ cat /etc/passwd | ./basic_md5_with_default_file.native
 b96af7576939a17ac4b2d4b6edb50ce7
```

Terminal ∗ command-line-parsing/run_basic_and_default_md5.out ∗ all code

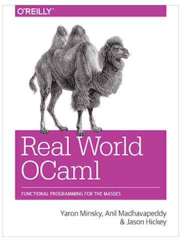
Sequences of Arguments

One last transformation that's useful is to obtain lists of anonymous arguments rather than a single one. As an example, let's modify our MD5 code to take a collection of files to process on the command line:

```
open Core.Std

let do_hash filename ic =
  let open Cryptokit in
  hash_channel (Hash.md5 ()) ic
  |> transform_string (Hexa.encode ())
  |> fun md5 -> printf "MD5 (%s) = %s\n" filename md5

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(empty +> anon (sequence ("filename" %: file)))
    (fun files () ->
       match files with
       | [] -> do_hash "-" In_channel.stdin
       | _  ->
         List.iter files ~f:(fun file ->
           In_channel.with_file ~f:(do_hash file) file
         )
    )

let () =
  Command.run ~version:"1.0" ~build_info:"RWO" command
```

OCaml ∗ command-line-parsing/basic_md5_sequence.ml ∗ all code

The callback function is a little more complex now, to handle the extra options. The `files` are now a `string list`, and an empty list reverts to using standard input, just as our previous `maybe` and `maybe_with_default` examples did. If the list of files isn't empty, then it opens up each file and runs them through `do_hash` sequentially.



Buy in print and eBook.

Login with GitHub to view and add comments

ADDING LABELED FLAGS TO THE COMMAND LINE

You aren't just limited to anonymous arguments on the command line. A *flag* is a named field that can be followed by an optional argument. These flags can appear in any order on the command line, or multiple times, depending on how they're declared in the specification.

Let's add two arguments to our `md5` command that mimics the Mac OS X version. A `-s` flag specifies the string to be hashed directly on the command line and `-t` runs a self-test. The complete example follows:

```
open Core.Std
open Cryptokit

let checksum_from_string buf =
  hash_string (Hash.md5 ()) buf
  |> transform_string (Hexa.encode ())
  |> print_endline

let checksum_from_file filename =
  let ic = match filename with
    | "-" -> In_channel.stdin
    | _   -> In_channel.create ~binary:true filename
  in
  hash_channel (Hash.md5 ()) ic
  |> transform_string (Hexa.encode ())
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    Command.Spec.(
      empty
      +> flag "-s" (optional string) ~doc:"string Checksum the given string"
      +> flag "-t" no_arg ~doc:" run a built-in time trial"
      +> anon (maybe_with_default "-" ("filename" %: file))
    )
    (fun use_string trial filename () ->
       match trial with
       | true -> printf "Running time trial\n"
       | false -> begin
           match use_string with
           | Some buf -> checksum_from_string buf
           | None -> checksum_from_file filename
         end
    )

let () = Command.run command
```

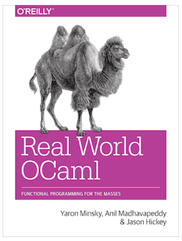
OCaml ∗ command-line-parsing/basic_md5_with_flags.ml ∗ all code

The specification now uses the `flag` function to define the two new labeled, command-line arguments. The `doc` string is formatted so that the first word is the short name that appears in the usage text, with the remainder being the full help text. Notice that the `-t` flag has no argument, and so we prepend its `doc` text with a blank space. The help text for the preceding code looks like this:

```
$ ./basic_md5_with_flags.native -help
 Generate an MD5 hash of the input data

   basic_md5_with_flags.native [FILENAME]

 === flags ===

   [-s string]    Checksum the given string
   [-t]           run a built-in time trial
   [-build-info]  print info about this build and exit
   [-version]     print the version of this build and exit
   [-help]        print this help text and exit
                  (alias: -?)

$ ./basic_md5_with_flags.native -s "ocaml rocks"
 5a118fe92ac3b6c7854c595ecf6419cb
```

Terminal ∗ command-line-parsing/run_basic_md5_flags_help.out ∗ all code

The `-s` flag in our specification requires a `string` argument and isn't optional. The Command parser outputs an error message if the flag isn't supplied, as with the anonymous arguments in


Buy in print and eBook.

Login with GitHub to view and add comments

earlier examples. Table 14.2, "Flag functions" contains a list of some of the functions that you can wrap flags in to control how they are parsed.

Table 14.2. Flag functions

Flag function	OCaml type
`required` *arg*	*arg* and error if not present
`optional` *arg*	*arg* `option`
`optional_with_default` *val arg*	*arg* with default *val* if not present
`listed` *arg*	*arg* `list`, flag may appear multiple times
`no_arg`	`bool` that is true if flag is present

The flags affect the type of the callback function in exactly the same way as anonymous arguments do. This lets you change the specification and ensure that all the callback functions are updated appropriately, without runtime errors.

GROUPING SUBCOMMANDS TOGETHER

You can get pretty far by using flags and anonymous arguments to assemble complex, command-line interfaces. After a while, though, too many options can make the program very confusing for newcomers to your application. One way to solve this is by grouping common operations together and adding some hierarchy to the command-line interface.

You'll have run across this style already when using the OPAM package manager (or, in the non-OCaml world, the Git or Mercurial commands). OPAM exposes commands in this form:

```
$ opam config env
$ opam remote list -k git
$ opam install --help
$ opam install cryptokit --verbose
```

Terminal ∗ command-line-parsing/opam.out ∗ all code

The `config`, `remote`, and `install` keywords form a logical grouping of commands that factor out a set of flags and arguments. This lets you prevent flags that are specific to a particular subcommand from leaking into the general configuration space.

This usually only becomes a concern when your application organically grows features. Luckily, it's simple to extend your application to do this in Command: just swap the `Command.basic` for `Command.group`, which takes an association list of specifications and handles the subcommand parsing and help output for you:

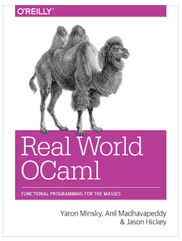
```
# Command.basic ;;
 - : summary:string ->
     ?readme:(unit -> string) ->
     ('main, unit -> unit) Command.Spec.t -> 'main -> Command.t
 = <fun>
# Command.group ;;
 - : summary:string ->
     ?readme:(unit -> string) -> (string * Command.t) list -> Command.t
 = <fun>
```

OCaml Utop ∗ command-line-parsing/group.topscript ∗ all code

The `group` signature accepts a list of basic `Command.t` values and their corresponding names. When executed, it looks for the appropriate subcommand from the name list, and dispatches it to the right command handler.

Let's build the outline of a calendar tool that does a few operations over dates from the command line. We first need to define a command that adds days to an input date and prints the resulting date:

```
open Core.Std

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
```



Buy in print and eBook.

Login with GitHub to view and add comments

```
    (fun base span () ->
       Date.add_days base span
       |> Date.to_string
       |> print_endline
    )

let () = Command.run add
```

OCaml ∗ command-line-parsing/cal_add_days.ml ∗ all code

Everything in this command should be familiar to you by now. Once you've tested it and made sure it works, we can define another new command that takes the difference of two dates. However, instead of creating a new binary, we group both operations as subcommands using `Command.group`:

```
open Core.Std

let add =
  Command.basic ~summary:"Add [days] to the [base] date"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
    (fun base span () ->
       Date.add_days base span
       |> Date.to_string
       |> print_endline
    )

let diff =
  Command.basic ~summary:"Show days between [date1] and [date2]"
    Command.Spec.(
      empty
      +> anon ("date1" %: date)
      +> anon ("date2" %: date)
    )
    (fun date1 date2 () ->
       Date.diff date1 date2
       |> printf "%d days\n"
    )

let command =
  Command.group ~summary:"Manipulate dates"
    [ "add", add; "diff", diff ]

let () = Command.run command
```

OCaml ∗ command-line-parsing/cal_add_sub_days.ml ∗ all code

And that's all you really need to add subcommand support! Let's build the example first in the usual way and inspect the help output, which now reflects the subcommands we just added.

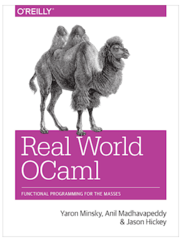
```
$ corebuild cal_add_sub_days.native
$ ./cal_add_sub_days.native -help
 Manipulate dates

   cal_add_sub_days.native SUBCOMMAND

 === subcommands ===

   add      Add [days] to the [base] date
   diff     Show days between [date1] and [date2]
   version  print version information
   help     explain a given subcommand (perhaps recursively)
```

Terminal ∗ command-line-parsing/build_cal_add_sub_days.out ∗ all code

We can invoke the two commands we just defined to verify that they work and see the date parsing in action:

```
$ ./cal_add_sub_days.native add 2012-12-25 40
 2013-02-03
$ ./cal_add_sub_days.native diff 2012-12-25 2012-11-01
 54 days
```

Terminal ∗ command-line-parsing/run_cal_add_sub_days.out ∗ all code


O'REILLY
Real World OCaml
FUNCTIONAL PROGRAMMING FOR THE MASSES
Yaron Minsky, Anil Madhavapeddy & Jason Hickey

Buy in print and eBook.

Login with GitHub to view and add comments

ADVANCED CONTROL OVER PARSING

The functions for generating a specification may seem like magic. In particular, even if you know how to use them, it's not entirely clear how they work, and in particular, why the types work out the way they do.

Understanding the details of how these specifications fit together becomes more useful as your command-line interfaces get more complex. In particular, you may want to factor out common functionality between specifications or interrupt the parsing to perform special processing, such as requesting an interactive passphrase from the user before proceeding. All of this is helped by a deeper understanding of the Command library.

In the following sections we'll explain the logic behind the combinators we've already described and show you some new combinators that let you use Command even more effectively.

The Types Behind Command.Spec

The Command module's safety relies on the specification's output values precisely matching the callback function which invokes the main program. In order to prevent any such mismatches, Command uses some interesting type machinery to guarantee they remain in sync. You don't have to understand this section to use the more advanced combinators, but it'll help you debug type errors as you use Command more.

The `Command.Spec.t` type looks deceptively simple: `('a, 'b) t`. You can think of `('a, 'b) t` here as a function of type `'a -> 'b`, but embellished with information about:

- How to parse the command line
- What the command does and how to call it
- How to autocomplete a partial command line

The type of a specification transforms a `'a` to a `'b` value. For instance, a value of `Spec.t` might have type `(arg1 -> ... -> argN -> 'r, 'r) Spec.t`.

Such a value transforms a main function of type `arg1 -> ... -> argN -> 'r` by supplying all the argument values, leaving a main function that returns a value of type `'r`. Let's look at some examples of specs, and their types:

```
# Command.Spec.empty ;;
 - : ('m, 'm) Command.Spec.t = <abstr>
# Command.Spec.(empty +> anon ("foo" %: int)) ;;
 - : (int -> '_a, '_a) Command.Spec.t = <abstr>
```
OCaml Utop ∗ command-line-parsing/command_types.topscript ∗ all code

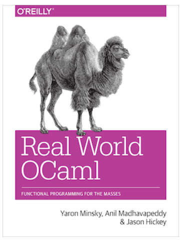
The empty specification is simple, as it doesn't add any parameters to the callback type. The second example adds an `int` anonymous parameter that is reflected in the inferred type. One forms a command by combining a spec of type `('main, unit) Spec.t` with a function of type `'main`. The combinators we've shown so far incrementally build the type of `'main` according to the command-line parameters it expects, so the resulting type of `'main` is something like `arg1 -> ... -> argN -> unit`.

The type of `Command.basic` should make more sense now:

```
# Command.basic ;;
 - : summary:string ->
     ?readme:(unit -> string) ->
     ('main, unit -> unit) Command.Spec.t -> 'main -> Command.t
 = <fun>
```
OCaml Utop ∗ command-line-parsing/basic.topscript ∗ all code

The parameters to `Spec.t` are important here. They show that the callback function for a spec should consume identical arguments to the supplied `main` function, except for an additional `unit` argument. This final `unit` is there to make sure the callback is evaluated as a function, since if zero command-line arguments are specified (i.e., `Spec.empty`), the callback would otherwise have no arguments and be evaluated immediately. That's why you have to supply an additional `()` to the callback function in all the previous examples.

Composing Specification Fragments Together



Login with GitHub to view and add comments

If you want to factor out common command-line operations, the ++ operator will append two specifications together. Let's add some dummy verbosity and debug flags to our calendar application to illustrate this.

```
open Core.Std

let add ~common =
  Command.basic ~summary:"Add [days] to the [base] date"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
      ++ common
    )
    (fun base span debug verbose () ->
       Date.add_days base span
       |> Date.to_string
       |> print_endline
    )

let diff ~common =
  Command.basic ~summary:"Show days between [date2] and [date1]"
    Command.Spec.(
      empty
      +> anon ("date1" %: date)
      +> anon ("date2" %: date)
      ++ common
    )
    (fun date1 date2 debug verbose () ->
       Date.diff date1 date2
       |> printf "%d days\n"
    )
```

OCaml ∗ command-line-parsing/cal_append.ml ∗ all code

The definitions of the specifications are very similar to the earlier example, except that they append a `common` parameter after each specification. We can supply these flags when defining the groups.

```
let () =
  let common =
    Command.Spec.(
      empty
      +> flag "-d" (optional_with_default false bool) ~doc:" Debug mode"
      +> flag "-v" (optional_with_default false bool) ~doc:" Verbose output"
    )
  in
  List.map ~f:(fun (name, cmd) -> (name, cmd ~common))
    [ "add", add; "diff", diff ]
  |> Command.group ~summary:"Manipulate dates"
  |> Command.run
```

OCaml ∗ command-line-parsing/cal_append.ml , continued (part 1) ∗ all code

Both of these flags will now be applied and passed to all the callback functions. This makes code refactoring a breeze by using the compiler to spot places where you use commands. Just add a parameter to the common definition, run the compiler, and fix type errors until everything works again.

For example, if we remove the `verbose` flag and recompile, we'll get this impressively long type error:

```
$ corebuild cal_append_broken.native
 File "cal_append_broken.ml", line 38, characters 45-52:
 Error: This expression has type
          (bool -> unit -> unit -> unit, unit -> unit -> unit) Command.Spec.t
        but an expression was expected of type
          (bool -> unit -> unit -> unit, unit -> unit) Command.Spec.t
        Type unit -> unit is not compatible with type unit
 Command exited with code 2.
```

Terminal ∗ command-line-parsing/build_cal_append_broken.out ∗ all code

While this does look scary, the key line to scan is the last one, where it's telling you that you have supplied too many arguments in the callback function (`unit -> unit` versus `unit`). If you started with a working program and made this single change, you typically don't even need to read the type error, as the filename and location information is sufficient to make the obvious fix.

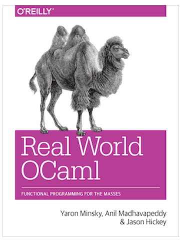

Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

Prompting for Interactive Input

The `step` combinator lets you control the normal course of parsing by supplying a function that maps callback arguments to a new set of values. For instance, let's revisit our first calendar application that added a number of days onto a supplied base date:

```
open Core.Std

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
    (fun base span () ->
       Date.add_days base span
       |> Date.to_string
       |> print_endline
    )

let () = Command.run add
```

OCaml ∗ command-line-parsing/cal_add_days.ml ∗ all code

This program requires you to specify both the `base` date and the number of `days` to add onto it. If `days` isn't supplied on the command line, an error is output. Now let's modify it to interactively prompt for a number of days if only the `base` date is supplied:

```
open Core.Std

let add_days base span () =
  Date.add_days base span
  |> Date.to_string
  |> print_endline

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      step
        (fun m base days ->
           match days with
           | Some days ->
             m base days
           | None ->
             print_endline "enter days: ";
             read_int ()
             |> m base
        )
      +> anon ("base" %: date)
      +> anon (maybe ("days" %: int))
    )
    add_days

let () = Command.run add
```

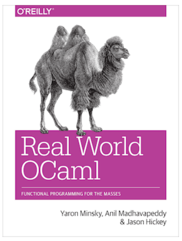
OCaml ∗ command-line-parsing/cal_add_interactive.ml ∗ all code

The `days` anonymous argument is now an optional integer in the spec, and we want to transform it into a nonoptional value before calling our `add_days` callback. The `step` combinator lets us perform this transformation by applying its supplied callback function first. In the example, the callback first checks if `days` is defined. If it's undefined, then it interactively reads an integer from the standard input.

The first `m` argument to the `step` callback is the next callback function in the chain. The transformation is completed by calling `m base days` to continue processing with the new values we've just calculated. The `days` value that is passed onto the next callback now has a nonoptional `int` type:

```
$ ocamlbuild -use-ocamlfind -tag thread -pkg core cal_add_interactive.native
$ ./cal_add_interactive.native 2013-12-01
 enter days:
 35
 2014-01-05
```

Terminal ∗ command-line-parsing/build_and_run_cal_add_interactive.out ∗ all code


Buy in print and eBook.

Login with GitHub to view and add comments

The transformation means that the `add_days` callback can just keep its original definition of `Date.t -> int -> unit`. The `step` function transformed the `int option` argument from the parsing into an `int` suitable for `add_days`. This transformation is explicitly represented in the type of the `step` return value:

```
# open Command.Spec ;;

# step (fun m (base:Date.t) days ->
  match days with
  | Some days -> m base days
  | None ->
     print_endline "enter days: ";
     m base (read_int ())) ;;
 - : (Date.t -> int -> '_a, Date.t -> int option -> '_a) t = <abstr>
```
OCaml Utop ∗ command-line-parsing/step.topscript ∗ all code

The first half of the `Spec.t` shows that the callback type is `Date.t -> int`, whereas the resulting value expected from the next specification in the chain is a `Date.t -> int option`.

Adding Labeled Arguments to Callbacks

The `step` chaining lets you control the types of your callbacks very easily. This can help you match existing interfaces or make things more explicit by adding labeled arguments:

```
open Core.Std

let add_days ~base_date ~num_days () =
  Date.add_days base_date num_days
  |> Date.to_string
  |> print_endline

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      step (fun m base days -> m ~base_date:base ~num_days:days)
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
    add_days

let () = Command.run add
```
OCaml ∗ command-line-parsing/cal_add_labels.ml ∗ all code

This `cal_add_labels` example goes back to our noninteractive calendar addition program, but the `add_days` main function now expects labeled arguments. The `step` function in the specification simply converts the default `base` and `days` arguments into a labeled function.

Labeled arguments are more verbose but can also help prevent errors with command-line arguments with similar types but different names and purposes. It's good form to use labels when you have a lot of otherwise anonymous `int` and `string` arguments.

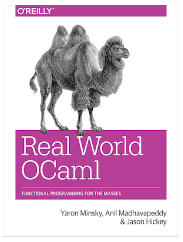
COMMAND-LINE AUTOCOMPLETION WITH BASH

Modern UNIX shells usually have a tab-completion feature to interactively help you figure out how to build a command line. These work by pressing the Tab key in the middle of typing a command, and seeing the options that pop up. You've probably used this most often to find the files in the current directory, but it can actually be extended for other parts of the command, too.

The precise mechanism for autocompletion varies depending on what shell you are using, but we'll assume you are using the most common one: **bash**. This is the default interactive shell on most Linux distributions and Mac OS X, but you may need to switch to it on *BSD or Windows (when using Cygwin). The rest of this section assumes that you're using **bash**.

Bash autocompletion isn't always installed by default, so check your OS package manager to see if you have it available.

Operating system	Package manager	Package
Debian Linux	`apt`	`bash-completion`
Mac OS X	Homebrew	`bash-completion`
FreeBSD	Ports system	`/usr/ports/shells/bash-completion`



Buy in print and eBook.

Login with GitHub to view and add comments

Once *bash* completion is installed and configured, check that it works by typing the `ssh` command and pressing the Tab key. This should show you the list of known hosts from your *~/.ssh/known_hosts* file. If it lists some hosts that you've recently connected to, you can continue on. If it lists the files in your current directory instead, then check your OS documentation to configure completion correctly.

One last bit of information you'll need to find is the location of the `bash_completion.d` directory. This is where all the shell fragments that contain the completion logic are held. On Linux, this is often in `/etc/bash_completion.d`, and in Homebrew on Mac OS X, it would be `/usr/local/etc/bash_completion.d` by default.

Generating Completion Fragments from Command

The Command library has a declarative description of all the possible valid options, and it can use this information to generate a shell script that provides completion support for that command. To generate the fragment, just run the command with the `COMMAND_OUTPUT_INSTALLATION_BASH` environment variable set to any value.

For example, let's try it on our MD5 example from earlier, assuming that the binary is called **basic_md5_with_flags** in the current directory:

```
$ env COMMAND_OUTPUT_INSTALLATION_BASH=1 ./basic_md5_with_flags.native
 function _jsautocom_76186 {
   export COMP_CWORD
   COMP_WORDS[0]=./basic_md5_with_flags.native
   COMPREPLY=($("${COMP_WORDS[@]}"))
 }
 complete -F _jsautocom_76186 ./basic_md5_with_flags.native
```

Terminal * command-line-parsing/md5_completion.out * all code

Recall that we used the `Arg_type.file` to specify the argument type. This also supplies the completion logic so that you can just press Tab to complete files in your current directory.

Installing the Completion Fragment

You don't need to worry about what the preceding output script actually does (unless you have an unhealthy fascination with shell scripting internals, that is). Instead, redirect the output to a file in your current directory and source it into your current shell:

```
$ env COMMAND_OUTPUT_INSTALLATION_BASH=1 ./cal_add_sub_days.native > cal.cmd
$ . cal.cmd
$ ./cal_add_sub_days.native <tab>
 add       diff      help      version
```

Terminal * command-line-parsing/cal_completion.out * all code

Command completion support works for flags and grouped commands and is very useful when building larger command-line interfaces. Don't forget to install the shell fragment into your global `bash_completion.d` directory if you want it to be loaded in all of your login shells.

Installing a Generic Completion Handler

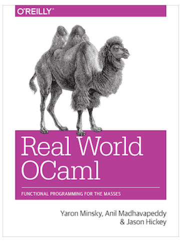
Sadly, **bash** doesn't support installing a generic handler for all Command-based applications. This means you have to install the completion script for every application, but you should be able to automate this in the build and packaging system for your application.

It will help to check out how other applications install tab-completion scripts and follow their lead, as the details are very OS-specific.

ALTERNATIVE COMMAND-LINE PARSERS

This rounds up our tour of the Command library. This isn't the only way to parse command-line arguments of course; there are several alternatives available on OPAM. Three of the most prominent ones follow:

The `Arg` module
The `Arg` module is from the OCaml standard library, which is used by the compiler itself to handle its command-line interface. Command is generally more featureful than Arg (mainly via



Buy in print and eBook.

Table of Contents

Login with GitHub to view and add comments

support for subcommands, the `step` combinator to transform inputs, and help generation), but there's absolutely nothing wrong with using Arg either.

You can use the `Command.Spec.flags_of_args_exn` function to convert Arg specifications into ones compatible with Command. This is quite often used to help port older non-Core code into the Core standard library world.

ocaml-getopt
`ocaml-getopt` provides the general command-line syntax of GNU `getopt` and `getopt_long`. The GNU conventions are widely used in the open source world, and this library lets your OCaml programs obey the same rules.

Cmdliner
Cmdliner is a mix between the Command and Getopt libraries. It allows for the declarative definition of command-line interfaces but exposes a more `getopt`-like interface. It also automates the generation of UNIX man pages as part of the specification. Cmdliner is the parser used by OPAM to manage its command line.

< Previous   Next >