16 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 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
- Build simple equivalents to the cryptographic `md5` and `shasum` utilities
- Demonstrate how to declare complex command-line interfaces in a type-safe and elegant way

16.1 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 macOS 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
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
let do_hash file =
  Md5.digest_file_blocking file |> Md5.to_hex |> print_endline
```

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 create a parser for the command line arguments. The module `Command.Param` provides a set of combinators that can be combined together to define a parameter parser for optional flags and positional arguments, including documentation, the types they should map to, and whether to take special actions such as pausing for interactive input if certain inputs are encountered.

16.1.1 Defining an Anonymous Argument

Let's build a parser for a command line UI with a single *anonymous* argument, i.e., an argument that is passed in without a flag.

```
let filename_param =
  let open Command.Param in
  anon ("filename" %: string)
```

Here, `anon` is used to signal the parsing of an anonymous argument, and the expression `("filename" %: string)` indicates the textual name of the argument and specification that describes the kind of value that is expected. The textual name is used for generating help text, and the specification, which has type `Command.Arg_type.t`, is used both to nail down the OCaml type of the returned value (`string`, in this case) and to guide features like input validation. The values `anon`, `string` and `%:` all come from the `Command.Param` module.

16.1.2 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 with `Command.basic`.

```
let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    (Command.Param.map filename_param ~f:(fun filename () ->
         do_hash filename))
```

The `summary` argument is a one-line description which goes at the top of the help screen, while the (optional) `readme` argument is for providing a more detailed description that will be provided on demand.

The final argument is the most interesting one, which is the parameter parser. This will be easier to understand if we first learn a bit more about the type signatures of the various components we've been using. Let's do that by recreating some of this code in the toplevel.

```
# let filename_param = Command.Param.(anon ("filename" %: string));;
val filename_param : string Command.Spec.param = <abstr>
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

The type parameter of `filename_param` is there to indicate the type of the value returned by the parser; in this case, `string`.

But `Command.basic` requires a parameter parser that returns a value of type `unit -> unit`. We can see that by using `#show` to explore the types.

```
# #show Command.basic;;
val basic : unit Command.basic_command
# #show Command.basic_command;;
type nonrec 'result basic_command =
    summary:string ->
    ?readme:(unit -> string) ->
    (unit -> 'result) Command.Spec.param -> Command.t
```

Note that the `'result` parameter of the type alias `basic_command` is instantiated as `unit` for the type of `Command.basic`.

It makes sense that `Command.basic` wants a parser that returns a function; after all, in the end, it needs a function it can run that constitutes the execution of the program. But how do we get such a parser, given the parser we have returns just a filename?

The answer is to use a `map` function to change the value returned by the parser. As you can see below, the type of `Command.Param.map` is very similar to the type of `List.map`.

```
# #show Command.Param.map;;
val map : 'a Command.Spec.param -> f:('a -> 'b) -> 'b Command.Spec.param
```

In our program, we used `map` to convert the `filename_param` parser, which returns a string representing the file name, into a parser that returns a function of type `unit -> unit` containing the body of the command. It might not be obvious that the function passed to map returns a function, but remember that, due to currying, the invocation of map above could be written equivalently as follows.

```
Command.Param.map filename_param ~f:(fun filename ->
  fun () -> do_hash filename)
```

16.1.3 Running 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
```

`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 the following `dune` file:

```
(executable
  (name       md5)
  (libraries  core)
  (preprocess (pps ppx_jane)))
```

At which point we can build and execute the program using `dune exec`. Let's use this to query version information from the binary.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
$ dune exec -- ./md5.exe -version
1.0
$ dune exec -- ./md5.exe -build-info
RWO
```

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. Dune provides a `dune-build-info` library[1] that automates this process for most common workflows.

We can invoke our binary with `-help` to see the auto-generated help.

```
$ dune exec -- ./md5.exe -help
Generate an MD5 hash of the input data

  md5.exe 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: -?)
```

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

```
$ dune exec -- ./md5.exe md5.ml
2ae55d17ff11d337492a1ca5510ee01b
```

And that's all it takes 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

let do_hash file =
  Md5.digest_file_blocking file |> Md5.to_hex |> print_endline

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

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

[1] https://dune.readthedocs.io/en/stable/executables.html#embedding-build-information-into-executables

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

16.1.4 Multi-Argument Commands

All the examples thus far have involved a single argument, but we can of course create multi-argument commands as well. We can make a parser for multiple arguments by binding together simpler parsers, using the function `Command.Param.both`. Here is its type.

```
# #show Command.Param.both;;
val both :
  'a Command.Spec.param ->
  'b Command.Spec.param -> ('a * 'b) Command.Spec.param
```

`both` allows us to take two parameter parsers and combine them into a single parser that returns the two arguments as a pair. In the following, we rewrite our `md5` program so it takes two anonymous arguments: the first is an integer saying how many characters of the hash to print out, and the second is the filename.

```
open Core

let do_hash hash_length filename =
  Md5.digest_file_blocking filename
  |> Md5.to_hex
  |> (fun s -> String.prefix s hash_length)
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Param.(
      map
        (both
           (anon ("hash_length" %: int))
           (anon ("filename" %: string)))
        ~f:(fun (hash_length, filename) () ->
          do_hash hash_length filename))

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

Building and running this command, we can see that it now indeed expects two arguments.

```
$ dune exec -- ./md5.exe 5 md5.ml
f8824
```

This works well enough for two parameters, but if you want longer parameter lists, this approach gets old fast. A better way is to use let-syntax, which was discussed in Chapter 8.1.3 (`bind` and Other Error Handling Idioms).

```
let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    (let open Command.Let_syntax in
    let open Command.Param in
    let%map hash_length = anon ("hash_length" %: int)
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
    and filename = anon ("filename" %: string) in
    fun () -> do_hash hash_length filename)
```

Here, we take advantage of let-syntax's support for parallel let bindings, using `and` to join the definitions together. This syntax translates down to the same pattern based on `both` that we showed above, but it's easier to read and use, and scales better to more arguments.

The need to open both modules is a little awkward, and the `Param` module in particular you really only need on the right-hand-side of the equals-sign. This is achieved automatically by using the `let%map_open` syntax, demonstrated below. We'll also drop the open of `Command.Let_syntax` in favor of explicitly using `let%map_open.Command` to mark the let-syntax as coming from the `Command` module

```
let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    (let%map_open.Command hash_length = anon ("hash_length" %: int)
     and filename = anon ("filename" %: string) in
     fun () -> do_hash hash_length filename)
```

Let-syntax is the most common way of writing parsers for `Command`, and we'll use that idiom from here on.

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

16.2 Argument Types

You aren't just limited to parsing command lines of strings and ints. There are some other argument types defined in `Command.Param`, like `date` and `percent`. But most of the time, argument types for specific types in `Core` and other associated libraries are defined in the module that defines the type in question.

As an example, we can tighten up the specification of the command to `Filename.arg_type` 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")
    (let%map_open.Command file =
       anon ("filename" %: Filename.arg_type)
     in
     fun () -> do_hash file)
```

This doesn't change the validation of the provided value, but it does enable interactive command-line completion. We'll explain how to enable that later in the chapter.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

16.2.1 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

let do_hash file =
  Md5.digest_file_blocking file |> Md5.to_hex |> print_endline

let regular_file =
  Command.Arg_type.create (fun filename ->
      match Sys.is_file filename with
      | `Yes -> filename
      | `No -> failwith "Not a regular file"
      | `Unknown ->
        failwith "Could not determine if this was a regular file")

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

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

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`:

```
$ dune exec -- ./md5.exe md5.ml
5df5ec6301ea37bebc22912ceaa6b2e2
$ dune exec -- ./md5.exe /dev/null
Error parsing command line:

  failed to parse FILENAME value "/dev/null"
  (Failure "Not a regular file")

For usage information, run

  md5.exe -help

[1]
```

16.2.2 Optional and Default Arguments

A more realistic `md5` binary could also read from the standard input if a `filename` isn't specified. To do this, we need to declare the filename argument as optional, which we can do with the `maybe` operator.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

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

But building this results in a compile-time error.

```
$ dune build md5.exe
File "md5.ml", line 15, characters 23-31:
15 |       fun () -> do_hash filename)
                             ^^^^^^^^
Error: This expression has type string option
       but an expression was expected of type string
[1]
```

This is because changing the argument type has also changed the type of the value that is returned by the parser. It now produces a `string option` instead of a `string`, reflecting the optionality of the argument. We can adapt our example to use the new information and read from standard input if no file is specified.

```
open Core

let get_contents = function
  | None | Some "-" -> In_channel.input_all In_channel.stdin
  | Some filename -> In_channel.read_all filename

let do_hash filename =
  get_contents filename
  |> Md5.digest_string
  |> Md5.to_hex
  |> print_endline

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

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

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

```
$ cat md5.ml | dune exec -- ./md5.exe
54fd98cd30f8faa76be46be0005f00bf
```

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.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

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

```
open Core

let get_contents = function
  | "-" -> In_channel.input_all In_channel.stdin
  | filename -> In_channel.read_all filename

let do_hash filename =
  get_contents filename
  |> Md5.digest_string
  |> Md5.to_hex
  |> print_endline

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

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

Building and running this confirms that it has the same behavior as before.

```
$ cat md5.ml | dune exec -- ./md5.exe
f0ea4085ca226eef2c0d70026619a244
```

16.2.3 Sequences of Arguments

Another common way of parsing anonymous arguments is as a variable length list. As an example, let's modify our MD5 code to take a collection of files to process on the command line.

```
open Core

let get_contents = function
  | "-" -> In_channel.input_all In_channel.stdin
  | filename -> In_channel.read_all filename

let do_hash filename =
  get_contents filename
  |> Md5.digest_string
  |> Md5.to_hex
  |> 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")
    (let%map_open.Command files =
       anon (sequence ("filename" %: Filename.arg_type))
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
      in
      fun () ->
        match files with
        | [] -> do_hash "-"
        | _ -> List.iter files ~f:do_hash)

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

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.

```
$ dune exec -- ./md5.exe /etc/services ./_build/default/md5.exe
MD5 (/etc/services) = 6501e9c7bf20b1dc56f015e341f79833
MD5 (./_build/default/md5.exe) = 6602408aa98478ba5617494f7460d3d9
```

16.3 Adding Labeled Flags

You aren't 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 macOS 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

let checksum_from_string buf =
  Md5.digest_string buf |> Md5.to_hex |> print_endline

let checksum_from_file filename =
  let contents =
    match filename with
    | "-" -> In_channel.input_all In_channel.stdin
    | filename -> In_channel.read_all filename
  in
  Md5.digest_string contents |> Md5.to_hex |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    (let%map_open.Command use_string =
       flag
         "-s"
         (optional string)
         ~doc:"string Checksum the given string"
     and trial = flag "-t" no_arg ~doc:" run a built-in time trial"
     and filename =
       anon (maybe_with_default "-" ("filename" %: Filename.arg_type))
     in
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
      fun () ->
        if trial
        then printf "Running time trial\n"
        else (
          match use_string with
          | Some buf -> checksum_from_string buf
          | None -> checksum_from_file filename))

let () = Command.run command
```

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:

```
$ dune exec -- ./md5.exe -help
Generate an MD5 hash of the input data

  md5.exe [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: -?)

$ dune exec -- ./md5.exe -s "ocaml rocks"
5a118fe92ac3b6c7854c595ecf6419cb
```

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 earlier examples. There are a number of other functions that you can wrap flags in to control how they are parsed:

- `required` `<arg>` will return `<arg>` and error if not present
- `optional` `<arg>` with return `<arg>` `option`
- `optional_with_default` `<val>` `<arg>` will return `<arg>` with default `<val>` if not present
- `listed` `<arg>` will return `<arg>` `list` (this flag may appear multiple times)
- `no_arg` will return a `bool` that is true if the 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.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

16.4 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 env
$ opam remote list -k git
$ opam install --help
$ opam install core --verbose
```

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 use `Command.group`, which lets you merge a collection of `Command.t`'s into one.

```
# Command.group;;
- : summary:string ->
    ?readme:(unit -> string) ->
    ?preserve_subcommand_order:unit ->
    ?body:(path:string list -> unit) ->
    (string * Command.t) list -> Command.t
= <fun>
```

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

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

let () = Command.run add
```

Everything in this command should be familiar to you by now, and it works as you might expect.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
$ dune exec -- ./cal.exe -help
Add [days] to the [base] date and print day

  cal.exe BASE DAYS

=== 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: -?)

$ dune exec -- ./cal.exe 2012-12-25 40
2013-02-03
```

Now, let's also add the ability to take the difference between two dates, but, instead of creating a new binary, we'll group both operations as subcommands using `Command.group`.

```
open Core

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date"
    Command.Let_syntax.(
      let%map_open base = anon ("base" %: date)
      and days = anon ("days" %: int) in
      fun () ->
        Date.add_days base days |> Date.to_string |> print_endline)

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

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

let () = Command.run command
```

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.

```
(executable
  (name       cal)
  (libraries  core)
  (preprocess (pps ppx_jane)))
```

```
$ dune exec -- ./cal.exe -help
Manipulate dates
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
  cal.exe 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)
```

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

```
$ dune exec -- ./cal.exe add 2012-12-25 40
2013-02-03
$ dune exec -- ./cal.exe diff 2012-12-25 2012-11-01
54 days
```

16.5 Prompting for Interactive Input

Sometimes, if a value isn't provided on the command line, you want to prompt for it instead. Let's return to the calendar tool we built before.

```
open Core

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

let () = Command.run add
```

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

let add_days base days =
  Date.add_days base days |> Date.to_string |> print_endline

let prompt_for_string name of_string =
  printf "enter %s: %!" name;
  match In_channel.input_line In_channel.stdin with
  | None -> failwith "no value entered. aborting."
  | Some line -> of_string line

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    (let%map_open.Command base = anon ("base" %: date)
     and days = anon (maybe ("days" %: int)) in
```

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
      let days =
        match days with
        | Some x -> x
        | None -> prompt_for_string "days" Int.of_string
      in
      fun () -> add_days base days)

let () = Command.run add
```

The `days` anonymous argument is now an optional integer in the spec, and when it isn't there, we simply prompt for the value as part of the ordinary execution of our program.

Sometimes, it's convenient to pack the prompting behavior into the parser itself. For one thing, this would allow you to easily share the prompting behavior among multiple commands. This is easy enough to do by adding a new function, `anon_prompt`, which creates a parser that automatically prompts if the value isn't provided.

```
let anon_prompt name of_string =
  let arg = Command.Arg_type.create of_string in
  let%map_open.Command value = anon (maybe (name %: arg)) in
  match value with
  | Some v -> v
  | None -> prompt_for_string name of_string

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    (let%map_open.Command base = anon ("base" %: date)
     and days = anon_prompt "days" Int.of_string in
     fun () -> add_days base days)
```

We can see the prompting behavior if we run the program without providing the second argument.

```
$ echo 35 | dune exec -- ./cal.exe 2013-12-01
enter days: 2014-01-05
```

16.6 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 macOS, 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`.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

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

- On Debian Linux, do `apt install bash-completion`
- On macOS Homebrew, do `brew install bash-completion`
- On FreeBSD, do `pkg install bash-completion`.

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 macOS, it would be /usr/local/etc/bash_completion.d by default.

16.6.1 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 `md5` in the current directory:

```
$ env COMMAND_OUTPUT_INSTALLATION_BASH=1 dune exec -- ./md5.exe
function _jsautocom_32087 {
  export COMP_CWORD
  COMP_WORDS[0]=./md5.exe
  if type readarray > /dev/null
  then readarray -t COMPREPLY < <("${COMP_WORDS[@]}")
  else IFS="
" read -d "" -A COMPREPLY < <("${COMP_WORDS[@]}")
  fi
}
complete -F _jsautocom_32087 ./md5.exe
```

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.

16.6.2 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:

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press

```
$ 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
```

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.

16.7 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 built on top of `Arg`, but you can also use `Arg` directly. You can use the `Command.Spec.flags_of_args_exn` function to convert `Arg` specifications into ones compatible with Command, which is a simple way of porting an `Arg`-based command line interface to `Command`.

ocaml-getopt[2] `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[3] 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.

https://doi.org/10.1017/9781009129220.019 Published online by Cambridge University Press