Command Line Application with Elixir

Pierre Sugar
pierre@sugaryourcoffee.de

January 2, 2015

Abstract

The programming world gets parallel because of processors with multiple cores. Most of the programming languages aren't build for multiprocessing and hence cannot use the performance that comes with the multi core processors. Erlang is one of the most sophisticated language build especially for multiprocessing. But Erlang has some get to used rules and is not so much fun to program with. This is where Elixir comes into play. Elixir combines the advantages of Erlang with the elegance of Ruby.

This tutorial is introducing into Elixir by developing a command line interface based on a suggestion from Dave Thomas' Elixir book.

1 Installation

`Elixir` is run on the `Erlang` virtual machine. That is we need `Elixir` and `Erlang` installed. The most up to date installation instructions for `Elixir` can be found at `http://elixir-lang.org/install.html` and for `Erlang` at `https://www.erlang-solutions.com/downloads/download-erlang-otp`.

To install `Erlang` we issue following commands from the command line

```
\$ wget http://package.erlang-solutions.com/erlang-solutions_1.0_all.deb
\$ sudo dpkg -i erlang-solutions_1.0_all.deb
\$ wget http://package.erlang-solutions.com/ubuntu/erlang_solutions.asc
\$ sudo apt-key add erlang_solutions erlang_solutions.asc
\$ sudo apt-get update
\$ sudo apt-get install erlang
```

Next we install `Elixir`

```
\$ wget http://packages.erlang-solutions._1.0_all.deb
\$ sudo dpkg -i erlang-solutions_1.0_all.deb
\$ sudo apt-get update
\$ sudo apt-get install elixir
```

Now we are ready to go.

2 Define the Project

The `National Climatic Data Center` or short `NOAA` is providing web services to retrieve climatic data from cities of the American continent. We want to retrieve this data with our application and display it on the console.

2.1 Project outline

The program runs from the command line. We will provide a location and the program will print the wheather data.

- Provide location $ noaa ¡location¿
- Parse the input
- Fetch the data from NOAA
- Extract the wheather data
- Print the results

But before we jump into the application development we need to understand the web service from `NOAA` to know what we actually have to provide as parameters.

2.2 Understanding the NOAA Web Service

To use the web service from `NOAA` we need to request a token. With each web service request we have to provide that token.

To request a token go to `https://www.ncdc.noaa.gov/cdo-web/token` and enter and submit your e-mail address. You will receive a token send to the provided e-mail address. This token you allways have to provide when sending a web service request to `NOAA`.

Let's check out the web service with `curl`. Information about how to use the web service can be found at `https://www.ncdc.noaa.gov/cdo-web/webservices/v2`.

To know wich locations are available to retrieve wheather data for we can retrieve the data with the `/locations` url.

```
\$ curl -H "token:pLCcTVobSdphuEvXyOyhkAbVlObmWQra" \
       http://www.ncdc.noaa.gov/cdo-web/api/v2/locations
{"results":[
 {"id":"CITY:AE000002",
  "name":"Ajman, AE",
  "datacoverage":0.6855,
  "mindate":"1944-03-01",
  "maxdate":"2014-12-29"},
 {"id":"CITY:AE000003",
  "name":"Dubai, AE",
  "datacoverage":0.6855,
  "mindate":"1944-03-01",
  "maxdate":"2014-12-29"},
  ...
 {"id":"CITY:AR000010",
  "name":"Mendoza, AR",
  "datacoverage":0.997,
  "mindate":"1959-10-01",
  "maxdate":"2014-12-29"},
```

```
 {"id":"CITY:AR000011",
  "name":"Neuquen, AR",
  "datacoverage":0.9814,
  "mindate":"1956-08-01",
  "maxdate":"2014-12-29"}],
 "metadata":{"resultset":{"limit":25,"count":38497,"offset":1}}}
```


As we can see the data is provided in a JSON format with `"results"` and `"metadata"` fields. But it seems there are loads of locations. That is our application also has to provided a way to list all available location data.

The data is provided in different datasets. Dependent the information we want to retrieve we have to know the datasets `datatypeid`. We can obtain the information with the `/datasets` url.


```
\$ curl -H "token:pLCcTVobSdphuEvXyOyhkAbVlObmWQra" \
        http://www.ncdc.noaa.gov/cdo-web/api/v2/datasets
{"results":[
  {"uid":"gov.noaa.ncdc:C00040","id":"ANNUAL","name":"Annual Summaries",
   "datacoverage":1,"mindate":"1831-02-01","maxdate":"2014-07-01"},
  {"uid":"gov.noaa.ncdc:C00861","id":"GHCND","name":"Daily Summaries",
   "datacoverage":1,"mindate":"1763-01-01","maxdate":"2014-12-31"},
  {"uid":"gov.noaa.ncdc:C00841","id":"GHCNDMS","name":"Monthly Summaries",
   "datacoverage":1,"mindate":"1763-01-01","maxdate":"2014-11-01"},
  {"uid":"gov.noaa.ncdc:C00345","id":"NEXRAD2",
   "name":"Weather Radar (Level II)",
   "datacoverage":0.95,"mindate":"1991-06-05","maxdate":"2014-12-31"},
  {"uid":"gov.noaa.ncdc:C00708","id":"NEXRAD3",
   "name":"Weather Radar (Level III)",
   "datacoverage":0.95,"mindate":"1994-05-20","maxdate":"2014-12-28"},
  {"uid":"gov.noaa.ncdc:C00821","id":"NORMAL_ANN",
   "name":"Normals Annual/Seasonal",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-01-01"},
  {"uid":"gov.noaa.ncdc:C00823","id":"NORMAL_DLY","name":"Normals Daily",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-12-31"},
  {"uid":"gov.noaa.ncdc:C00824","id":"NORMAL_HLY","name":"Normals Hourly",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-12-31"},
  {"uid":"gov.noaa.ncdc:C00822","id":"NORMAL_MLY","name":"Normals Monthly",
   "datacoverage":1,"mindate":"2010-01-01","maxdate":"2010-12-01"},
  {"uid":"gov.noaa.ncdc:C00505","id":"PRECIP_15",
   "name":"Precipitation 15 Minute",
   "datacoverage":0.25,"mindate":"1970-05-12","maxdate":"2013-07-01"},
  {"uid":"gov.noaa.ncdc:C00313","id":"PRECIP_HLY",
   "name":"Precipitation Hourly",
   "datacoverage":1,"mindate":"1900-01-01","maxdate":"2013-10-01"}],
 "metadata":{"resultset":{"limit":25,"count":11,"offset":1}}}
```


This information we have to additionally provide to the location information for that we want to retrieve data from. Hence we also have to provide a way to retrieve possible datasets with our application.

From the `location` information we can obtain the min and max date weather data is available for each country. From the `dataset` information we can obtain

which datasets are available. Based on that information we call the `data` url to retrieve the weather data.

And finally to retrieve actual weather data we invoke the `/data` url with the `datasetid` GHCND and the `locationid` CITY:000019 for Munich.

```
\$ curl -H "token:pLCcTVobSdphuEvXyOyhkAbVlObmWQra" \
        http://www.ncdc.noaa.gov/cdo-web/api/v2/data?datasetid=GHCND&\
        locationid=CITY:GM000019&startdate=2014-10-01&enddate=2014-10-31
{"results":[
  {"station":"GHCND:GM000004199","value":14,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":193,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":120,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":3,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":191,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GME00111524","value":104,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-01T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":195,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":105,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":184,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GME00111524","value":73,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-02T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"PRCP","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
  "datatype":"SNWD","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GM000004199","value":182,"attributes":",,E,",
  "datatype":"TMAX","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GM000004199","value":89,"attributes":",,E,",
  "datatype":"TMIN","date":"2014-10-03T00:00:00"},
  {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
```

```
 "datatype":"PRCP","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":0,"attributes":",,E,",
 "datatype":"SNWD","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":184,"attributes":",,E,",
 "datatype":"TMAX","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GME00111524","value":67,"attributes":",,E,",
 "datatype":"TMIN","date":"2014-10-03T00:00:00"},
 {"station":"GHCND:GM000004199","value":0,"attributes":",,E,",
 "datatype":"PRCP","date":"2014-10-04T00:00:00"}],
"metadata":{"resultset":{"limit":25,"count":248,"offset":1}}}
```


Now we have all information we need to build our command line interface which will, from the current information state, provide these functions.

- List a specified count of available `datasets`
- List all or selected count of available `locations`
- Search for a specific city and print the `locationid`
- Get the wheather data for a specific location base on the `locationid`

The invocation variants are listed in the table 1 on page 5.

Table 1: Command line interface

Command	**Description**
noaa datasets -c 10	Lists 10 datasets
noaa datasets	Lists all available datasets
noaa locations -c 10	Lists 10 locations
noaa locations	List all available locations
noaa locations -s Munich	Searches of the Vancouver `locationid`
noaa data -l CITY:GM000019 -d GHCND -f 2014-10-01 -t 2014-10-31	Print weather data for Munich (-l) for the specified dataset (-d) between 1st (-f) and 31st (-t) of October

3 Create the Project

`Elixir` comes with the tool `Mix` to create and manage `Elixir` projects.

To create a project we issue the `Mix` command `new`. But first we change directory where we want to have created our project tree.

```
\$ cd ~/Learn/Elixir/noaa
\$ mix new noaa
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/noaa.ex
* creating test
```

```
* creating test/test_helper.exs
* creating test/noaa_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

  cd noaa
  mix test

Run 'mix help' for more commands.

\$
```

Now we have a fully implemented project tree and we can right away run our first test.

```
\$ cd noaa
\$ mix test
Compiled lib/noaa.ex
Generated noaa.app
.

Finished in 0.07 seconds (0.07s on load, 0.00s on tests)
1 tests, 0 failures

Randomized with seed 963329
\$
```

But before we actually start we want to set up version control for `noaa`.

```
\$ git init
\$ git add .
\$ git commit -am "Initial commit of noaa"
```

4 Implement the Command Line Interface

Let's try behaviour driven development (BDD) with `Elixir`. `Elixir` comes with a test environment called `EXUnit`. With BDD we start with tests first and then implement the code. So let's try that.

Listing 1: test/cli_test.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
  end

  test ":datasets returned with default count when only dataset is given" do
```

```
  end

  test ":datasets returned with count if dataset and count is given" do
  end

  test ":help returned if datasets w/o correct switches is given" do
  end

  test ":location returned with default count if only location is given" do
  end

  test ":location returned with count if location and count is given" do
  end

  test ":location returned with city location and city is given" do
  end

  test ":help returned if location w/o correct switches is given" do
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
  end

  test ":help returned if data w/o correct switches is given" do
  end

end
```

This is a bare test file with only a description we want to test. Let's run it with

```
\$ mix test
** (CompileError) test/cli_test.exs:4: module Noaa.CLI is not loaded and could
not be found
    (stdlib) lists.erl:1352: :lists.mapfoldl/3
    (stdlib) lists.erl:1353: :lists.mapfoldl/3
```

No surprise so far as we don't have an implementation of `Noaa.CLI` yet. So the next step is to get it compiled. To do this we have to implement `Noaa.CLI` next. As per convention the main source code of an application is saved into `lib/project_name`. Our proeject's name is `noaa`, hence our source code directory is named `lib/noaa`. Also a each module is saved to a separate file and each module is name spaced with the project name. So our first module for the command line interface (CLI) is called `noaa.CLI`. Table 2 on page 8 shows a summary of the conventions of `Elixir` projects.

Listing 2: lib/noaa/cli.ex

```
defmodule Noaa.CLI do

  @default_count 10
```

Table 2: Conventions for Elixir Projects

Command	Description
`lib/noaa/`	Directory for main source code
`noaa.CLI`	Each module is name spaced with the project name
	Each module lives in an own file

```
@moduledoc """
Handle the command line parsing and dispatching to the respective functions
that list the weather conditions of provided cities.
"""

def run(argv) do
  parse_args(argv)
end

@doc """
`argv` can be one of the following options.

* datasets  --count   COUNT
* locations --count   COUNT
* locations --search  CITY
* data      --dataset DATASET --location LOCATION --from DATE --to DATE

Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
`{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
"""
def parse_args(argv) do
  parse = OptionParser.parse(argv,
                             switches: [ help:      :boolean,
                                         datasets:  :boolean,
                                         locations: :boolean,
                                         data:      :boolean,
                                         count:     :integer,
                                         search:    :string,
                                         dataset:   :string,
                                         location:  :string,
                                         from:      :string,
                                         to:        :string ],
                             aliases:  [ h:         :help,
                                         c:         :count,
                                         s:         :search,
                                         d:         :dataset,
                                         l:         :location,
                                         f:         :from,
                                         t:         :to ])

end
```

end

With `cli.ex` in place we run the test again.

```
\$ mix test
Compiled lib/noaa/cli.ex
Generated noaa.app
test/cli_test.exs:4: warning: unused import Noaa.CLI
...........

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
11 tests, 0 failures

Randomized with seed 289389
```

This looks good our tests ran successfully without errors - but actually this is still boring. But now we get into the excitement. We now add our first actual test for `:help`.

Listing 3: test/cli_test1.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["--help", "anything"]) == :help
    assert parse_args(["-h",     "anything"]) == :help
  end

  test ":datasets returned with default count when only dataset is given" do
  end

  test ":datasets returned with count if dataset and count is given" do
  end

  test ":help returned if datasets w/o correct switches is given" do
  end

  test ":location returned with default count if only location is given" do
  end

  test ":location returned with count if location and count is given" do
  end

  test ":location returned with city location and city is given" do
  end

  test ":help returned if location w/o correct switches is given" do
  end
```

```
  test ":data returned with -s, -d, -b and -e if all required data is given" do
  end

  test ":help returned if data w/o correct switches is given" do
  end

end
```

Note that we didn't implement the function in `lib/noaa/cli.ex`. This is why our test at that moment fails when we run `mix test`.

Listing 4: mix test

```
.....

  1) test :help returned by option parsing with -h and --help options (CliTest)
     test/cli_test.exs:6
     Assertion with == failed
     code: parse_args(["--help", "anything"]) == :help
     lhs:  {[help: true], ["anything"], []}
     rhs:  :help
     stacktrace:
       test/cli_test.exs:7

.....

Finished in 0.1 seconds (0.1s on load, 0.01s on tests)
11 tests, 1 failures

Randomized with seed 305907
```

To make the test pass we will implement the parser section for `:help`.

Listing 5: lib/noaa/cli2.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    parse_args(argv)
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count   COUNT
  * locations --count   COUNT
```

```
  * locations --search  CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                               switches: [ help:      :boolean,
                                           datasets:  :boolean,
                                           locations: :boolean,
                                           data:      :boolean,
                                           count:     :integer,
                                           search:    :string,
                                           dataset:   :string,
                                           location:  :string,
                                           from:      :string,
                                           to:        :string ],
                               aliases:  [ h:         :help,
                                           c:         :count,
                                           s:         :search,
                                           d:         :dataset,
                                           l:         :location,
                                           f:         :from,
                                           t:         :to ])

    case parse do
      { [ help: true ], _, _ }
        -> :help
      { [ datasets: true, count: count ], _, _ }
        -> { :dataset, count }
      { [ datasets: true ], _, _ }
        -> { :dataset, @default_count }
      { [ locations: true, count: count ], _, _ }
        -> { :location, count }
      { [ locations: true ], _, _ }
        -> { :location, @default_count }
      { [ locations: true, location: location ], _, _ }
        -> { :location, location }
      { [ data: true,
          dataset: dataset,
          location: location,
          from: from,
          to: to ], _, _ }
        -> { :data, dataset, location, from, to }
      _ -> :help
    end
  end

end
```

When we run the test again we will see all tests pass.

Listing 6: mix test

```
...........

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
11 tests, 0 failures

Randomized with seed 962325
```

Now we implement the rest of the tests.

Listing 7: test/cli2_test

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["--help", "anything"]) == :help
    assert parse_args(["-h",     "anything"]) == :help
  end

  test ":datasets returned with default count when only dataset is given" do
  end

  test ":datasets returned with count if dataset and count is given" do
  end

  test ":help returned if datasets w/o correct switches is given" do
  end

  test ":location returned with default count if only location is given" do
  end

  test ":location returned with count if location and count is given" do
  end

  test ":location returned with city location and city is given" do
  end

  test ":help returned if location w/o correct switches is given" do
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
  end

  test ":help returned if data w/o correct switches is given" do
  end

end
```

When we run the tests they should all fail except the one for `:help`.

Listing 8: mix test

```
.

  1) test :datasets returned with default count when only dataset is given (Cli
     test/cli_test.exs:11
     ** (CaseClauseError) no case clause matching: {[datasets: true], [], []}
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:12

.

  2) test :data returned with -s, -d, -b and -e if all required data is given (
     test/cli_test.exs:35
     ** (CaseClauseError) no case clause matching: {[data: true, dataset: "ABCD
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:36



  3) test :datasets returned with count if dataset and count is given (CliTest)
     test/cli_test.exs:15
     ** (CaseClauseError) no case clause matching: {[datasets: true, count: 5],
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:16



  4) test :location returned with default count if only location is given (CliT
     test/cli_test.exs:20
     ** (CaseClauseError) no case clause matching: {[locations: true], [], []}
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:21



  5) test :location returned with city location and city is given (CliTest)
     test/cli_test.exs:29
     ** (CaseClauseError) no case clause matching: {[locations: true, location:
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:30
```

```
  6) test :location returned with count if location and count is given (CliTest
     test/cli_test.exs:24
     ** (CaseClauseError) no case clause matching: {[locations: true, count: 8]
     stacktrace:
       (noaa) lib/noaa/cli.ex:45: Noaa.CLI.parse_args/1
       test/cli_test.exs:25



Finished in 0.3 seconds (0.3s on load, 0.05s on tests)
8 tests, 6 failures

Randomized with seed 569703
```

To make them pass we implement the rest of the command line interface in `lib/cli.ex`

Listing 9: lib/cli3.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count    COUNT
  * locations --count    COUNT
  * locations --search   CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                               switches: [ help:      :boolean,
                                           datasets:  :boolean,
                                           locations: :boolean,
                                           data:      :boolean,
```

```
                                        count:      :integer,
                                        search:     :string,
                                        dataset:    :string,
                                        location:   :string,
                                        from:       :string,
                                        to:         :string ],
                                aliases:  [ h:          :help,
                                        c:          :count,
                                        s:          :search,
                                        d:          :dataset,
                                        l:          :location,
                                        f:          :from,
                                        t:          :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, dataset, location, from, to }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
  IO.puts """
  noaa fetches weather data from NOAA

  usage: noaa command args
```

```
    noaa --datasets  [ --count [ count | #{@default_count} ] ]
    noaa --locations [ --count [ count | #{@default_count} ] ] | --search city ]
    noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                     --to YYYY-MM-DD
    """
    System.halt(0)
  end

  def process({:datasets, count}) do
    "#{count} datasets"
  end

  def process({:locations, count}) when is_integer(count) do
    "#{count} locations"
  end

  def process({:locations, city}) do
    "#{city} location"
  end

  def process({:data, values}) do
    "Data for specified city"
  end
end
```

And we see them all pass. There is a specialty regarding the `data` command. This has more than one switch. Currently the switches have to be provided in a specific order. This is not user friendly. We should only make sure that all switches are provided. We write a test where we provide the switches in an arbitrary sequence.

Listing 10: test/cli_test3.exs

```
defmodule CliTest do
  use ExUnit.Case

  import Noaa.CLI, only: [ parse_args: 1 ]

  test ":help returned by option parsing with -h and --help options" do
    assert parse_args(["--help", "anything"]) == :help
    assert parse_args(["-h",     "anything"]) == :help
  end

  test ":datasets returned with default count when only dataset is given" do
    assert parse_args(["--datasets"]) == { :dataset, 10 }
  end

  test ":datasets returned with count if dataset and count is given" do
    assert parse_args(["--datasets", "--count", "5"]) == { :dataset, 5 }
    assert parse_args(["--datasets", "-c",      "5"]) == { :dataset, 5 }
  end
```

```
  test ":location returned with default count if only location is given" do
    assert parse_args(["--locations"]) == { :location, 10 }
  end

  test ":location returned with count if location and count is given" do
    assert parse_args(["--locations", "--count", "8"]) == { :location, 8 }
    assert parse_args(["--locations", "-c",      "8"]) == { :location, 8 }
  end

  test ":location returned with city location and city is given" do
    assert parse_args(["--locations",
                       "--location",
                       "Munich"]      ) == { :location, "Munich" }
  end

  test ":data returned with -s, -d, -b and -e if all required data is given" do
    assert parse_args(["--data",
                       "--dataset",  "ABCD",
                       "--location", "Munich",
                       "--from",     "2014-12-24",
                       "--to",       "2015-01-01"]) == { :data,
                                                         "ABCD",
                                                         "Munich",
                                                         "2014-12-24",
                                                         "2015-01-01" }
  end

  test ":data returned with values provided in arbitrary sequence" do
    assert parse_args(["--data",
                       "--location", "Munich",
                       "--from",     "2014-12-24",
                       "--to",       "2015-01-01",
                       "--dataset",  "ABCD"]      ) == { :data,
                                                         "ABCD",
                                                         "Munich",
                                                         "2014-12-24",
                                                         "2015-01-01" }
  end

end
```

If we run the test we see it fail.

Listing 11: mix test

```
Compiled lib/noaa/cli.ex
Generated noaa.app
........

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
```

```
8 tests, 0 failures

Randomized with seed 126754
```

To make it pass we have to refactor our code where we do the parsing. We replace the _ and the `:data` parts with a function `parse_remains` that is trying to match the command line arguments that haven't been matched by the case statements before.

Listing 12: lib/cli4.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count    COUNT
  * locations --count    COUNT
  * locations --search   CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
    parse = OptionParser.parse(argv,
                               switches: [ help:      :boolean,
                                           datasets:  :boolean,
                                           locations: :boolean,
                                           data:      :boolean,
                                           count:     :integer,
                                           search:    :string,
                                           dataset:   :string,
                                           location:  :string,
                                           from:      :string,
                                           to:        :string ],
                               aliases:  [ h:         :help,
                                           c:         :count,
```

```
                                        s:          :search,
                                        d:          :dataset,
                                        l:          :location,
                                        f:          :from,
                                        t:          :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, dataset, location, from, to }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
  IO.puts """
  noaa fetches weather data from NOAA

  usage: noaa command args

  noaa --datasets  [ --count [ count | #{@default_count} ] ]
  noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
  noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                   --to YYYY-MM-DD
  """
  System.halt(0)
end
```

```
  def process({:datasets, count}) do
    "#{count} datasets"
  end

  def process({:locations, count}) when is_integer(count) do
    "#{count} locations"
  end

  def process({:locations, city}) do
    "#{city} location"
  end

  def process({:data, values}) do
    "Data for specified city"
  end
end
```

If we run the test again we see it pass.

Listing 13: mix test

```
.............

Finished in 0.1 seconds (0.1s on load, 0.00s on tests)
13 tests, 0 failures

Randomized with seed 773652
```

Next we fetch the data from `NOAA`.

5 Fetch Weather Data from NOAA

The next step is to process the parsed command line data. So we add a `process` function to our `run` function in the `Noaa.CLI` module. But before we do that we write the test.

Listing 14: test/cli_process_test.exs

```
defmodule CliProcessTest do
  use ExUnit.Case
  import Noaa.CLI, only: [ process: 1 ]

  test "process :datasets to fetch datasets" do
    result = Noaa.CLI.process({:datasets, 10})

    assert result == "10 datasets"
  end

  test "process :locations to fetch locations" do
    result = Noaa.CLI.process({:locations, 10})

    assert result == "10 locations"
```

```
  end

  test "process :locations to search for a city" do
    result = Noaa.CLI.process({:locations, "Munich"})

    assert result == "Munich location"
  end

  test "process :data to fetch data for a specified city" do
    result = Noaa.CLI.process({:data,
                               ["GHCND",
                               "CITY:GM000019",
                               "2014-10-01",
                               "2015-01-01"]})

    assert result == "Data for specified city"
  end
end
```

When we run the test we see it fail. To make the test pass we implement the functionality in the `Noaa.CLI` module.

Listing 15: lib/noaa/cli5.ex

```
defmodule Noaa.CLI do

  @default_count 10

  @moduledoc """
  Handle the command line parsing and dispatching to the respective functions
  that list the weather conditions of provided cities.
  """

  def run(argv) do
    argv
    |> parse_args
    |> process
  end

  @doc """
  `argv` can be one of the following options.

  * datasets  --count    COUNT
  * locations --count    COUNT
  * locations --search   CITY
  * data      --dataset DATASET --location LOCATION --from DATE --to DATE

  Return the tuple of `{:dataset, COUNT}`, `{:locations, COUNT}`,
  `{:locations, CITY}`, `{:data, DATASET, LOCATION, DATE, DATE}` or :help.
  """
  def parse_args(argv) do
```

```
  parse = OptionParser.parse(argv,
                             switches: [ help:       :boolean,
                                         datasets:   :boolean,
                                         locations:  :boolean,
                                         data:       :boolean,
                                         count:      :integer,
                                         search:     :string,
                                         dataset:    :string,
                                         location:   :string,
                                         from:       :string,
                                         to:         :string ],
                             aliases:  [ h:          :help,
                                         c:          :count,
                                         s:          :search,
                                         d:          :dataset,
                                         l:          :location,
                                         f:          :from,
                                         t:          :to ])

  case parse do
    { [ help: true ], _, _ }
      -> :help
    { [ datasets: true, count: count ], _, _ }
      -> { :dataset, count }
    { [ datasets: true ], _, _ }
      -> { :dataset, @default_count }
    { [ locations: true, count: count ], _, _ }
      -> { :location, count }
    { [ locations: true ], _, _ }
      -> { :location, @default_count }
    { [ locations: true, location: location ], _, _ }
      -> { :location, location }
    _ -> parse_remains(parse)
  end
end

def parse_remains([ data: true,
                    dataset: dataset,
                    location: location,
                    from: from,
                    to: to ]), do: { :data, dataset, location, from, to }

def parse_remains({ parse, _, _ }) do
  parse_remains(Enum.map([:data, :dataset, :location, :from, :to],
                         fn(x) -> List.keyfind(parse, x, 0) end))
end

def parse_remains(_), do: :help

def process(:help) do
```

```
    IO.puts """
    noaa fetches weather data from NOAA

    usage: noaa command args

    noaa --datasets  [ --count [ count | #{@default_count} ] ]
    noaa --locations [ --count [ count | #{@default_count} ] | --search city ]
    noaa --data      --dataset dataset --location location --from YYYY-MM-DD \
                     --to YYYY-MM-DD
    """
    System.halt(0)
  end

  def process({:datasets, count}) do
    "#{count} datasets"
  end

  def process({:locations, count}) when is_integer(count) do
    "#{count} locations"
  end

  def process({:locations, city}) do
    "#{city} location"
  end

  def process({:data, values}) do
    "Data for specified city"
  end
end
```

When we run the test again we see it pass.