The Art of Computational Science

The Kali Code

vol. 8

ACS Data Format:
Self-Describing Data Files

Piet Hut and Jun Makino

September 13, 2007

Contents

Preface **5**

0.1 Acknowledgments 5

1 A Hierarchical Data Format **7**

1.1 Thinking Ahead 7

1.2 Tree Structure 8

1.3 Physical Quantities 9

1.4 A Particle Format 11

1.5 Name Spaces 12

1.6 Compromise 13

1.7 Hierarchical particle structures 14

1.8 Extended Tag Names 15

2 Particle Output **17**

2.1 Getting Started 17

2.2 A Single Write Method 18

2.3 Writing `to_s` 19

2.4 Methods f_to_s and f_v_to_s 21

2.5 Testing 22

2.6 The File `iobody1.rb` 24

3 Using the `to_s` Method **27**

3.1 xxx 27

4 Nested Output **29**

4.1 ACS Output 29
4.2 Nbody Output 29
4.3 Example: Plummer's Model 29

5 Particle Input **31**
5.1 How to Read 31
5.2 Passing the Type 32
5.3 Handling the Header 34
5.4 Handling Other Lines 35
5.5 Testing 37

6 A Scratch Pad **39**
6.1 Extra Information 39
6.2 Two Possibilities 40
6.3 More Possibilities 42
6.4 A Box 44
6.5 Onward 45
6.6 Testing 47

7 Nested Input **51**
7.1 xxx 51

8 Introduction **53**
8.1 xxx 53
8.2 xxx 54

9 Literature References **55**

Preface

In this volume, Alice and Bob develop a robust and flexible ACS data format, and implement basic I/O modules, based on that format. The current version of this volume contains only half of the material that we plan to include here, in a future release.

[add reference to starlab, for hierarchical particle data structure idea]

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 the first half of this volume was written, during a visit in May 2004, made possible by the kind invitations to both of us by Professor Masao Ninomiya.

Piet Hut and Jun Makino

Kyoto, July 2004

Chapter 1

A Hierarchical Data Format

1.1 Thinking Ahead

Alice: Hi, Bob! How are things?

Bob: Hi, Alice! As far as our project goes, things are fine. As for chores, there are two referees reports I still haven't written, an NSF proposal that is due soon, a committee meeting for which I have to prepare, and so on.

Alice: Business as usual, I take it! I have quite a backlog of things-to-do too. However, if we would wait for everything to be cleared out of the way, we would never get around to do some useful work. Shall we just sit down and see how far we can get, for the rest of the afternoon?

Bob: Yeah, why not. I'm pretty excited by how much we got done so far, and it is high time we get some form of graphics working, so that we can see what we've been doing so far.

Alice: I agree. But before rushing into that, let us take stock of what we have done so far. We have written a two-body orbit integrator, which is quite flexible in the sense that it has a choice of algorithms. On the other hand, the interface of the program with the user and with external data is quite primitive.

Bob: Well, it's a toy model, isn't it, what else do you expect? And why should that hold us back from visualizing our orbits right away?

Alice: I'm trying to think ahead, to see how our toy model may evolve. Even though we call our programs toy models, if we are successful in our design, it is likely that they will be used quite widely, and that they will be modified for purposes we can't even think about right now.

Bob: That would be nice, but why should we care about that now?

Alice: If we don't look ahead, and let things simply evolve, it is quite likely that

we will design a better I/O format, before too long. Also, we will probably want to implement the option to invoke our programs with command line arguments, rather than always having to change the parameters in the driver file, as we have done so far. If we change I/O format next month, and command line options the month after that, then we will wind up with three different versions of each program, doing more or less the same thing.

This proliferation of versions will be a source of confusion, especially for new users. What is worse, if we write a graphics interface, we will have to write at least two parallel versions, one for the original I/O data format, and one for the later one; and more if we keep changing the data format once more.

Bob: But even if we would try to be a bit more careful now, how can we guarantee that we won't change our mind about the data format, say, half a year from now? It would be very hard to predict what our needs might be by that time.

Alice: This is indeed a challenge, but I think it is a challenge we can live up to. If we design our data format with sufficient care and generality and flexibility, we might be able to absorb even quite unexpected future additions, without having to change the data framework.

Bob: Are you thinking about a self-describing data format, like the FITS format that many observers use?

Alice: Indeed, but then more powerful and more flexible. Something like XML would be a natural choice.

1.2 Tree Structure

Bob: Hmm. The only thing I know about XML is what I have seen in the VOTable format for the virtual observatory initiative. It seems to be modeled on the HTML format for web pages.

Alice: I don't know much about XML either. The basic structure is indeed list-based, like Lisp, and like HTML. Unlike HTML, XML is more consistent in always requiring closing brackets of the same type to follow an opening bracket. From what I have seen of it, the basic idea seems sound and very general. What worries me a bit is that there are still relatively few tools and applications, especially in the open source world.

Bob: I must admit, it did strike me as rather wordy, and I'm not sure that we want to stick to such a precise protocol.

Alice: We could make a compromise, by designing something simpler and more straightforward, but in such a way that we can translate it easily into XML. We can then implement both versions for our I/O routines, our own format as well as an XML format.

Bob: The question is, do we want to have two parallel systems? I guess it would be okay. There is something to say for having a full XML compatible version, since XML is likely to remain a generally accepted format for quite a while to come. And at the same time, I like to have something a bit more user friendly to work with.

Alice: The only constraint would be to give our own format a lisp-like tree structure, like XML, by providing opening and closing tags.

Bob: That would make sense. The first thing I would vote for, is to leave out the full `</something>` description after a `<something>` opening, since that is certainly unnecessarily tedious. And we can always automatically reconstruct the full ending tags, when we translate into XML.

Alice: I agree. How about having a "begin something" followed by only an "end", and not a full "end something"?

Bob: I like that, since it fits in nicely with the Ruby syntax. So we just have to introduce a few special key words, to signal the start of the data for a single particle, and for a whole N-body system.

Alice: And also for the beginning and end of a file, of our type. We need the first line of a such a file to be "begin our-system". Now how shall we call our system?

Bob: If we want to be modest, we can call it "toy-simulations". Or more grandiosely, "toy-computational-science", if we follow your vision and claim that we can apply it to any and all form of scientific simulations.

Alice: I know that you are joking, but frankly, I see no reason why our approach should not be widely applicable to all of astrophysics, or physics, or natural science for that matter. But to call it a toy may give the wrong emphasis.

Bob: If not a toy, you could call it real science. But the science of computational science sounds silly. Hey, how about art? The Art of Computational Science.

Alice: I like that! ACS, for short. Okay, the first line for any file in our new date format will then be `begin ACS`, and the last line will be `end` :

```
begin ACS
  . . .
end
```

1.3 Physical Quantities

Bob: Now how do we specify the connection between hard numbers and the physical variables that they contain the values of? If we want to be strict, we could write

```
begin mass
```

```
  0.1
end
```

and similarly

```
begin position
  1.0  0.0  0.5
end
```

but that seems to be a bit too much of a good thing.

Alice: Yes, it would be better if we could introduce an in-line notation. Even so, it would be good to keep the form you just wrote down as a legal option within our system. The in-line notation would then be just syntactic sugar.

Bob: Indeed nicely Ruby-like. What is happening today? We seem to be in agreement all the time!

Alice: It's because you agreed so quickly to device a standard format. I expected that you would put up a fight and resist such extra overhead!

Bob: I would have, a few years ago. But in the mean time, I've been bitten too often by the problems that can happen when you use different file formats without properly distinguishing them. And the notion of self-describing data is something I was familiar with through FITS files. So you see, I based my agreement on real experience, and not on a wish to be theoretically pure in some way or other.

Alice: Good theoretical ideas arise from the distillation of a large body of collective experience, so in that sense too, I agree with you. In any case, I'm sure we'll develop differences again pretty soon. Moving right along, do you have a suggestion for an inline notation?

Bob: How about

```
mass=0.1
```

Alice: I like the idea of introducing an equal sign, to separate variable and value, making it look again rather Ruby like. But I don't like the look of

```
position=1.0 0.0 0.5
```

I suggest that we either introduce commas to separate the three numbers on the right, and allow spaces around the `=` sign.

Bob: I prefer spaces over commas. Okay, let's make both legal, what you just wrote with the equal sign crammed in, as well as

```
position = 1.0 0.0 0.5
```

In other words, an equal sign is the delimiter between variable and value, a single space is a delimiter between components of a vector, and in addition you can add as many spaces elsewhere as you like. So the following would be fine too:

```
position  =  1.0   0.0 0.5
```

1.4 A Particle Format

Alice: fair enough; let it be so. Now how shall we bundle all these numbers into one particle?

Bob: We can just call it `particle`. Here is one particle, all by itself in a file:

```
begin ACS
  begin particle
    mass = 0.1
    position = 1.0 0.0 0.5
    velocity = 0.0 1.0 0.0
  end
end
```

Alice: That looks nice. And if we don't want to use the syntactic sugar, we can write:

```
begin ACS
  begin particle
    begin mass
      0.1
    end
    begin position
      1.0 0.0 0.5
    end
    begin velocity
      0.0 1.0 0.0
    end
  end
end
```

Bob: Yeah, it is good to keep this as a legal option, but I doubt whether anyone will ever use that. In fact, once we allow levels of syntactic sugar, how about introducing an even more compact version:

```
begin ACS
```

```
  begin particle
    mass = 0.1 ; position = 1.0 0.0 0.5 ; velocity = 0.0 1.0 0.0
  end
end
```

Alice: Good idea! That again conforms with Ruby usage, and it is nice to have freedom of expression.

1.5 Name Spaces

Bob: If you really want to give me freedom of expression, I have another suggestion. I think that the above version is still a bit verbose. How about using `part` instead of `particle` and `pos` instead of `position`. We could even use `m` instead of mass, abbreviate things further:

```
begin ACS
  begin part
    m = 0.1 ; r = 1.0 0.0 0.5 ; v = 0.0 1.0 0.0
  end
end
```

Alice: nonono, that's really bad programming. Right now, we know what we mean, but later on, when we get stellar evolution to mix with stellar dynamics and who knows what other types of complications, we will soon run out of the 26 letters of the alphabet! I feel strongly that we should keep using full English words.

Bob: your prediction that we would soon disagree has come true even quicker than I would have thought! I must say, I really like the compact notation. But I don't seem to feel as strongly about it as you do. Oh, well, let's stick with the longer descriptions then.

Alice: And come to think of it, `ACS` is too short a name too. Something like `ArtCompSci` or even `art-of-computational-science` would be much better.

Bob: Now you're really pushing it! And it is my turn to feel strongly; I like to be able to talk about what I use, and to talk about an ACS file rolls off the tongue much more easily than talking about artcompsci files, let alone art-of-computational-science files.

Alice: Before we settle this issue, let us think ahead a bit further. The reason to choose the name ACS is that we want to introduce our ideas as templates for use in computational science in general, not only in astrophysics. If a chemist or biologist will start using our system, in future extended versions, they are likely to use a world like `Particle` for their own purpose.

Bob: So you are suggesting an extra level of headers, effectively something like a type of `namespace`, as you have in C++?

Alice: Yes, that would be a good idea, I think. And in that case, I wouldn't mind keeping the short version `ACS`. Even if for some reason some people would introduce files for Advanced Computational Software that would also be called `ACS`, and if we and they would start using XML, there would be no confusion. If we would get one of their files, than the next line would not have the proper name space tag that we require for our particular use.

1.6 Compromise

Bob: I'm glad you found a compromise! But how many levels of name space do you want to introduce? I could argue that some day you will do hydrodynamics with SPH particles, that may have a rather different structure than our particles. If we give you total freedom, I bet you will come up with something unwieldy pretty soon.

Alice: Let me not disappoint you. If we introduce another abbreviation, `DSS` for simulations involving dense stellar systems, surely short enough for your taste, then how about:

```
begin ACS
  begin astrophysics
    begin DSS
      begin stellar_dynamics
        begin particle
          mass = 0.1
          position = 1.0 0.0 0.5
          velocity = 0.0 1.0 0.0
        end
      end
    end
  end
end
```

Bob: Are you serious?

Alice: Well, a typical data file will contain quite a number of particles, possibly with many more variables than we have here, so the total length of the file won't change much, even with these three levels of name spaces in between `ACS` and `particle`.

Bob: Even so, this is too much of a proliferation, certainly at this stage.

Alice: Perhaps. I don't insist that we will implement all these levels right away, since we don't know at this stage how our system will evolve. But at least we would be prepared to move in the direction of a full hierarchy of name spaces, if the need arises. And this implies that we should not make any design decisions that would make a future implementation difficult.

Bob: Fair enough. And I am willing to make a compromise by allowing the `DSS` level of name space to be present right now. It is indeed short, and it tells us about the broad type of application within ACS. So this will give us:

```
begin ACS
  begin DSS
    begin particle
      mass = 0.1
      position = 1.0 0.0 0.5
      velocity = 0.0 1.0 0.0
    end
  end
end
```

Alice: If we have to add levels of name spaces in the future, we'll have to write a few short conversion programs. One will add the extra levels to old data files, to make them compatible with new programs. Another conversion program will subtract the extra levels from new data files in order to be read by old programs.

Bob: That should be easy to do.

1.7 Hierarchical particle structures

Alice: Now that we can define a single particle, we also need a way to define a whole N-body system.

Bob: Introducing a tag `System` would be confusing, since it would be too general a name. How about `Nbody`?

Alice: Here is an idea. We can use the same tag `Particle` in an hierarchical way. A two-body system could be written as:

```
begin ACS
  begin DSS
    begin particle
      begin particle
        mass = 0.1
        position = 1.0 0.0 0.5
        velocity = 0.0 1.0 0.0
      end
      begin particle
        mass = 0.3
        position = -1.0 0.0 0.5
        velocity = 0.0 -0.2 0.0
      end
    end
```

```
  end
end
```

Bob: I see. And the higher-level particle could be interpreted as the center-of-mass particle of the two-body systems. It could carry its own information. In this case that would be:

```
begin ACS
  begin DSS
    begin particle
      mass = 0.4
      position = -0.5 0.0 0.5
      velocity = 0.0 0.1 0.0
      begin particle
        mass = 0.1
        position = 1.0 0.0 0.5
        velocity = 0.0 1.0 0.0
      end
      begin particle
        mass = 0.3
        position = -1.0 0.0 0.5
        velocity = 0.0 -0.2 0.0
      end
    end
  end
end
```

Alice: Yes, perfect.

1.8 Extended Tag Names

Bob: One problem with using the name `particle` hierarchically is that it can lead to confusion as to which `particle` is which. For example, if you want to set up two galaxies, and let them approach each other to simulate a collision, each galaxy would be represented by one `particle` structure that would in turn contain many `particle`'s, one for each body used in the simulation. And the whole system would be represented by a top `particle`.

Alice: Ah, here is an idea! We can allow a second name in our `particle` tag, to indicate the particular type of `particle` we are dealing with. Your example would then become:

```
begin ACS
  begin DSS
```

```
    begin particle simulation
      begin particle galaxy
        begin particle star
        end
        begin particle star
        end
        . . .
        begin particle star
        end
      end
      begin particle galaxy
        begin particle star
        end
        begin particle star
        end
        . . .
        begin particle star
        end
      end
    end
  end
end
```

Bob: Great idea. In that way, programs that only need to know that a particular object is a `particle` can just read the first word in the tag. Other programs that need more information, or a human reader inspecting the data file, can read the subsequent information; we might even allow arbitrary many tags. An star on the asymptotic giant branch could be:

```
begin ACS
  begin DSS
    begin particle star giant AGB
      mass = 0.1
      position = 1.0 0.0 0.5
      velocity = 0.0 1.0 0.0
    end
  end
end
```

Alice: I like that. I think we are getting there.

Chapter 2

Particle Output

2.1 Getting Started

Bob: Let's get some actual work done, after all our talking, last time. Shall we code up an I/O implementation of our ACS data format we designed yesterday?

Alice: Yes. We definitely need to see a working model, before we can go any further. Let's start with the `Body` class we have been using so far for a single particle. We may as well strip off everything but the I/O part, and see whether we can rewrite that into our new format.

Bob: Okay. Let's call it `iobody.rb`. And since we will keep adding and changing things, it is probably a good idea to keep a number of versions around, so that we can always go back to see what we did earlier. We can start with a file called `iobody1.rb`, and after we have some rudimentary functionality we just freeze it, and don't modify it anymore. From then on we will work on `iobody2.rb`, and so on.

Alice: That's a good idea, to keep a trail of previous versions.

Bob: So here is our starting point, a copy of what we did before, which we can call `iobody0.rb`:

```
require "old_vector.rb"

class Body

  attr_accessor :mass, :pos, :vel, :acc

  def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
    @mass, @pos, @vel = mass, pos, vel
  end
```

```
  def to_s
    "  mass = " + @mass.to_s + "\n" +
    "   pos = " + @pos.join(", ") + "\n" +
    "   vel = " + @vel.join(", ") + "\n"
  end

  def pp               # pretty print
    print to_s
  end

  def simple_print
    printf("%24.16e\n", @mass)
    @pos.each{|x| printf("%24.16e", x)}; print "\n"
    @vel.each{|x| printf("%24.16e", x)}; print "\n"
  end

  def simple_read
    @mass = gets.to_f
    @pos = gets.split.map{|x| x.to_f}.to_v
    @vel = gets.split.map{|x| x.to_f}.to_v
  end

end
```

2.2 A Single Write Method

Alice: Ah, look, we started with two different ways of outputting our data: we had a pretty way to list the data, using the `pp` command, for pretty printing, and we had a raw way to dump all the significant digits, using `simple_print`.

Bob: Yes, it's all coming back now. And given that we have introduced a self-describing format, I guess we don't need two different methods anymore.

Alice: Even so, it would be nice to control the number of digits. A human reader may want to see only a few of the most significant digits, whereas you need full double precision when you want to pipe data in and out of programs read by the computer, so that you don't loose accuracy.

Bob: But instead of writing different methods, it would be better to have only one method, with the number of digits as an argument. Also, let us follow the idea shown above with `to_s`, let us write the output data first onto a string. We can then use another method to print out that string, or to write it to a file, as the case may be.

Alice: Good idea. And by calling that method `to_s`, we can type `print b` for a particle `b`, since the Ruby command `print` by default looks for a member function `to_s`.

Bob: Let us recall how we want a single particle to appear in the output. We had decided on:

```
begin ACS
  begin DSS
    begin particle star giant AGB
      mass = 0.1
      position = 1.0 0.0 0.5
      velocity = 0.0 1.0 0.0
    end
  end
end
```

Let's not worry about how to print the first two lines; those will be taken care of by a higher-level function. What our `to_s` should do is just write the middle five lines, the contents of a `Body` instance.

Alice: But with the proper indentation, which will depend on information that is only available from outside the `iobody.rb` context. The calling function should provide the base amount of indentation, to start with.

Bob: Indeed. And while we're giving the users the freedom to specify the numbers of digits of precision, we may as well allow them to specify the incremental indentation between the `begin particle` line and the subsequent lines. How about something like

```
def to_s(precision = 16, base_indentation = 0, additional_indentation = 2)
. . .
end
```

Alice: Good. That makes it clear that we intend to give 16 digits of precision by default, enough to cover double-precision notation. You're quick at figuring out how to implement this, why don't you fill in the dots?

2.3 Writing `to_s`

Bob: How about this? As we discussed, I've put this now in file `iobody1.rb`:

```
def to_s(precision = 16, base_indentation = 0, additional_indentation = 2)
  subtag = if @type then " "+@type else "" end
  indent = base_indentation + additional_indentation
```

```
    return " " * base_indentation + "begin " + TAG + subtag + "\n" +
      f_to_s("mass", mass, precision, indent) +
      f_v_to_s("position", pos, precision, indent) +
      f_v_to_s("velocity", vel, precision, indent) +
      " " * base_indentation + "end" + "\n"
  end
```

Alice: I see that you return the string with all the output information in the last logical line, which is actually wrapped over the last five lines before the end.

Bob: Yes. I start by adding `base_indentation` number of blank spaces. Ruby, with the principle of least surprise, lets you do that by typing `" " * base_indentation`.

Alice: I find it quite surprising that you can just multiply a string with a number in such a simple way, because I'm not used to that convenience in other languages. But you're right, it does look very natural.

Bob: I then have to provide the main tag `particle`, which I have encoded as a `Body` class constant, by adding the following line to the `Body` class:

```
TAG = "particle"
```

The rest of the tag, in our example `star giant AGB`, which I call the type, in our case the type of particle, I assume will be stored in an instance variable `@type`. By default, when you create a vanilla flavor `Body` instance, there is no extra type information, so `@type = nil`. I added `@type` to the list of accessor macros:

```
attr_accessor :mass, :pos, :vel, :acc, :type
```

If a type is specified, then the string `@type` is inserted after the string `TAG`, with a space in between, as you can see in the `if` clause; the `else` clause does not add anything.

Alice: I must admit, that first line in `to_s` is a bit confusing, but I guess I can make sense of it. What appears to the right of the `=` sign is a normal if-else construction, but without the usual indentation.

Bob: Yes, it seemed a bit wasteful of space to use five lines for what can be easily written in just one line. But note that I added the word `then`, which you don't use when you write it over several lines. Ruby insists on using `then` for inline constructs like this, since otherwise it would not know how to separate the condition from the resulting action.

Alice: But I'm surprised that you can just assign the results of the if-else construction to a variable.

Bob: A nice feature of Ruby, which will feel very natural once you have used it a few times. Here is what I could have written more explicitly:

```
if @type
  subtag = " "+@type
else
  subtag = ""
end
```

In inline-version that would have become

```
if @type then subtag = " "+@type else subtag = "" end
```

But don't you think this is more short and simple:

```
subtag = if @type then " "+@type else "" end
```

Alice: Shorter yes, but simpler only once you get used to it. Okay, I see what is happening in this method. You have postponed the real work to the two methods `f_to_s` and `f_v_to_s`. A nice example of top-down programming!

2.4 Methods f_to_s and f_v_to_s

Bob: The real work is actually very simple, since we've done it already in our previous version. Here is the first method:

```
def f_to_s(name, value, precision, indentation)  # from floating-point number
  " " * indentation +
    name + " = " + sprintf(" %#{precision+8}.#{precision}e\n", value)
end
```

Alice: So I guess `f` stands for floating-point format, and `f_to_s` indicates a conversion from a floating point number to a string. That makes sense, as a first step toward the more general `to_s` with converts the whole `Body` content to a string. In fact `to_s` could be called `body_to_s`.

Bob: Ah, but here is where Ruby's method notation shines: you invoke the method `to_s` for a particular `Body` instance `b` by writing `b.to_s`, which when you read it aloud sounds like b-to-s, and does what you expect it to do.

Alice: You're right. It is all very logical and consistent – and concise as well. I like it.

Bob: Here is the second method:

```
def f_v_to_s(name, value, precision, indentation)   # from floating-pt vector
  " " * indentation + name + " = " +
    value.map{|x| sprintf(" %#{precision+8}.#{precision}e", x)}.join + "\n"
end
```

Alice: I see. Earlier we have used a `to_v` method as an extra method for the class `Array`, which is in fact a type of a-to-v method, or array-to-vector. But as you reminded me, a particular array `a` will be converted by writing `a.to_v` which sounds just right. And now you are using the same logic to define a v-to-s method, from vector to string.

Bob: Yes, and I thought it would be more consistent to stress the fact that we are not dealing with any type of vector, but with a vector that has floating point values in it. Hence the name `f_v_to_s`.

Alice: But we use vectors exclusively for physical quantities, that are always represented as floating point variables. Is it really necessary to add this `f_` to stress that we are dealing with floating point numbers? You could as well write `f_p_n_v_to_s` for floating-point-number-valued-vectors.

Bob: Ah, but look at the definition of the `Vector` class; you will find no mention there of floating point variables. So it does make sense to add that we are doing an extra conversion. You can also look at the `simple_read` input method that we defined before. The position, for example, was read in as follows:

```
  @pos = gets.split.map{|x| x.to_f}.to_v
```

So you see, from that point of view it is natural to make a combination like `to_f_v`, as we will undoubtedly do later on in our new read method. For our write method this means that `f_v_to_s` is natural.

Alice: I see your point. But how about making it a bit more compact, like `fv_to_s`? I would prefer that, it is visually more pleasing.

Bob: But logically less correct, I would say.

Alice: Hmm, I don't think so. But you wrote it, and it's not that important, so let's do it your way.

2.5 Testing

Alice: We still need a method to do the actual output. Let me try something. How about this:

```
  def write(file = $stdout, precision = 16,
            base_indentation = 0, additional_indentation = 2)
    file.print to_s(precision, base_indentation, additional_indentation)
  end
```

Bob: Yes, that should work. By default this will print to the standard output, and if you provide a file name, the output will be stored in that file.

Alice: Let's test it. Here is a test file `test.rb`

```
require "iobody1.rb"

b = Body.new(1, [2,3], [4.5, 6.7])
b.write
```

And here is the result:

```
|gravity> ruby test.rb
begin particle
  mass =    1.0000000000000000e+00
  position =    2.0000000000000000e+00   3.0000000000000000e+00
  velocity =    4.5000000000000000e+00   6.7000000000000002e+00
end
```

Bob: Looks good! Let's give it a more modest accuracy. Given the order of the arguments to `write`, this means that we now have to explicitly supply the file name `stdout`:

```
require "iobody1.rb"

b = Body.new(1, [2,3], [4.5, 6.7])
b.write($stdout, 4)
```

Let's test it:

```
|gravity> ruby test.rb
begin particle
  mass =    1.0000e+00
  position =    2.0000e+00   3.0000e+00
  velocity =    4.5000e+00   6.7000e+00
end
```

Alice: Just what it should be. Let's see whether the indentation works:

```
require "iobody1.rb"

b = Body.new(1, [2,3], [4.5, 6.7])
b.write($stdout, 4, 20, 4)
```

```
|gravity> ruby test.rb
                    begin particle
                        mass =    1.0000e+00
                        position =    2.0000e+00   3.0000e+00
                        velocity =    4.5000e+00   6.7000e+00
                    end
```

Bob: Perfect. I think we've done enough writing now. Time to start reading in our new data format!

Alice: I agree. But just to see the whole landscape, can you show me what the file `iobody1.rb` looks like now?

2.6 The File `iobody1.rb`

Bob: My pleasure:

```
require "old_vector.rb"

class Body

  TAG = "particle"

  attr_accessor :mass, :pos, :vel, :acc, :type

  def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
    @mass, @pos, @vel = mass, pos, vel
  end

  def to_s(precision = 16, base_indentation = 0, additional_indentation = 2)
    subtag = if @type then " "+@type else "" end
    indent = base_indentation + additional_indentation
```

```
    return " " * base_indentation + "begin " + TAG + subtag + "\n" +
      f_to_s("mass", mass, precision, indent) +
      f_v_to_s("position", pos, precision, indent) +
      f_v_to_s("velocity", vel, precision, indent) +
      " " * base_indentation + "end" + "\n"
  end

  def f_to_s(name, value, precision, indentation)  # from floating-point number
    " " * indentation +
      name + " = " + sprintf(" %#{precision+8}.#{precision}e\n", value)
  end

  def f_v_to_s(name, value, precision, indentation)   # from floating-pt vector
    " " * indentation + name + " = " +
      value.map{|x| sprintf(" %#{precision+8}.#{precision}e", x)}.join + "\n"
  end

  def write(file = $stdout, precision = 16,
            base_indentation = 0, additional_indentation = 2)
    file.print to_s(precision, base_indentation, additional_indentation)
  end

end
```

Chapter 3

Using the to_s Method

3.1 xxx

aha:

```
require "old_vector.rb"

class Body

  TAG = "particle"

  attr_accessor :mass, :pos, :vel, :acc, :type

  def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
    @mass, @pos, @vel = mass.to_f, pos.to_v, vel.to_v
    @type = nil
  end

  def to_s(precision = 16, base_indentation = 0, additional_indentation = 2)
    subtag = if @type then " "+@type else "" end
    indent = base_indentation + additional_indentation
    return " " * base_indentation + "begin " + TAG + subtag + "\n" +
      mass.to_s("mass", precision, indent) + "\n" +
      pos.to_s("position", precision, indent) + "\n" +
      vel.to_s("velocity", precision, indent) + "\n" +
      " " * base_indentation + "end" + "\n"
  end

  def write(file = $stdout, precision = 16,
            base_indentation = 0, additional_indentation = 2)
```

```
    file.print to_s(precision, base_indentation, additional_indentation)
  end

end
```

aha:

```
: inccode: .vector.rb
```

aha!

Note: to_f and to_v in initializer. Let them find the bug themselves!

Let us run the same test as before:

```
require "iobody2.rb"

b = Body.new(1, [2,3], [4.5, 6.7])
b.write
```

And here is the result:

```
|gravity> ruby test.rb
begin particle
  mass =    1.0000000000000000e+00
  position =    2.0000000000000000e+00   3.0000000000000000e+00
  velocity =    4.5000000000000000e+00   6.7000000000000002e+00
end
```

and also the more complex test:

```
require "iobody2.rb"

b = Body.new(1, [2,3], [4.5, 6.7])
b.write($stdout, 4, 16, 4)
```

```
|gravity> ruby test.rb
                begin particle
                    mass =    1.0000e+00
                    position =    2.0000e+00   3.0000e+00
                    velocity =    4.5000e+00   6.7000e+00
                end
```

Chapter 4

Nested Output

4.1 ACS Output

4.2 Nbody Output

4.3 Example: Plummer's Model

Nice to get some real results already!

Chapter 5

Particle Input

5.1 How to Read

Alice: Hi, Bob, time to write some input routines?

Bob: I already got started. However, I realize that input is quite a bit more complicated than output, because the input routine has to recognize hierarchical structures properly.

Alice: In general, the problem with input is that you have no control over what you might find. While doing output you can decide to just write positions and velocities, for example, but while reading in an existing file, you may find that someone wrote accelerations as well.

Bob: Good point. And if we pipe the data from one program to the other, we don't want to loose any data. So it would be best to let the input routine read in everything, and to let the output routine simply echo whatever it did not understand.

Alice: I agree. We'll have to add that to the `to_s` method that we just wrote. But let us first write the input method, so that we can see what form this blind data handling will take. What have you written so far?

Bob: I started a new file `iobody3.rb`, in which I first copied what we had done already in `iobody2.rb`. The last thing we did there was to write a `write` function, for output to a file or to the standard output. The natural thing to do next was to write a `read` function. As before, the default input choice would be the standard input, but you can also give a file name instead, to read data from a file.

Remember that our `write` method looked like this:

```
def write(file = $stdout, precision = 16,
```

```
            base_indentation = 0, additional_indentation = 2)
    file.print to_s(precision, base_indentation, additional_indentation)
  end
```

So my first thought was that the `read` method should have the following simple form:

```
  def read(file = $stdin)
  . . .
  end
```

since precision and indentation will be already provided by the data that are being read in.

Alice: Sounds plausible.

5.2 Passing the Type

Bob: However, I realized that we need an extra argument. It took me a while to figure this out, and I saw it only when I tried to imagine how the method `read` will be called. At some higher level, a command will be given to read a whole data file, which will start as something like

```
begin ACS
  begin DSS
    begin particle globular_cluster
    . . .
      begin particle star giant AGB
        mass = 0.1
        position = 1.0 0.0 0.5
        velocity = 0.0 1.0 0.0
      end
    . . .
    end
  end
end
```

On that level, the first two lines will be read and discarded, since they only function as a safety check, guaranteeing that we are dealing with the right type of data file. This higher-level method will then read in the line

```
    begin particle globular_cluster
```

and only at that point it becomes clear that the `read` method of the `Body` class has to be invoked. But at that point the extra information containing the type of the particle has already been read in. The third line tells us that the particle stands for the center of mass of a globular cluster, which contains many other particles. And the next line that starts with `begin particle` contains our previous friend, the asymptotic giant branch star.

Alice: I see what you mean. In the case of that last star, the method `read` for the star will be asked to start reading the next input lines, and it will only see

```
  mass = 0.1
  position = 1.0 0.0 0.5
  velocity = 0.0 1.0 0.0
end
```

It can hand control back to the calling function when it encounters the first `end`, but indeed it will never know that it was an AGB star.

Bob: Exactly. So that additional information has to be passed to the star. The easiest and most general way is to pass the whole last line that was read in by the calling function. I therefore decided that the `read` method should have the form

```
def read(header, file = $stdin)
. . .
end
```

where in the case of our star the variable `header` contains the whole line:

```
begin particle star giant AGB
```

Alice: That makes sense.

Bob: By the way, you mentioned that control can be handed back to the calling function when the first `end` is encountered, but that is not quite true. As we saw above, a single `particle` block can contain several other `particle` subblocks. For example, when the calling function asks the first particle, of type `globular_cluster`, to read in its data, that particle will encounter the stars that belong to the globular cluster. For each star, it will call the `read` method for the appropriate `Body` instance, and continue after that method returns. But since that star's `read` method has gobbled up its own `end` line, the globular cluster `read` method will never see those embedded `end` lines. Indeed, the first `end` it encounters will be its own proper `end`.

Alice: So I was right, but for the wrong reason.

Bob: It's always nice to be lucky!

Alice: I'd rather be right, though. Can you show me your `read` implementation?

5.3 Handling the Header

Bob: Here it is:

```
def read(header, file = $stdin)
  raise unless header =~ /^\s*begin\s+particle/
  a = header.split
  if a.size > 2
    a.shift
    a.shift
    @type = a.join(" ")
  end
  loop {
    s = file.gets
    name = s.split[0]
    case name
      when /^mass/
        @mass = s_to_f(s)
      when /^position/
        @pos = s_to_f_v(s)
      when /^velocity/
        @vel = s_to_f_v(s)
      when "begin"
        subread(file, s)
      when "end"
        break
      else
        raise
    end
  }
end
```

Alice: I see. First you process the `header` which tells you what type of particle you are dealing with and then you enter into a loop called `loop` that processes subsequent lines from the file or input stream, until you encounter a final `end`.

Bob: Yes, and the first thing to do is to check whether the `header`, which is really an echo of the last line read, really starts with the words `begin particle`. In general there will be white space in front, because of the indentation, and one or more pieces of white space in between `begin` and `particle` but everything else is only allowed to follow those two words.

What you see at the right hand side of the first line of the method is the `raise` command, which raises an error if the `header` does not have the correct form. In Ruby, `unless` is the opposite of `if`, a nice feature, since otherwise we would

have needed to say "if not of the correct form" and "unless of the correct form" is more natural.

The combination `=~` compares a string on the left with a regular expression on the right, and returns a `nil` value if the string does not correspond to the regular expression, with is surrounded by two slashes.

Alice: Let me try to remember my regular expressions. The first up arrow `^` means that it must match the beginning of the string. Then `\s` stands for a white space, a blank or a tab, followed by a `*` which tells us that we can expect zero or one or more of those white spaces, while the `+` after the second `\s` tells us that there should be at least one, and possible more, white spaces between `begin` and `particle`.

So for a string to match this regular expression, it should start with zero or more white spaces, followed by the word `begin`, followed by one or more white spaces, followed by the word `particle`, and optionally followed by whatever else you like.

Bob: Indeed. Now if that test is passed successfully, the header is split of by the `String` method `split` into an array of words, as we have seen before. If there are only the two words `begin` and `particle` no further action is taken. But if there is at least one more word, then all those extra words are glued together with `join`, to form the type of the particle, stored in the instance variable `@type`. We have to give `join` an argument to separate the individual words, and a single blank space is the most natural choice.

5.4 Handling Other Lines

Alice: So now we enter a loop, I presume. What does `loop` mean?

Bob: It stands for an endless loop. In C you would write `while(1)` or `for{;;}` depending on your taste, but Ruby has a more clean and direct construct. The only way to leave the loop `loop` is to break out explicitly: you break with `break`.

Alice: Ruby seems to do exactly what you tell it to do! The `case` statement reminds me of the `switch` statement in C and C++.

Bob: Yes. In this case, `case` is used to compare the first word of each new line with the three legal choices we have so far for the names of input and output data: mass, position and velocity. When one of these choices is encountered, the appropriate helper method kicks in, a simple translator to floating point for the mass, and a translator to a floating point vector for position and velocity. Here is the first one:

```
def s_to_f(s)                                   # string to floating-point number
  s.split("=")[1].to_f
end
```

This time we give `split` an explicit argument. The default has been to consider white space as the separator between words, but now we use an equal sign `=` as a separator. The name of the variable, in our case the string `"mass"` is returned as the value of `s.split("=")[0]`. Everything to the right of the equal sign, in our case the value of the mass, is returned as `s.split("=")[0]`, and promptly converted to floating point format with `.to_f`.

Alice: Everything, unless there are more equal signs, in which case the following pieces would wind up in `s.split("=")[2]`, etc.

Bob: True. Normally there should be one and exactly one equal sign, and I could have checked that by checking and raising an error condition in case `s.split("=").size != 2`. However, at this stage I'm happy to live a bit dangerously. And besides, the extra stuff will be ignored, since it is not used.

Alice: Even so, I wouldn't want to continue if there is was an second equal sing in the input line, since that would mean that there would be something very seriously wrong. Let's get back to error handling soon; we really should teach the students some defensive programming, instead of just assuming that the world is a paradise, and that nobody will hand you wrong data.

Bob: I agree. But first things first: let's get things to work. Here is my vector version:

```
def s_to_f_v(s)                                 # string to floating-point vector
  s.split("=")[1].split.map{|x| x.to_f}.to_v
end
```

As before, it uses the `map` method to convert each component of the vector from a string to a floating point number.

Alice: And when you encounter a new `begin` before you have reached the `end` of the current particle data input, you interpret that as a new particle that is contained in the current particle, in the same way that a star is contained in a star cluster.

Bob: Indeed. Such a new particle should normally be indented one level further, since it lies one level deeper in the hierarchy of particles, but I don't check for that, since I don't want to insist on a particular style of indenting. Style is nice for human readers, but computers should be able to handle anything logically reasonable.

Alice: And `subread` is the method that handles those particles deeper in the hierarchy, under the current particle. What does it do?

Bob: I don't know yet. I just put it there as a stub, a place holder to remind us that we still had some work to do there. Hmm. Now that you ask me, I guess

it should just be `read` again, a recursive invocation of the same method that we are currently executing.

Alice: Yes, but you first have to know where to put the date you read for the daughter particle. I think you will have to create such a particle, to start with, and then hand execution to that particle.

Bob: That sounds right. Let's do that in a moment.

Alice: Well done! This is clean and elegant, but I'm still worrying about handling unusual lines. If somebody hands you a line with

```
acceleration = 0.1 -0.3
```

your program will raise an error. But there is nothing wrong with providing extra information. How about creating a little scratch pad where you store all the lines for which you did not know what to do with them. You can just keep this scratch pad lying around, and by the time you do an output, you echo its contents so that the information is passed on correctly to the next function, which may know what to do with it.

Bob: That is a good idea too. Let's do that in a moment too, in another moment.

5.5 Testing

Alice: Did you test your code?

Bob: Not yet, but it's high time. Let's try a write-read cycle.

Alice: You mean, you will output the content of one `Body` instance with its `write` method and then read it in again into another `Body` instance using that instances `read` function?

Bob: Conceptually, yes, but in practice it is more complicated. Remember that we haven't yet written the higher level function that handles the top level lines `begin ACS` and all that. That function gobbles up not only those higher levels lines, but also the header of the particle data. Let's see.

Alice: Ah, of course, you're right. This means that we will have to pass the header by hand, for now, in the `read` command, after which we can feed in the remaining lines from the standard input, or from a file, if we wish.

Bob: Exactly. Let's check it, by writing another test file `test.rb`:

```
require "iobody3.rb"

b = Body.new
b.read("begin particle star giant AGB")
b.write
```

And here is the result:

```
|gravity> ruby test.rb
   mass =      1
   position =  2  3
   velocity =  4.5  6.7
   end
begin particle star giant AGB
  mass =    1.0000000000000000e+00
  position =    2.0000000000000000e+00   3.0000000000000000e+00
  velocity =    4.5000000000000000e+00   6.7000000000000002e+00
end
```

Alice: Just what it should be. After typing in those four lines, with whatever indentation, `b.write` generates the right output. Congratulations!

Bob: Let's see what happens if we give a wrong input line.

Alice: Okay. How about giving it a supersymmetric particle, for a change?

```
require "iobody3.rb"

b = Body.new
b.read("begin sparticle")
b.write
```

And here is the result:

```
|gravity> ruby test.rb
./iobody3.rb:40:in `read': unhandled exception
from test.rb:4
```

Bob: It works, but I admit, this type of error message is less than helpful.

Alice: A third thing to do in yet another moment. So here is our to-do list:

- implement a scratch pad
- write the `subread` method
- improve the error handling

Bob: And then of course we have to write the higher level read functions to. Okay, one thing at a time!

Chapter 6

A Scratch Pad

6.1 Extra Information

Bob: Our first to-do time was to implement a scratch pad, to contain unrecognized data that can be read in, and to reproduce those safely at the time of output. I will copy the file `iobody3.rb`, with which we have been working so far, into a new file `iobody4.rb`.

Alice: Let's call the scratch pad `rest`, since it will contain the rest of all that we read in, whatever doesn't fit our expectations. We can make it one big string, to which we keep adding whatever input line that we don't recognize.

I don't think we should put any *a priori* limitations on what such lines could contain. Obvious choices are lines like

```
acceleration = 0.1 -0.3
density = 345.18
```

for physical quantities, or something like

```
neighbors = 15 18 23
```

for a list of nearest neighbors, with particles identified by their numbers, or in any other way for that matter, for example:

```
neighbors = star5  star7  GMC3  triple8
```

As long as other appropriate programs can handle that format, the I/O routines for the `Body` class don't have to worry about them.

Bob: But how do you know which line makes sense? You don't have the information to know what all the other programs can handle. What if someone was planning to write an email, reporting on a run, and somehow by mistake got the text mixed up with the input for a `Body` instance. Surely it would be an error to read in:

```
velocity = 1 0
acceleration = 0.1 -0.3
Hi Joe!  Look what a cool run I just produced.
This AGB star is out of control!  Man, it's
evolving with a time step much less than a year,
and shrinking.  This way we'll never reach the
horizontal branch.  Meanwhile, wanna come over
for a beer?
```

Alice: Actually, this does not have to count as an error at all. What if the writer is planning to send Joe a data set, and would want to communicate to Joe what the data are all about? It would be much safer to add this message to the actual data file, rather than sending it in a separate email. We all know how many emails we get every day, and how difficult it is to retrieve the right one, after weeks or months.

Bob: But there is no way of knowing the form that a human narrative can take. People might write `!@#*!!` if they are in a bad mood. By letting people write what they want you would allow literally any line!

6.2 Two Possibilities

Alice: Why not?

Bob: I know a good reason why not: next thing you know is to see someone type `begin position`, for example the last three lines above could have equally well been:

```
and shrinking.  This way we'll never reach the
begin position of the horizontal branch.  Meanwhile,
wanna come over for a beer?
```

Remember that we allowed both

```
position = 1 2
```

and also

```
begin position
  1 2
end
```

So the above version of the email chat would be very dangerous.

Alice: You have a point there. So we have to be a bit more careful. How about allowing only two types of 'rest' lines: either such a line should have an equal sign in it, to indicate that it is of the form `name = value`, or its first word should be `begin`, in which case we read in everything until we encounter the next same-level `end`.

So the above data could take as a legal form:

```
velocity = 1 0
acceleration = 0.1 -0.3
begin story
  Hi Joe!  Look what a cool run I just produced.
  This AGB star is out of control!  Man, it's
  evolving with a time step much less than a year,
  and shrinking.  This way we'll never reach the
  begin position of the horizontal branch.  Meanwhile,
  wanna come over for a beer?
end
```

Bob: I like the idea of extending the notion of self-describing data. Your story idea will introduce self-chatting data!

Alice: I prefer self-narrating data. Good! Shall we introduce a `Story` class?

Bob: Huh? Why? We had just decided that we will put all the *rest* lines on a big pile, and store that in a string called `rest`. Why would you suddenly want to give extra structure to that string??

Alice: Because we have to distinguish the story structure above from other unrecognized structures. For example, imagine that some other program includes information about a few multipoles for the internal structure of our star. That might take a form such as:

```
begin multipoles
  begin monopole
    1.5
  end
  begin quadrupole
    0.3
  end
end
```

Now there are two possibilities. Here is the first one. You can read in the *rest* data in a hierarchical way, in which you keep track of how many levels deep you go with the `begin` and `end` statements. In that case you can read the complete multipole information, even if your program has not the foggiest idea of what multipoles are, just by counting levels and stopping when you encounter the first `end` on the same level as the `begin` that was associated with `multipoles`.

Using the first possibility, however, will let your program crash when you try to read in the narrative above, starting with `begin story`. The input mechanism sees `begin position` later on, and it will presume that the last `end` belongs to the same level of `begin position`, so it will keep searching for the extra `end` that it expects to correspond to `begin story`.

The second possibility is to ignore any `begin` statement at the start of a line, and just to keep reading on till you find an `end`, all alone on a line. That would solve the problem for the story above. And in practice, even if you would write a single "end" at the end of a sentence, most likely you would put a period or question mark or exclamation mark after the end.

So the second possibility is pretty safe, although not completely safe, as far as a story goes. But what is much worse, this second method fails miserably when reading in the multipoles. It would stop at the first `end`, then assume that `begin quadrupole` would be a new item, read that in until its proper `end`, and then it would encounter the final `end`. At that point it would think it had read the whole particle structure, which is not the case.

6.3 More Possibilities

Bob: I see. Hmmm. That is tricky. And you were thinking to solve that by introducing a special `Story` class, for which you use the second solution, whereas you use the first solution for all other cases?

Alice: Yes, that was my first thought. But now that I have put it all on the table, I'm not so sure whether that would be a good solution. For one thing, it is not completely fail-safe: an email, say, can indeed contain the word `"end"` instead of `"end."`. Many young people these days seem to completely ignore punctuation.

Bob: And many old people can leave out a period, as a typo, especially when they forget to bring their reading glasses.

Alice: Just you wait! Before too long you'll have to choose between bifocals and reading glasses. But I guess we both agree that my initial thought is not safe enough. Hmmm.

Bob: There is another problem with your initial thought. What if you want to keep a log of previous commands, as in the Unix history mechanism? You might want to include that as

```
begin history
  make_binary -M 2 -m 3
  integrate -t 10
  find_orbital_elements
end
```

It is possible that such a list of commands would include a command called `begin`. Why not? So it is not only the `begin story` that would need a `Story` class. you would need a `History` class, and so on, one class each for each different type of application. I don't like that.

Alice: Well, what else do you propose?

Bob: One possibility would be to check indentation. If you encounter an `end` on the same level of indentation as the rest of the lines within a block, it does not mean that the block ends; it only counts as a real end if it is indented by one or more spaces less. Similarly, a `begin` should only signify the beginning of a new block if the next line is indented by one or more spaces, compared to the line starting with `begin`.

Alice: But then the higher-level program should pass more than just the header, as we have implemented above. It should read one more input line before it can decide that `begin particle` really meant the start of a new particle structure, or whether it was part of a chatty email as we saw above. But once it had read in that line, it has to pass it to the `read` function of `Body`. So we would get something like

```
def read(header, next_line, file = $stdin)
. . .
end
```

And what is worse, instead of going directly into the loop `loop`, you would have to first process this `next_line` before you can pick up more lines with `file.gets`

Bob: I agree, that will make things ugly. Hmmm again. Well, perhaps we can invent more complex words than `begin` and `end`. If we write `acs_begin` or even `!@#*!!_acs_begin` and similarly `!@#*!!_acs_end` we would be safe enough. What is the chance that someone would type those combination of characters by chance in a chatty email?

Alice: Not in an normal email, no. But if someone will include a piece of natural language text to explain what a bunch of data represent, and if the data are in our acs format, chances are that that person may also explain how to read and write those data. And, guess what, that person will have to write *exactly* the expressions `!@#*!!_acs_begin` and `!@#*!!_acs_end` . . .

Bob: Yes, that is a catch. I don't see an way around that. But hey, wait, there is a way! We can ask this person to write something like \`!@#*!!_acs_begin`, and provide a way to translate that into the proper `acs_begin` when the story is being processed for a human reader.

But the more I think about that, the more I dislike the idea. Who would want to look at files that have `\!@#*!!_acs_begin` and `\!@#*!!_acs_end` everywhere in them? ACS will get a bloody bad reputation with what looks like curse words sprinkled in everywhere.

6.4 A Box

Alice: It seems that we're running out of options.

Bob: And yet we have to solve it, at least if we want to allow self-narrating data. I must say, I got warmed to the idea, and I don't like to give that up, just because we have some difficulty figuring out how to implement it.

Alice: You said you didn't like to make a whole slew of exceptional cases, for `begin story` and `begin history` and what not. Here is an alternative. Let us protect the content of a story or a history or whatever by somehow putting it into a safe box, wrapping it up in something . . .

Bob: . . . by putting four lines around the text as in a children's drawing? I wish we could do that.

Alice: A line!

Bob: A line?

Alice: You found the solution! Or more accurately, one quarter of what you just found is the solution. We need to put a vertical line in front of the text, at the left-hand margin. In other words, a comment symbol in front of each line.

Bob: Ah, of course, like you use a `#` in Ruby or a `C` in Fortran or a `//` in C++ or a percent sign in Latex. Yes, I like that. In that way we can allow any part of a story or history or anything else to be commented out, so to speak, making both `begin` and `end` invisible for the I/O routines.

Alice: So our example for the *rest* data could become

```
velocity = 1 0
acceleration = 0.1 -0.3
begin story
  |Hi Joe!  Look what a cool run I just produced.
  |This AGB star is out of control!  Man, it's
  |evolving with a time step much less than a year,
  |and shrinking.  This way we'll never reach the
  |begin position of the horizontal branch.  Meanwhile,
  |wanna come over for a beer?
end
```

You could it even put in a real box, as a children's drawing, if you want:

```
velocity = 1 0
acceleration = 0.1 -0.3
begin story
  +-----------------------------------------------------+
  |Hi Joe!  Look what a cool run I just produced.       |
  |This AGB star is out of control!  Man, it's          |
  |evolving with a time step much less than a year,     |
  |and shrinking.  This way we'll never reach the       |
  |begin position of the horizontal branch.  Meanwhile,|
  |wanna come over for a beer?                          |
  +-----------------------------------------------------+
end
```

Bob: Very funny. But yes, you could use any symbol you like. The least obtrusive would be a period, just as the Unix system does for files that are normally invisible:

```
velocity = 1 0
acceleration = 0.1 -0.3
begin story
  .Hi Joe!  Look what a cool run I just produced.
  .This AGB star is out of control!  Man, it's
  .evolving with a time step much less than a year,
  .and shrinking.  This way we'll never reach the
  .begin position of the horizontal branch.  Meanwhile,
  .wanna come over for a beer?
end
```

Alice: And whatever symbol you use, nothing will match `begin` and `end` anywhere. I think we have found a fail-safe solution! A nice surprise, after we both thought that we were stuck.

6.5 Onward

Bob: Isn't it interesting? You can use comments in a program for many years and never give it much thought. But when you have to design a special data format, as we are doing for ACS, you are in fact designing a kind of mini-language. So we have just reinvented the wheel! Now I can appreciate much better the role of comment conventions in computer languages.

Alice: Of course, anyone using the data will still have to find a way to strip the comment symbols off, if they want to work with clean text.

Bob: However, that is less urgent. In this last example, the leading periods are almost invisible.

Alice: For some purposes, yes, but for other applications I'm sure that you may want to implement a way to get rid of the comment characters.

Bob: If you like. But first onward to get to graphics. Any good software project can be stalled completely by implementing a surplus of features before you really need them – and most of those turn out not to be want you want anyway, when you later look back on them. I've seen that happening.

Alice: I agree. Where were we? We decided to create a scratch pad named `rest` for all the rest of the lines that `Body` could not understand.

Bob: How about this: we can add an instance variable `@rest` for the `Body` class, in the form of one big string. Initially each `Body` will be created with an empty string:

```
def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
  @mass, @pos, @vel = mass, pos, vel
  @type = nil
  @rest = ""
end
```

The only line that we need to change in the `read` method, is to replace the old

```
        else
          raise
```

in `iobody3.rb` by

```
        else
          if s =~ /^\s*\w+\s*=/
            @rest += s
          else
            raise
```

in our new `iobody4.rb`

The first line tests whether the unknown line has an equal sign in it. If so, the whole line is appended to the `@rest` string. If not, it really is an error.

Alice: Simple indeed! But we have to make a change in our output mechanism as well.

Bob: That should be simple too. After the current lines in `to_s` that handle the known quantities

```
      f_to_s("mass", mass, precision, indent) +
      f_v_to_s("position", pos, precision, indent) +
      f_v_to_s("velocity", vel, precision, indent) +
```

we can add a similar line for the unknown quantities, where the only information that needs to be passed in the amount of indentation `indent`:

```
      rest_to_s(indent) +
```

and if I regularly express myself as follows

```
  def rest_to_s(indent)
    @rest.gsub(/^\s*/, " "*indent)
  end
```

it should all work. The command `gsub` globally substitutes however many initial blank spaces there may be in any line within the string `@rest` by the proper indentation length.

6.6 Testing

Alice: Let's see whether it all works.

Bob: I'll write a test file `test.rb`:

```
require "iobody4.rb"

b = Body.new
b.read("begin particle star giant AGB")
b.write
```

And here is the result:

```
|gravity> ruby test.rb
      mass =      1
      nearest_neighbor = 365
      position =  2  3
      velocity =  4.5  6.7
      density = 3.2e-07
    end
```

```
begin particle star giant AGB
  mass =    1.0000000000000000e+00
  position =    2.0000000000000000e+00   3.0000000000000000e+00
  velocity =    4.5000000000000000e+00   6.7000000000000002e+00
  nearest_neighbor = 365
  density = 3.2e-07
end
```

Alice: Looking good! Shall we try some bad indentation, to see whether it will get corrected?

```
|gravity> ruby test.rb
      mass =      1
nearest_neighbor = 365
               position =  2  3
  velocity =  4.5  6.7
        density = 3.2e-07
end
begin particle star giant AGB
  mass =    1.0000000000000000e+00
  position =    2.0000000000000000e+00   3.0000000000000000e+00
  velocity =    4.5000000000000000e+00   6.7000000000000002e+00
  nearest_neighbor = 365
  density = 3.2e-07
end
```

Bob: So far, so good. Let's try to give a story line, without a proper `begin story` header, to see whether we get a proper error message.

```
|gravity> ruby test.rb
      mass =      1
      nearest_neighbor = 365
      position =  2  3
      velocity =  4.5  6.7
      this is a rather large star
      density = 3.2e-07
    end
./iobody4.rb:71:in ‘read’: unhandled exception
from ./iobody4.rb:53:in ‘loop’
from ./iobody4.rb:53:in ‘read’
from test.rb:4
```

Alice: Indeed: that is indeed the number of the last line in

```
else
  if s =~ /^\s*\w+\s*=/
    @rest += s
  else
    raise
```

But we should really provide a more user friendly error message, that does not require counting lines of source code.

Bob: Let us first handle proper stories, starting with `begin story`, as well as other particles that might be embedded within our current particle data, as members of a star cluster.

Alice: Yes, and these two points are indeed what was left from our previous todo list:

- write the `subread` method
- improve the error handling

Chapter 7

Nested Input

7.1 xxx

TODO: HIGHER LEVEL ACS READ-IN, HASH TABLE, file testing appendices, XML versions of ACS format I/O, also HISTORY

nil nil

Chapter 8

Introduction

8.1 xxx

```
|gravity> ruby sim2acs.rb < cube1.in > tmp1.out
==> Conversion from simple N-body format to ACS data format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
|gravity> ruby sim2acs.rb < cube1.in | ruby acs2sim.rb | ruby sim2acs.rb > tmp2.out
==> Conversion from simple N-body format to ACS data format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Conversion from simple N-body format to ACS data format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Conversion from ACS data format to simple N-body format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
|gravity> diff tmp[12].out
```

```
|gravity> ruby sim2acs.rb < cube1.in | ruby acs2sim.rb > tmp3.out
==> Conversion from ACS data format to simple N-body format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Conversion from simple N-body format to ACS data format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
|gravity> ruby sim2acs.rb < cube1.in | ruby acs2sim.rb | ruby sim2acs.rb | ruby ac
==> Conversion from simple N-body format to ACS data format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Conversion from ACS data format to simple N-body format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Conversion from simple N-body format to ACS data format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Conversion from ACS data format to simple N-body format <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
|gravity> diff tmp[34].out
```

8.2 xxx

Chapter 9

Literature References

[to be provided]