The Art of Computational Science

The Kali Code

vol. 6a

User Interface: Command Line Arguments I

Piet Hut and Jun Makino

September 13, 2007

Contents

Preface

The current volume contains a rather fancy parser of command line arguments, together with a detailed help facility. We have found such a system to be indispensable when running large numbers of simulations, using different codes to generate initial data, evolve the N-body systems, and analyze the results. Therefore, we decided to take the time to develop and introduce the whole system here in detail. An additional advantage is that we can show the power and practical usefulness of Ruby, not only for prototyping the scientific modules in a simulation, but also for helping out with the important task of administering N-body runs.

A good piece of software should administer the runs in such a way as to keep the astrophysicists protected from the details, so that they can give their full attention to the scientific problems at hand. If you are not interested in the details of how this administration is carried out, it would be sufficient to just skim this volume, reading only chapters 1, 2, and 11. However, if you plan to use Ruby extensively for your own simulation needs, in new and creative ways, it probably will pay off to spend some time looking at the other chapters as well, in order to get an idea of the rich pallet of possibilities that Ruby has to offer. It may well give you some new ideas for your own applications.

0.1 Acknowledgments

Besides thanking our home institutes, the Institute for Advanced Study in Princeton and the University of Tokyo, we want to convey our special gratitude to the Yukawa Institute of Theoretical Physics in Kyoto, where this volume was written, during a visit in June 2004, made possible by the kind invitations to both of us by Professor Masao Ninomiya.

We thank Bill Guindon for his comments on the manuscript.

Piet Hut and Jun Makino

Kyoto, July 2004

Chapter 1

Command Line Arguments

1.1 A New Approach

Bob: Hi Alice! Look what I've done, since we last met.

Alice: What have you done?

Bob: I added command line arguments to our latest N-body code. I was getting so tired of having to edit a line in our driver file, each time we were doing a different run. It was high time that we made this switch. Now we can instruct the code on the command line what options and value to give to the `evolve` method.

Alice: Can you show me what you did?

Bob: Here is the file `rkn1.rb`, which reads the command line, and then invokes `evolve`. But before showing you the contents, let me first show you how it works. Here is an example:

```
|gravity> ruby rkn1.rb -o 2.1088 -e 2.1088 -t 2.1088 < figure8.in
eps = 0
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
```

```
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

You can see from the values that were echoed that I just ran a fourth-order Runge-Kutta, for 1/3 of an orbit of a figure-eight triple configuration. And by the way, you can see from the output that I have reproduced the same positions and velocities as before, as a test that the code still works correctly.

1.2 Flexibility

Alice: So how do you ask the code to use, say, a leapfrog integrator instead of your default fourth-order Runge-Kutta, perhaps with a ten times smaller time step?

Bob: Easy! This is what you type:

```
|gravity> ruby rkn1.rb -o 10 -e 2.1088 -t 2.1088 -m leapfrog -d 0.0001 < figure8.i
eps = 0
dt = 0.0001
dt_dia = 2.1088
dt_out = 10.0
dt_end = 2.1088
init_out = false
x_flag = false
method = leapfrog
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.1088, after 21088 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
```

```
             E_tot - E_init = -6.35e-13
  (E_tot - E_init) / E_init = 4.93e-13
```

You see, I added the option `-o 10` to suppress the snapshot output. This makes the interface much more flexible than it was before.

Alice: What exactly is the meaning of `-o`, and so on?

Bob: rather than answering you, let me ask the code. It even has a help function:

```
|gravity> ruby rkn1.rb -h
usage: rkn1.rb [-h (for help)] [-s softening_length] [-d step_size]
         [-e diagnostics_interval] [-o output_interval]
         [-t total_duration] [-i (start output at t = 0)]
         [-x (extra debugging diagnostics)]
         [-m integration_method]
```

You see now what I did. By adding the option `-o 10` to the command line in my last little run above, I asked the program to postpone the first output to the time `t = 10` which is later than the time `t = 2.1088` at which I had ordered the program to halt and to give energy diagnostics. In that way, I suppressed the output of the snapshot, so that we could concentrate on looking only at the energy.

1.3 Various Options

Alice: Adding a help facility is a great improvement, I agree! But what would happen if I would have typed `--help` instead? I would not have guess that the help option would have been the old-fashion Unix-style `-h`.

Bob: Try it!

Alice: Okay:

```
|gravity> ruby rkn1.rb --help
usage: rkn1.rb [-h (for help)] [-s softening_length] [-d step_size]
         [-e diagnostics_interval] [-o output_interval]
         [-t total_duration] [-i (start output at t = 0)]
         [-x (extra debugging diagnostics)]
         [-m integration_method]
```

Ah, the same answer. Good! But your `help` answer is not complete: it suggests that you can only use single letter options.

Bob: You're right. I could have added that explicitly. But in a way, it is there already. Try your "`--`" notation with the longer words that appear in the help answer.

Alice: You mean:

```
|gravity> ruby rkn1.rb --total_duration 1 --output_interval 10 < figure8.in
eps = 0
dt = 0.001
dt_dia = 1
dt_out = 10.0
dt_end = 1.0
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 1000 steps :
  E_kin = 1.22 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
```

I yes, that indeed works. But what about the `-i` option? That was a flag, if I remember it correctly; if you set it, you got the initial output. What is the long version of the command to set the flag?

Bob: Let me look at the code. Ah, the long option is `--initial_output`. I could have written that in the `help` output, but I decided that that would be too cryptic. If you would have gotten `[-i initial_output]` as part of the answer, that probably would not made much sense. Instead, I let the `help` option echo the sentence `[-i (start output at t = 0)]`.

1.4 A Critical Attitude

Alice: Well, I applaud the *idea* of using command line options, and I also very much like the addition of a `help` function. Both aspects are essential in a good user interface. However, if you don't mind me saying so, your current `help` facility still needs a lot of help.

Bob: It's just my first try. In fact, please be critical! I would love to provide a good user interface. For one thing, if we don't, my students will keep knocking on my door to ask me how to use all these codes. If we can make things more transparent, it will actually save me a lot of time, if that means that the students can figure out things for themselves.

Alice: You really want me to be really critical? You may regret asking!

Bob: Sure, go ahead, be critical!

Alice: Okay! Then I won't hold back. I already mentioned a few things that were not clear, and certainly not yet documented, but now that you challenge me, why don't we go through the code you wrote, and I will critically appraise your whole approach!

Bob: You look as if you mean business. But I have nothing to hide. Here is the code, in file `rkn1.rb` It is only a couple pages. Let me print it out first, and then we can walk through it.

```
require "rknbody.rb"

def print_help
  print "usage: ", $0,
    " [-h (for help)] [-s softening_length] [-d step_size]\n",
    "         [-e diagnostics_interval] [-o output_interval]\n",
    "         [-t total_duration] [-i (start output at t = 0)]\n",
    "         [-x (extra debugging diagnostics)]\n",
    "         [-m integration_method]\n"
end

require "getoptlong"

parser = GetoptLong.new
parser.set_options(
  ["-d", "--step_size", GetoptLong::REQUIRED_ARGUMENT],
  ["-e", "--diagnostics_interval", GetoptLong::REQUIRED_ARGUMENT],
  ["-h", "--help", GetoptLong::NO_ARGUMENT],
  ["-i", "--initial_output", GetoptLong::NO_ARGUMENT],
  ["-m", "--integration_method", GetoptLong::REQUIRED_ARGUMENT],
  ["-o", "--output_interval", GetoptLong::REQUIRED_ARGUMENT],
  ["-s", "--softening_length", GetoptLong::REQUIRED_ARGUMENT],
  ["-t", "--total_duration", GetoptLong::REQUIRED_ARGUMENT],
  ["-x", "--extra_diagnostics", GetoptLong::NO_ARGUMENT])

def read_options(parser)
  dt = 0.001
  dt_dia = 1
  dt_out = 1
  dt_end = 10
  eps = 0
  init_out = false
  x_flag = false
  method = "rk4"

  loop do
    begin
      opt, arg = parser.get
      break if not opt

      case opt
      when "-d"
        dt = arg.to_f
      when "-e"
        dt_dia = arg.to_f
      when "-h"
        print_help
        exit         # exit after providing help
      when "-i"
        init_out = true
      when "-m"
        method = arg
      when "-o"
        dt_out = arg.to_f
      when "-s"
        eps = arg.to_f
      when "-t"
        dt_end = arg.to_f
      when "-x"
        x_flag = true
      end

    rescue => err
      print_help
      exit           # exit if option unknown
    end

  end

  return eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag, method
end

eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag, method =
   read_options(parser)

STDERR.print "eps = ", eps, "\n",
      "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
       "dt_out = ", dt_out, "\n",
       "dt_end = ", dt_end, "\n",
       "init_out = ", init_out, "\n",
       "x_flag = ", x_flag, "\n",
       "method = ", method, "\n"

include Math

nb = Nbody.new
nb.simple_read
nb.evolve(method, eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
```

1.5 An Overview

Alice: Can you show me roughly how it all works? No need to go into all the details, right now, since we'll probably want to change the functionality soon. If you can just show me the flow of control, that would be fine.

Bob: Don't worry, I'll give you an overview, just enough information to start your critical quest!

The first line of the file `rkn1.rb` reads:

```
require "rknbody.rb"
```

which means that it reads in the file `rknbody.rb`, which is just a straight copy from the file `rknbody9.rb`, where we had just introduced softening as an extra option. Remember, that file contained the `Body` and `Nbody` definitions. The file `rknbody.rb` will be the beginning of our N-body library.

Alice: That is something I definitely approve of, to keep the best bits and piece of our prototyping, and to use those to build up a library.

Bob: Until now, we have used a special driver file where we wrote down the arguments that were given to the method `evolve`. In this case, these arguments will be plucked from the command line, through the method `read_options`, the longest method in this file. But before we get there, it will be easiest to read the file starting at the end.

The last three lines:

```
nb = Nbody.new
nb.simple_read
nb.evolve(method, eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
```

are exactly the same as the last three lines of the old driver file that we used to invoke the softened version of our N-body code. Just above that, we report the parameter values that are actually used in the run:

```
STDERR.print "eps = ", eps, "\n",
      "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "init_out = ", init_out, "\n",
      "x_flag = ", x_flag, "\n",
      "method = ", method, "\n"
```

These are also exactly the same as what was written in the old driver. However, working our way back up, from here on things are different. Where we wrote down the values of all parameters by hand in the old driver code, here there are assigned by a call to `read_options` with a single argument, called `parser`, as follows:

```
eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag, method =
   read_options(parser)
```

The method `read_options` is written just above this call. At the top of that method, all default values are assigned to the parameters that will become the arguments for the call to `evolve`. Then the method enters a loop in which all command line options are being read in. They are being parsed, as they say in computer science, which just means that their meaning is being interpreted in the correct way.

Alice: And I see that you have defined `parser` as a global variable. Somehow this variable, which is an instance of a class called `GetoptLong`, knows how to gather the information about all legal options in your code, both the one-letter versions starting with a single "`-`" sign, as well as the longer command line arguments that start with "`--`".

Bob: Yes, this is a piece of magic that is provided by a package that I loaded through the statement near the top:

```
require "getoptlong"
```

Alice: What does this package do, and roughly how does it work?

Bob: I must admit, I did not really look at it. I was browsing in a few Ruby books, and I came across this example. It seemed really handy, so I copied it. And the example was easy enough to change to my own requirements. The important thing is: it works! And it has made my life already a lot easier.

1.6 A Beautiful Violation

Alice: All of that I don't doubt. And as I already mentioned, this is definitely going in the right direction, but if you want my honest opinion . . .

Bob: . . . nothing less!

Alice: . . . then I must say: your writing style is a terrible violation of the DRY principle. Or perhaps I should say: a beautiful example, since it violates the Don't Repeat Yourself so flagrantly!

Look, each option, from "`-h`" all the way to "`-m`", occurs three times: in the `print_help` method at the top of the file, then immediately after that in the `parser.set_options` block, and then once again in the method following just below, `read_options`.

This means that you have created an ideal breading ground for bugs! How easy it will be to add or change an option, and to forget to make the addition or change in all three places. Or worse, to make different modifications in different places. Just a single typo will be enough. And you're likely to make typos because it is such boring to repeat the work for changing the same option in three different places.

Bob: Well, that may all be true, but aren't you chasing after a form of Utopia? There is a reason to do things three times. In the first occurrence, in `print_help`, we provide a help facility for the user. In the second occurrence, in `parser`, we tell the parser what to expect on the command line, and to hand it to us. In the third occurrence, in `read_options` we interpret what the `parser` has just handed us, and in doing so we prepare for the proper execution of `evolve`.

I admit that I am handling each option three times, but frankly, I doubt that we have a choice. Look. This driver file `rkn1.rb` does three things: 1) it instructs the parser to deliver information, in a precise way, so that it will get the correct information; 2) it interprets the information and then passes it on to the `evolve` method, again in a precise way, as needed by `evolve`; 3) it also is friendly enough to share all that information with the user, through a help facility.

I simply don't see how we can dispense with providing the central information to these three players: the parser, the evolver, and the user. You have been stressing modularity so often. As I understand modularity, this means that the different modules don't know from each other how they deal with the information that they get. And this then means that someone has to play the role of go-between. You talked about interface specifications. Well, here my driver program forms a type of interface between parser, evolver, and user.

1.7 Arguments about Arguments

Alice: All of that is true. The parser should not and cannot know about the way the options are going to be used by the `evolve` function, and neither of those two should know about friendly user interfaces and how exactly such a user interface implements a help facility.

Bob: So you agree that someone somewhere has to do a three-way translation of the information?

Alice: Yes.

Bob: Good! And I think it is not fair to say that I violated the DRY principle. It is true that I *mentioned* each option three times. But each time I did something quite different with it. So I did not *repeat* anything, really. It would be like receiving a dollar bill from someone, showing it to you to let you know what I got, and then spending it somewhere in a store. Each of those three actions is different and totally independent of the other two. So none of these actions can be said to repeat any of the other ones.

Alice: No, at this point I disagree. You did repeat yourself, in the code.

Bob: Before disagreeing about the code, can you tell me what is wrong with my simple analogy of the dollar bill?

Alice: That was not a good analogy. In fact, there you did *not* repeat yourself. You mentioned the word 'dollar bill' only once! If your sentence would have reflected your style of code writing, you might have said something like the following: "tt would be like receiving a dollar bill from someone, showing the dollar bill to you to let you know what I got, and then spending the dollar bill somewhere in a store." Notice how strange that would sound. In that case you would have mentioned the words 'dollar bill' three times, in a way that sounds odd, since people would expect to hear it only once.

In other words, I don't object against using the information dealt with in the command line options three times. You are right: we have no choice there. This is a direct consequence from the fact that we are coding in a modular way, and that information has to be passed from module to module. What I do object against is the fact that you are explicitly using the same *label* three times. That is where you are inviting mistakes to happen.

Bob: I still don't see how I can pass the same information around if I don't use the same label. Your analogy is fishy.

Alice: It was your analogy.

Bob: Ah, but you twisted it around. In a natural language, like English, you can replace the words 'dollar bill' by 'it', and somehow native speakers can figure out what each 'it' refers to. But in a computer language that doesn't work. A computer language has to be precise. Even Ruby is hopefully unambiguous in its meaning, unlike natural languages!

Alice: I don't think we will convince each other on the level of analogies. Why don't we sit down and see whether we can adapt your first command line argument parser, in such a way that we avoid repetition.

Bob: If you think that can be done, fine. I don't think you can succeed, but I'm willing to give it a try. I agree that these arguments about parsing command line arguments can only be decided by hard-nosed coding examples.

Chapter 2

Encapsulating Information

2.1 A Soft Violation

Bob: Hi Alice, are you still convinced that you find a way to avoid repetition in a command line option parser?

Alice: Hi Bob! I gave it a good deal of thought, and I came to the conclusion that we were both right.

Bob: How can that be?

Alice: You were right in pointing out that the three places in your program did three quite different things: parsing information, passing it on to the computer, and optionally passing it to the user through a help facility.

At the same time, I was right in pointing out the danger of having the information about that information scattered around in those three different places. I called it a violation of the DRY principle, the notion of Don't Repeat Yourself. But I guess you did not commit a hard violation of the principle, since you did not literally repeat yourself.

Perhaps you could call it a soft violation. The problem I objected to was the fact that you mentioned the same option in three different places. And even though you did something different concerning that option in each of those different places, there still is the danger that if you change the functionality of that option, you can introduce subtle bugs if you don't update all three places correctly.

Bob: Yes, I agree that there is that danger.

Alice: In fact, when I looked at your code, I realized that you actually deal with each option *four* times! I only saw the first three, when I scanned the actual mention of each option, but at the end you use the option information once again.

Bob: Can you show me?

2.2 Four Occurrences

Alice: Take the time step information, for example. First it appears in your `help` description of the `-d` option:

```
def print_help
  print "usage: ", $0,
    " [-h (for help)] [-s softening_length] [-d step_size]\n",
    "         [-e diagnostics_interval] [-o output_interval]\n",
    "         [-t total_duration] [-i (start output at t = 0)]\n",
    "         [-x (extra debugging diagnostics)]\n",
    "         [-m integration_method]\n"
end
```

Then it occurs a second time as the first option listed in the call to the parser, which have loaded through the `getoptlong` package:

```
require "getoptlong"

parser = GetoptLong.new
parser.set_options(
  ["-d", "--step_size", GetoptLong::REQUIRED_ARGUMENT],
  ["-e", "--diagnostics_interval", GetoptLong::REQUIRED_ARGUMENT],
  ["-h", "--help", GetoptLong::NO_ARGUMENT],
  ["-i", "--initial_output", GetoptLong::NO_ARGUMENT],
  ["-m", "--integration_method", GetoptLong::REQUIRED_ARGUMENT],
  ["-o", "--output_interval", GetoptLong::REQUIRED_ARGUMENT],
  ["-s", "--softening_length", GetoptLong::REQUIRED_ARGUMENT],
  ["-t", "--total_duration", GetoptLong::REQUIRED_ARGUMENT],
  ["-x", "--extra_diagnostics", GetoptLong::NO_ARGUMENT])
```

The third time, we encounter this same `-d` option in the inner loop of the `read_options` method, in the lines:

```
      case opt
      when "-d"
        dt = arg.to_f
```

What I had not realized yesterday was the fact that at the end, you echo all the values that are set, before you start the integration. That is certainly a good thing to do, since it allows the user to see explicitly what parameter values were used by the integrator, including default values that were not set by the user. But you see, again the time step shows up, this time not through a mention of `-d`, but through the second line in the initial print statement:

```
STDERR.print "eps = ", eps, "\n",
      "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "init_out = ", init_out, "\n",
      "x_flag = ", x_flag, "\n",
      "method = ", method, "\n"
```

2.3 Danger

Bob: But why should that bother you?

Alice: imagine that you want to change the internal way to store the data. Or more seriously, imagine that someone else wants to adapt your program for additional applications, and therefore wants to change the internal way to store the data. Instead of assigning the time step to the variable `dt`, perhaps she wants to assign that value to a variable `dynamics_dt`, since she also has a stellar evolution module for which she is using a time step `evolution_dt`.

Now she has to realize that she has to change the appropriate line in the inner loop of the `read_options` method, and in the second line of the `STDERR.print` command. In those two places, `dt` occurs *three* times, and she has to realize that she has to change two of the three as follows: in the `read_options` place she has to write:

```
        dynamics_dt = arg.to_f
```

and in the `STDERR.print` command, she has to write:

```
      "dt = ", dynamics_dt, "\n",
```

Do you see the potential for confusion, and hence mistakes.

Bob: Hmmm. I must admit that there is that possibility, yes.

Alice: So this is what I meant when I said that we were both right. You are doing four *different* things in four different places, so in that sense you are not repeating anything. At the same time, you deal with the *same* variable in those four different places, either through their command line option or their internal representation.

Bob: You are saying that I do *repeatedly* something different with the same piece of information.

Alice: Exactly.

Bob: And it would be better if we could avoid that.

Alice: Indeed.

Bob: But I come back to my question: how can we avoid that? It would be terribly clumsy to do all four actions for the first option first, then to do those for actions for the second option, and so on. In that way, you could keep the information for one option all close together, but you would have to write a copy of all four commands for each new option!

Alice: That would not be a good solution, I agree. We have to think of something better.

2.4 A Matter of Principle

Bob: You always like to come up with principles. Can't you think of a principle that will help us out here?

Alice: Now that you challenge me, perhaps we can use Ruby itself as an example. Ruby is built on the principle of indirect addressing, which is why it is so flexible. Perhaps we could avoid using the information itself in the four places that I pointed out. How about storing the real information in a fifth place? If we can find a way to access that fifth place in the other four instances where we need the information, we can keep things under control.

Bob: Ah, you mean that the code writer has write access to the data in that fifth place, while the other four places only have read access to those data?

Alice: I guess, yes, that is a good way to put it. I mentioned the case of someone wanting to adapt the code for a different purpose. Rather than changing it in the two different places I mentioned above, writing

```
        dynamics_dt = arg.to_f
```

and

```
      "dt = ", dynamics_dt, "\n",
```

all she would have to do is to go to the storage place where all the real information is kept, and change one line there. If the information was kept there as:

```
  Internal variable:  dt
```

then all she would have to do is change this line to:

```
  Internal variable:  dynamics_dt
```

Bob: I think I begin to see how this could be implemented. The two lines you mentioned could be replaced by:

```
        time_step_variable_name = arg.to_f
```

and

```
      "dt = ", time_step_variable_name, "\n",
```

and somewhere an action would be taken that would replace `time_step_variable_name` everywhere with whatever the code writer would have specified in the `Internal variable:` slot for the time step option. In the above case, `time_step_variable_name` would be replaced everywhere by `dynamics_dt`.

Alice: Yes. This is the principle of indirect addressing, or the principle of indirection, I guess.

Bob: That sounds like a lack of direction to me. But lets forget about naming your principle – we could leave that open as well, and indirectly address your principle later.

Alice: I see you still don't like principles, but you must admit, this one gave us a new idea.

2.5 An Option Block

Bob: I admit. So how to go about this? Ah, each option would have such a definition, right? And each option would need more than the information of the name of the *internal* variable. At a minimum, it would have to know about the name of the *external* handle, namely the name of the option on the command line argument; in our case `-d` or in longer form `--step_size`.

Alice: I can see from the look in your eyes that you're getting ready to code something up!

Bob: And each option would need a type, for the input to work properly. Even though Ruby has dynamic typing, someone somewhere has to tell Ruby that the number of particles, when read in from the command line, has to be converted to an integer, while the name of the integration method retains the type of a string, and the time step size becomes a floating point number.

Hmm. This becomes really interesting. So each option will be characterized by a single block of information, which could be an instance of a new class. And it could contain help information as well!

Alice: Ah, yes, of course. And if you would change either the functionality or just the names of some of the information in a block, you would naturally modify the help message as well, to reflect the changes you made. Since everything lives together in one paragraph, so to speak, there is no obstacle against keeping things up-to-date together.

Bob: We could even have more than one help level. For example, typing "-h" could lead to a one-line help message being displayed, while typing "–help" could give you a more detailed multi-line message.

Alice: So all the information would be bundled: the internal representation, to be used by the computer when running a program, the specification of the command line interface, to get the information from the user into the computer, and help messages, to get information back to the user. I like it!

Bob: Not only that, there is another flow of information back to the user, when a program starts running and echoes its initial state, as you pointed out earlier. So each block could have a 'print name' for its internal variable as well. For example, the number of particles could be specified on the command line by typing `-n 3` for an N-body system, or `--number_of_particles 3` or something like that. Internally the variable storing that information might be `n_part`. But when you echo the initial state, it is much more natural to just type `N = 3`. This could be specified by a block with the lines

```
Short name:          -n
Long name:           --number_of_particles
Value type:          int
Default value:       1
Global variable:     n_part
Print name:          N
Description:         Number of particles
Long description:
  Number of particles in the N-body system,
  that is generated by this program.  The
  positions will be chosen at random within
  a sphere of unit radius, and the velocities
  will be set to zero.
```

The `Description` content can then be displayed after a `-h` request, and the

`Long description` content appears when you give a `--help` request. I started the latter on a new line, since it will be the only piece of information that will need to be spread out over more than one line, and starting it at the beginning of the line will allow us to format it properly for display.

2.6 Growing a Manual

Alice: Very nice! In fact, when you have several options, and you give each option such a detailed `Long description`, those together in effect form a kind of man page, like in the Unix system, a form of manual page that summarizes the interface to the program. And keeping all that information within the code itself will be a form of insurance.

We all know of cases where the manual page for a code says one thing, while the code does something else, because someone modified the code and did not bother to update the man page. Now if the manual information lives in the very same place where the actual information about the main variables is kept, there is no longer *any* barrier against keeping information up to date.

Bob: Even I can bring up the discipline to keep documentation up to date in that way, I expect. And now that you mention man pages, a natural thing would be to add examples in the `Long description`. How about:

```
Long description:
  Number of particles in the N-body system,
  that is generated by this program.  The
  positions will be chosen at random within
  a sphere of unit radius, and the velocities
  will be set to zero.

    Example: "ruby mk_cold_collapse -n 100"

    will generate a cold system containing 100
    particles.
```

Alice: So you allow blank lines in the output. Well, why not. If we consider each help message to form a part of a manual, it would only be natural to allow new paragraphs and blank lines to appear. It certainly will make things more readable. We just have to be careful to find a unique way to get the information listed so as not to confuse two options.

Bob: Easy! We can insist on *two* blank lines between options. If we insist that the `Long description` allows appears at the very end of a block, then a single blank line means that the `Long description` still continues, whereas a double blank line means that we now start the next block, for a new option.

Alice: I think you have found a great way to make a top-down specification for a user interface for all of our programs! Before we write a program to parse all that information, how about going all the way, top-down wise? We may as well specify the whole series of option blocks for our N-body program. Once we are happy with that, we can implement a parser, and then use all that to replace your previous driver `rkn1.rb`.

Bob: Good! In that way, it will be easier to write the parser, with a concrete example in front of us. I can just take the options from `rkn1.rb`, fill in the blanks for the variables, and weave the appropriate help texts into each block.

Alice: Go right ahead!

2.7 An Full Option List

Bob: Okay. This is the type of code that writes itself, once you get the idea!

Alice: As long as you write it, I can maintain your illusion that the program writes itself.

Bob: What about this, for our N-body program? You see: I am adding a top-level not-an-option option, which only contains two entries, `Description` and `Long description` to tell us what the whole code is doing, before getting into the detailed information of each option. This could be the opening paragraph of the manual page.

And come to think of it, let me put everything in one long 'here document'. That will make it easy to pass this option block list around, as a single string.

```
options_definition_string = <<-END

  Description:          The simplest ACS N-body code
  Long description:
    This is the simplest N-body code provided in the ACS environment
    (ACS: Art of Computational Science; cf. "http://www.ArtCompSci.org").
    It offers a choice of integrators, for constant shared time steps.


  Short name:           -m
  Long name:            --integration_method
  Value type:           string
  Default value:        rk4
  Global variable:      method
  Print name:                                 # blank: suppresses glob. var. name
  Description:          Integration method
  Long description:
    There are a variety of integration methods available, including:
```

```
    Forward Euler:            forward
    Leapfrog:                 leapfrog
    2nd-order Runge Kutta:    rk2
    4th-order Runge Kutta:    rk4


Short name:           -d
Long name:            --step_size
Value type:           float
Default value:        0.001
Global variable:      dt
Description:          Integration time step
Long description:
  In this code, the integration time step is held constant,
  and shared among all particles in the N-body system.


Short name:           -e
Long name:            --diagnostics_interval
Value type:           float
Default value:        1
Global variable:      dt_dia
Description:          Diagnostics output interval
Long description:
  The time interval between successive diagnostics output.
  The diagnostics include the kinetic and potential energy,
  and the absolute and relative drift of total energy, since
  the beginning of the integration.
      These diagnostics appear on the standard error stream.
  For more diagnostics, try option "-x" or "--extra_diagnostics".


Short name:           -o
Long name:            --output_interval
Value type:           float
Default value:        1
Global variable:      dt_out
Description:          Snapshot output interval
Long description:
  The time interval between output of a complete snapshot
  A snapshot of an N-body system contains the values of the
  mass, position, and velocity for each of the N particles.

      This information appears on the standard output stream,
  currently in the following simple format (only numbers):
    N:            number of particles
    time:         time
    mass:         mass of particle
    position:     x y z : vector components of position of particle
    velocity:     vx vy vz : vector components of velocity of particle
    mass:         mass of particle
    ...:          ...

  Example:

     2
     0
     0.5
    7.3406783488452532e-02  2.1167291484119417e+00 -1.4097856092768946e+00
    3.1815484836541341e-02  2.7360312082526089e-01  2.4960049959942499e-02
     0.5
   -7.3406783488452421e-02 -2.1167291484119413e+00  1.4097856092768946e+00
   -3.1815484836541369e-02 -2.7360312082526095e-01 -2.4960049959942499e-02


Short name:           -t
Long name:            --total_duration
Value type:           float
Default value:        10
Global variable:      dt_end
Description:          Duration of the integration
Long description:
  This option allows specification of the time interval, after which
  integration will be halted.


Short name:           -s
Long name:            --softening_length
Value type:           float
Default value:        0
Global variable:      eps
Description:          Softening length
Long description:
  This option sets the softening length used to calculate the force
  between two particles.  The calculation scheme comforms to standard
  Plummer softening, where rs2=r**2+eps**2 is used in place of r**2.


Short name:           -i
Long name:            --init_out
```

```
  Value type:           bool
  Global variable:      init_out
  Description:          Output the initial snapshot
  Long description:
    If this flag is set to true, the initial snapshot will be output
    on the standard output channel, before integration is started.


  Short name:           -x
  Long name:            --extra_diagnostics
  Value type:           bool
  Global variable:      x_flag
  Description:          Extra diagnostics
  Long description:
    The following extra diagnostics will be printed:

      acceleration (for all integrators)
      jerk (for the Hermite integrator)


  END
```

Alice: Wonderful! That contains all the information needed for a computer as well as for a human reader. How nice! You can just go through it and already get a good feeling for what the code is doing, without reading any line of code yet.

Just one question: why is there a minus sign before the `END` in the beginning of the specification of the 'here document'?

Bob: Oh, hyphen means that we can put the `END` anywhere on a line, not necessarily flush with the left margin. In other words, it does not have to start in the first column, in the old language of punch cards, but it can appear indented and still be recognized as the proper `END`. And as you can see, I indeed ended the 'here document' with a few spaces in front of the ending `END`, since it looked more natural in that way.

Alice: Now all we have to do is implement it, by writing a parser.

Bob: I'm happy to give that a try. Now that we have specified the procedure, it shouldn't be too hard to write the code to make it all come alive. Given the flexibility of Ruby, and a healthy dose of regular expression magic, this should be doable. And I'm sure glad we don't have to code this up in C++ or Fortran!

Alice: Indeed, this is the ideal task for a scripting language. Even the name fits: we have just produced a script for specifying an N-body dance.

Bob: Okay, let me give it a shot. Next time we meet I should have at least something workable.

Chapter 3

Implementation: `Clop` Entry Points

3.1 A New Driver

Alice: Hi Bob! How's it going with your attempt to implement the option block idea?

Bob: Well, it took me quite a bit longer than I thought, but that was mainly because I got more and more ideas for further improvements, while I was writing the parser. And now I'm really hooked to the use of a scripting language! It was wonderful to see how easy it was to add functionality, and to change prototype behavior on the fly, just to try out various options.

Alice: Ah, I can see that your parsing code grew as a result.

Bob: Yes, but not as much as I would have expected. Compared to my much simpler parser, the length grew by only a factor four, and even that is not a fair comparison at all, since I used a piece of canned magic before, by adding

```
require "getoptlong"
```

Who knows how long that code is. In contrast, this time I wrote everything myself, and the functionality is vastly increased.

Alice: Can you walk me through the code, to show me the new magic?

Bob: I'm glad to do so! Let me just follow the flow of control, right from the beginning.

In my new driver, `rkn2.rb`, I now start as follows:

```
require "rknbody.rb"
require "clop.rb"
```

You see, I'm including the `Body` and `Nbody` classes, as before, in the first line, and I'm including the new parser that I wrote, which lives in `clop.rb`. You will appreciate the modularity: In `clop.rb` there is no knowledge about N-body systems; in fact, a chemist or a biologist could use `clop.rb` equally well for completely different purposes.

Alice: Hear, hear!

Bob: I thought you would like that. Now following those two lines, there appears this one long 'here document' that we already wrote before, containing the full list of option blocks. Then, the only thing left in this file `rkn2.rb`, are the following lines:

```
parse_command_line(options_definition_string)

include Math

nb = Nbody.new
nb.simple_read
nb.evolve($method, $eps, $dt, $dt_dia, $dt_out, $dt_end, $init_out, $x_flag)
```

So that is all there is to it! This is the whole driver. It contains a two lines at the start to specify what needs to be included, a few lines at the end, the rest is one long list of option blocks in a single 'here document'. And all the work is done in the file `clop.rb` that contains the parser.

3.2 Invoking the Parser

Alice: So the last three lines are almost the same as the last three lines in your first attempt at parsing the command line, in file `rkn1.rb`:

```
nb = Nbody.new
nb.simple_read
nb.evolve(method, eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
```

The only difference is that all the parameters of the call to `evolve` are now global variables, if I remember Ruby's convention correctly.

Bob: Yes, in Ruby, a variable starting with a dollar sign is by definition a global variable. Normally I would not like to use global variables, but here it seemed like a natural way to get the information from the parse file `clop.rb` back into our driver. The alternative would have been to turn each variable into a method that interrogates the class `Clop` that is hiding inside `clop.rb`. Instead of using the global variable `$dt`, we could define an instance `my_clop.dt`, and so on.

You might argue that these variables are what the user is providing for a particular run, and while the run is running, these variables contain the only information to the program available from the outside world; all other information is local to the program. So to use global variables may even be natural.

Alice: I agree that this is one of the few places where global variables seem like a reasonable solution. Although I don't like them in general, I also don't like to stick to literally to any principle, even the principle 'thou shalt not use global variables'.

Bob: Another meta principle, not to stick to any principle?

Alice: Watch out, if you apply that to itself, you may get into a paradox!

Bob: Like the question "who shaves the barber?" if the barber shaves everyone who doesn't shave himself. But let's not get into that.

Alice: Now all the magic occurs because of the one call

```
parse_command_line(options_definition_string)
```

I see that you take the one humongous 'here document' string, and feed it into this method, that must be defined inside the file `clop.rb`.

Bob: Indeed! Time to open that file, and to show you what is going on. Instead of going through it from beginning to end, let me walk through the file, following the flow of the logic, starting with the method that is called here.

Alice: I'm all ears and eyes!

3.3 The Clop Class

Bob: The file `clop.rb` contains three things: there is the definition of a class called `Clop`, in front of that there is the definition of a helper calls `Clop_Option`, and after that there is a very short piece, namely the following three line definition:

```
def parse_command_line(def_str)
  Clop.new(def_str, ARGV)
end
```

Alice: That's all that happens, in order to parse the command line? This method just creates a new instances of the class `Clop`, and that's it?

Bob: That's it. Note that two essential pieces of information are passed to that new instance. The first argument contains the string with the whole list of option blocks, that was defined in the driver. That was the one and only argument passed from the driver to the `clop.rb` file. The second argument is `ARGV`, the array that contains the command line, broken up in space separated pieces.

Alice: So that is very similar to C.

Bob: Yes, except that `ARGV[0]` is already the first argument to the program, not the program name, as a C programmer might expect. So if you give a command:

```
  ruby test.rb -x -o out_file
```

then `ARGV[0] = "-x"`, `ARGV[1] = "-o"`, and `ARGV[2] = "out_file"`. Effectively what has happened is that the piece of the command line that follows the program name is treated as a string, on which the command `split` is run. In the above case, when we call the remainder of the command line, after `test.rb`, `str`, then `ARGV` is the same as the array `a` that we would obtain from the statement:

```
  a = str.split
```

The `split` command splits the one string `str` into an array of smaller strings, where blank spaces function as separators defining the extent of each smaller string.

Alice: So the logic here is that you create a new instance of the class `Clop`, and you give it all the information that it needs: the 'here document' that contains the complete interface information of our N-body code, and the `ARGV` array that contains the full information of what the user wrote on the command line. And somehow everything else happens as a side effect of creating this new `Clop` instance.

Bob: Yes. I did it that way so that you don't have to bother anymore later on about `Clop` classes. You just create one, and then you can already discard it, since upon creation it has done all its work. Let me show how.

Alice: By the way, why the name `Clop` ?

Bob: Ah, I should have mentioned that. `Clop` stands for Command Line Option Parser.

Alice: I should have guessed.

Bob: A class name `Command_Line_Option_Parser` just sounded a bit too long for my taste. On the other hand, feel free to change the name that way, if you like. In true object-oriented and modular way, the name of the class is not visible to the user. Instead, the user just gives the command

```
parse_command_line(options_definition_string)
```

Still, as you know, I prefer more terse names, hence `Clop`.

3.4 Creating a `Clop` Instance

Alice: So what happens when you create a new `Clop` instance?

Bob: Here is the initializer for the `Clop` class:

```
  def initialize(def_str, argv_array = nil)
    parse_option_definitions(def_str)
    if argv_array
      parse_command_line_options(argv_array)
    end
    print_values
  end
```

Alice: It indeed seems to do all the work required: first it parses all the definitions from the option block list from the N-body driver, then it parses all the options given on the command line.

Bob: And finally it echoes all the values that it gives back to the driver. Some of the values will be specified by the user. Other values, not specified by the user, will retain their default value. By echoing the whole set, the user will know exactly how the N-body integration got started, with what set of initial parameters.

Alice: But where does this method give those values back to the driver?

Bob: Ah, global variables, remember? Nothing is passed back explicitly. It is just made visibly globally. That's why the driver could simply give the command:

```
nb.evolve($method, $eps, $dt, $dt_dia, $dt_out, $dt_end, $init_out, $x_flag)
```

Alice: Ah, yes, of course. One more advantage of global variables. Once you have decided to go that dangerous path, you might as well enjoy it.

Okay, the logic is still crystal clear, so far. Let us start with the first command. How do you parse the option definitions?

3.5 Parsing Option Definitions: the Idea

Bob: Before showing you the method, let me first explain the idea. The full list with all the option blocks is contained in the single string `def_str`. What we would like to do is to cut up this list in two steps. The first logical step would be to divide the full string into shorter strings, one for each option. The second logical step would be to split each option string into lines, so that you can parse the meaning of each line.

Now a more practical approach would be to reverse the order. It is much easier to split the original `def_str` string immediately into individual lines. You can do that with the `split` method we just talked about: by default it cuts up a string wherever a blank space appears, but if you give it an argument, such as a newline `\n`, it will cut the string wherever in encounters the symbol specified in the argument.

In other words, the command

```
a = def_str.split("\n")
```

will produce an array of single lines, that together make up the original list of option blocks.

So now we have to go back to the step we skipped: we have to stitch the lines together that belong to a single option. To do this, we hand the whole array of lines to another method, which is so friendly as to take off enough lines from the array as are needed to reconstruct a single option block. That friendly method then passes back that single option, as a nice package, while leaving all the unrelated lines on the array of lines. After each call to this method, the line of arrays shrinks, until the whole array has been eaten up, and we are left with a stack of package, one for each option.

Now what I call a package is – you guessed it – an instance of a new class, called `Clop_Option`. It is a helper class, used by the `Clop` class, to wrap up all the information for a single option. The `Clop` class itself contains an array of instances of `Clop_Option`.

Alice: Just like an N-body system is represented by an instance of the class `Nbody`, which contains an array of instances of the `Body` class, one instance for each particle.

3.6 Parsing Option Definitions: the Method

Bob: Exactly. And here is the method that parses the option definitions.

```
def parse_option_definitions(def_str)
  a = def_str.split("\n")
  @options=[]
  while a[0]
```

```
      if a[0] =~ /^\s*$/
        a.shift
      else
        @options.push(Clop_Option.new(a))
      end
    end
  end
```

What I just described as the friendly method that wraps related lines into a single package is nothing else but . . . the initializer for the `Clop_Option` class! I use the same approach that we started with, one level lower. On the top level all the parsing work, for all options, was done as a side effect of creating a single `Clop` instance. On this level here, the parsing work for a single option is done as a side effect of creating an instance of the `Clop_Option` class.

Alice: All very clear. So you create an empty array of options, called `@options`, an instance variable within the `Clop` class. As long as there is anything left on the array of single lines, you traverse the `while` loop. Only when `a[0] = nil`, in other words when the array of lines has been picked empty of lines, and nothing is left anymore, do you end your work.

Now within the `while` loop, whenever you encounter a line that is completely blank, you discard it. That is what the lines

```
      if a[0] =~ /^\s*$/
        a.shift
```

mean, right?

Bob: Right. The regular expression indicates lines that contain zero or more blanks, between begin and end of a line. The symbol `\s` stands for any type of white space, such as a single space or a tab. The symbol ^ at the beginning of a regular expression `/^.../` means the beginning of a line, while the symbol `$` means the end of a line. The symbol `*` as usual means zero or more instances of the previous symbol, so `\s*` means any number of spaces or tabs, possibly zero.

Taken together, the regular expression `/^\s*$/` corresponds to any line that looks blank to the eye, whether it is a null string `""` or a string with a few blanks like `"  "` or a string containing tabs as well, like `" \t \t\t "`. Now whenever such a line is encountered, the array method `shift` is called in the second line above, which simply discards the first element of the array. As a result, the new element `a[0]` now contains what used to be stored in `a[1]`, `a[1]` contains what used to be in `a[2]`, and so on. The array consequently has become one element less in length.

Alice: And as soon as a non-blank line is encountered, you create a new instance of `Clop.Option`.

Bob: Yes, and I give the line array `a` as an argument to the of `Clop.Option`. This is the friendly function that gobbles up as many lines as needed to complete a well wrapped single option.

Alice: Ah, this means that it stops when it encounters two blank lines.

Bob: Yes, since we had agreed that that would be the sign that would separate two different option blocks. But the `Clop.Option` initializer is even friendlier than that: it also stops when there is something wrong with the syntax of the option that it is trying to wrap up. It doesn't just wrap any random bunch of double-blank-line-separated stuff.

So we can be assured that when `Clop_Option.new` returns, we have a valid new option package, in the form of a new instance of the `Clop.Option` class, and we can safely add that to the array of options called `@options`, using the command

```
        @options.push(Clop_Option.new(a))
```

Alice: Okay, I get it! In a moment, I would like to see how `Clop.Option` does its work, but for now, let us assume it knows what it is doing, and let us look at the second action that the initializer for `Clop` itself is performing.

3.7 Parsing Command Line Options: the Idea

Bob: Again, let me lay out the logic first. After the definitions of all options have been read in and parsed, it is time to see which options the user has actually specified, and to take the corresponding actions, such as modifying the default values of the appropriate global variables, or providing help of one type or another, as the case may be.

So there are two steps to the process of parsing the command line options: first make an inventory of options specified, and then take the appropriate actions. If a help request is encountered in the first step, the second step consists of printing out the corresponding help message(s). If no help is requested, the second step consists of initializing the proper global variables.

The first step is carried out in a loop. At the beginning of the loop, the first element of the `ARGV` array is examined. Depending on the option found, the correct action is taken. For example, if an option is found that does not require a value, this option is assumed to be a boolean variable, in other words a flag. Such a flag is by default set to be `false`, but when the option is encountered, the value of the flag is set to be `true`. If an option does require a value, another element is taken from the `ARGV` array, and properly interpreted.

This last element can be a bit complex, since some values may be spread over several elements of the `ARGV` array. For example, if a vector is specified, through `-v [ 1, 2, 3 ]`, several elements from the array have to be parsed until the closing `]` symbol is encountered.

Alice: But you could have required the reader to put the whole vector into a string, as follows: `-v "[ 1, 2, 3 ]"`.

Bob: Yes, and that is also a legal option. However, I wanted to make the parser really general, and I also wanted to free the reader from thinking about such aspects as how the command line would be parsed. In the spirit of Ruby, I prefer to download as much of the complexity of the interface to the code behind the interface, keeping the interface itself as natural as can be. Rather than training the user to add those double quotes, I would rather train the computer to figure out what to do even without quotes.

Alice: And as long as you insist that every vector starts with an opening square bracket and ends with a closed square bracket, there is no ambiguity.

Bob: Exactly. Ambiguity would be impossible to correct, of course. But as long as everything is unambiguous, I prefer the parser to do the hard work.

Now all of what I have just mentioned is still part of step one. Step two is more straightforward: You just ask each option to initialize its own global variable. And here you don't care whether such an option still has its default value, or whether that value has been modified through a command line option that was just read in.

Alice: Okay, got it! Let me see how you coded this.

3.8 Parsing Command Line Options: the Method

Bob: Here is the actual method:

```
def parse_command_line_options(argv_array)
  while s = argv_array.shift
    if s == "-h"
      parse_help(argv_array, false)
      exit
    elsif s == "--help"
      parse_help(argv_array, true)
      exit
    elsif i = find_option(s)
      parse_option(i, s, argv_array)
    else
      raise "\n  option \"#{s}\" not recognized; try \"-h\" or \"--help\"\n"
    end
  end
  initialize_global_variables
end
```

As long as there is something left in the array that contains all the command line bits and pieces, you take the next piece, call it `s`, and inspect that string. Now there are four possibilities. It could be a request for short help, in the form of a `-h` string; or it could be a request for long help, in the form of a `--help` string; or it could be the beginning of a regular option; or none of these three. In the last case, an error is reported, and the program is halted. The command `raise` prints the string that follows it, and stops execution of the code.

Alice: The call to `find_option` takes only one argument, while `parse_option` takes three arguments. Why is that?

Bob: The string `s` should contain one or two hyphens, followed by the name of the option, and that unique name is enough to determine which option we are dealing with. Therefore `find_option` takes only one argument, namely `s`, and returns the number of the option, `i`, which is simply the index of the option in the array of options. Remember that `Clop` has an instance variable `@options` for the option array, and the number `i` just means that we are dealing with option `@options[i]`.

However, knowing which option we have just encountered is not enough to completely parse the information for that option. In general, the next element in the `ARGV` array will contain the new value for that option. And as I just mentioned, in the case of a vector value, that value may be distributed over an unknown number of further `ARGV` array elements. Therefore the call to `parse_option` needs to receive both `i` and `argv_array`.

Alice: Yes, but why do you give it `s` as well? Haven't you squeezed all the information out of it by finding out which option it refers to? If `s = "-d"` or `s = "--step_size"`, there is no need to pass that string `s` on to `parse_option`.

Bob: Ah, you are completely right in those both cases. But there are other cases!

Alice: I can see that you are proud of having find a clever solution for something. But for what? There are only two cases for *any* option; either it is a one-letter option, starting with a single hyphen; or it is a multi-letter option, starting with a two hyphens

Bob: Right.

Alice: Right? So, then why pass it on?

Bob: Imagine that a user wants to set up a three-body system, and tries to give that option as `-n3` . . .

Alice: . . . instead of the more proper `-n 3`. I see. Yes, that makes sense. I like that! It is another example where you could have trained the user to always leave a blank space between an option and a value, but why do that? Better let the computer figure it out. And in that case, of course `parse_option` needs to have access to the string `s`, just in case not all the information has been squeezed out of it. It may still contain the value of the option.

Bob: Right! Of course, this only applies to one-letter options. In this case, too, we cannot allow ambiguity. An option specification like `-n3` is unambiguous, but writing `--number_of_particles3` would be confusion. It could refer to a boolean flag with a name `number_of_particles3`. An unlikely name in this case, but there are other option names that could naturally take a number, such as `--high5` or `--loveU2`, which may or may not be defined as boolean. So I only allow leaving out a space in the case of one-letter options.

Alice: And finally you initialize all global variables through a call to `initialize_global_variables`. No arguments needed, since the variables are global, and we deal with all of them. I like the long names you have chosen for your methods. That really helps in following the flow of the logic!

3.9 Printing Values

Bob: Thanks! Now let us go back to the initializer for the `Clop` class, where all the action started. Let me show it again:

```
  def initialize(def_str, argv_array = nil)
    parse_option_definitions(def_str)
    if argv_array
      parse_command_line_options(argv_array)
    end
    print_values
  end
```

We have now seen, in outline, how the options definitions are parsed, until the definition string `def_str` has been eaten up, and how the command line options are parsed, until the array containing command line fragments, `argv_array`, has been digested. All that is left to do at that point is to print the values and, you guessed it, that is done with the method `print_values`:

```
  def print_values
    @options.each{|x| STDERR.print x.to_s}
  end
```

You see, this is a very simple method: it just gives an order to each option to print its own value. Remember, we want the output of each program to start with a list of values used, to remind the user what the initial state is that the program starts out with.

Alice: And the actual work is done through `print_value`, which must be a method associated with the `Clop_option` class.

Bob: Exactly. It is time that we look at that class as well. Here we have reached the end of the top level tour.

Alice: Thank you! Now I see clearly how you have laid out the program. Indeed: time to open some of the black boxes that you have mentioned so far.

Bob: Yes, these boxes were left in the dark so far. Now let there be light!

Chapter 4

The First Journey: Clop, the Non-help Part

4.1 Three Journeys

Alice: I have enjoyed getting a bird's eye view of your clop.rb file. Let's get a little closer to the ground now. Where shall we swoop down?

Bob: I suggest that we continue our tour on the level of the Clop class, before descending all the way to the internal workings of the individual options, the machinery of which is contained in the Clop_option class.

However, more than halve of the Clop class code lines are dedicated to the help facility. It is not necessary to look at these lines in order to understand how normal options are being parsed. So I suggest that we continue our tour in three easy journeys. First we inspect how a normal option is handled on the Clop level. Second, we descend to the Clop_option level, to see how the corresponding option block is parsed and used. Third, we go back to the Clop level in order to figure out how the help facilities works.

Alice: Sounds good to me!

Bob: The first journey is by far the simplest, and shortest. Of the three actions ordered in the Clop initializer:

```
  def initialize(def_str, argv_array = nil)
    parse_option_definitions(def_str)
    if argv_array
      parse_command_line_options(argv_array)
    end
    print_values
  end
```

we have already seen how the first action parse_option_definitions consisted in handing all the work to the initializer one level lower, through a call to Clop_Option.new. So that part will be visited in our second journey.

Similarly, we have seen that the request for the third action also was handed down directly to the individual options on the Clop_Option level. All we have to do in our first journey is to figure out how the method parse_command_line_options works.

4.2 Inspecting find_option

Alice: Can you show me this method again?

Bob: Here it is:

```
def parse_command_line_options(argv_array)
  while s = argv_array.shift
    if s == "-h"
      parse_help(argv_array, false)
      exit
    elsif s == "--help"
      parse_help(argv_array, true)
      exit
    elsif i = find_option(s)
      parse_option(i, s, argv_array)
    else
      raise "\n  option \"#{s}\" not recognized; try \"-h\" or \"--help\"\n"
    end
  end
  initialize_global_variables
end
```

The first two if and elsif branches concern the help facility, which we will address in our third journey. So we only have to inspect the following three methods here, during our first journey: find_option and parse_option and initialize_global_variables.

Here is the first one:

```
def find_option(s)
  i = nil
  @options.each_index do |x|
    i = x if s == @options[x].longname
```

```
      if @options[x].shortname
        i = x if s =~ Regexp.new(@options[x].shortname) and $‘ == ""
      end
    end
    return i
  end
```

Alice: The top part is clear. You hand it a string that contains something like `"-d"` or `"--step_size"`. I presume that the option class `Clop_option` has a method `longname` that returns exactly the string `"--step_size"` and a method `shortname` that similarly returns `"-d"`.

Bob: Well presumed!

Alice: Now if the option is recognized as the long name version of option `i` in the option array, the value `i` is returned, as it should be. But what happens with the short name?

Ah, wait, before you answer my question, let me think. This must be connected with the fact that you allow for short options to be glued to their values. For example `"-d0.001"` would be a valid format.

Bob: Indeed, even though a user would not be likely to write it that way, since it does look a bit confusing. However, if we allow `"-n3"`, we should allow `"-d0.001"` as well.

Alice: Agreed. So I understand that you want to check only whether the `-d` part is present in the string `s`, while that string is allowed to contain more. Now you do that by turning the `shortname` of the option into a regular expression.

Bob: Yes: if you want to compare two strings, the proper and clean way to do so in Ruby is to change the string at the right-hand side into a regular expression. This is like converting a integer into a floating point number. In a way, nothing changes, except that now it has become an instance of a different class. For the number, an `Int` instance has become a `Float` instance, and here in our case, a `String` instance has become a `Regexp` instance.

Alice: and the comparison operator `=`∼ returns `true` if `@options[x].shortname` is indeed contained in the string `s`.

Bob: Yes, except that it returns the position of the first character of the match, rather than `true`. But what concerns us here is that it does *not* return `nil`, which would be interpreted sa `false`; anything that is not `nil` or `false` is considered to be `true`. Even the null string `""` is `true` in Ruby, another thing to watch out for if you are a C programmer.

Alice: And a more logical use of the notion of `true`, if you ask me. A non-null string string is still more than nothing.

Bob: Yes, I agree, though it took me a while to shake the C habit.

4.3 The Last Cryptic Bit

Alice: Now I think I understand all about this `find_option` method, except for that last cryptic bit, `and $‘ == ""`. What is that doing there? And what *does* it do?

Bob: Ah, that is a nice addition, if I may say so myself. At first I had not put that in, but when I looked at this method, without that addition, I had the feeling that something wasn’t right. When I thought about it, I realized that there was still a possibility for ambiguity.

Alice: Like?

Bob: Like having an option with a long form `--number_of_particles` and a short form `-n`. Can you see what would happen in that case?

Alice: Let me inspect. Ah! Yes, of course. In the case of the long form, you still match correctly against `-n`, as the second and third character of the long form. How devious!

But wait a minute. If you first check the long form, you could bypass the check for the short form, by turning the two `if` statements into an `if...else` statement.

Bob: Yes, that would work in the specific case I just mentioned, where there is only a confusion between the two ways of writing the *same* option. But what if there is a possible confusion between two different options?

Here is an example. Let there be another option with a long form called `--neutron_star_type`. Now that option, too, matches `-n`. So we have to protect different options from each other, and we cannot assume safety just by shadowing the short option check by the long option check.

Alice: You are right! But I still don’t understand the syntax of your solution. I would have checked whether the match started at the beginning. Didn’t you say that the match attempt returns the position of the first character of a successful match?

Bob: Indeed. And you are right. I could have written

```
        i = x if (s =~ Regexp.new(@options[x].shortname)) == 0
```

However, I preferred to use the `$‘` variable. After every successful match, the matched part of the string is assigned to the variable `$&`, while the part of the string before the match is assigned to `$‘` and the part of the string following the match to `$’`. So I just checked whether `$‘` was equal to the empty string:

```
        i = x if s =~ Regexp.new(@options[x].shortname) and $‘ == ""
```

Alice: I see. That is good to know. I guess those rather cryptic shorthands are borrowed from Perl.

Bob: I think so.

Alice: Okay, I now fully understand how `find_option`. On to the next station of our first journey!

4.4 Inspecting `parse_option`

Bob: Here is the next station. After we know which option we are dealing with, we have to parse it. This happens in the following method:

```
def parse_option(i, s, argv_array)
  if @options[i].type == "bool"
    @options[i].valuestring = "true"
    return
  end
  if s =~ /^-[^-]/ and (value = $') =~ /\w/
    @options[i].valuestring = value
  else
    unless @options[i].valuestring = argv_array.shift
      raise "\n  option \"#{s}\" requires a value, but no value given;\n" +
            "  option description: #{@options[i].description}\n"
    end
  end
  if @options[i].type =~ /^float\s*vector$/
    while (@options[i].valuestring !~ /\]/)
      @options[i].valuestring += " " + argv_array.shift
    end
  end
end
```

Now this is a bit more complicated, since there are several forks in the road. The first fork is related to the question: is the type of the option boolean? In other words: are we dealing with a flag? A flag can only be `true` or `false`. By name the flag as a command line option, the user intends to set the flag, *i.e.*, to the value `true`. By leaving out that option, the user intends to keep the default value `false`.

For example, in our N-body code, the user can ask for extra diagnostics by including the option `-x`, which leads to the corresponding global variable `$x_flag` as we have specified already. By default `$x_flag = false`. If the option `-x` is encountered, we have to change this variable to `$x_flag = true`.

This happens by setting the `valuestring` of the boolean option to `true` as you can see at the beginning of the code fragment above.

Alice: This valuestring is probably implemented as a string @valuestring within the Clop_option class, and there that string is used later to obtain the actual value?

Bob: All correct, as we will see during our second journey, but you don't have to rely on that, on this level: it could have been implemented in a different way, as far as the Clop class is concerned. The only important thing is that there is a 'setter' method provided for the Clop_option class, that somehow sets the internal information of the Clop_option instance in such a way as to guarantee that the boolean value of the option, when asked for later, will return true.

Hmm, that sounded more complicated than it really is. Often things are much clearer on the code level than when you try to express it in words.

Alice: The same is true in mathematical equations, of course, once you understand all the symbols . . .

Bob: . . . and once you are sufficiently familiar with manipulating the symbols that they are becoming old friends.

Alice: Yes, until that point it is still helpful to have clumsy sentences in a natural language to help you get the idea. So, please continue to be clumsy, and tell me what happens next. We have encountered a fork in the road. It the option is boolean, we set it to true without needing to read anything more from the command line, and we happily return.

Bob: And if the option is not boolean, we take the other fork in the road, by continuing the travel through the method parse_option.

4.5 Extracting the Value: Normal Case

Alice: Ah, I see, if the type of the option is *not* boolean, you have to extract the value from the next little bit of command line information, by accessing arg_array. But wait a minute, I see two lines where you assign something to @options[i].valuestring, no, three lines; one at the very bottom too.

Ah, that last one deals with vectors, and you already explained that vectors are special, in that their value can be spread out over different bits of string in the command line. So let's leave that for later. But what about these two assignments of @options[i].valuestring right in the middle?

Bob: The main assignment, the one you should look at first, is this one:

```
unless @options[i].valuestring = argv_array.shift
  raise "\n  option \"#{s}\" requires a value, but no value given;\n" +
        "  option description: #{@options[i].description}\n"
```

In most cases, after encountering a new option name, you just read in the value corresponding to that option, as the next little string that came from the command line. If there is nothing left to be parsed on the command line, that just means that the user has forgotten to provide a value: an error message is printed, and execution of the code is halted.

Alice: But what happens if the user provides a next option, instead of the value for the previous option? Imagine that the user writes `-n -x`.

Bob: In that case, an attempt will be made to set the number of particles to `-x`, which will result in something silly. But hey, we can't protect the user from all possible errors! I don't know how to anticipate on this level what is and is not correct. Others, using this code in the future, will undoubtedly use it for more general purposes than I can currently envision, so I don't want to constrain too much what can and cannot be said.

Alice: Hmmm. You could at least insist that a valid number would be provided when the type of a variable is given as an `int` or `float`.

Bob: Perhaps. We could come back to those questions later, and try to make everything industrial-strength. For the time being, I'm happy if everything works under reasonably normal circumstances with reasonably intelligent users.

Alice: Well, if you talk about users that don't make errors, then I have to conclude that *nobody* fits the criterion of being 'reasonably intelligent'. But okay, for now let's move on. I'd probably want to come back to this point later, though.

4.6 Extracting the Value: Compact Case

Bob: Now if you look just above the two lines I quoted above, you find:

```
if s =~ /^-[^-]/ and (value = $') =~ /\w/
  @options[i].valuestring = value
```

This addresses the case where a one-character option is used, without any space separating the option and the value, as in `-n3`, a very compact notation which we already discussed before.

Alice: What is the meaning of this funny looking repetition of the symbols `^-`? They occur twice, with a square bracket in between, and a closing bracket at the end, as `^-[^-]`.

Bob: This is one of the *most* confusing aspects in the notation of regular expressions, this overloading of the meaning of the up-arrow `^`. In fact, the two up-arrows here are two *completely* different things. In order to see this, let us inspect the whole regular expression:

```
/^-[^-]/
```

The first ^ specifies the beginning of the string. The presence of - immediately following means that the string has to start with a - sign. Now the square brackets are normally used to give you a choice, as in [aei] or [a-f]. In [aei] it is understood that any of the three letters a or e or i could be present and still form a match. And in [a-f], any letter in the range a, b, c, . . . , f would form a valid match.

Alice: Yes, that notation I am familiar with. But how can you start at the beginning of a line for the second time.

Bob: You don't. Within square brackets, the up-arrow ^ has the effect of negating the meaning of the next character. So the combination [^-] simply means: any character *but* the - character!

In other words, by writing

```
  if s =~ /^-[^-]/
```

we ask whether it is true that the string s begins with a hyphen, but does not begin with two consecutive hyphens. Let me show you:

```
|gravity> irb
irb(main):001:0> "-n" =~ /^-[^-]/
=> 0
irb(main):002:0> "--nono" =~ /^-[^-]/
=> nil
```

Alice: Ah, very nice, though difficult to parse for a human like me.

Bob: You'll get used to it.

4.7 Interesting or Confusing?

Alice: Now that I understand the first half of the first line, let me stare at both lines again:

```
    if s =~ /^-[^-]/ and (value = $') =~ /\w/
      @options[i].valuestring = value
```

You have told me that the variable $' contains the rest of the string, the part after the part which matched. So if we start with the option "-n", and if we insist that it should start with one and only one hyphen, then $' = n, right?

Bob: Wrong.

Alice: Huh?

Bob: Try it!

Alice: Okay:

```
|gravity> irb
irb(main):001:0> s = "-n"
=> "-n"
irb(main):002:0> s =~ /^-[^-]/
=> 0
irb(main):003:0> $'
=> ""
```

Hey, that is strange! Why should it be the empty string? What happened to `n` ?

Bob: Why don't you try the compact option-value notation `-n3`

Alice: Here goes:

```
irb(main):004:0> s = "-n3"
=> "-n3"
irb(main):005:0> s =~ /^-[^-]/
=> 0
irb(main):006:0> $'
=> "3"
```

Somehow the `n` gets eaten up and disappears without a trace, but the `3` survives.

Bob: What happened is that the matching attempt `s =~ /^-[^-]/` involves *two* characters: first the hyphen and then the next character, for which it is checked that it is not a hyphen.

Alice: Ah, although in plain English we can describe this match as 'a check that there is one and only one hyphen', in fact it is a match where the first two characters are being checked as being an ordered pair 'hyphen followed by non-hyphen.'

Now I see what happened. And since this all happens in the case of a one-character option, the non-hyphen that gets eaten is the option character, so that what is left is exactly the value that needs to be assigned to the variable corresponding to the option.

So what you do at the end of this complicated line, is that you check whether the remainder, stored in `$'` contains at least one alphanumeric character or underscore, which is what the `\w` stands for.

Bob: Exactly.

Alice: Okay, I see now what happens. But I think you could have written this in a simpler way.

Bob: How?

Alice: Instead of

```
    if s =~ /^-[^-]/ and (value = $') =~ /\w/
```

you could have used

```
    if s =~ /^-\w/ and (value = $') =~ /\w/
```

Bob: Ah, I had not thought about that. I guess I was just to fixated on hyphenation! But, now that I figured out how to do it, I find my double hat trick, or double up arrow if you like, quite elegant. Or at least interesting.

Alice: I just find it confusing, rather than interesting, but to each his own taste! Let's move on to the last case, at the end.

4.8 Extracting the Value: Vector Case

Bob: This is a lot simpler. Here we are dealing with the case that the option type is that of a `float vector`, a vector of the type we have defined before, with components that are all floating point numbers. As I already mentioned, a vector on the command line should be given in Ruby array notation, with the numbers enclosed between square brackets, `[]`.

There is a lot of freedom for the user: the vector can be written as a string, like `"[3, 5]"`, or without those double quotes directly as `[3, 5]`. The numbers can be comma separated, but they can also just be space separated, as in `[3 5]`. Spaces are allowed next to the brackets: `[ 3 5]` and `[3 5 ]` and `[ 3 5 ]` are all equally fine.

There is one catch to be aware of, when you leave of the double quotes: on the command line `[ 3,5]` and `[3, 5]` and `[3,5 ]` are all fine, but `[3,5]` is likely to give you an error message.

Alice: Why?

Bob: It depends on the Unix shell you use, but chances are that the shell tries to interpret this as an attempt to address files in the current directory. Unless you happen to have a file with the name `3` or a file with the name `5`, and expression on the command line containing `[3,5]` will probably generated a short dry message `No match`.

Alice: That's good: short and simple, and it makes it clear that there is no subtle Ruby bug involved.

As for your implementation, let me look at what you wrote for vector parsing:

```
      if @options[i].type =~ /^float\s*vector$/
        while (@options[i].valuestring !~ /\]/)
          @options[i].valuestring += " " + argv_array.shift
```

You allow some flexibility in writing the type: it could be `float vector` or `float vector` or even `float vector`.

Bob: Sure, it would seem to restricted to insist on one literal way of writing it. I can easily see someone adding an extra space between the two words, and perhaps a tab or whatever would strike them as looking better. I have consistently given the users that freedom, also in parsing the lines within the `Clop_Option` class, as we will see in our next journey.

Alice: And then you keep shifting new content from the `ARGV` array until you finally encounter a string that contains a closing square bracket `]`. During that whole process, you keep adding what you find to the `valuestring` of the option you are working with, so that you build up the whole vector again, from the bits and pieces from the command line that were stored in successive elements of the `ARGV` array, here called `argv_array`.

One last question: why don't you just string those strings together? What is the need for adding a " " between the bits and pieces?

Bob: If all the vector elements were comma separated, as in `[2,3,4]`, there would be no need to do so. However, I give the user the flexibility to use a space separated notation as well. Take the example of a vector written as `[2 3]`. In the `ARGV` array, this will be distributed over two elements, the first being `"[2"` and the second one `"3]"`. Now if you would just string those two strings together, as you suggested, you would get `"[23]"`, a one-dimensional vector with one element, `23`. Not what you wanted.

Alice: I see. Good! Now I believe there is one station left on our first journey?

4.9 Inspecting `initialize_global_variables`

Bob: Indeed. Here is the last method:

```
  def initialize_global_variables
    @options.each{|x| x.initialize_global_variable}
    check_required_options
  end
```

I am asking each option to do its own work, initializing the internal variables that contain the external information presented on the command line. For example, the time step value may have been presented on the command line as `-d 0.001`, and in order to give that value to the method `evolve` that we have used to integrate an N-body system, we have to store it somewhere in a variable, which we would normally called `dt` or something like that.

As we have seen in our new driver, for convenience we chose to use global variables, which means the name actually has to be something like `$dt`. Now the conversion from the string `-d 0.001` to the actual floating point numerical value `$dt = 0.001` is being done within the `Clop_Option` class, by its method `initialize_global_variable`.

Alice: And then you check something about the options. What do you mean with 'required'?

Bob: For many options, a default value is specified, in the original definition string. However, for some options there is no natural default. For example, you may not want to specify a default value for an output file. Not only is there no obviously appropriate name for such a file, you don't want to risk overwriting another file that the user may have added to the current directory. In that case, you could start the entry for the output file option in the original here document with:

```
Short name:          -o
Long name:           --output_file_name
Value type:          string
Default value:       none
```

The convention used here is that an error message will be generated if the user does not provide a specific name for the output file on the command line. And the method `check_required_options` checks that all the original `none` specifications have indeed been overwritten by values provided on the command line.

Alice: Instead of `check_whether_all_required_options_have_been_provided`, you abbreviated the method name. Even for me an eight-word name would have been too long. Can you show me the code?

Bob: Here it is, all very straightforward.

```
  def check_required_options
    options_missing = 0
    @options.each do |x|
      if x.valuestring == "none"
        options_missing += 1
        STDERR.print "option "
        STDERR.print "\"#{x.shortname}\" or " if x.shortname
        STDERR.print "\"#{x.longname}\" required.  "
        STDERR.print "Description:\n#{x.longdescription}\n"
      end
    end
    if options_missing > 0
      STDERR.print "Please provide the required command line option"
      STDERR.print "s" if options_missing > 1
      STDERR.print ".\n"
      exit(1)
    end
  end
```

Alice: All clear! I think this finishes our first journey?

Bob: Yes, time to finally descend into the `Clop_Option` class.

Chapter 5

The Second Journey: Clop_option

5.1 Code Listing

Alice: Time to open the black box that contains the helper class that does all the work behind the scenes, for each individual option. Can you show me the whole class definition, so that I get an idea of what it looks like, before we inspect each method?

Bob: Here you are:

```
class Clop_Option

  attr_reader :shortname, :longname, :type,
              :description, :longdescription, :printname, :defaultvalue
  attr_accessor :valuestring

  def initialize(def_str)
    parse_option_definition(def_str)
  end

  def parse_option_definition(def_str)
    while s = def_str.shift
      break if parse_single_lines_done?(s)
    end
    while s = def_str.shift
      break if s =~ /^\s*$/ and  def_str[0] =~ /^\s*$/
      @longdescription += s + "\n"
    end
```

```
end

def parse_single_lines_done?(s)
  if s !~ /\s*(\w.*?)\s*\:/
    raise "\n  option definition line has wrong format:\n==> #{s} <==\n"
  end
  name = $1
  content = $'
  case name
    when /Short\s+(N|n)ame/
      @shortname = content.split[0]
    when /Long\s+(N|n)ame/
      @longname = content.split[0]
    when /Value\s+(T|t)ype/
      @type = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
      @valuestring = "false" if @type == "bool"
    when /Default\s+(V|v)alue/
      @defaultvalue = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
      @valuestring = @defaultvalue
    when /Global\s+(V|v)ariable/
      @globalname = content.split[0]
    when /Print\s+(N|n)ame/
      @printname = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
    when /Description/
      @description = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
    when /Long\s+(D|d)escription/
      @longdescription = ""
      return true
    else
      raise "\n  option definition line unrecognized:\n==> #{s} <==\n"
  end
  return false
end

def initialize_global_variable
  eval("$#{@globalname} = eval_value") if @globalname
end

def eval_value
  case @type
    when "bool"
      eval(@valuestring)
    when "string"
      @valuestring
    when "int"
      @valuestring.to_i
      when "float"
        @valuestring.to_f
      when /^float\s*vector$/
        @valuestring.gsub(/[\[,\]]/," ").split.map{|x| x.to_f}.to_v
      else
        raise "\n  type \"#{@type}\" is not recognized"
    end
  end

  def add_tabs(s, reference_size, n)
    (1..n).each{|i| s += "\t" if reference_size < 8*i}
    return s
  end

  def to_s
    if @type == nil
      s = @description + "\n"
    elsif @type == "bool"
      if eval(@valuestring)
        s = @description + "\n"
      else
        s = ""
      end
    else
      s = @description
      s = add_tabs(s, s.size, 4)
      s += ": "
      if @printname
        s += @printname
      else
        s += @globalname
      end
      s += " = " unless @printname == ""
      s += "\n  " if @type =~ /^float\s*vector$/
      s += "#{eval("$#{@globalname}")}\n"
    end
    return s
  end

end
```

5.2 Parsing An Option Definition

Alice: That's not as long as I thought it would be.

Bob: One of the great things of Ruby: because the notation is so compact, and because you don't have to worry about types and declarations and all that sort of stuff, you can write quite powerful codes in just a few pages.

Alice: Let's step through the `Clop_option` class. The initializer starts off just like it did on the higher `Clop` class level. In that case, the first line was:

```
    parse_option_definitions(def_str)
```

while here we have only one line, the same apart from the final "s":

```
  def initialize(def_str)
    parse_option_definition(def_str)
  end
```

And that makes sense, since by definition this helper class takes care of only one option at a time.

Bob: The next method shows how the parsing gets started:

```
  def parse_option_definition(def_str)
    while s = def_str.shift
      break if parse_single_lines_done?(s)
    end
    while s = def_str.shift
      break if s =~ /^\s*$/ and  def_str[0] =~ /^\s*$/
      @longdescription += s + "\n"
    end
  end
```

Remember how we wrote the definition of an option block: first we write one line for each piece of information, such as

```
 Short name:           -o
```

or

```
 Value type:           string
```

Only at the end do we allow an arbitrarily long multi-line description of what the option is all about. That was the line called `Long Description`. It contains the information that will be echoed when we ask for `--help` on the command line.

This means that the parsing process is somewhat different from the single-line instructions and for the last multi-line block. The method `parse_single_lines_done?`

takes care of the single lines, while the last few lines of the parse_option_definition method take care of the multi-line block.

Of course, I could have written a separated parse_multiple_lines method, but that seemed to be a bit of overkill, given that the work can be specified in just a few lines:

```
     while s = def_str.shift
       break if s =~ /^\s*$/ and  def_str[0] =~ /^\s*$/
       @longdescription += s + "\n"
     end
```

You just keep taking lines off from the def_str that contained the whole here document, and when you encounter two successive blank lines, you stop. Remember, we had agreed that two blank lines would signal the start of a new option block.

5.3 Are we Done Yet?

Alice: So all the rest of the parsing is done in the method parse_single_lines_done?. Why the question mark at the end of the name?

Bob: This is a nice feature of Ruby, that it allows you to add a question mark or exclamation mark at the end of the name. You can't use it as a general character in the middle of a name; it can only appear at the end. Its use is to communicate to the human reader something of the intention of the program: in this case, you might guess that a boolean value is being returned by this method. If the value is true, then indeed we are done parsing the single lines. If the value is false, we aren't done yet.

Alice: I like that, that does make the intention clearer.

Bob: Here is the method:

```
   def parse_single_lines_done?(s)
     if s !~ /\s*(\w.*?)\s*\:/
       raise "\n  option definition line has wrong format:\n==> #{s} <==\n"
     end
     name = $1
     content = $'
     case name
       when /Short\s+(N|n)ame/
         @shortname = content.split[0]
       when /Long\s+(N|n)ame/
         @longname = content.split[0]
       when /Value\s+(T|t)ype/
```

```
        @type = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
        @valuestring = "false" if @type == "bool"
      when /Default\s+(V|v)alue/
        @defaultvalue = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
        @valuestring = @defaultvalue
      when /Global\s+(V|v)ariable/
        @globalname = content.split[0]
      when /Print\s+(N|n)ame/
        @printname = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
      when /Description/
        @description = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
      when /Long\s+(D|d)escription/
        @longdescription = ""
        return true
      else
        raise "\n  option definition line unrecognized:\n==> #{s} <==\n"
    end
    return false
  end
```

5.4 A Non-Greedy Wild Card

Alice: I see that you start off with another exercise in regular expressions, but this one puzzles me:

```
    if s !~ /\s*(\w.*?)\s*\:/
```

Why do you add a ? after an * ? That seems to be redundant. The * tells you to expect zero or more instances of the previous character, while the ? tells you to expect just zero or one instances. No, I take that back, it is not even redundant, it seems wrong, since ? would be expected to follow the previous character, and here the * is in the way.

Bob: You should consider the combination of the two characters as one unit: *? is defined as a 'non-greedy' version of the * wild-card character.

Alice: Non-greedy?

Bob: Yes. Normally, the wild card notation is interpreted in a 'greedy' way: it gobbles up as much as it can.

Alice: I would call it a 'hungry' way in that case. Can you give me a simple example of the difference?

Bob: Sure. Let's use our friend irb again:

```
  |gravity> irb
```

```
irb(main):001:0> s = "abc:def:xyz"
=> "abc:def:xyz"
irb(main):002:0> s =~ /.*:/ ; $'
=> "xyz"
irb(main):003:0> s =~ /.*?:/ ; $'
=> "def:xyz"
```

In the first regular expression, I ask for a match with an arbitrary number of characters of any type, followed by a colon. The period can stand for any character except a new line. As you can see, after the match, what is left over is `"xyz"` so the match went all the way to the second colon.

Now in the second regular expression I have added the question mark to make the match non-greedy. In this case, the string `"def:xyz"` remains, which means that the match only included `"abc:"`. This was the first match that satisfied the minimal requirement of having an arbitrary number of characters ending with a colon. Our non-greedy operator `*?` was satisfied at this point, while its greedy colleague `*` kept looking for a longer match, and indeed found one.

Alice: Very nice to have the option to stop early. And in our case, this means that you are allowed to include colons in the definitions, without confusing the parser, right?

Bob: Indeed. Every line among the single-line definitions has the structure:

```
<name>    : <content>
```

I do not allow a colon ”`:`” to appear in the ‘name’ part of the line, but I do allow colons to appear in the ‘content’ part. This is yet another example of trying not to limit the user unnecessarily. An example I thought about is where someone might want to define a classification for stars, and for some reason decides that it is convenient to use colons. Options to assign stars of different classes could take on the form:

```
--star_type "star : MS"
```

for a main sequence star, or

```
--star_type "star : MS : ZAMS"
```

as a further specialization, to indicate a zero-age-main-sequence star. A giant on the asymptotic giant branch could be specified as:

```
--star_type "star: giant: AGB"
```

In all these cases, the non-greedy parser instruction will extract the content part of the line correctly.

Alice: I like the idea of keeping maximum flexibility for option specifications, rather than excluding characters like a colon. Good! And I see in the next line that you raise an error if you find no colon at all.

5.5 Extracting the Name from a Definition

Bob: Yes. And if a colon is found, everything before the first colon is assigned to the variable `name` and everything after that colon to the variable `content`.

Alice: I understand how `content` gets its content, so to speak, since `$'` is by definition what is left over after the match. And I also understand that `$&` would not have been a good choice for `name`, since it would have included the colon. I probably would have started with `$&` and stripped off the last character.

Bob: That would still not be right, since in most cases you would have wound up with a name that contained trailing spaces. You could have taken those off too, of course, but I found a quicker way to do everything at once. They key is given in the use of the parenthesis in the first line:

```
if s !~ /\s*(\w.*?)\s*\:/
```

Alice: That one line is a rich line indeed! What does `(\w.*?)` mean?

Bob: in general, parentheses in a regular expression can be used for two purposes: they allow you to group characters together and they also allow you to collect particular parts of the match results that you might be interested. An example of the first use is to write `/(na)*/`. This specifies that the group of letters `na` is to be repeated an arbitrary number of times. In a word like `banana`, it matches against the `nana` part. An example of the second use is what we see here in the code.

When parts of a regular expression are put within parentheses, the variable `$1` will be given the string that matches the content of the first set of parentheses, the variable `$2` will receive a string containing the content of the second parentheses delimited match, and so on. Here there is only one set of parentheses, enclosing whatever appears after initial white spaces, and before the first colon.

To be specific, a match against the `(\w.*?)` part requires there to be at least one alphanumeric character or underscore, corresponding to `\w`, followed by arbitrary characters. Since the `:` in the regular expression `/\s*(\w.*?)\:/` is placed outside the parentheses, the colon does not appear in the value of the variable `$1`, but everything else up to the colon does appear, apart from possible white space before the colon. Therefore, `$1` will contain the complete name, with any leading or trailing white space removed.

Actually, removing those leading and trailing white space characters was not really necessary, as you will see below, since we're only matching the 'name' part of the definitions against various possibilities, and those matches would work fine with blank space left in place. I just decided to be extra neat, for a change.

5.6 Extracting the Content from a Definition

Alice: Let me summarize the idea. Each option definition, apart from the exceptional Long Description lines, has the one-line structure:

```
<name>    : <content>
```

You have now successfully extracted the name from an individual line, and now you enter a `case` switch, in which you are going to check which name it is you have extracted, and depending on the name, you're going to do something with the corresponding content.

Bob: Precisely. Let us walk through the different possibilities. It helps to remind ourselves of the structure of a typical option block. Here is what we could have written for the step size specification:

```
Short name:          -d
Long name:           --step_size
Value type:          float
Default value:       0.001
Global variable:     dt
Print name:          delta t
Description:         Integration time step
Long description:
  In this code, the integration time step is held constant,
  and shared among all particles in the N-body system.
```

You can see in the listing of the method `parse_single_lines_done?` above that each of the one-line definitions is being treated in the correct way.

Alice: Let me check. For the long and short name, you allow both spellings, "name" and "Name", in the 'name' part of the definition. And since there are no blanks allowed in the name of the option, it is safest to split off only the first contiguous non-blank character set from the content string. That makes sense. And then you assign the actual name, in this case either `-d` or `--step_size` to an instance variable of the class `Clop_Option`: `@shortname` or `@longname`, respectively.

Bob: Yes. And I could have done a better job in checking for errors, but you have to stop somewhere. If someone would write a definition as:

```
Long name:           --step size
```

the results would be `@longname = --step`, and the string `"size"` would be discarded.

5.7 Two Types of Mistakes

Alice: I agree that there is no point to make things completely robust in an iron clad way at this point. Perhaps in a year or so, when we decide to use this program indefinitely, we can come back and make things more sturdy.

But wait a minute. Is that really correct what you just said? If someone would have typed `"--step size"` on the command line, then only the string `"--step"` would have been handed to our helper class `Clop_Option`, by the parser in class `Clop`, and there still would be no need to use the `split` method here since in this case the variable `content` would contain only the string `"--step"`, and `split` would not change anything.

Bob: All true, but I think you are confusing two different things. First we talked about the *writer* of a program writing an option block that contains a mistake, in the form of `--step size` as the choice of long name for an option. The mistake here is to leave a space between the two words, rather than an underscore. An underscore would have had the same effect of making things more readable, as compared to the simplest choice of `--stepsize`, but an underscore counts as a non-blank, so for Ruby `"step_size"` is still a single word, while `"step size"` would be considered to be two words.

Now there is a separate mistake that you brought up, where the *user* of a program would give a command line that includes, for example, `"--step size 0.01"`. Perhaps the user saw the option description of the writer, and followed it blindly, not realizing that it was faulty. Or perhaps the program would have a correct option definition, given as `"step_size"`, but the user overlooked the underscore. In either case, what will happen is that an instance of the class `Clop_Option` will be created, with a long option name `"step"`. Next, the string `"size"` will be parsed, as the next element in the `ARGV` array, and an attempt will be made to convert that to a floating point number.

Alice: And that will fail.

Bob: Not necessarily. Again, I could have checked whether a string has the correct format for a floating point number, but I don't think I've been quite so meticulous. However, the next step will definitely go wrong: even if somehow `"size"` is converted to some kind of number, and assigned to a variable associated with the option `"--step"`, then the parser of the `Clop` class will read in `"0.01"`, trying to make sense of that as an option. And of course, this is not a valid option. It does not even contain a hyphen.

Alice: What will go wrong in that case?

Bob: The method `find_option`, which we looked at in our first journey, would not find a match with any known option, and so it would return `nil`. As a result, the method `parse_command_line_options` would raise an error, and halt the program after printing the string

```
  "option "0.01" not recognized"
```

Alice: Good to know that such mistakes would be caught, and what is more, would lead to understandable error messages.

By now, I think I need a break. Something tells me this will be a long journey!

Bob: I'm afraid it will be, so perhaps we should split it up into subjourneys.

Chapter 6

More Parsing of Single Lines

6.1 Recognizing the Type

Alice: I'm ready to continue our exploration of the `parse_single_lines_done?`. So far we have only looked at the way it deals with short and long names.

Bob: Yes, let us continue our `case` study. Before getting into the third `when` statement, first a bit of background. So far we have extracted the essential part of the `content` string by splitting off its first word, using `content.split[0]`, but now we will encounter several cases where we have to be more careful than that.

If the 'name' part of a definition has the form `"Value Type"` or `"Value type"`, we have to do more than splitting off the first blank-separated substring, since some types can legally contain more than one word. A physical vector is represented in our programs as an object of class `Vector` with floating point numbers as elements. I reserved the expression `"float vector"` for this type, since in the future we might also want to deal with `"int vector"` or perhaps `"complex vector"` or even `"quaternion vector"` once we implement regularization techniques.

Alice: Let me try to understand the regular expression soup that is supposed to extract the multi-word type information here, in the line

```
      @type = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
```

You take the string `content`, and apply two substitutions to it. First you take one or more white spaces at the very beginning of the string, if you find them there, and replace them by the null string; in other words, you take off all leading white space.

But why did you write `/^\s+/` instead of `/^\s*/` ? If there would be no white space at all, you could still replace it by the null string. Replacing nothing by nothing would not hurt anyone, would it?

Bob: I suppose you are right. Yes, that must be true; either way would work.

Alice: Now for the second substitution, there you use parentheses again. You talked about two different uses, but this seems to be yet another use.

Bob: You are right, I probably should have mentioned this as a third way of using parenthesis. Here the vertical line indicates a choice: it is a type of logical 'or' operator. This regular expression matches in two different cases: when the expression in parentheses is replaced by what is written to the left of the vertical line, or when it is replaced by what is written to the right of that line.

Let us first look at the left side. This substitution would be equivalent to `sub(/\s*#.*/,"")`.

Alice: Ah, you strip off a Ruby comment, something that starts with a `#` sign, and is followed by whatever: an arbitrary number of arbitrary characters, indicated by "`.*`". In addition, you remove blank space leading up to the comment.

Bob: Indeed. Now the second possibility for a match is that the right side of the vertical bar springs into action, in which case the substitution would be equivalent to `sub(/\s*$/,"")`.

Alice: And that would only remove whatever blank space there may be just before the end of the string, indicated by `$`. Okay, got it!

Bob: Note that this third `when` statement does something more than assigning the content of the definition line to the instance variable `@type`. If the type happens to be `bool`, the following action is undertaken:

```
    @valuestring = "false" if @type == "bool"
```

The variable `@valuestring` will be given the string `"false"`. This guarantees that any logical flag will always be set to be false, unless the user specifically mentions the flag as an option, in which case the corresponding variable `@valuestring` will be set to be `"true"`, as we will see later.

Alice: So this frees the writer of the option block from having to add the line

```
  Default value:        false
```

In the definition of an option block with a boolean value.

Bob: Exactly. The writer can still add this, but it is not necessary. And since a flag as a command line option does not take a value on the command line, it would be easy to forget to include this line in the definition. Therefore I have automated the process here.

Alice: What do you mean with “a flag does not take a value on the command line”?

Bob: Remember the example where the user can ask for extra diagnostics, by adding `-x` to the command line? If we really wanted to treat it on the same level as the time step choice, say, where you would write `-d 0.001`, we should ask the user to write `-x true`. But that would be unnecessarily wordy. By mentioning `-x` the user already requests something extra to be done. Not mentioning this option would leave the value false, mentioning it would make it true. The advantage of a boolean variable is that there is no third option, so no need to specify anything!

Alice: Okay, that is clear, and yes, I am all for automatizing, whenever a situation is unambiguous.

6.2 Default Values

Bob: Now we can quickly walk through the remaining `when` statements. When we encounter a `Default Value` or `Default value`, written before the first colon, we assign the content corresponding to that name to the varialbe `@defaultvalue`, and then we immediately give that same value to `@valuestring`.

Alice: In that way, if no command line option for that variable is specified, the program will run with the default value.

Bob: Exactly. If the default time step would be `0.01`, and someone would give the command line option `0.001`, then the `@valuestring` would be changed from `"0.01"` to `"0.001"`. We will see below how that happens.

Alice: But why do you keep two instance variables here, both `@defaultvalue` and `@valuestring` ? You could just have used only one, `@valuestring`. If you give that string the default value at first, and then you allow it to be overridden, it will always have the intended value. What need is there to remember the original default value separately, in a variable called `@defaultvalue` ?

Bob: Originally I had only one value, just as you suggested, but then I realized that it would be nice to let the help facility tell you what the default values are.

Alice: But if you ask for help, by typing `ruby some_program.rb -h`, or by typing `ruby some_program.rb --help`, you don’t change any of the default values, so you could ask the help facility to echo whatever is stored in the `@valuestring`, can’t you? I still don’t see the need for a separate `@defaultvalue` variable.

Bob: You’re reproducing my stream of thoughts, when I wrote this. I had the same idea, initially, but then I realized that someone could type

```
|gravity> ruby some_program.rb -d 0.1 --help
```

The option parser would first encounter `-d 0.1` and set the `@valuestring` for the time step option to `"0.1"`, overriding the default value `"0.01"`.

Alice: But why would anybody want to do that?

Bob: Two answers. First, if something like this can happen, it will, and a good attitude in defensive programming is to be prepared for such eventualities.

Alice: If someone would be writing a book about our dialogues, that line would be put into my mouth! Since when are you relying on principles, such as defensive programming?

Bob: I just wanted to make sure you were still listening. Well, let me give you the more important second reason: it is the use of the 'bang bang' command in Unix.

Alice: You mean typing `!!` in order to repeat the previous command?

Bob: Yes. It is quite natural for someone to run a program first, and then to become curious about one of the options. The easiest thing to get help is then to a help request to the previous command. What I mean is:

```
|gravity> ruby some_program.rb -d 0.1 -q
clop.rb:148:in `parse_command_line_options':
  option "-q" not recognized (RuntimeError)
|gravity> !! -h
```

This is a much quicker way to get help than to retype the whole line:

```
|gravity> ruby some_program.rb -h
```

Alice: I see what you mean. In that case, the time step would have been reset already, from the default value to `0.1`. So therefore you want to store the default value in a separate memory place. That makes sense.

6.3 A Matter of Principles

Bob: Moving right along, assigning a global variable name to the option is what happens next.

Alice: So the variable `@globalname` will contain the name of the actual variable that will in turn contain the value of whatever quantity will be associated with the option.

Bob: Yes, I guess this is an example of what the redirection principle you like so much. It was quite natural to write it this way. After all, the option definitions we are dealing with here form a type of template. And here we are parsing the contents of the template. Everything is just one step further removed. Instead of a pair {variable, value}, we here have a triple {variable-name-holder, variable, value}. We'll come back to this when we will see how the variables get their values.

The need for this extra meta level can also be seen from the fact that we are not only dealing with a code user, giving command line options, and a code writer, giving the option definitions, but with a third person, me in fact, a kind of meta writer. What I just the writer of a code that incorporates this new option mechanism is a writer with respect to the user of that code, but is at the same time a user of this parsing code. A meta user if I am the meta writer.

Anyway, this gets too complicated. All I wanted to say is that it is unavoidable to have three levels instead of the usual variable-value. And three levels allow at least two such pairings, and therefore we need a form of redirection these pairings. So while I'm not a man of many principles, I do see the need for your redirection principle here.

Alice: You mean the *indirection* principle, as in indirect addressing. But you could call it the redirection principle I guess, since you are redirecting the flow of control from one name to another. But since I have never heard anyone talk about 'redirect addressing', I'll stick with indirection.

Bob: Sounds too close to indigestion. What would be the best name? Instead of giving fixed directions as to where to find the value of a variable, we are giving indirect directions in order to redirect the computer to look in a different place. But we are still giving directions, neither indirections nor redirections.

Alice: Enough of that – you are convincing me not to mention principles too often! But wait, this time you started it, talking about principles.

Bob: Okay, we'll move on. As we discussed before, every option has a special 'print name', that determines how the information will appear on the output. We gave the example of an N-body system, where you expect the particle number to be printed as `N = 100`, not as `n_part = 100` or something cryptic like that. In this case, `N` is the print name. And since it could have more than one word, as in `particle number`, we have to use the same substitution tricks as we have used before.

Alice: That double substitution line occurs four times, wouldn't it be nicer to write a one-line method for that?

Bob: That occurred to me, but I decided against it. I could have written

```
  def read_content(str)
    str.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
  end
```

and then the print name, for example, would be extracted as

```
        @printname = read_content(content)
```

But I concluded that the improvement in readability did not outweigh the extra complexity of introducing yet another method. Having to let your eyes jump around to too many places also decreases readability.

Alice: I agree, this is a tough call, either way would have been fine.

6.4 Descriptions

Bob: Finally, we read in the one-line short description, if the name field contains the word `Description`, and we start reading in the long description if the word field contains the words `Long Description`, or `Long description`.

Alice: Wait a minute, the beginning of a long description *also* contains the word `Description`, after the word `Long`. Would that not confuse your parser, who could mistakenly interpret it as a short description?

Bob: No, since a long description is only allowed at the end of an option block. This has to be the case, since the only way to find the end of a multi-line long description is to look for two consecutive blank lines. And when those lines are found, you are by definition at the end of the option. Therefore, the method `parse_single_lines_done?` will always first encounter the short one-line `Description` before it finds the multi-line `Long Description`.

Alice: If there is a short description, that is. If there would only be a long description, then `parse_single_lines_done?` would read the first line of it, and consider it to be a short description.

Bob: And then it would not recognize the next lines, which would form the actual long description itself, and it would raise an error, printing `option definition line unrecognized:\n==> (next line) <==\n`. So it would be clear that there would be something wrong.

However, there would be no point in writing *any* option block without a short description. An important part of our whole approach is to provide a good help facility. If you want to cut corners, you could imagine leaving out the long description, and only giving a short one-line description. But it wouldn't make sense to be lazy and not write a short description, and yet to go out of your way to write a long description.

Alice: Logically, yes, but I can easily imagine myself to make a mistake, and to just forget to include a short description.

Bob: Well, in that case you would get an error message, which would remind you to mend the error of your way.

Alice: Fair enough. Talking about the long description, if you find the words `Long description`, you just set the `@longdescription` variable to contain an empty string? Here is the action that follows the recognition of a `Long description` :

```
        @type = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
```

Bob: Yes, I create the empty string in order to have a place to start adding the multi-line actual description to, line by line. This happens in the previous method, `parse_option_definition`, remember? Here it is again:

```
  def parse_option_definition(def_str)
    while s = def_str.shift
      break if parse_single_lines_done?(s)
    end
    while s = def_str.shift
      break if s =~ /^\s*$/ and  def_str[0] =~ /^\s*$/
      @longdescription += s + "\n"
    end
  end
```

Note that the value `true` is being returned when a `Long description` is detected. This causes the first `while` loop in `parse_option_definition` to end, with the `break` statement. As a result, the second `while` loop is entered, which reads the lines of the long description, one by one, until two blank lines following each other are found.

6.5 Reflections on Ruby

Alice: I think I now understand how you parse the single lines. That was quite an adventure! Still, now that I have gone through it, I must say that it all looks quite straightforward.

Bob: It is the longest method in the file. If there would have been a natural way to split it up into smaller methods, I would have done so. But since we are naturally dealing with a long `case` statement, I did not see a good way to break this up.

Alice: Since when are you concerned with writing short and concise pieces of code? I thought you didn't particularly like the notion of trying to be modular. And now you talk like Mr. Modularity himself!

Bob: I know, I'm a bit surprised too, I must admit. But in Ruby, I somehow find myself writing methods that are much shorter than the subroutines I would write in Fortran, or functions in C. I wonder why that is.

Alice: I think the language invites you to think in a more structured way. In contrast, Fortran and C try to please the compiler, not the user, and as a result you get used to rather more complex ways of expressing yourself. In the process writing can easily run away to cover many dozens of lines.

Bob: And in addition, when you start a new subroutine or function, you have to be careful about type declarations and all that, which make you think twice whether you want to jump through those extra hoops. In Ruby, in contrast, it is the easiest thing in the world to split off a few lines and call it a method.

Alice: Yes, I've noticed that. And it helps not only the writer, but also the reader: if the names of those short methods are well chosen, they effectively serve as comments. You've done very well here, in choosing your names!

Bob: I must admit, it took me a while. When I first started writing all this, I had introduced different names, but half-way I found that I got quite confused myself about what was doing what, exactly. I found that by changing the names to something closer to what was being done in each method, the whole logic became much clearer.

Alice: Another way in which Ruby naturally invites better code writing. It would be impossible to know beforehand how each method should be named since you don't know what it is going to do until you are well underway with the prototyping process. By inviting you to write many small methods, Ruby also invites you to rename them appropriately.

In contrast, when everything is done within three of four humongous subroutines, you can get used to names like `firstblock` and `secondblock` and not even think about changing those later.

Bob: You mean `fstblk` and `scdblk`, if you really want to convey the flavor. Okay, enough meta-talk. Let's continue our second journey: we are well past half-way now, so let's finish the journey.

6.6 A Comic Book Code Line

Alice: Now the next method, short as it is, puzzles me a bit:

```
  def initialize_global_variable
    eval("$#{@globalname} = eval_value") if @globalname
  end
```

If this would be the first line I would ever see of Ruby, I would think it would be someone swearing, in a comic strip. Just to see a succession of these six characters, `("$#{@`, is quite something to behold!

Bob: I hadn't thought about that, but yes, that does like a bit funny, doesn't it? If there is ever going to be a contest for piling non-alphanumeric symbols on top of each other, this one may have a chance, though I bet that clever programmers would be able to come up with something much longer. Anyway, this method does what I have advertised: it goes from variable-name-holder to variable to value. Let us take it apart, and translate what we find.

If an option wants to initialize its global variable, say `dt` for the time step size, it first looks at the variable `@globalname` that contains the string with the name of the global variable, as extracted from the option block in the definition string. In this case, the string will hold `"dt"`. Note that this is a string, and not yet a variable. It is what we called the variable-name-holder before.

As usual, `#{@globalname}` evaluates `@globalname` and substitutes the result back into the string that it is part of. But in this case, there is a `$` in front of the result of the evaluation. In our example, `#{@globalname}` would give just the two characters `dt`, while `$#{@globalname}` results in `$dt`.

So this is the first step in our double evaluation program: we have gone from the variable-name-holder `@globalname`, containing the string `"dt"` to *naming* the actual variable `$dt`. We have not yet *created* that variable. We have only prepared its name, `$dt`, as part of a longer string. But look at the beginning of the program line: `eval` is a method that takes a string, and then executes its content as if it were a normal line in a program.

In our example, this line will thus begin with:

```
eval("$dt = eval_value")
```

This is equivalent to the following program line:

```
$dt = eval_value
```

Now in *this* form, you see that we have actually created a new global variable `$dt`.

Alice: I am beginning to understand the process. It is quite amazing what is going on here, as a result of just one line of Ruby code. There are not just two evaluations implied, but *three*. First the value of the variable `@globalname` is extracted, through `#{@globalname}`. Then the string, which it is part of, is evaluated with the `eval` command. But as part of this second evaluation, you execute the method `eval_value`. Something tells me that this method does a third evaluation.

Bob: You are right. This method takes the variable that is called `@valuestring`, which as the name indicates is a string that contains the value associated with the option – either the default value, or the value that the user has supplied through the command line. What the method does is evaluating that string, once again using the `eval` command.

Let us forget for a moment about changes coming from the command line, and focus on default initialization of the global variable associated with an option. Then the logic is that the option block specifies the sentences:

```
Default value:        0.001
Global variable:      dt
```

The value `"0.001"` and the name `"dt"`, both of them strings, are read in as the values of the variables `@valuestring` and `@globalname`, respectively.

Alice: And what I called the three evaluations are the evaluation of those last two variables, to recover the two strings, then an evaluation of the string that says `"$dt = 0.001"` to the actual command `$dt = 0.001`. Yes, I think I now see the logic clearly.

And now that I see it, I also realize why things have to be so complex. Since this command line option parser cannot have any knowledge about any of the variables and values, it has to pass both the variables and values around in meta-variables, one meta-variable containing the variable name and one meta-variable containing the value of the value, so to speak, the string that contains the value. And since we are dealing with the value of a value, we have to evaluate that in order to get the actual value back.

6.7 Evaluating Values

Bob: Yes, that is a good summary. And that is exactly why I choose the name `eval_value` for the next method: it is evaluation the *value* of the value string that contains the actual value. Hard to say all this in words. You did quite well! Even so, you must admit that the Ruby one-line summary is a lot shorter than the English summary you just gave.

Alice: And unambiguous, unlike the English sentence. Good! Time to look at how you implemented the method `eval_value`. If all it does is use the `eval` method to go from a string to its value, why then did you write a separate method for it? It seems that you are really getting addicted to writing short methods!

Bob: Not *so* short, actually: you forget here that we have to use a third piece of information: we cannot evaluate the value of the global variable unless we know its type, knowledge of which is encoded in the string `@type`, as we have seen while parsing the single lines.

Alice: Ah, of course, yes. And different types lead to different actions.

Bob: Indeed. Here is the method:

```
  def eval_value
    case @type
      when "bool"
        eval(@valuestring)
      when "string"
        @valuestring
      when "int"
        @valuestring.to_i
      when "float"
```

```
        @valuestring.to_f
      when /^float\s*vector$/
        @valuestring.gsub(/[\[,\]]/," ").split.map{|x| x.to_f}.to_v
      else
        raise "\n  type \"#{@type}\" is not recognized"
    end
  end
```

In the case of a boolean variable, we can indeed apply `eval`. If the value is true, for example, this will just lead to:

```
eval("true") = true
```

In the case of a variable that is a string, nothing needs to be done, since a string is already a string. In the case of a number, the value string is converted using the `to_i` method if we are dealing with an integer, or `to_f` if we have a floating point number.

Things are a wee bit more complicated when we have a vector with floating point values. In that case we have to do three operations.

First we remove the square brackets and commas, if present. The simplest way to get rid of them is to replace each of those with a blank space `" "`. Just deleting them would be dangerous, since then `1,2`, for example, would become `12`; `1 2` on the other hand preserves the meaning that we are dealing with two separate components.

The second step involves converting each component from a piece of string to a floating point number, after having cut up the string using the `split` method, to create an array of little strings, and the `map` method to apply the `to_f` conversion to each of the elements of the array.

Finally, the third step consists in converting the array to a proper vector, an instance of our `Vector` class, using our `to_v` converter.

Alice: And the impressive thing is that all three steps are done in just one line of code.

Bob: Yes, one line that contains no less than five methods that are applied in turn! What is even more impressive is that the whole thing is still quite readable.

Alice: Yes, let me try to 'read' it: you start with a string, and after substituting some things, you split it into pieces, map the float converter to each piece, and then apply the vector converter.

Bob: And in that way you naturally evaluate a string to create a value of type 'float vector.' It all makes sense!

Alice: It's hard to believe that I have been writing code for so many years using more low-level languages like C++ and Fortran.

Bob: And hard to go back, although we'll have to, probably, to get a reasonable speed.

Alice: We may be pleasantly surprised. This whole parsing program, for example, is executed only once, at the beginning of running a program. There is absolutely no need to speed this up. Whether it takes a microsecond to run or a millisecond, who cares! Factors of a hundred or more of potential speed-up are only important in the domain of minutes and more. It is nice to change a minute of run time into a second, for sure, but to change a second into a few milliseconds is useless.

Bob: I hope you're right. We really should look into this speedup business soon, though.

Alice: I agree.

Chapter 7

Initial State Output

7.1 The to_s Method

Bob: We are approaching the end of the second journey. There is only one method left, to_s, apart from a little helper method, add_tabs, that acts as a little accountant, keeping care of how many tabs to add to make the output pretty. Let me first show the little helper:

```
def add_tabs(s, reference_size, n)
  (1..n).each{|i| s += "\t" if reference_size < 8*i}
  return s
end
```

Alice: That last one is keeping tabs on tabs, right? Well, I'm happy to skip that one for now. I can easily judge from the output whether you did a good job there or not. Can you show me the to_s method? Judging from the name, it converts the result of an option block into a string. But what kind of string? As if we haven't dealt with enough strings already!

Bob: It prepares the string that the program will print out at the start, as an echo of its initial state. In our example case of a time step, it will print something like:

```
dt = 0.001
```

Alice: And that is the only thing the user needs to know, after having used an option – or after having left its value to be the default value, as the case may be. Fine. That method should be quite short, no?

Bob: No, not really. It started off short and sweet, but it grew and grew while I was improving the code. There is quite a bit of bookkeeping that needs to be done here. Here is the code:

```
def to_s
  if @type == nil
    s = @description + "\n"
  elsif @type == "bool"
    if eval(@valuestring)
      s = @description + "\n"
    else
      s = ""
    end
  else
    s = @description
    s = add_tabs(s, s.size, 4)
    s += ": "
    if @printname
      s += @printname
    else
      s += @globalname
    end
    s += " = " unless @printname == ""
    s += "\n  " if @type =~ /^float\s*vector$/
    s += "#{eval("$#{@globalname}")}\n"
  end
  return s
end
```

7.2 The Header Option

Alice: It *is* longer than I thought. And it is not just a matter of many `when` options in a `case` statement: there seems to be some genuine complexity here.

Bob: Let me run you through the various switches. The first `if` statement concerns the description of the program as a whole. We have decided that the here document that contains this one long list of all options will start off with information about the program as a whole. This could take the form of:

```
Description:          A code doing such-and-such
Long description:
  This is a code doing such-and-such for the purpose of so-and-so.
  This code may come in handy if you want to do this-and-that.
  Be warned that it may not always work.  And beware of the dog.
```

This is an unusual option block, as option blocks go. Parsing so far has posed no problem, since we could parse the `Description` and `Long Description` like we parsed any other option. Also, when we talked about the `eval_value` method, we could forget about this header option that is a not-an-option option.

Alice: Could we, really? Now that I look at `eval_value` again, I see that it raises an error if `@type` is not one of the known values. But this header option has no type at all! So it surely will raise an error.

Bob: It *will* raise an error when it is invoked for the header option. But the key here is that it will never be invoked! I now realize that we forgot to talk about that. We were so distracted by what you called the comic book sequence, `("$#{@`, that we did not really finish looking at that line till the end. It came from the one-line method `initialize_global_variable`:

```
  def initialize_global_variable
    eval("$#{@globalname} = eval_value") if @globalname
  end
```

Note that it *only* invokes the `eval_value` method *if* the variable `@globalname` exists. And that variable springs into existence only when `parse_single_lines_done?` encounters a line in the definition string that starts with `Global Variable` or `Global variable`. And since the header option block contains no such line, the `eval_value` method will never be called for that option.

Alice: All is well then. I had completely forgotten about the header option. And I now see the meaning of the `if` statement at the top of `to_s`:

```
:include .clop.rb-14
```

The header option is the only one that does not carry a type, so at the start of the program, the one-line short description will be echoed.

Bob: Indeed. What will happen is:

```
|gravity> ruby some_code.rb
A code doing such-and-such
Time to stop integration        :
First interesting variable      : x  =  1.0
Second interesting variable     : y  =  3.14
. . .
```

After the one-line summary of what the code is about, it will start listing the values of the global variables.

Alice: But using their print name.

7.3 A Boolean Option

Bob: Exactly. We'll get to that in a moment. We're done with the case of the header option. Let us have a look at what needs to happen for an option of type boolean:

```
    elsif @type == "bool"
      if eval(@valuestring)
        s = @description + "\n"
      else
        s = ""
      end
```

If the value is true, in other words if `@valuestring == "true"`, that means that the user has invoked this option on the command line. In that case, a short message should be printed out. Ah, I see that the example I just gave only contained a header option and specific values of non-boolean type. Let me throw in a boolean option:

```
|gravity> ruby some_code.rb
A code doing such-and-such
First interesting variable      : x  =  1.0
Extra diagnostics will be provided
Second interesting variable     : y  =  3.14
. . .
```

This 'Extra diagnostics' line would be the result of the user providing a `-x` option, for example, on the command line. Now if the user chooses not to provide that option, there is nothing to report. Also, in that case, the option will retain its default value, which for a boolean variable is always `false`. In that case, the `else` part of the `if...else` statement will kick in, and only the empty string will be added to the string `s` that will be returned by `to_s`: nothing will be added at all.

Alice: Continuing down the `to_s` function, we have dealt with the cases of the header option and of boolean options. All other options are dealt with in the final `else` clause of the initial `if` statement.

Bob: Yes, because all other options have genuine values that need to be reported as such.

7.4 A Hack

Alice: The following lines:

```
s = @description
s = add_tabs(s, s.size, 4)
s += ": "
if @printname
  s += @printname
else
  s += @globalname
end
```

must result in producing the example line

```
  First interesting variable      : x
```

Right?

Bob: Right indeed. The description of the variable is followed by some white space counting magic followed by a colon, and then we get the proper print name, if it is provide. If not, we just use the internal name for the global variable associated with this option.

Alice: But I'm puzzled about what follows next:

```
s += " = " unless @printname == ""
```

What is the meaning of the `unless` here?

Bob: Ah, this is another 'feature' that I built in. I was thinking about running an N-body code . . .

Alice: . . . which is how we got into all this . . .

Bob: . . . yes, hard to believe, I feel like I'm turning into a software engineer. But just like an observer needs a telescope, which requires quite a bit of hardware engineering, a theorist interested in simulations needs a software environment, which requires quite a bit of software engineering.

Alice: Hear, hear!

Bob: So, thinking about running an N-body code, I realized that I would like to see the time step echoed, as well as the other physical parameters, in the form of `description :  name = value`, as in

```
  Integration time step         : dt = 0.001
```

but for the case of an option `-o output_file`, it would look a bit strange to have in the initial state list the line:

```
  Name of the outputfile        : output_file = run.out
```

It would be much more natural to have:

```
  Name of the outputfile        : run.out
```

since there is no reason in this case to echo the global variable name used to hold the output file name, nor is there any appropriate print name I could think of. In this case, the description says it all!

So I decided to build in a way to block the appearance of "`name = `" for such a case.

Alice: How did you do that?

Bob: I did not feel like adding yet another variable, like `@no_name_requested`. So I made an inventory of possible cases. If an option *does* have a print name, that name will be used. If it does *not* have a print name, the global variable name will be used. And then I realized the solution: if an option has a print name of length zero, just the empty string `""`, that could be interpreted as a request to remain silent and not print anything, neither the name, which is already nothing, nor the equal sign normally following it.

I admit, it is a bit of a hack, but it works. So if `@printname == ""` there is no need to put an `=` sign after the description, and that what is expressed in the line with the `unless` in it that you asked about:

```
       s += " = " unless @printname == ""
```

7.5 A Vector Option

Alice: Yes, it is a hack alright, but if it works, it works.

Bob: It works.

Alice: Good! But now I'm puzzled by the next line in the `to_s` code:

```
       s += "\n  " if @type =~ /^float\s*vector$/
```

What does that do?

Bob: I again had in mind our experience with an N-body code. When we print out a vector, we may want to have full machine accuracy, double precision. And if we do a three-dimensional simulation, which is normally the case, we need to print three numbers, each of which will take a space of two dozen characters. That together will already span a normal output line. Therefore, if we start a line with a description, the `float vector` will run off the page. To prevent that, I added a new line, just after the colon, for these kind of vectors.

As an example for an N-body code, where you could specify choosing three particles, and a shift in the center of mass position as follows:

```
ruby some_N_body_code.rb -n 3 -v [ 3, 4, 5 ]
```

You will then have the following initial state print-out:

```
Number of particles           : N = 3
Shifts center of mass by      : rcom =
  3.0000000000000000e+00  4.0000000000000000e+00  5.0000000000000000e+00
```

Alice: That does look much better than running off the page, I agree.

Bob: Somehow we are still under the influence of the original 80 column limitation of Fortran, it seems.

Alice: I admit that I always try to keep my output fitting within 80 columns. For of habit, I guess. And I don't like people sending me email that runs over 80 columns either.

7.6 A Pyramid of Evaluations

Bob: All a matter of taste. Well, one more line to go, at the end of `to_s` :

```
      s += "\n  " if @type =~ /^float\s*vector$/
```

This must finally produce the actual value, that what appears to the right of the equal sign in the initial state output.

Bob: Yes, and it does so by a three-stage evaluation, all bundled in one statement line.

Alice: Wow, that looks impressive. This will be my final test to see whether I now really understand what is going on. A concrete example will help. In the case of a time step size option, we have

```
@globalname = "dt"
```

The first evaluation produces the string

```
"$#{@globalname}" = "$dt"
```

which is a string holding the name of the global variable `$dt`.

The second evaluation is a call to `eval` and it produces:

```
eval("$#{@globalname}") = eval("$dt") = $dt
```

the global variable itself.

Then the third evaluation produces a string that contains the *value* of this global variable:

```
"#{eval("$#{@globalname}")}\n" = "#{eval("$dt")}\n" = "#{$dt}\n" = "0.01\n"
```

with a new line character at the end, to finish of the line.

Bob: Congratulations! You passed the exam.

Alice: And I also realize that, if we wanted to actually use the value of the time step here, we would have to do a fourth evaluation, to go from the string `"0.01"` to the value `0.01`. Let me just write it down, to see what it looks like. And I can forget about the new line here. The value should be:

```
eval("#{eval("$#{@globalname}")}") = 0.01
```

Bob: That looks right to me.

Alice: So you go from a variable to a string to a variable to a string to a variable. Well, well, and you told me the whole thing works?

Bob: Yes, it works. I'll give you a demonstration after we have completed our three journeys. This, by the way, is the end of the second journey, which is by far the longest one.

Alice: Not too surprising, since it is a journey into the kitchen, so to speak, to sea what is actually cooking in the various pots and pans. The other two journeys take place in the restaurant, where you're dealing with the food and the menu, but not the details of the preparation.

Bob: Indeed, the third journey will be shorter.

Chapter 8

The Third Journey: `Clop`, the Help Part

8.1 Two forms of Help

Alice: Hi Bob! Time for our third and last journey.

Bob: Hi Alice! Yes, we covered everything as far as parsing the options is concerned: both the definitions of the options by the writer of a program, and the way the options are set on the command line by the user of a program.

The only thing left to do is to see how the help mechanism is implemented. This is all done on the top level, within the `Clop` class that we visited in our first journey. Do you remember the flow of control? The initializer of the `Clop` class did three things, by invoking the following three methods: `parse_option_definitions`, `parse_command_line_options`, and `print_values`.

Of these three methods, all the work for the first and third one is done in the lower-level class `Clop_Option` that we visited at length in our second journey. Only the middle one, `parse_command_line_options`, required more work on the top-level, as we saw in the first journey, where we skipped the 'help' part. Let me print out this method again:

```
def parse_command_line_options(argv_array)
  while s = argv_array.shift
    if s == "-h"
      parse_help(argv_array, false)
      exit
    elsif s == "--help"
      parse_help(argv_array, true)
      exit
      elsif i = find_option(s)
        parse_option(i, s, argv_array)
      else
        raise "\n  option \"#{s}\" not recognized; try \"-h\" or \"--help\"\n"
      end
    end
    initialize_global_variables
  end
```

There is only one method that we have to inspect, `parse_help`. The first argument is the array `ARGV` that contains the pieces of the command line, a bunch of little strings that have been extracted from the command line by cutting it wherever blank space appeared. The second argument is a flag that determines whether we want to have extensive help information. The `-h` option asks for a minimal amount of help information, in short form, whereas the `--help` option asks for the long form of help.

Alice: No surprises here: all very clear.

8.2 Help for Selected Options

Bob: Before inspecting the `parse_help` method, let me first describe the idea behind the help facility as I have implemented it. If you type:

```
|gravity> ruby some_program.rb -h
```

you will get a one-line help message about each possible option. However, if you prefer to have less output, and you are only interested in one option, you can type:

```
|gravity> ruby some_program.rb -h -o
```

and this will give you only one line of output, a short description of the `-o` option. This is especially useful for the case of long help, since the command:

```
|gravity> ruby some_program.rb --help
```

may well generate a few pages of help, if you have a well documented program with many options. In that case it will be much easier to find what you want if you only order help for the options you are interested in.

Alice: Can you get help for, say, three options?

Bob: Yes: if you type:

```
|gravity> ruby some_program.rb --help -a -b --some_other_option
```

you will get get long help for all three options, but not for the other options that are not mentioned. Note that you can mix short and long descriptions of options. How you name an option, long or short, has no influence on the help output. If you use `-h` you only get one-liners, and if you use `--help` you get long information, several lines per option, independent of whether you use long or short names for the options themselves.

Alice: What will happen if you type:

```
|gravity> ruby some_program.rb -o -h
```

Bob: In that case, it is treated as if you would have typed:

```
|gravity> ruby some_program.rb -h
```

which means that you get one-line help for all options. The reason is that such a line as you just gave can naturally be generated through the use of a Unix `!!` command, as we already discussed, and the presence of the `-o` is likely to be a matter of laziness, rather than significance. If your previous command has been

```
|gravity> ruby some_program.rb -o
```

and if you then decide you what short help for that particular option, you have to type:

```
|gravity> !! -h -o
```

which the Unix shell translates into:

```
|gravity> ruby some_program.rb -o -h -o
```

and is then interpreted by the `Clop` help parser as if you had typed:

```
|gravity> ruby some_program.rb -h -o
```

8.3 Parsing Help

Alice: That is clear and reasonable. Can I see the actual help parser?

Bob: Here is the `parse_help` method:

```
def parse_help(argv_array, long)
  all = true
  while s = argv_array.shift
    if i = find_option(s)
      all = false
      print_help(long, i)
    end
  end
  print_help(long) if all
end
```

The variable `all` is a boolean flag. If it is `true`, we will print help information for all options. If help is requested only for selected options, this flag will be set to the `false` value. We start off with `all = true`. Then we inspect the `ARGV` array, and if we find one or more options present on the command line following the help request, then we offer selected help for each option encountered, through a call to `print_help`, while setting the `all` flag to be `false`.

The first argument of `print_help` passes on the flag which we received earlier, specifying whether we want to have the long form of help. The second, optional argument of `print_help` contains the number `i`, specifying the selected option. If we invoke the method `print_help` without a second argument, it is assumed that we want to have help for all options. Here is the method:

```
def print_help(long, i = nil)
  if i
    STDERR.print help_string(@options[i], long)
  else
    @options.each{|x| STDERR.print help_string(x, long)}
  end
end
```

Alice: I see what you mean with the optional second argument: if you leave that one out, it will be set to `nil`, the `if` test will fail, and so the `else` branch will be taken, and all options are printed out. However, if you specify an option `i`, the `if` test is successful, and only that option will be printed. In both cases you use the same procedure: you print a string on the standard error stream, provided by the method `help_string`.

8.4 Printing Help: the Idea

Bob: Yes, and before showing you how that method is implemented, let me sketch the idea behind it. Let us start with the short help form, invoked by `-h`. There are three types of options, that have to be treated differently.

First, there are the run-of-the-mill options, such as the time step size. I decided to give it a short help string as follows:

```
  -d  --step_size         : Integration time step    [default: 0.001]
```

Both forms of the command-line version of the options are shown, followed by the short description of the option, and then between square brackets the default value is shown.

Second, there are the boolean options, such as the request for extra diagnostics. I decided to let that generate the following short help string:

```
  -x  --extra_diagnostics : Extra diagnostics
```

For a boolean option, the default value is always `false`, so there is no need to list that.

Third, there is the header option, which does not have any value, and therefore also not a default value; in fact it does not have a way of writing it as a command line option. It really is a not-an-option option, since it only contains short and long description strings. Therefore, this is the only thing that will be printed. In short form, it could be just:

```
  A code doing such-and-such
```

To summarize, a code that has only these two options, together with this header option, will provide the following short help:

```
  |gravity> ruby some_program.rb -h
    A code doing such-and-such
    -d  --step_size         : Integration time step    [default: 0.001]
    -x  --extra_diagnostics : Extra diagnostics
```

Alice: I like the layout. Great! How did you implement it?

8.5 Printing Help: the Method

Bob: Here is the method:

```
   def help_string(option, long_flag)
     s = ""
     if option.type
       s += option_name_string(option)
     end
```

```
    if option.type or not long_flag
      s += "#{option.description}"
      s += default_value_string(option)
      s += "\n"
    end
    if long_flag
      s += "\n#{option.longdescription}\n"
    end
    return s
  end
```

I start with a null string `s`. If we are dealing with a header option, there is no type, and there are also no command line option names, such as `-d` or `--step_size`. So only if there is a type, we add those command line option names; this is taken care of by the following method:

```
  def option_name_string(option)
    s = ""
    if option.shortname
      s += "#{option.shortname}  "
    end
    s += "#{option.longname}"
    s = option.add_tabs(s, s.size, 3)
    s += ": "
    return s
  end
```

My assumption is that every option has a long way to call it, as in `--step_size`, but it may or may not have a short way `-d`. After all, some people don't like one-letter options.

Alice: People like me.

Bob: And others, like me, who do like short options, may literally run out of options if they write a program with more than 26 options.

Alice: If you really write such a program, I suggest that you cut it up into smaller pieces: 26 options strikes me as too much of a good thing.

Bob: Don't be so sure: there are plenty of programs that you use every day that have loads of options. Most of them you'll never use, but occasionally there you may hit upon a need for an arcane option, and then you'll be happy if it is provided. In any case, this is what I decided: short options are optional, pun not intended, while long options are not optional, but required.

Alice: With the exception of the header option, which never will invoke the `option_name_string` since the first `if` statement in `help_string` prevents it from doing so. Good! And finally, you add however many tabs are needed, through your `add_tabs` helper method in the `Clop_Option` class, and then a colon. So that is what produces the left-hand side of each normal option help output, for the short help version at least.

Bob: And for the long help version as well. We have not used the information from the `long_flag`, so far. So in both cases, for short and long help, the first `if` statement in `help_string` will provide the left hand side, up to the colon, of a line such as:

```
  -d  --step_size         : Integration time step    [default: 0.001]
```

8.6 What is Needed When

Alice: What is the meaning of the second `if` statement in `help_string`:

```
if option.type or not long_flag
  s += "#{option.description}"
  s += default_value_string(option)
  s += "\n"
end
```

Bob: If the `if` statement tests `true`, the following three lines are executed, which do what they say they do: they print the description of the option, to the right hand side of the colon, followed by the default value. This then finishes the one-line short help version.

Now the `if` test requires some explanation. For all options, except the header file, there is a type associated with the option, so the `if` test returns `true` and the three lines are executed. The only possible exception is the case where the option is the header option. In that case we have to discriminate between two possibilities.

If we request short help, we *do* want a short description of what the program is all about. So in that case we *do* want to execute the three lines following the `if` statement. Or more precisely, we want the first and the last line to be executed; there is no default value, so the method that is invoked in the second line, `default_value_string` has the responsibility to do nothing in the case of a header option.

However, if we request long help, there is no point in presenting both the short and the long description of what the program is doing, so we skip the short description, and move on directly to the long description. This is the reason for the complex looking `if` statement. It only tests false if `option.type == false` *and* `long_flag == true`, that is when the option is a header option, with the request for long help.

Alice: For all other options, when you ask for long help, you provide short help as well, for good measure?

Bob: Yes, I decided to do that. One reason is that gives us a quick way to get the default value information right at the top. Another reason is that the short information then acts as a type of title line for the longer help paragraph that follows. Here is an example of what you could expect for, say, the time step information, first for short help:

```
|gravity> ruby some_program.rb -h -d
  -d  --step_size         : Integration time step    [default: 0.001]
```

and then for long help:

```
|gravity> ruby some_program.rb --help --step_size
  -d  --step_size         : Integration time step    [default: 0.001]

  In this code, the integration time step is held constant,
  and shared among all particles in the N-body system.
```

Notice that I include a blank line between the short information part and the actual long information part, as answer to a long help call. This makes everything a bit more structured, and therefore easier to read, when you are faced with a whole bunch of options.

Alice: And the last part is what is printed out as a result of testing the third `if` statement in `help_string`:

```
    if long_flag
      s += "\n#{option.longdescription}\n"
    end
```

8.7 The Finishing Touch

Bob: Yes. And the only thing I haven't shown you yet is the method that prints the last part of the right-hand side of a short help line, where the default value is given:

```
  def default_value_string(option)
    s = ""
    if option.type and option.type != "bool"
      reference_size = "#{option.description}".size + 2
      s = option.add_tabs(s, reference_size, 4)
      s += " [default: #{option.defaultvalue}]"
    end
    return s
  end
```

If we are dealing with a header option, or an option with a boolean type, no default should be printed. The actual value is evaluated in the line

```
        s += " [default: #{option.defaultvalue}]"
```

and the rest is just bookkeeping stuff to get the blank lines and tabs all positioned correctly.

Alice: Congratulations again, Bob! This is a great tool that you have created. It will be so nice to have a common user interface for all the programs that we are going to write from now on. For each program, we will now what to expect: how to specify the options, and how to get information about the options through the fancy help mechanism you have developed here.

Most importantly, it will invite us to provide the right type of a help descriptions right away in the same file as where we write a piece of code. Rather then trying to write a separate manual, and then forgetting to update it, we can now provide the important information in a 'here document' as part of the same file in which we store the code lines that do the work. Excellent!

Bob: Well, you shouldn't give me all the credit. We developed the idea to provide option blocks together. And in fact, it was your criticism that prevented me from being happy with my original form of a command line parser, so you were the one who started it all!

Alice: Thanks, but I think I played the easier role. You implemented it all.

Bob: It depends on what you're good at, I guess. I find it easier to code something up, once I get the basic idea. What I find much harder is to get out of an accepted mind set, and to look at a problem with fresh eyes.

Alice: Watch out, Bob! This is the second time that you sound out of character. Everyone would have expected that you would not be that much interested in thinking about accepted mind sets, let alone trying to get out of one. As I noticed earlier, if someone were writing a book about our discussion, they would have put such a line in my mouth.

Bob: Well, it's good that nobody will be doing such a silly thing!

Chapter 9

A Built-In Test Facility

9.1 Testing Without a Driver

Alice: Have you tested the whole parsing mechanism that you have written?

Bob: Yes, and as far as I can see, it all works as advertised. At first, while I was writing the `Clop` class, I put the definition of the class, as well as the definition of the helper class `Clop_Option`, in the file `clop.rb`, and I used a different file to test the whole thing. That other file contained a driver with a long 'here document', and a call to `parse_command_line`, the only method that is used to let `Clop` do its work.

But after a while, it occurred to me that that I might as well add the driver to the end of the `clop.rb` file, in a clever way. You see, the reason I did not do it right away, is that I did not want to prevent another program from including the `clop.rb` file. A typical application program, such as an N-body code, can include a line at the top that reads:

```
require "clop.rb"
```

And if the file `clop.rb` includes a test driver at the end, that driver will be included in the application file, which is not what we want.

Alice: And you found a solution, to have your cake and eat it, that is to include the driver part and yet make it invisible for the application program?

Bob: How did you guess! That is exactly what I did. The trick was to add the following statement, after the definition of the classes and everything else that has to be included in the application program, and before the start of the driver block:

```
if __FILE__ == $0
```

Here the global variable `$0` contains the name of the program that you are running. If you run the `clop.rb` file directly, by typing:

```
|gravity> ruby clop.rb
```

then Ruby will give the string `"clop.rb"` to the variable `$0`. However, if you include the line

```
  require "clop.rb"
```

in a file called `some_other_program.rb`, the value of `$0` will be the string `"some_other_program.rb"`.

The key here is that the variable `__FILE__` *always* gets the value of the file in which it occurs. So the variable `__FILE__` in the file `clop.rb` will *always* get the content `"clop.rb"`, independently of whether you run `clop.rb` directly, or whether you run another program that includes a `require` statement for `clop.rb`.

Alice: Therefore, only when you run `clop.rb` directly, with the explicit command `ruby clop.rb`, is the equality guaranteed. Clever indeed! So if that is what you did, can I try it?

Bob: Please do!

9.2 Required Options

Alice: I will start without any options, to see what happens:

```
|gravity> ruby clop.rb
option "-n" or "--number_of_particles" required.  Description:
    Number of particles in an N-body snapshot.

option "-o" or "--output_file_name" required.  Description:
    Name of the snapshot output file.
    The snapshot contains the mass, position, and velocity values
    for all particles in an N-body system.

Please provide the required command line options.
```

Wow, that's a lot more than I expected. Where did that all come from?

Bob: In defining the option blocks, you can specify the default value `none`, which means that no default is given, which in turn means that the user should provide one. I included various options in the here document of my test driver, and two of them I gave the the default value `none`.

The fact that you see so much output is a result of the action of the method `check_required_options`, which prints out the whole long description, to tell you in detail what type of options you should minimally provide.

Alice: Let me see what happens if I provide the first option:

```
|gravity> ruby clop.rb -n3
option "-o" or "--output_file_name" required.  Description:
    Name of the snapshot output file.
    The snapshot contains the mass, position, and velocity values
    for all particles in an N-body system.

Please provide the required command line option.
```

It still complains about the other option, as it should. And it even knows about English grammar. Look: it talks at the end about the missing 'option' rather than the missing 'options', as it did when two options were missing. Nice touch.

Bob: Thanks! I like to get such details straight. A matter of craftsmanship, as they used to call it. Why don't you add the second option too? No output file will be created here; it is just a test.

Alice: Okay, this should stop the complaints:

```
|gravity> ruby clop.rb -n3 -o tmp.out
Command line option parser
Softening length: eps = 0.0
Time to stop integration: t = 10.0
Number of particles: N = 3
Shifts center of mass velocity: vcom =
  3.04.05.0
Name of the outputfile: tmp.out
Star type: star_type = star: MS
```

Ah, the whole list of default options, and on top a one liner that describes what this program is doing. And everything is lined up perfectly. Indeed a matter of craftmanship, I would say.

9.3 A Boolean Option

Bob: Glad you like it! How about trying out a boolean value?

Alice: Sure. Let's see. But, wait a minute. All boolean values by default are set to `false`. And boolean options are only reported in the initial state printout if they are `true`. So we have a catch 22 here: if I don't give the boolean options, they will never appear, so how do I know how to ask for those options. How do I find out what their command line names are?

Bob: Ahem.

Alice: You're not going to help me and give me a hint at least?

Bob: Going to *help* you?

Alice: Ah, of course, the *help* facility. I can ask the program itself. Okay:

```
|gravity> ruby clop.rb -h
Command line option parser
-s  --softening_length: Softening length [default: 0.0]
-t  --end_time: Time to stop integration [default: 10]
-n  --number_of_particles: Number of particles [default: none]
-x  --extra_diagnostics: Extra diagnostics
-v  --shift_velocity: Shifts center of mass velocity [default: [3, 4, 5]]
-o  --output_file_name: Name of the outputfile [default: none]
--star_type: Star type [default: star: MS]
```

Ah, there it is: our good friend `--extra_diagnostics`, or simply `-x`. That must be a boolean option.

Bob: Try it!

Alice: By just adding `-x` at the end, yes? That's easy, as long as I keep remembering to add the required `-n` and `-o` options as well.

Bob: Don't worry, the program will remind you if you don't.

Alice: Let there be boolean initial state output:

```
|gravity> ruby clop.rb -n3 -o tmp.out -x
Command line option parser
Softening length: eps = 0.0
Time to stop integration: t = 10.0
Number of particles: N = 3
Extra diagnostics
Shifts center of mass velocity: vcom =
  3.04.05.0
Name of the outputfile: tmp.out
Star type: star_type = star: MS
```

And so there is! This is fun.

9.4 A Vector Option

Bob: How about shifting the velocity of the center of mass?

Alice: With `-v` or `--shift_velocity`, I see from the short help output. Let me try the longer form. I see that the default value for this vector has three components. Am I allowed to work in two dimensions as well, for a three-body scattering experiment in a plane, say, or do you insist on working in three dimensions, in which case I could set the third component equal to zero?

Bob: Why don't you ask the program?

Alice: You're getting mischievous. Ask the program? Ah, with the *long* version of help perhaps? Will that tell me?

Bob: Remember, if you type `--help --some_option`, you will get a long form of help for that option.

Alice: Let me try:

```
|gravity> ruby clop.rb --help --shift_velocity
-v  --shift_velocity: Shifts center of mass velocity [default: [3, 4, 5]]

    The center of mass of the N-body system will be shifted by this amount.
    If the vector has fewer components than the dimensionality of the N-body
    system, zeroes will be added to the vector.
    If the vector has more components than the dimensionality of the N-body
    system, the extra components will be disgarded.
```

Now *that* is impressive. It answers all that I wanted to know and more!

Bob: I can't guarantee that all my programs will be *that* helpful, but at least I made a start here.

Alice: Here is my attempt at a two-dimensional shift:

```
|gravity> ruby clop.rb -o tmp.out -n 3 --shift_velocity [2, 3]
Command line option parser
Softening length: eps = 0.0
Time to stop integration: t = 10.0
Number of particles: N = 3
Shifts center of mass velocity: vcom =
  2.03.0
Name of the outputfile: tmp.out
Star type: star_type = star: MS
```

And it behaves as I expected it to.

9.5 A Star Type Option

Bob: Why don't you try to play with the type I introduced for classifying stars. I added that as another test for the whole concept of a general parser.

Alice: Classifying stars? Okay, I got it now: I won't ask you any questions anymore, I'll just ask the program. First I have to find which option I should be dealing with:

```
|gravity> ruby clop.rb -h
Command line option parser
-s  --softening_length: Softening length [default: 0.0]
-t  --end_time: Time to stop integration [default: 10]
-n  --number_of_particles: Number of particles [default: none]
-x  --extra_diagnostics: Extra diagnostics
-v  --shift_velocity: Shifts center of mass velocity [default: [3, 4, 5]]
-o  --output_file_name: Name of the outputfile [default: none]
--star_type: Star type [default: star: MS]
```

I see: `star_type`. And this happens to be an example of an option that does not have a short name. After all, both `-s` and `-t` are taken already, and rather than inventing an unmemorable name like `-q` for the star type, it makes more sense to stick to a longer name that has a clearer meaning. I completely agree.

The next step is to find out more about this particular option:

```
|gravity> ruby clop.rb --help --star_type
--star_type: Star type [default: star: MS]

    This options allows you to specify that a particle is a star, of a
    certain type T, and possibly of subtypes t1, t2, ..., tk by specifying
    --star_type "star: T: t1: t2: ...: tk".  The ":" separators are allowed
    to have blank spaces before and after them.

      Examples: --star_type "star: MS"
                --star_type "star : MS : ZAMS"
                --star_type "star: giant: AGB"
                --star_type "star:NS:pulsar:millisecond pulsar"
```

And here we have an example of allowing extra colons in the content of the definition of an option, as you discussed when you introduced the concept of a non-greedy operator in a regular expression. Let me try one of the examples:

```
|gravity> ruby clop.rb -o tmp.out -n 3 --star_type star : MS : ZAMS
clop.rb:143:in `parse_command_line_options': (RuntimeError)
  option ":" not recognized; try "-h" or "--help"
from clop.rb:115:in `initialize'
from clop.rb:267:in `new'
from clop.rb:267:in `parse_command_line'
from clop.rb:377
```

9.6 No Comment

Hey, what happened? Your long help facility must have given me the wrong answer! I tried one of the examples given there.

Bob: Not quite.

Alice: What do you mean? I did not make any spelling mistake: here it is: `star_type` is spelled correctly, and so is `star`. And I thought I had a freedom in choosing whatever else would follow.

Bob: Why don't you look at what the error message is telling you.

Alice: You really want the program to help me, and I must admit, it would be an accomplishment if your error messages would tell me what went wrong, without you doing the hand holding. Let's see. The program complains that the option `:` is not recognized. But you told me that the non-greedy operator would take care of any and all extra colons!

Bob: No comment.

Alice: Okay, okay, I'll struggle on. The first `:` I gave was the one following the word `star`. I have a space between `star` and `:`, but that space also occurred in the example suggested by the long help output.

Still, I have to do something, wiggle some wires somewhere, to learn more, so let me write the same thing without a space, just to see whether that makes a difference.

```
|gravity> ruby clop.rb -o tmp.out -n 3 --star_type star: MS : ZAMS
clop.rb:143:in `parse_command_line_options': (RuntimeError)
  option "MS" not recognized; try "-h" or "--help"
from clop.rb:115:in `initialize'
from clop.rb:267:in `new'
from clop.rb:267:in `parse_command_line'
from clop.rb:377
```

Now it complains that option `MS` is not recognized. Hmmm. What would happen if I don't give any spaces at all?

```
|gravity> ruby clop.rb -o tmp.out -n 3 --star_type star:MS:ZAMS
Command line option parser
Softening length: eps = 0.0
Time to stop integration: t = 10.0
Number of particles: N = 3
Shifts center of mass velocity: vcom =
  3.04.05.0
Name of the outputfile: tmp.out
Star type: star_type = star:MS:ZAMS
```

Hey, now it works! And the star type is being assigned correctly. It seems that adding a space somewhere triggers a protest, and the specific protest is that whatever comes after the first space is not recognized.

I'm still in the dark. Let me look again at the error message. I can just add a space somewhere, and I'm sure the program will complain again:

```
|gravity> ruby clop.rb -o tmp.out -n 3 --star_type star:MS :ZAMS
clop.rb:143:in ‘parse_command_line_options’:  (RuntimeError)
  option ":ZAMS" not recognized; try "-h" or "--help"
from clop.rb:115:in ‘initialize’
from clop.rb:267:in ‘new’
from clop.rb:267:in ‘parse_command_line’
from clop.rb:377
```

Hmmmm. `option ":ZAMS" not recognized`. AAH!! It is an *option* that is not recognized. The command line parser receives the command line in a form that is already cut up wherever there is white space. So only the `star:MS` is considered to be the *value* of the option `--star_type` and whatever comes next would normally be a new *option*, starting with a hyphen, and more than that, an option that can be recognized by the program. And `:ZAMS` is not even an option.

9.7 The Answer

Bob: See: you got it under your own steam. I'm glad to see that combination of the help facility and the error messages were enough to let you figure it out.

Alice: But I still have a question. There is something wrong with your long help form, since the example you provided contained blank spaces!

Bob: Why don't you check it again, to make sure.

Alice: I *am* sure. Here it is, this is how we got started:

```
|gravity> ruby clop.rb --help --star_type
--star_type: Star type [default: star: MS]

    This options allows you to specify that a particle is a star, of a
    certain type T, and possibly of subtypes t1, t2, ..., tk by specifying
    --star_type "star: T: t1: t2: ...: tk".  The ":" separators are allowed
    to have blank spaces before and after them.

      Examples: --star_type "star: MS"
                --star_type "star : MS : ZAMS"
                --star_type "star: giant: AGB"
                --star_type "star:NS:pulsar:millisecond pulsar"
```

You see, a space between `star` and `:` and between . . .

Bob: Yes?

Alice: . . . the whole argument is enclosed between double quotes, making it a *single* element in the array `ARGV` of command line options.

Bob: Yes!

Alice: Here we go again:

```
|gravity> ruby clop.rb -o tmp.out -n 3 --star_type "star : MS : ZAMS"
Command line option parser
Softening length: eps = 0.0
Time to stop integration: t = 10.0
Number of particles: N = 3
Shifts center of mass velocity: vcom =
  3.04.05.0
Name of the outputfile: tmp.out
Star type: star_type = star : MS : ZAMS
```

That's much better. You were right, I was wrong. I guess I was misled by the vector example.

Bob: That is an exception, and it only works because the parser keeps parsing until it encounters a closing square bracket `]`. With the `--star_type` option there would be no way to know when to stop parsing.

Alice: You *could* make the parser more fancy, by letting it continue to parse until it encounters a new option, or the end of the string.

Bob: And how would it recognize a new option?

Alice: A leading hyphen would do the trick, wouldn't it?

Bob: And how about a number like `-1` ?

Alice: Ah yes, it is more complicated than I thought. That would look like an option, but actually would be a legal value.

Bob: I think I *could* indeed do something, but it would not be so simple. For example, insisting that an option would start with a letter, not a number, would take care of the negative number problem. But there may be other problems as well. It just seems simpler to insist on using double quotes. My attempt to make life easier for vectors was perhaps already too much of a concession.

Chapter 10

The `Clop` Code

10.1 Test Drivers

Alice: I'm really happy with the command line option parser. We'll use it from now on. And your demonstration of the help facility was impressive, I must say.

Bob: You mean *your* demonstration.

Alice: I guess so; you were letting me struggle all by myself! But somehow there was just enough information delivered to let me figure it out. And I must say, I like the idea of having each class include its own test driver, at the end of the file that holds the class. Your trick of including the line

```
if __FILE__ == $0
```

was a great idea. I think we should do that from now on for any file that contains class definitions that will be used by other files.

Bob: Yes, I'm planning to do that.

Alice: Can you show me what you wrote, in `clop.rb`, as the test driver?

Bob: Here is the whole content of `clop.rb`, with the test driver at the end. Note that the first four lines of the 'here document' are:

```
Description: Command line option parser, (c) 2004, Piet Hut & Jun Makino, ACS
Long description:
  Test program for the class Clop (Command line option parser),
  (c) 2004, Piet Hut and Jun Makino; see ACS at www.artcompsi.org
```

Alice: Who are those two people?

Bob: Oh, I just chose two pen names for us; as long as we are just building toy models, we may as well use toy names for our products, don't you think?

Alice: But why a Dutch and a Japanese name?

Bob: Why not? I had to pick something.

Alice: I might have picked a Liechtensteiner and a Fijian, but I won't argue with your taste. Can you show me the code?

10.2 Code Listing

Bob: Here it is:

```
require "vector.rb"

class Clop_Option

  attr_reader :shortname, :longname, :type,
              :description, :longdescription, :printname, :defaultvalue
  attr_accessor :valuestring

  def initialize(def_str)
    parse_option_definition(def_str)
  end

  def parse_option_definition(def_str)
    while s = def_str.shift
      break if parse_single_lines_done?(s)
    end
    while s = def_str.shift
      break if s =~ /^\s*$/ and  def_str[0] =~ /^\s*$/
      @longdescription += s + "\n"
    end
  end

  def parse_single_lines_done?(s)
    if s !~ /\s*(\w.*?)\s*\:/
      raise "\n  option definition line has wrong format:\n==> #{s} <==\n"
    end
    name = $1
    content = $'
    case name
      when /Short\s+(N|n)ame/
        @shortname = content.split[0]
      when /Long\s+(N|n)ame/
        @longname = content.split[0]
      when /Value\s+(T|t)ype/
      @type = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
      @valuestring = "false" if @type == "bool"
    when /Default\s+(V|v)alue/
      @defaultvalue = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
      @valuestring = @defaultvalue
    when /Global\s+(V|v)ariable/
      @globalname = content.split[0]
    when /Print\s+(N|n)ame/
      @printname = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
    when /Description/
      @description = content.sub(/^\s+/,"").sub(/\s*(#.*|$)/,"")
    when /Long\s+(D|d)escription/
      @longdescription = ""
      return true
    else
      raise "\n  option definition line unrecognized:\n==> #{s} <==\n"
  end
  return false
end

def initialize_global_variable
  eval("$#{@globalname} = eval_value") if @globalname
end

def eval_value
  case @type
    when "bool"
      eval(@valuestring)
    when "string"
      @valuestring
    when "int"
      @valuestring.to_i
    when "float"
      @valuestring.to_f
    when /^float\s*vector$/
      @valuestring.gsub(/[\[,\]]/," ").split.map{|x| x.to_f}.to_v
    else
      raise "\n  type \"#{@type}\" is not recognized"
  end
end

def add_tabs(s, reference_size, n)
  (1..n).each{|i| s += "\t" if reference_size < 8*i}
  return s
end
```

```
  def to_s
    if @type == nil
      s = @description + "\n"
    elsif @type == "bool"
      if eval(@valuestring)
        s = @description + "\n"
      else
        s = ""
      end
    else
      s = @description
      s = add_tabs(s, s.size, 4)
      s += ": "
      if @printname
        s += @printname
      else
        s += @globalname
      end
      s += " = " unless @printname == ""
      s += "\n  " if @type =~ /^float\s*vector$/
      s += "#{eval("$#{@globalname}")}\n"
    end
    return s
  end

end

class Clop

  def initialize(def_str, argv_array = nil)
    parse_option_definitions(def_str)
    if argv_array
      parse_command_line_options(argv_array)
    end
    print_values
  end

  def parse_option_definitions(def_str)
    a = def_str.split("\n")
    @options=[]
    while a[0]
      if a[0] =~ /^\s*$/
        a.shift
      else
        @options.push(Clop_Option.new(a))
      end
    end
  end

  def parse_command_line_options(argv_array)
    while s = argv_array.shift
      if s == "-h"
        parse_help(argv_array, false)
        exit
      elsif s == "--help"
        parse_help(argv_array, true)
        exit
      elsif i = find_option(s)
        parse_option(i, s, argv_array)
      else
        raise "\n  option \"#{s}\" not recognized; try \"-h\" or \"--help\"\n"
      end
    end
    initialize_global_variables
  end

  def print_values
    @options.each{|x| STDERR.print x.to_s}
  end

  def find_option(s)
    i = nil
    @options.each_index do |x|
      i = x if s == @options[x].longname
      if @options[x].shortname
        i = x if s =~ Regexp.new(@options[x].shortname) and $` == ""
      end
    end
    return i
  end

  def parse_option(i, s, argv_array)
    if @options[i].type == "bool"
      @options[i].valuestring = "true"
      return
    end
    if s =~ /^-[^-]/ and (value = $') =~ /\w/
      @options[i].valuestring = value
    else
      unless @options[i].valuestring = argv_array.shift
        raise "\n  option \"#{s}\" requires a value, but no value given;\n" +
              "  option description: #{@options[i].description}\n"
      end
    end
    if @options[i].type =~ /^float\s*vector$/
      while (@options[i].valuestring !~ /\]/)
        @options[i].valuestring += " " + argv_array.shift
      end
    end
  end

  def initialize_global_variables
    @options.each{|x| x.initialize_global_variable}
    check_required_options
  end

  def check_required_options
    options_missing = 0
    @options.each do |x|
      if x.valuestring == "none"
        options_missing += 1
        STDERR.print "option "
        STDERR.print "\"#{x.shortname}\" or " if x.shortname
        STDERR.print "\"#{x.longname}\" required.  "
        STDERR.print "Description:\n#{x.longdescription}\n"
      end
    end
    if options_missing > 0
      STDERR.print "Please provide the required command line option"
      STDERR.print "s" if options_missing > 1
      STDERR.print ".\n"
      exit(1)
    end
  end

  def parse_help(argv_array, long)
    all = true
    while s = argv_array.shift
      if i = find_option(s)
        all = false
        print_help(long, i)
      end
    end
    print_help(long) if all
  end

  def print_help(long, i = nil)
    if i
```

```
      STDERR.print help_string(@options[i], long)
    else
      @options.each{|x| STDERR.print help_string(x, long)}
    end
  end

  def help_string(option, long_flag)
    s = ""
    if option.type
      s += option_name_string(option)
    end
    if option.type or not long_flag
      s += "#{option.description}"
      s += default_value_string(option)
      s += "\n"
    end
    if long_flag
      s += "\n#{option.longdescription}\n"
    end
    return s
  end

  def option_name_string(option)
    s = ""
    if option.shortname
      s += "#{option.shortname}  "
    end
    s += "#{option.longname}"
    s = option.add_tabs(s, s.size, 3)
    s += ": "
    return s
  end

  def default_value_string(option)
    s = ""
    if option.type and option.type != "bool"
      reference_size = "#{option.description}".size + 2
      s = option.add_tabs(s, reference_size, 4)
      s += " [default: #{option.defaultvalue}]"
    end
    return s
  end

end

def parse_command_line(def_str)
```

```
  Clop.new(def_str, ARGV)
end

if __FILE__ == $0

  options_definition_string = <<-END

  Description: Command line option parser
  Long description:
    Test program for the class Clop (Command line option parser),
    (c) 2004, Piet Hut and Jun Makino; see ACS at www.artcompsi.org

    This program appears at the end of the file "clop.rb" that contains
    the definition of the Clop class.
    By running the file (typing "ruby clop.rb"), you can check whether
    it still behaves correctly.  Maximum help is provided by the command
    "ruby clop.rb --help".


  Short name:           -s
  Long name:            --softening_length
  Value type:           float              # double/real/...
  Default value:        0.0
  Global variable:      eps                # any comment allowed here
  Description:          Softening length   # and here too
  Long description:                        # and even here
    This option sets the softening length used to calculate the force
    between two particles.  The calculation scheme comforms to standard
    Plummer softening, where rs2=r**2+eps**2 is used in place of r**2.


  Short name:           -t
  Long name:            --end_time
  Value type:           float
  Default value:        10
  Global variable:      t_end
  Print name:           t
  Description:          Time to stop integration
  Long description:
    This option gives the time to stop integration.


  Short name:           -n
  Long name:            --number_of_particles
  Value type:           int
  Default value:        none
```

```
Global variable:      n_particles
Print name:           N
Description:          Number of particles
Long description:
  Number of particles in an N-body snapshot.


Short name:           -x
Long name:            --extra_diagnostics
Value type:           bool
Global variable:      xdiag
Description:          Extra diagnostics
Long description:
  The following extra diagnostics will be printed:

    acceleration (for all integrators)
    jerk (for the Hermite integrator)


Short name:           -v
Long name:            --shift_velocity
Value type:           float vector          # numbers in between [ ] brackets
Default value:        [3, 4, 5]
Global variable:      vcom
Description:          Shifts center of mass velocity
Long description:
  The center of mass of the N-body system will be shifted by this amount.
  If the vector has fewer components than the dimensionality of the N-body
  system, zeroes will be added to the vector.
  If the vector has more components than the dimensionality of the N-body
  system, the extra components will be disgarded.


Short name:           -o
Long name:            --output_file_name
Value type:           string
Default value:        none
Global variable:      output_file_name
Print name:                                 # no name, hence name suppressed
Description:          Name of the outputfile
Long description:
  Name of the snapshot output file.
  The snapshot contains the mass, position, and velocity values
  for all particles in an N-body system.
  Long name:             --star_type             # no short option given here
  Value type:            string
  Default value:         star: MS                # parser cuts only at first ":"
  Global variable:       star_type
  Description:           Star type
  Long description:
    This options allows you to specify that a particle is a star, of a
    certain type T, and possibly of subtypes t1, t2, ..., tk by specifying
    --star_type "star: T: t1: t2: ...: tk".  The ":" separators are allowed
    to have blank spaces before and after them.

      Examples: --star_type "star: MS"
                --star_type "star : MS : ZAMS"
                --star_type "star: giant: AGB"
                --star_type "star:NS:pulsar:millisecond pulsar"

  END

  parse_command_line(options_definition_string)

end
```

Chapter 11

Literature References

[to be provided]