2 Abstraction

Divide-and-conquer reasoning breaks enigmas into manageable problems. When the reasoning is represented as a tree, the manageable problems become the leaf nodes of the tree, and they are conceptually simpler than the original problem or its intermediate subproblems. For example, the length of a classical symphony is a simple concept compared to the data capacity of a CDROM.

Being simpler, it is more likely than the parent nodes to be used in another calculation. Imagine that you are an architect designing a classical concert hall. One task is to ensure sufficient airflow to handle the heat produced by 1500 audience members during a concert. But how long is a concert? Reuse the symphony leaf node from the CDROM-capacity estimate. Concerts often include a symphony before or after a break (the intermission), with a comparably long other half, so a rough concert duration 2.5 hours.

Creating and using such reusable parts is the purpose of our second tool for organizing complexity: abstraction. Abstraction is, according to the *Oxford English Dictionary* [29]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] describe:

So after reading this section, and the comments, I think, but I'm not sure if I got this concept correctly (it's a bit confusing to read at first). Is the idea of abstraction to basically make "templates" in order to reduce/simplify/organize a task, and so that things lower on the "tower" are unimportant to continue? If that is the case, then this example makes sense finally at the end, by showing that a program/code designed to do a task does it better than just using a more general program that cannot do the specific tasks? I'm trying to think of other more, non-obvious real life uses of abstraction and am having a hard time thinking of a good one?

> A resistor of a specified resistance is an abstraction of what is really an analog transfer curve with tolerances. All the elements are abstractions. We can think about them without having to think about all their quarks and gluons or draw abstract diagrams like O-H-O. Scaling up, we can think about a bucket of water without imagining every molecule and trying to keep track of its position. Think of it as collapsing a lot of information into a more manageable concept. It's key to engineering so you probably do it all the time without thinking.

I think I'm having a lot of trouble seeing how this section fits in with approximations. It seems like the main point of the chapter is that you can write code to produce a tree easily. I also don't quite understand what abstraction in this context is.

This is a very vague, hand-waving definition of abstraction.

what is this boxes package? and how do we know which level of abstraction to look at without testing?

how did you come up with these lines? which level of abstraction are you looking at? I find this to be a very confusing example overall...I didn't realize all the scripts above were describing parts of an abstraction tower.

aren't most drawing programs designed this way? maybe I don't understand exactly what you mean by graphical captive UI, but this way the user has lots of options to format the tree as he/she wished.

So is abstraction using precreated shortcuts in creative ways? Also, I imagine it took a while to make the program. Doesn't it kinda cancel out?

2 Abstraction

Divide-and-conquer reasoning breaks enigmas into manageable problems. When the reasoning is represented as a tree, the manageable problems become the leaf nodes of the tree, and they are conceptually simpler than the original problem or its intermediate subproblems. For example, the length of a classical symphony is a simple concept compared to the data capacity of a CDROM.

Being simpler, it is more likely than the parent nodes to be used in another calculation. Imagine that you are an architect designing a classical concert hall. One task is to ensure sufficient airflow to handle the heat produced by 1500 audience members during a concert. But how long is a concert? Reuse the symphony leaf node from the CDROM-capacity estimate. Concerts often include a symphony before or after a break (the intermission), with a comparably long other half, so a rough concert duration 2.5 hours.

Creating and using such reusable parts is the purpose of our second tool for organizing complexity: abstraction. Abstraction is, according to the *Oxford English Dictionary* [29]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] describe:

could we go over divide and conquer vs. abstraction conceptually? they overlap on some key parts and differ in some other key parts. I'd just like to see that line between them more distinctly

I still don't understand abstraction after reading this section...

Kind of confused...mostly because I've never done programming but does this seek to compare abstraction to reusing portions of the GUI computer program?

I'm very confused, can we have a non programming example to explain abstraction? I think I understand the point of abstraction but the programming portion has completely lost me.

Unlike the MH example in class, it seems that using abstraction to make a tree is longer and more time consuming than using a prescribed program. Is there an example where using abstraction is time efficient?

2 Abstraction

Divide-and-conquer reasoning breaks enigmas into manageable problems. When the reasoning is represented as a tree, the manageable problems become the leaf nodes of the tree, and they are conceptually simpler than the original problem or its intermediate subproblems. For example, the length of a classical symphony is a simple concept compared to the data capacity of a CDROM.

Being simpler, it is more likely than the parent nodes to be used in another calculation. Imagine that you are an architect designing a classical concert hall. One task is to ensure sufficient airflow to handle the heat produced by 1500 audience members during a concert. But how long is a concert? Reuse the symphony leaf node from the CDROM-capacity estimate. Concerts often include a symphony before or after a break (the intermission), with a comparably long other half, so a rough concert duration 2.5 hours.

Creating and using such reusable parts is the purpose of our second tool for organizing complexity: abstraction. Abstraction is, according to the *Oxford English Dictionary* [29]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] describe:

Comments on page 1

Read this introduction to our next chapter (the reading is 4 pages) and submit the memo by 9am on Wednesday (2/17).

Isn't this intro still mostly about divide-and-conquer? Maybe it should go back in the first chapter?

I expected this to involve using diagrams to abstract, not using abstraction on the process of making diagrams. – I take this back in part because I didn't know the 2.1 section on diagrams hadn't started here.

These last few sections have been particularly CS-based; I know that many of the students have not enjoyed it as much.

These last few sections have been particularly CS-based; I know that many of the students have not enjoyed it as much.

If the window is left open, the new readings don't show up unless you hit refresh. It's almost tricked me a few times.

This is an absolutely lovely breakdown of divide-and-conquer; it's simple, concise, and covers the main points. It'd be helpful to see it earlier in the d-and-c section.

Can't an architect use divide and conquer for this question?

Yes, I think that a divide and conquer approach could be used for this. It does seem like a difficult calculation though.

strangely i don't ever remember being hot during a concert

I think that it's because it's a problem that has already been figured out. Concert halls have been around so long that not much thought has to be given about airflow.

2 Abstraction

Divide-and-conquer reasoning breaks enigmas into manageable problems. When the reasoning is represented as a tree, the manageable problems become the leaf nodes of the tree, and they are conceptually simpler than the original problem or its intermediate subproblems. For example, the length of a classical symphony is a simple concept compared to the data capacity of a CDROM.

Being simpler, it is more likely than the parent nodes to be used in another calculation. Imagine that you are an architect designing a classical concert hall. One task is to ensure sufficient airflow to handle the heat produced by 1500 audience members during a concert. But how long is a concert? Reuse the symphony leaf node from the CDROM-capacity estimate. Concerts often include a symphony before or after a break (the intermission), with a comparably long other half, so a rough concert duration 2.5 hours.

Creating and using such reusable parts is the purpose of our second tool for organizing complexity: abstraction. Abstraction is, according to the *Oxford English Dictionary* [29]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] describe:

So, the ones we reuse a lot (in this class and life in general) are the ones we should memorize? Like the number of seconds in a year?

By using them often, you're also more likely to remember them rather than memorizing by rote.

Agreed - I think we'll just see them often enough that over time we'll remember them. Plus, I think we can raise flags in our minds that say "hey, this number sounds useful!" - seconds in a year, population of the US, etc should all fall into that category. In this case, where we talk about CD-ROM capacity, I'm sure it's fine to just look it up.

It would be nice if you finished solving the problem in this section.

i suppose it just depends on which values you are more familiar with.

I agree, I think it's important to emphasize that there is not one way to break down these problems, but instead it relies a lot on the pieces of information you individually know.

Maybe it will become more clear as I continue reading, but so far, I'm struggling to understand the difference between divide and conquer and abstraction.

This technique seems like it comprises divide and conquer, making trees, and robustness. I'm struggling to see what else it contains/employs besides these three things...?

How does this connect to divide and conquer? meaning treat the leaves as separate things?

I thought today's example about tracing the origin of species and spread of the bible was a great example of this

I've never been a fan of dropping dictionary definitions. But the idea of Abstraction is to generalize your concept, and I think this article just put in the only definition of abstraction that fits for their use.

I'm not sure that a dictionary definition is really the most useful definition of abstraction here. It might work better to just define it within the context of our goals towards estimation.

Yeah I feel like its kind of vague in terms of usefulness...

2 Abstraction

Divide-and-conquer reasoning breaks enigmas into manageable problems. When the reasoning is represented as a tree, the manageable problems become the leaf nodes of the tree, and they are conceptually simpler than the original problem or its intermediate subproblems. For example, the length of a classical symphony is a simple concept compared to the data capacity of a CDROM.

Being simpler, it is more likely than the parent nodes to be used in another calculation. Imagine that you are an architect designing a classical concert hall. One task is to ensure sufficient airflow to handle the heat produced by 1500 audience members during a concert. But how long is a concert? Reuse the symphony leaf node from the CDROM-capacity estimate. Concerts often include a symphony before or after a break (the intermission), with a comparably long other half, so a rough concert duration 2.5 hours.

Creating and using such reusable parts is the purpose of our second tool for organizing complexity: abstraction. Abstraction is, according to the *Oxford English Dictionary* [29]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] describe:

I understand the idea of considering things independently- but isn't the whole point of divide and conquer to consider things as they relate to the goal? In order to divide and conquer, we must think about what smaller things contribute/ relate to the bigger picture so it's hard to then not consider them as parts of a whole.

I agree–it sounds a lot like divide and conquer, but I think abstraction goes deeper than divide and conquer. in divide and conquer we're looking into different components but in abstraction we're finding different ways of approaching a problem by seeing what it breaks down into...

I agree as well, it seems like abstraction is saying "step away and look again" rather than "how can we break this into bite sized chunks"

Agreed. I see there being somewhat of a balance- if things are too broad they aren't very useful, but if they are too specific they can't be reused. The trick is to find things that are small enough that they are manage-able and still be able to use the same numbers in a variety of applications

I agree that this is ambiguous - perhaps Sanjoy should use his own words instead of a dictionary definition to clarify confusion.

I also thought that the definition of abstraction kind of sounds like the same thing as divide and conquer. In my other classes abstraction is a way of viewing a variable at an acceptably high enough level. For example, when i learned about circuits, we "abstracted" away from analyzing it using really horrendous Maxwell equations, but "abstracted" components using values of Resistance, Inductance, and Capacitance. Those values of R,L, and C contain a lot of physics, but since we don't really care about the physics to solve the final solution, abstraction was used to lump unnecessary.

I don't really think this explains abstraction at all. I've read the entire document and I'm still not really sure what it is or how it helps in making approximations. It looks like it could be useful in making a diagram or a program, but how does that relate to approximating?

Is it possible to include an approximation-related example? Since it is the introduction, I see how the coding example can explain the concept. But it took me several readings to understand it.

Why is it helpful to think of something independently when it goes with something else to give you an answer.

2 Abstraction

Divide-and-conquer reasoning breaks enigmas into manageable problems. When the reasoning is represented as a tree, the manageable problems become the leaf nodes of the tree, and they are conceptually simpler than the original problem or its intermediate subproblems. For example, the length of a classical symphony is a simple concept compared to the data capacity of a CDROM.

Being simpler, it is more likely than the parent nodes to be used in another calculation. Imagine that you are an architect designing a classical concert hall. One task is to ensure sufficient airflow to handle the heat produced by 1500 audience members during a concert. But how long is a concert? Reuse the symphony leaf node from the CDROM-capacity estimate. Concerts often include a symphony before or after a break (the intermission), with a comparably long other half, so a rough concert duration 2.5 hours.

Creating and using such reusable parts is the purpose of our second tool for organizing complexity: abstraction. Abstraction is, according to the *Oxford English Dictionary* [29]:

> The act or process of separating in thought, *of considering a thing independently of its associations;* or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs. [my italics]

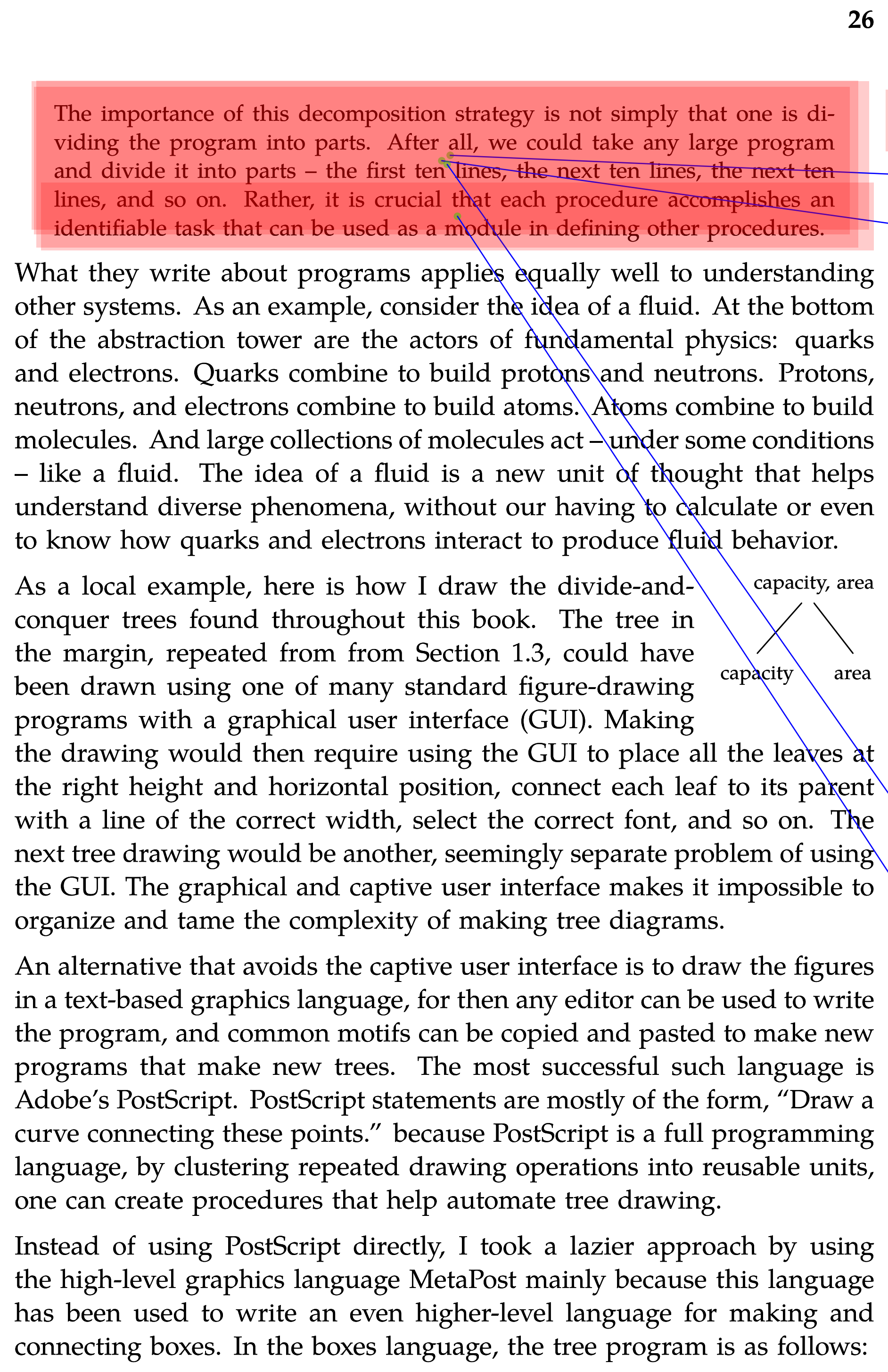
The most important characteristic of abstraction is reusability. As Abelson and Sussman [1, s. 1.1.8] describe:

This is a great word to describe the use of abstraction. An abstracted element needs to be a contained entity. Something kind of like UNIX programs that can be reused for different tasks.

This ties in nicely with the modular part of the UNIX philosophy. Maybe this section should come before that one?

Or at least this section should briefly mention the connection to UNIX.

So this makes it sound like they want us to remember every estimation we ever made. I understand it's easier if you already estimated to just use it again, but I think it's pretty unpractical in reality.

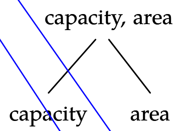


26

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.

As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.



An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

I am starting to get a little confused here, connecting the dots.

this really clarifies a lot

This is much more useful than the Oxford dictionary definition.

I agree especially when considering how we are using the word.

I agree as well. Is it really necessary to have the dictionary definition? I found the OED def to be a little broad given the course.

I also agree! The Oxford definition probably shouldn't be used and this put in its place.

i disagree. the first definition explains "abstraction" while this one is a more intuitive practice.

I like having the two definitions since the first one describes the general definition of abstraction while the second tells how it relates to what we are learning.

Well, this definition is the object oriented programming version specifically.

I also think that the oxford definition is kind of useless. I actually just skimmed over it because it seemed fairly complex and I assumed it would be better explained later.

I like the dictionary definition because it gives a technical introduction to what you're actually trying to accomplish. After knowing the process, the description gives a more thorough definition of the thought process.

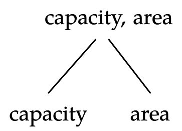
I was thinking the same like. Methods pop right into mind.

So abstraction goes further than divide and conquer- it is a type of divide and conquer but goes further to generalize each solution

abstraction!!! It's pretty cool that divide and concur methods of approximation are analogous to methods in computer programing

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

Choosing appropriate modules seems to be the critical and important step in this process. However, sometimes it can be very difficult to decide exactly how to make these modules. Are there any rules or guidelines to ensure that you modularize things in a way that is most helpful?

One way is to watch your own actions with slightly blurry vision. If you find yourself doing the same thing, or making the same calculation a second or third time, then you are likely to have a spot to make an abstraction.

The reason I say "slightly blurry vision" as well is that if you look too closely, then you'll never think you are doing something for a second time. Rather, you'll find all the differences. But with blurry vision, those details will hopefully disappear ("get abstracted away") and you'll be left with the common features.

i feel like this material is much more accessible to the 6.055 students than to the 2.038 students.

Yes... I'm a little nervous about this unit because I have no programming experience.

I think he's more interested in the ideas he's already explained than in the actual example. He's just trying to show how you can actually use stuff we've learned to solve problems.

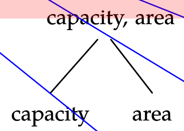
I am not sure if that's true though, the text does go into detail...

i think the point of the statement "what they write about programs applies equally well to understanding other systems" is supposed to reassure us about the fact that we don't just have to be course 6 to understand and benefit from this.

Can we see an example of abstraction (or any other unit for that matter) that applies mech e concepts?

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

You mention this "tower" later on in the section and I think it really helped me understand this better; that things on the bottom of the tower become "abstracted" (quarks are not important to the understanding of fluids, but they still exist). I think if the tower analogy is defined more clearly this would be really helpful for a lot of people!

I agree. The appearance of the 'abstraction tower' here seems rather abrupt and caused me to look through previous paragraphs thinking that I had missed its definition already.

A figure for this example of an 'abstraction tower' would also be awesome.

so is a tower what's used for abstraction and a tree for divide and conquer?

I agree with the other comments here - the "abstraction tower" suddenly appears with no explanation whatsoever. A diagram or short description would be very helpful.

perhaps "abstract tower" (in quotes) is better? you would take it more metaphorically rather than expecting a definition. after all, everyone knows that a tower is.

This is a much better visualization of abstraction for more visual people.

why are we starting from the bottom here? are we going backwards from the divide and conquer method- I don't know why you would ever start with the smallest form of matter anyway- it seems to trivial to me (sorry I don't really like chemistry)

I like that you use such a simple example to follow the definition, but is there one that would be even more accessible. It helps for people to be able to visualize things and it was hard for me to visualize the relationships between the elements of this example because of my backgrounds.

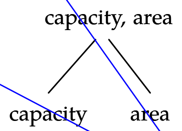
I actually thought this was a very helpful example. It breaks down something as complicated as a fluid to suff as simple as quarks. (I am also taking some course 8 classes, so I love this analogy)

I REALLY liked this example. Maybe I am just a physics major, but I would like to point out it was one of my favorite examples yet on the reading.

Using "like" makes it sound as if there is merely an analogy between the behavior of molecules and the behavior of fluids. But large collections of molecules do not just act like a fluid (under the right conditions), they actually are a fluid.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

I don't understand this reference.

In this example, I see the abstraction tower. It's pretty clear to me the different levels of "detail". But I do not understand the tree language one below that well...

This was a helpful example; generic examples are always excellent for keeping a reader's attention.

so is an abstraction is an easy to work with model? Like how we use circuits to model conduction, or control systems, or eletromagnetic phenomena.

Yeah, and that's why the resistors, capacitors, and inductors we use are part of the "lumped matter" abstraction.

I feel that the example of the fluid is more relevant and intuitive than the concept of abstraction in programming

I feel like that's the point of abstraction. Quarks, and molecules would be useless to use in order to understand airflow but as they are looked at as a fluid these problems can be solved. I see abstraction more as a tool for creating intuitive solutions then the solution itself.

I agree with this abstraction is more of a method, or a means to an end.

This fluid example helped me wrap my head around the subject more.

this makes me want to yell: Abstraction! yippie

but you still need to understand fluid behavior in order to use it.

I guess this is the most direct contrast to the divide and conquer, where the point is to know exactly how they interact so you can do calculations

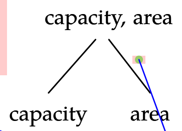
At the same time, we should recognize that the ends of abstraction is the same as divide-and-conquer: get onto terms with which you are comfortable generating numbers.

How can this help generate numbers f we are just examining big-picture interactions? Maybe an example with numbers would be helpful in this intro, just give a taste of what is to come and show what can be accomplished with this method.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.

As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.



An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

So in this example we are referring to leaves as the quarks and electrons? This abstraction is very clear, however, just to reaffirm your point as made in the intro, specifically stating their leaf characteristics would be helpful.

I don't understand what this is serving as an example of. I thought it would be clarified later, but I got lost in the details of creating the tree and don't understand what exactly this example is supposed to represent.

Just curious, what exactly does "local" mean here? Local to the text, or local as in limited-scope?

I think it means local as in it's only looking at a small part of a larger tree? Edit: OK I take that back, I'm not sure anymore.

Local is not the best word here, but it seems to mean specific to the text, as opposed to a grander physics topic like sub-atomic particles.

also confused.

I am almost positive he's just trying to say "an example from this class" or "an example from the text"

Just my two cents, but maybe it would be better to use a different example of abstraction than to describe how to draw the divide-and-conquer trees in the book. On the first read I was confused because we were previously talking about divide-and-conquer as an approximation method and now we're taking it to a new level with abstraction, but the abstraction of how to draw divide-and-conquer trees. Of course on my second read it was very clear, but it did take me a second read.

I understand you want to lay a foundation for not using captive user interfaces, or at least for why they are counterproductive as far as this text is concerned, how ever this paragraph seems very forced and mostly unneeded. You could just cut this entire paragraph and start the next with the first sentence from this paragraph.

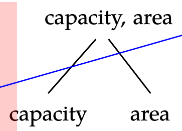
Agreed - this paragraph seems entirely unnecessary.

What does this have to do with abstraction?

While a tree as reference is nice since its the basis for the example, its necessity/relation to abstraction isnt defined so maybe an improvement in that area would make its presence more understandable

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

this tree bothers me because it is very off center

Should be: tree to the right, as it is not technically in the margin.

Ah, good point, I'm sure most people would figure it out but this should be rectified.

typo. figured if this was to be in your book you should know.

Thanks for finding that. I make so many of that kind of mistake that I once wrote a 19-line Python script to check for double words like 'from from' or 'the the'.

I'll dust it off and run it on the TeX file to see if there are other examples. Ah, there is one more in the next section that you'll get.

I would use a different example.

I agree, this example seems like a odd at first glance and I had to reread it again to understand the application of it.

I think it only makes sense in the context of yesterday's lecture (especially the 'captive user interface' part), and should definitely be explained better or changed.

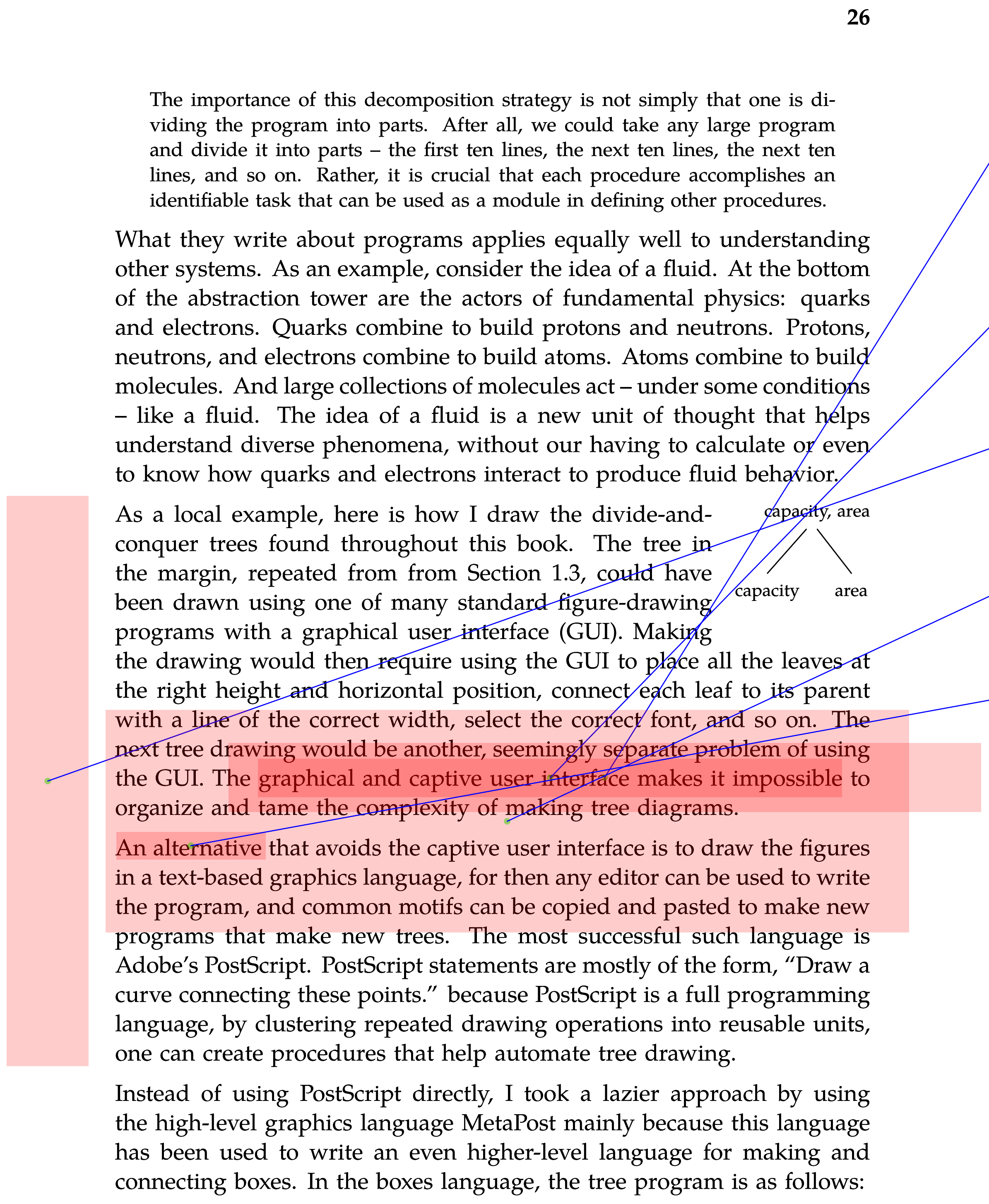
Do you mean that you have to start over every time you wish to make a new figure? Because you could create a template, even with a GUI program...

Or you could just copy and paste.

it may just be that I kinda enjoy this kind of thing, but I'd much rather do this kind of tedious (but simple) work than thing too hard about how to code it...There is also that whole copy&amp;paste thing is useful for. also peons...peons are nice.

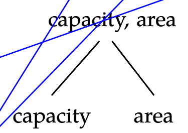
what does captive mean here and why is it a bad thing?

I am also a little unsure of what this reference means.
I believe it means the same thing as GUI, and as discussed in the last lecture, although it is useful, it is extremely limited by what the original designers chose to implement.



> The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

You make a valid point about how using Postscript allowed you to save time but I feel that your assertions about GUIs comes across as being unnecessarily partisan.

I think he is just rying to show other problems through which you can also solve with divide and conquer.

I can see how this would apply to non-extensible GUI applications, but many popular graphical programs now have extensive plugin support. In theory, wouldn't it be possible to write a tree-diagram plugin that would be arguably more straightforward for users with little programming experience?

Here it's good that you compare divide and conquer to abstraction, but the example is difficult to understand. Maybe if this was accompanied by some visual comparison it would be simpler for people that don't understand coding. For example, a divide and conquer tree and its analagy in abstraction.

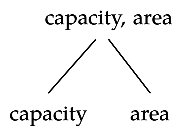
This advocacy of alternative computational methods seems very one-sided. I agree with later posts that this gets in the way of explaining abstraction. Sussman talks about electrical/electronics components, which I think is a much easier idea to grasp. We can specify an AND gate without having to describe the components every time.

I think I missed whatever link there was between abstraction and tree/coding/captive user interfaces. Why is this relevant? And how is it abstraction?

I am curious about the intended audience for this book. Is it intended for only this class that you teach at MIT, or would you want it to be a text that a similar class at a different university could use? If it is the latter, these programming examples make the book less accessible. I have no programming experience and often get hung up trying to understand how the systems in the examples work rather than relating them to the actual concepts. I agree with what others are saying about actually liking the captive user interfaces because, for my purposes, they work better than anything else, partially because I don't know any programming and would be dreadfully inefficient at any other method.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

Couldn't a text-based graphics abstraction be more difficult for some people to understand? I realize that certain UI's may be limited, and text is one of the easiest forms of input, but depending on the program's audience, couldn't a visual tree-diagram editor be more effective?

I agree that sometimes text-based graphics could be harder to understand. I am personally a visual thinker and graphical analyses always make more sense to me. I have also used PostScript and while the language was easy to use, it was hard to think about and construct a graphic with PostScript. Wasn't it just discussed in lecture the other day how humans in general learn better from visual graphics?

The same arguments have been debated about LaTeX word-processors. While, Microsoft Word uses a more visual way of editing papers, TeX based processors use a more programming oriented way of editing papers. The tradeoff is immediate visual effect to saving time in abstracting to multiple applications or longer documents.

abstraction of a figure- the abstract, divided problem becomes creating one node/edge?

So does that mean that any instance of breaking things down into more manageable subunits is an instance of trees/dividing and conquering?

I don't know much about programming languages, but is Adobe's PostScript cleary the most successful language for this? Shouldn't all programming languages be built from basic fundamental building blocks that are versatile?

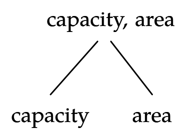
no period or capitalize Because

Maybe in the future you could assig a homework that actually uses some programming in either unix or postcript. I believe every MIT student should know how to jot down a few lines of code.

Can we see examples of different diagrams drawn by these programs? Maybe that is unnecessary.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

what does this mean? is python a full programming language? how is it different from non-full programming language?

I think a full programming language means it has the constructs to make it easy for you to solve a wide variety of programming problems. Python is full programming, but i believe SQL is not

I feel like it might be a better idea to reference a more common programming language. PostScript makes sense, but it seems most people are unfamiliar with it. What about referencing a more common language like Java or C++? One could make a very convincing argument about the use of libraries or packages as a basis for abstraction.

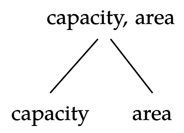
Although PostScript might not be as well known, I feel like it is used as a really good example in this section. In fact, it gives an example of abstraction in a programming language that is tied back to the tree diagrams that we've seen in the previous sections of the reading.

The example is too technical for me to follow. Something more general would be better.

Key word that gets to your point, meaning it can be repetitive, but it is a little misleading, just b/c I have the pieces that are applicable for many complex situations it is not "automated."

> The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

This "lazier approach" is still over my head since I am not familiar with programming concepts. I think (just like the last section) that some readers are going to get lost and either get frustrated or just totally skip over the section if they don't understand the programming language.

Agreed. For me, it would be much more work and time intensive to learn how to program to get to the stage where I could do this than just to use a fairly set UI. Especially where there are so many out there that specialized in certain things.

I disagree. Without knowing anything about the languages, I was able to piece together that the alternative was a manual means of making the trees that were copied and pasted. I also understood that he took a shortcut by using a program that had some sort of abilities built in to make the trees.

I think some people have the sour taste of the UNIX example in their mouths, but this is fairly straightforward if you just re-read it a few (3) times.

It seems hard to revert thinking back to 'programming' thinking after having been spoiled with GUIs for so long. I agree that some might skip it because of not understanding programming, but I also agree with the point that it is very important to have that thinking, and that it's a tragedy that we are so keep on using GUIs. Since learning programming, my thinking has become cleaner, and not just for computer skills. It's an improved way of going about life tasks.

Can you post links to where we can learn more about the different coding languages/programs/etc mentioned? I'm not familiar with a lot of them, or have a passing familiarity, and would like to learn more about them.

Good idea. I've added several to the course website. I'll upload it from my laptop to the MIT webserver shortly.

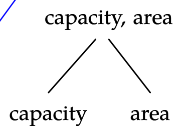
Are we going to continue with a lot of computer programming material? I don't really understand this stuff.

cut mainly...unless you are going to give us a second reason, the word is useless.

I don't have a problem with coding as an example as much as I do biology. I have an unreasonable distaste for that science.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts – the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

What they write about programs applies equally well to understanding other systems. As an example, consider the idea of a fluid. At the bottom of the abstraction tower are the actors of fundamental physics: quarks and electrons. Quarks combine to build protons and neutrons. Protons, neutrons, and electrons combine to build atoms. Atoms combine to build molecules. And large collections of molecules act – under some conditions – like a fluid. The idea of a fluid is a new unit of thought that helps understand diverse phenomena, without our having to calculate or even to know how quarks and electrons interact to produce fluid behavior.



As a local example, here is how I draw the divide-and-conquer trees found throughout this book. The tree in the margin, repeated from from Section 1.3, could have been drawn using one of many standard figure-drawing programs with a graphical user interface (GUI). Making the drawing would then require using the GUI to place all the leaves at the right height and horizontal position, connect each leaf to its parent with a line of the correct width, select the correct font, and so on. The next tree drawing would be another, seemingly separate problem of using the GUI. The graphical and captive user interface makes it impossible to organize and tame the complexity of making tree diagrams.

An alternative that avoids the captive user interface is to draw the figures in a text-based graphics language, for then any editor can be used to write the program, and common motifs can be copied and pasted to make new programs that make new trees. The most successful such language is Adobe's PostScript. PostScript statements are mostly of the form, "Draw a curve connecting these points." because PostScript is a full programming language, by clustering repeated drawing operations into reusable units, one can create procedures that help automate tree drawing.

Instead of using PostScript directly, I took a lazier approach by using the high-level graphics language MetaPost mainly because this language has been used to write an even higher-level language for making and connecting boxes. In the boxes language, the tree program is as follows:

though, I must say that dealing with many languages/programs I haven't yet encountered does add a level of complexity to the homework and reading.

I feel like maybe using energy or something more broad as an intro to abstraction might be useful. I know as a non course 6 student abstraction and recursion were hard for me to grasp at first and a more global example might be beneficial in this section.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

Comments on page 3

This seems pretty unnecessary. Perhaps some pseudocode?

Perhaps you could have little boxes on the sides or before the chapters with code saying what language, which shells to use, and how to run them, in the event that readers want to give it a whirl.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

I have no idea what this means

well, the comments walk you through it. First, the contents of the 3 text boxes are specified, then their locations relative to one another, then it draws the text boxes, and then it draws the branch lines.

This assumes you know that the % before a line means its a comment.

It might be better to write an example (like this one) in pseudocode. While the comments do walk you through it, you can't assume that every reader is going to have the requisite coding experience to realize what exactly is going on.

As mentioned above, I think it would be helpful to "guide" students through the code in this book by explaining in detail what's going on. From reading these posts it seems that many students are confused or discouraged when they see large amounts of code with few comments to help them understand.

While this is a course 6 class, not everyone taking the class has coding experience - why not write a line by line description of what the program does instead of providing code? I would guess that there exist MIT students who have never coded.

I agree that maybe this should be more of a written pseudocode. I understood this because of my programming experience, but someone without may have problems even recognizing what is commented in this code.

It seems like a lot of my thought in reading this is aimed at understanding the code and not the apporoximation method driving the code.

The author can't hold the readers hand at every pond crossing. If the reader doesn't have any idea what the text means (i.e. they have literally never written ANY computer code in their lives), they can still understand it based on the two previous relatively-simple paragraphs.

I agree. While I understand what this code is accomplishing in it's entirety. I don't know what a lot of the specific functions and things mean and I think their presence detracts from my understanding of the concept because I feel like I should try to understand the code.

I am on the same boat about how even though I understand what this code is doing, it is still a new language to me and detracts from my understanding as I try to figure out and picture what each line is doing.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

I have to side with the not explaining it is much,If it's written in psuedocode, it loses a good chunk of the purpose of the code, and it's not intended to necessarily be understood for the coding factor, but for a general overview of the format, supplemented by the comments.

having limited coding experience, i was able to pick this apart but i can imagine how hard this would be for a person who hasn't coded and how important pseudocode would be for the understanding of this. i think that the code should stay but perhaps on the right hand side (as if the upper part of this page is split into two columns) there could be italicized psuedocode or an image showing what's happened after that chunk of code

What about making the code block more of a figure than a part of the text? This way a small caption could be applied with a few short sentences acting as a way to guide the reader through the code, much like psuedocode would do.

I would have just draw it in paint

you don't have to understand exactly waht it meants; only the gist of it. By placing exact code verbatim in the text, Sanjoy makes the application of abstraction seem more realistic. I don't think he should withhold realism from his examples, but rather suggest sources of more detailed explanations for students that need it. In our case, these comment boxes provide a good place to suggest more material and clarifications.

This is both confusing and kind of comes out of nowhere. There has to be a better/easier way to explain abstraction than with this complicated jumble of coding.

Agreed. It would be nice if the code was graphically commented or hashed out a little more. As a Course 2 student, this makes virtually no sense and has compromised my understanding of the rest of this section.

not familiar with this language, i doubt a lot of people are, maybe should have used a simpler diagram creating software, like powerpoint, etc?

It would be nice to have interactive code so we can actually understand the programming better for those of us who are unfamiliar with coding.

Since this will be in a textbook, prseumably, this might be a difficult concept.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

I don’t understand several of these lines.

just as a side note, I really do not have any idea how to look at this or any lines of code and get anything out of it, I need someone to explain it to me in person, reading it on a page only makes me more confused

I know that LATEX can be used to make similar PDFs. Would it be possible to do the trees in LATEX (or a higher level tree language based on LATEX) as well?

I understand this paragraph, but I don’t see how the coding part above relates to abstraction, or how it is necessary in this article (I’m one of those who doesn’t have much coding experience, so I could be missing the point completely...)

The idea is by writing higher level code we can avoid the mess of error prone code that we would need to accomplish simple tasks. By the end of the section this all becomes clear.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

This sounds really confusing. Is this really relevant? Does it help us to understand abstraction or approximation?

I can kind of see how it's an example of abstraction based on how it's an example of defining how different levels of the tree are related to each other.

Agreed... I understand the example but I still don't understand abstraction, and this isn't really helping me understand it at all. It think the examples tend to lose their purpose of being "examples" to demonstrate how a concept works, and kind of become there own thing... which is interested, but not helping me with abstraction. Perhaps giving a short summary of the example, relating it to abstraction, and the delving in deeper?

I think this paragraph is helpful for understanding how higher and lower levels of code work, which is an essential part of abstraction.

I agree; it seems difficult to see how this connects to the "bigger picture"

I think we've seen this same problem earlier in the notes - although some of these explanations are quite interesting in their own right, well written, and quite thorough, they are not always relevant to the discussion at hand.

Yeah it's kind of overwhelming looking at this text trying to understand it and also trying to connect it.

Even though this is slightly overwhelming it does do a bit to show how he is already using abstraction to make tree drawing much easier than using PostScript and slightly easier than using MetaPost by using the boxes language. Later on when he adds a tree language over boxes he attempts to show how adding that level of abstraction makes creating a tree even *more* easy than it was.

While I agree the example as walked through is rather confusing this paragraph does help show abstraction - possibly a better way to represent this would be by some sort of graphic? Possibly a "capabilities funnel" or "lines of code required comparison" of sorts showing the relative capabilities of the methods and how eventually x lines of tree language make y lines of boxes language make z lines of MetaPost and so on

so complicated to explain...

This is a very good observation I had not noticed

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

there's a wrong level of abstraction? isn't the only rule that the smaller sections function as expected?

I am also confused about this. I understand that something can be abstracted to where it is not as useful but how useful or modular does something have to be to be on the correct level of absraction? Why is this at the wrong level?

i think by "wrong" he means that the level of abstraction is not optimal for this example's purpose. i'd imagine that the correct level of abstraction would allow a person to perform their task without an unnecessary amount of repeated text/info. it's probably less definite and more "what feels right", if it seems that there's an unnecessary amount of code for a simple job, it might be at the wrong level of abstraction

Highlighting these repetitions would be helpful for the elementary programmer.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

What? What's a repeating motif? And how does it imply that its at the wrong level of abstraction?

I think a repeating motif is a section of code that is used over and over again with few differences. This "motif" can easily be written as a function in a higher level programming language.

Recursion is another way to abstract away repetition, and it's particularly helpful with tree structures since the same things are happening over and over at different levels of the tree.

I'm not sure because I'm not experienced with coding, but I think what the author means is that if you have code that comes up over and over again, you can write another program or method that handles that code–like in the last reading, the author was saying make each program do one task.

I think it means something that one should use a "for" loop for, in higher levels of program languages. It repeats, so you shouldn't have to copy-paste it again and again

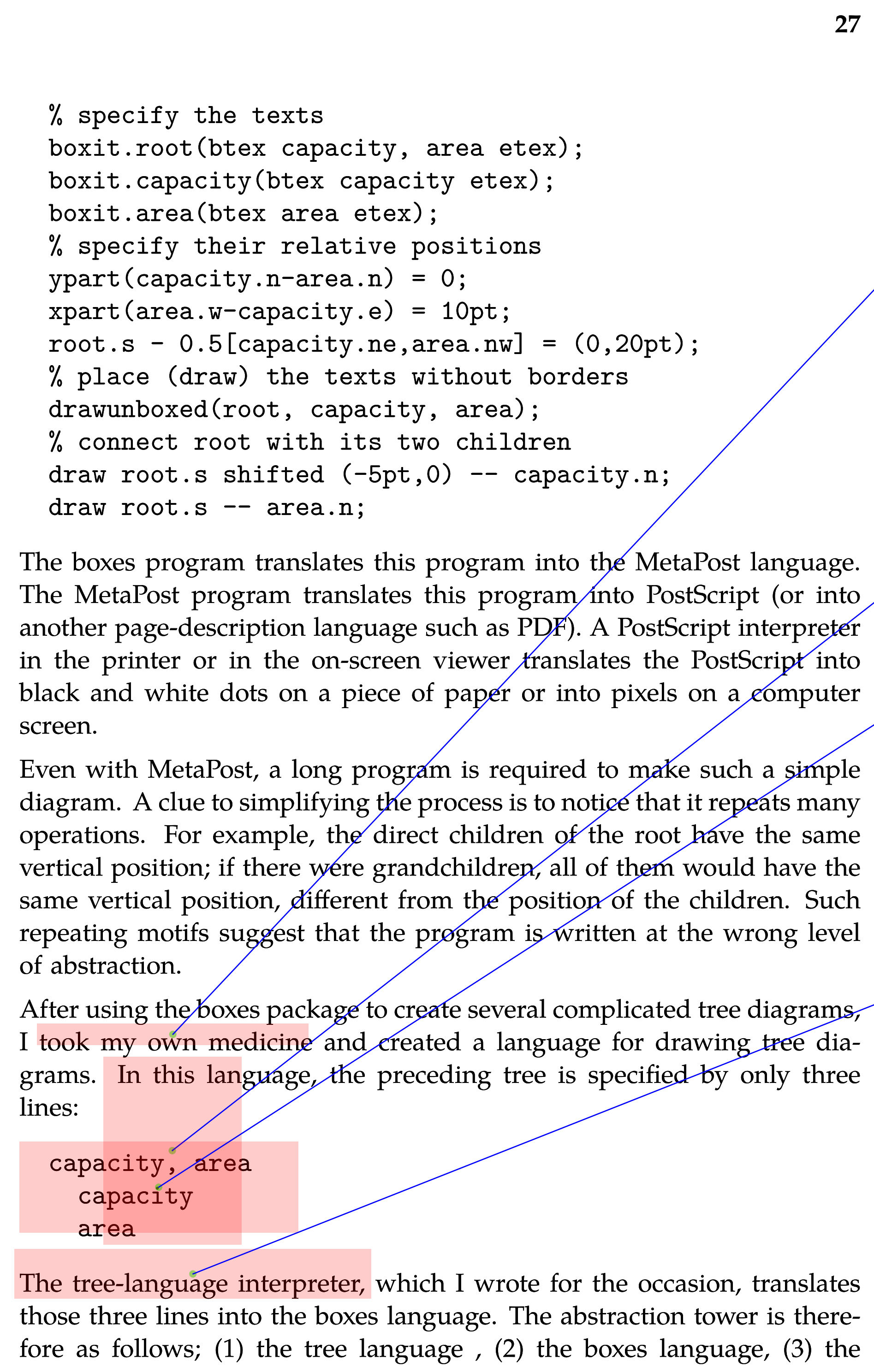
It doesn't even need to be in a "for" loop... sometimes you call the same function at various points in a program, and it is useful to have it as a separate function to call as opposed to repeating those lines of code over and over again... but there doesn't need to be a pattern in when they are called as using a "for" loop implies

I think the previous comment is correct - this doesn't necessarily mean use a 'for' loop, but you should write a program that takes in certain inputs. You can alter those inputs every time you call the program so that the program does what you want it to.

I would argue that a for loop isn't really abstraction and I don't see how a for loop would inherently change the level of abstraction of its specific functionality. I thought of that being done by making a parent class that has common functions for a type of objects. Then the children inherit that functionality without having to rewrite any code. Or something.

I am having trouble making a connection between what I am reading now and abstraction.

I am having trouble making a connection between what I am reading now and abstraction.



27

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

This sounds somewhat awkward to me, or it might just be an idiom that I haven't heard much.

I agree. I thought to "take ones own medicine" means to do something that you told someone else to do, and the idiom isn't very clear in this context.

This makes perfect sense to me, based on context clues. If anything, you might want to change "medicine" to "advice" for people that are struggling.

I understand what you mean, but I think "taking one's own medicine" has a negative connotation from its usage in other places.

Perhaps skip almost everything between where the problem is introduced and the part where you turn graphical trees into outline-type description. Then you can explain how the outline is created by abstracting from the tree. (Though perhaps a better example all together would be better?)

How do these three lines specify the positions and number of the boxes for tree? I don't understand - It says that the preceding tree is specified by only these three lines, I must be missing something.

The top and least indented line goes up top. The next two lines are indented, so that means they go below "capacity, area". Since "capacity" is above "area" and they both have the same indentation, then they go in the same level, but "capacity" goes to the left of "area". In summary, indentation determines what goes below it predecessors, and the order at which you write the words (with same indentation) just go right after each other from left to right. I hope this helped.

this tree-language interpreter was how long in itself? i.e.- was the time spent creating it cost-effective?

It is about 140 lines of Python. It has been very time effective! I just checked my tree directory (if the projector works, I'll show it in class) and it has 34 tree files, one per tree. They total 241 lines. They get turned into 34 files using the boxes language, for a total of 803 lines – and those lines are hard to write correctly.

Furthermore, if I decide to change how the trees are displayed – for example, sideways instead of vertically – I have to change just one program (the translator) rather than 34 individual files.

```
% specify the texts
boxit.root(btex capacity, area etex);
boxit.capacity(btex capacity etex);
boxit.area(btex area etex);
% specify their relative positions
ypart(capacity.n-area.n) = 0;
xpart(area.w-capacity.e) = 10pt;
root.s - 0.5[capacity.ne,area.nw] = (0,20pt);
% place (draw) the texts without borders
drawunboxed(root, capacity, area);
% connect root with its two children
draw root.s shifted (-5pt,0) -- capacity.n;
draw root.s -- area.n;
```

The boxes program translates this program into the MetaPost language. The MetaPost program translates this program into PostScript (or into another page-description language such as PDF). A PostScript interpreter in the printer or in the on-screen viewer translates the PostScript into black and white dots on a piece of paper or into pixels on a computer screen.

Even with MetaPost, a long program is required to make such a simple diagram. A clue to simplifying the process is to notice that it repeats many operations. For example, the direct children of the root have the same vertical position; if there were grandchildren, all of them would have the same vertical position, different from the position of the children. Such repeating motifs suggest that the program is written at the wrong level of abstraction.

After using the boxes package to create several complicated tree diagrams, I took my own medicine and created a language for drawing tree diagrams. In this language, the preceding tree is specified by only three lines:

```
capacity, area
  capacity
  area
```

The tree-language interpreter, which I wrote for the occasion, translates those three lines into the boxes language. The abstraction tower is therefore as follows; (1) the tree language , (2) the boxes language, (3) the

This sounds like an important term, like a tree.. What is it?

Was this term ever defined explicitly as different from abstraction? Is it just abstractions stacked upon abstractions?

I assume that the pixels/specks are the base of our new tower? A diagram, rather than parentheses and numbers, would do well here.

I think it's ok the way it is. I don't think you get any appreciable benefit from a diagram here–the simple list is fine.

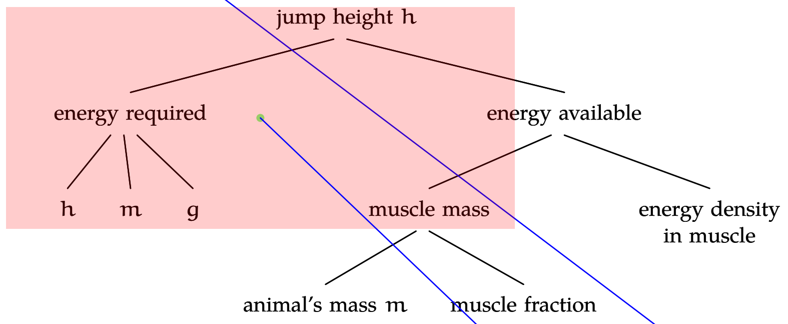
I disagree with the previous comment - a picture would be nice, if only to see what's at the "top" and the "bottom" of the tower - visualization and images are more helpful for me.

I agree that if the term is used, there should be a diagram of the abstraction tower both when it's introduced and in this specific example. This would illustrate the hierarchy. The earlier comment is correct that the tree language is the highest level of abstraction, so it would be at the top and pixels at the bottom. Rather than a tower, it could also be represented by a pyramid, because as indicated, the short tree program abstracts away many more lines of PS code.

This description of the abstraction tower is very helpful and this numbered list is an excellent way to summarize the levels of abstraction described in the previous paragraphs. A diagram may be helpful if the reader is not really experienced in the concept of abstraction applied to programming, but I don't find it necessary.

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the boxes language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

The method is a bit clearer here and comes together as you describe how you completed the problem. I think the use of a computer engineering problem is good because it shows the versatility of this method. However, it would help if many of the more technical processes were also summed up in a more simple manner.

Don't understand the last two as well- how can you relate pixels on a screen as being a lower abstraction level than a computer language?

I forgot that we weren't still talking about trees since this example of abstraction is all about the trees that were used to describe divide and conquer.

The coding example somewhat makes sense, but I don't think it should be related to trees since we've been associating them with divide-and-conquer.

Me too. This section has a lot of different concepts in it between the examples and relating to previous chapters. It's hard to keep track of which one we're focusing on.

I agree. Seeing the tree makes my mind jump back to divide and conquer and made it confusing as to what exactly we were looking at.

Why weren't they included in Chapter 1? I would include them and then in Chapter 2 reveal just how you made them.

Ok, I only followed all the following by reading the bottom paragraph. Perhaps rearranging the graphics would help and I lost where our 5 levels of abstraction went...

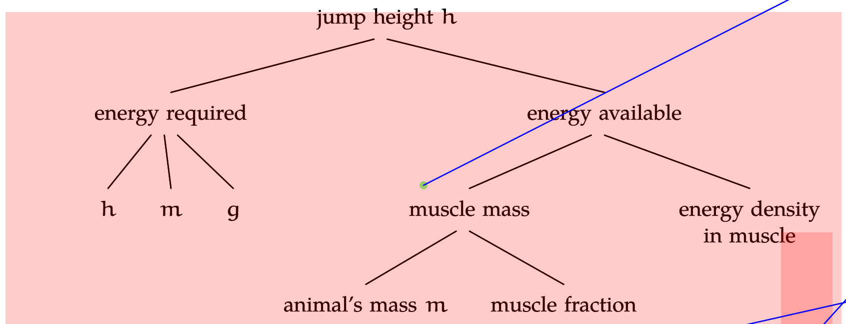
I couldn't find this example in any of the previous sections. Is this a section that we haven't covered yet? In that case why is it being used without any introduction? This slightly confused me, as I don't remember doing anything involving an animal jumping.

i think this is a later section in the book. if we had the whole thing, i'm sure we could flip forward and check it out.

I agree that abstraction is a powerful tool that can be very useful. However, I think for its introduction, another example would be better to use. Since we just finished discussing trees with divide and conquer, the use of a tree here sort of confuses and blends the two topics together. Understandably the topics are very closely related, but perhaps a different opening example would help differentiate them.

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the `boxes` language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

I'm not sure what this example is doing here...it's great to see a tree, but we've been seeing them for the whole previous chapter and we know that you can draw them. Without any introduction or qualification, it seems like this example just comes out of nowhere.

I think there might be a better example to start with, perhaps one with less code (although I understand most of the code) and more of a direct comparison to divide and conquer, and how this can be sued to solve problem that more people understand (for example, the oil barrel problem or something similar)

I agree. I think too much coding can easily confuse some audience like me who are not so familiar with coding. I think it's better to start with non-coding problems so that it's suitable for the general audience

And look like python, indentation-wise. It would be nice to mention the indents as an organizational tool!

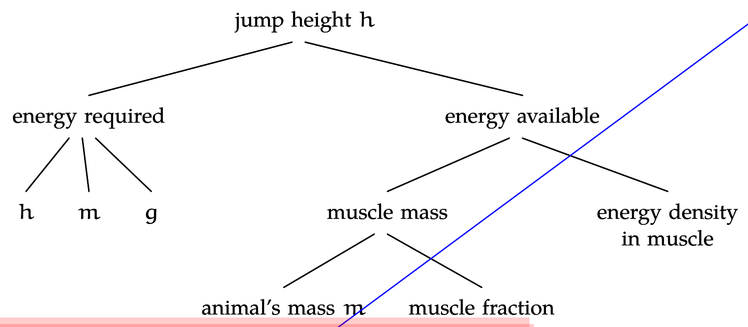
it bothers me that you are using a tree from a future section to explain this...i'd use the tree from part 1.3 or 1.4

I like the comparison here. It's good to see the tree and also the same in the tree mini-language.

Why does this h have to be between dollar signs? It's not in a separate tree..

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the boxes language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

what do these $ signs mean? maybe it's better to annotate them on the side?

In TeX, one typesets mathematics by enclosing it in dollar signs (that tells the TeX translator, "Here be mathematics. Turn it into nicely typeset text."). So I adopted that convention here. Thus, \$h\$ means the variable "h" (in italics). Similarly, \$a=bc\$ would typeset the equation a=bc.

In fact, the lines of the tree file get passed to TeX for typesetting, so I didn't even have to do anything special to use the math typesetting features of TeX (TeX is a useful abstraction!).

In TeX, one typesets mathematics by enclosing it in dollar signs (that tells the TeX translator, "Here be mathematics. Turn it into nicely typeset text."). So I adopted that convention here. Thus, \$h\$ means the variable "h" (in italics). Similarly, \$a=bc\$ would typeset the equation a=bc.

In fact, the lines of the tree file get passed to TeX for typesetting, so I didn't even have to do anything special to use the math typesetting features of TeX (TeX is a useful abstraction!).

I generally found that the examples here made sense, but I didn't feel that it really got to the heart of what abstraction is. We saw a way it was used to create a new programming language, but I think that without any coding experience, the abstraction used there would have been entirely lost on me.

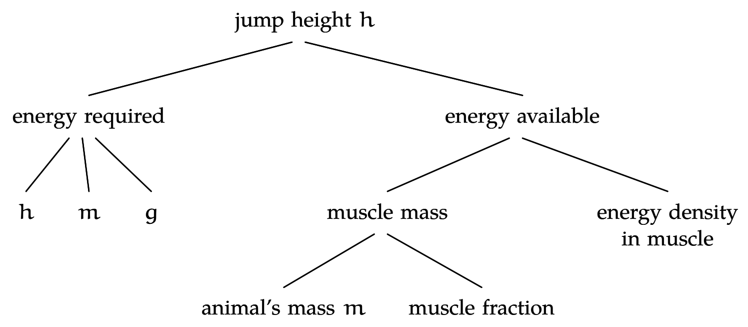
is this is coded tree?? thats kinda cool

I would kind of be interested in seeing this example before the first, because it wasn't until here that I actually fully understood the point you were trying to get across. I'm sort of interested in what would happen if you presented this example starting with this layer of abstraction and then exploring what is actually under the hood.

I see intuitively how this language relates to the tree, but obviously that's because this is some language the author made himself just for this purpose. I do not understand at all how he got from the 1732 lines of PostScript code down to these 10 lines.

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the `boxes` language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

I am a little confused as to which language this is in. Obviously it says "tree minilanguage", but I am confused as to whether this is what the author invented, or some combination of all of the other random languages discussed before, a little clarification would be helpfull. In general I would like a little more background when the author claims to invent a new language/method, becuase although it is clear to the author, I often find it confusing, which part he/she invented, why, etc...

I figured the author was talking about the language he invented, but I can see how this could be a bit misleading. However, the important part of this sentence is how the author hints that this process made constructing tree diagrams much easier than before. I don't think the reference takes away from the point that abstraction simplified the problem greatly.

It might not be quite as simple to understand as you think. I think you should explain more thoroughly how the program knows which lines go where on the tree. You should actually mention the indentations in the text somewhere. And you should also specify what the dollar signs indicate.

I think mentioning that you can use the indentations recursively might be useful. Also, you have to notice that the h, m, and g are in a different font (the dollar signs in TeX represent "math mode", and he mentioned TeX earlier... I assume they mean the same thing.)

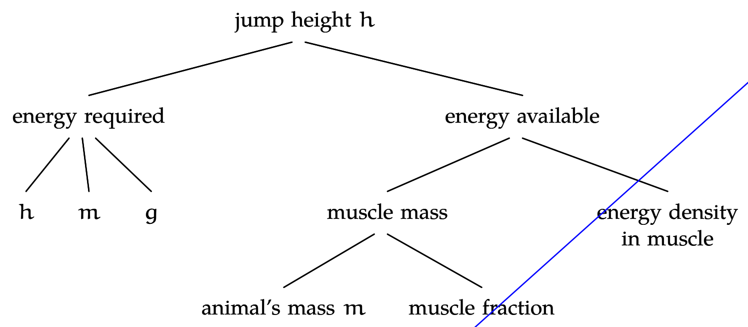
I agree, if I didn't see the tree right above it I don't think I would have been able to look at this code and draw the tree myself.

what are the dollar signs for? Is there a reason you need these special characters to specify single characters

It's just code, I'd guess it makes the letters italic or in the font that they're in

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
  jump height $h$
    energy required
      $h$
      $m$
      $g$
    energy available
      muscle mass
        animal's mass $m$
        muscle fraction
      energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the `boxes` language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

I'm probably in the MIT minority, but every time I see computer code I get lost. Don't know what any of it means.

Why, this one simply has the indentations defining the levels of the tree.

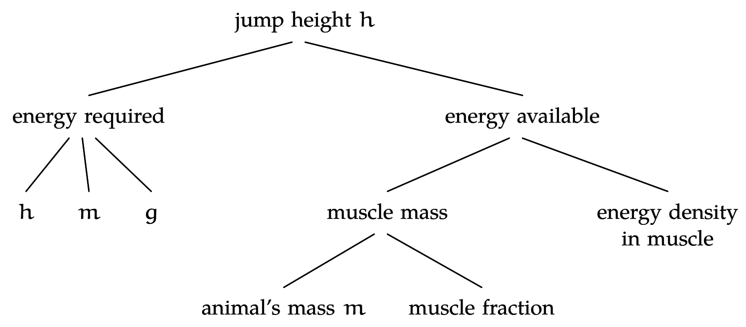
I agree with the first comment, if I were just given the code and not the tree, I'm not sure I would have known what I was supposed to do. The tree is helpful and I'm glad it comes before the coding because it makes the code easier to understand. That being said, even code that ends up being more simple like this one is intimidating to look at when you aren't familiar with programming.

Understandably, not everyone is familiar with programming and it can seem confusing to those not familiar with it. However, it is worthwhile to remember that the code presented here isn't in a familiar programming language that many people know, like C, Java, or Python. In the previous page, it is mentioned that this "language" was made up by the author. So, I have never seen this language or code before, and yet I can analyze its content and structure and make guesses about what it means. I feel like a lot of people in this class see code and panic and forget to spend the time trying to figure it out. If you encounter a new mathematical or engineering formula, do you panic because you've never seen it before? No, you slowly go through it term by term until you understand the whole thing.

Uh, that said, coding isn't so much about learning a language so much as a way of thinking... so if you've never coded before this is definitely hard to understand as an example, especially when you're still trying to wrap your head around the idea of abstraction. I guess what I mean that it doesn't help if the example you use to explain a confusing topic is also confusing.

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the boxes language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

I agree with the previous two comments. I think Anon1 is simply pointing out, as I have also observed, that any time there is a snippet of code, there are at least 5 comments about how it's confusing and asking why the code is there. I think this is something perhaps Professor Sanjoy should address directly, but I would like people to remember that this is a designated course 6 and course 2 class, both of which require you to have some exposure to coding (matlab in course 2), so really, there's no reason people should be so surprised by a few lines of some meta code (or even less so of unix command line prompts in the previous section). That said, I do grant Anon2 that reading and understanding code comes much easier to some people than others. If you're having trouble with this bit of code, just think of it as a list. Everyone knows how to use lists to outline ideas, and this is no different. The further indented the line, the more nested and subsidiary the element.

I guess this is part of the reason the class is in Course 6 as well. I think it's good to have Course 6 and Course 2 examples. But I am tempted to argue that Course 2 ish examples make sense to Course 6 while Course 6 examples make less sense to Course 2 students. What do you think?

Overall, the example is pretty easy to follow, but I'm not sure how well it actually illustrates -how- to abstract. It pretty much just says: graphs are hard, so we abstract to this outline form through (black box). As long as a later section tells us how to practically use abstraction, this is fine.

Why does one leaf have a variable and other don't. I feel like each end leaf should consist of a variable. Also are the indents defining whether a new node is made?

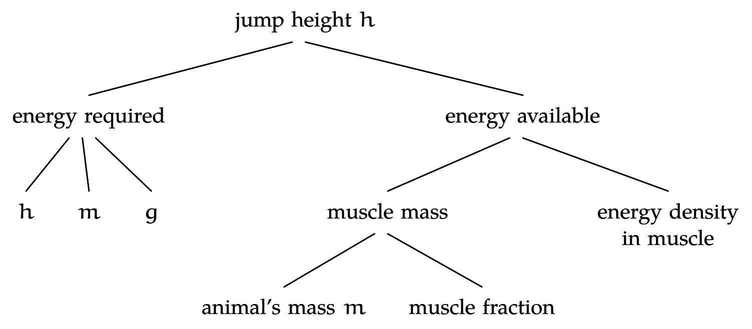
This is a much better example than the capacity, area one.

I think it confuses me more actually... maybe since the capacity one was something I could better visualize.

but since this minilanguage is not real, are we merely to take from this that we should be writing as elegant programs as possible (i.e. dividing and conquering with intelligent redundancy?)

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the `boxes language`. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

why is the boxes language error-prone? Is it always error-prone, or just in this particular case?

I think he was just pointing out that writing those 34 lines of code by hand would most certainly introduce errors the first time through, whereas using a layer of abstraction (his metacode), he was able to create those 34 lines in a fasion that did not produce errors, a sign of a good abstraction.

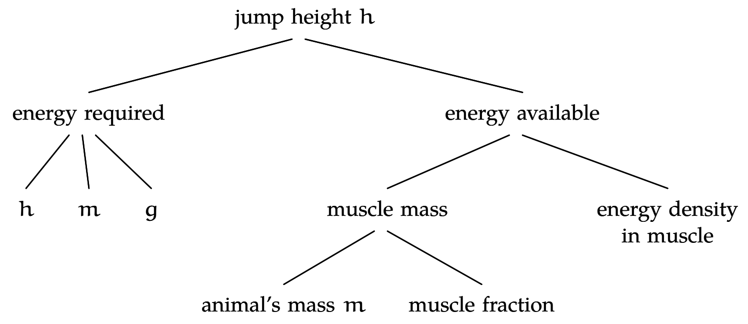
That is cool.

Agreed, this is a great closing paragraph and helps helps point out the now obvious benefits of abstraction.

Excellent closing though I feel that maybe this sort of growing code cost could have aided earlier in understanding the long list of boxes over MetaPost over PostScript. I do agree that this really nails the benefits of abstraction

MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the boxes language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

That's pretty cool, but I'm still waiting to see how this chapter and the last fit into the scope of this course.

The last chapter discussed divide and conquer and this one is discussing abstraction?

I like the way the divide and conquer section was set up with the concept explained with an estimation example and then used the code as an elaboration on the topic in a different section. I think a simpler "warm up" explanation of abstract would be good before adding the complexities of coding.

Agreed - it might have been better to begin this chapter by framing abstraction as useful for approximation. As it is, I think many people will read this chapter with the impression that abstraction is useful for coding.

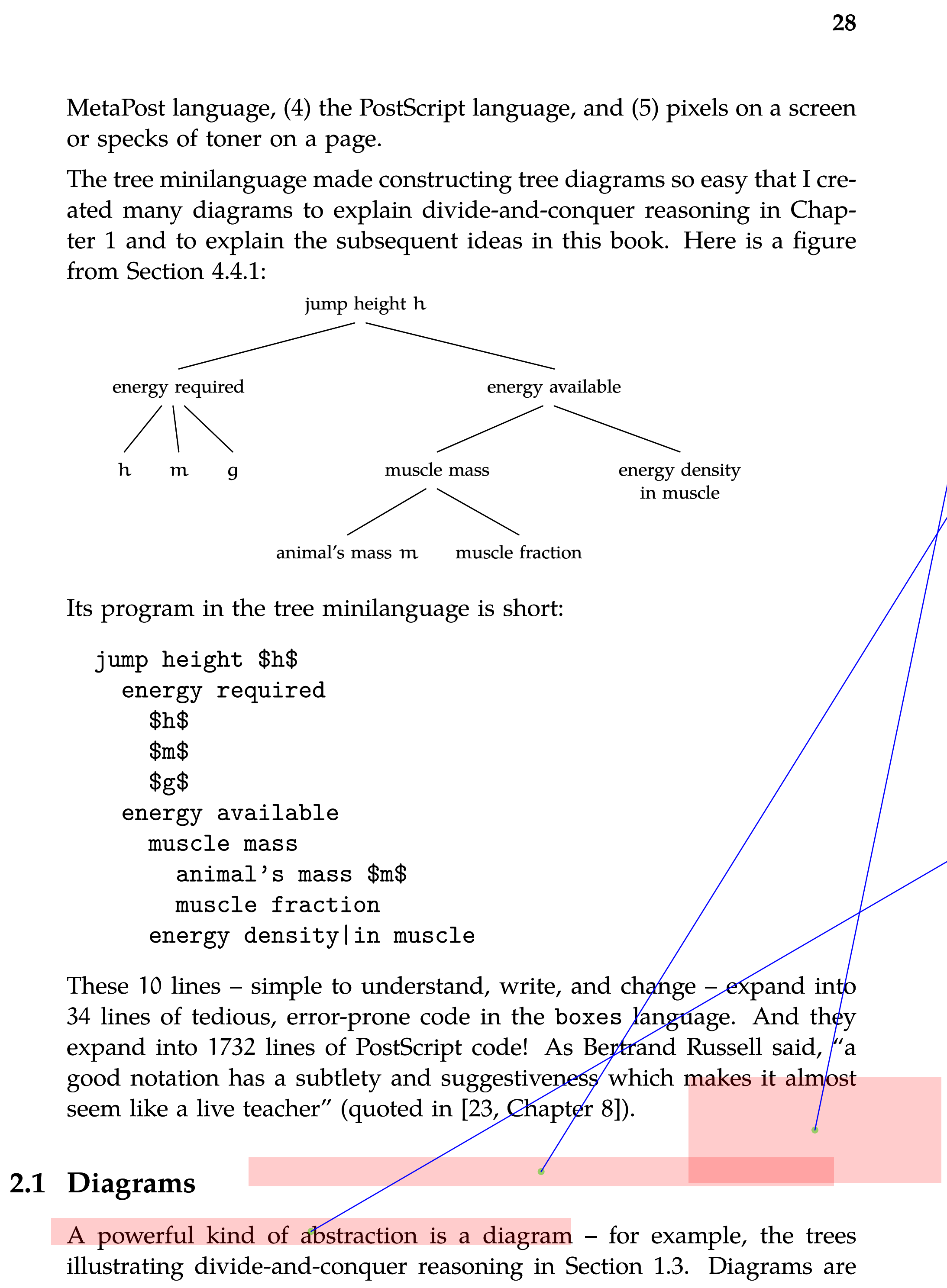
I agree. When the chapter began with abstraction, I was thinking approximation methods that can be abstracted to solve different types of problems. All this discussion of GUIs lost me, and I still don't understand what it has to do with abstraction.

I agree. I understand that this is the beginning of a new chapter and a new concept. However, in the introduction, it was confusing about how "abstraction" related to divide and conquer, where it fit into the picture.

Again, I agree. I think this in part stems from the fact that this new chapter begins with a definition of divide and conquer.

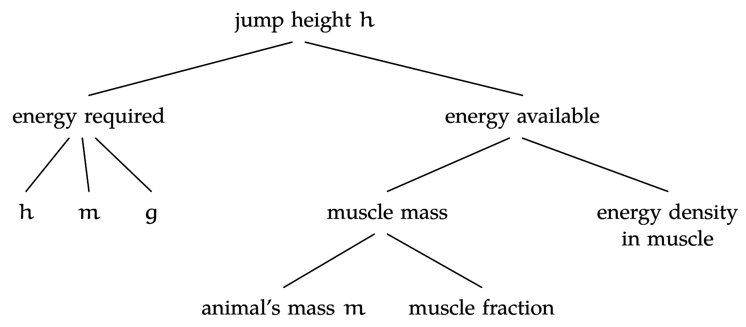
I agree that a little explanation of why abstraction is an important technique in approximation could have been useful at the beginning. Other than that, I thought the PostScript to tree minilanguage example was an excellent way to introduce the concept of abstraction especially to people who are not familiar with the concept from programming experience.

i love this explanation...really makes your point well and it's a little scary to think about...but cool



MetaPost language, (4) the PostScript language, and (5) pixels on a screen or specks of toner on a page.

The tree minilanguage made constructing tree diagrams so easy that I created many diagrams to explain divide-and-conquer reasoning in Chapter 1 and to explain the subsequent ideas in this book. Here is a figure from Section 4.4.1:



Its program in the tree minilanguage is short:

```
jump height $h$
  energy required
    $h$
    $m$
    $g$
  energy available
    muscle mass
      animal's mass $m$
      muscle fraction
    energy density|in muscle
```

These 10 lines – simple to understand, write, and change – expand into 34 lines of tedious, error-prone code in the `boxes` language. And they expand into 1732 lines of PostScript code! As Bertrand Russell said, "a good notation has a subtlety and suggestiveness which makes it almost seem like a live teacher" (quoted in [23, Chapter 8]).

2.1 Diagrams

A powerful kind of abstraction is a diagram – for example, the trees illustrating divide-and-conquer reasoning in Section 1.3. Diagrams are

this seems pretty inefficient. Aren't we normally striving to make things simpler and efficient? Also I am a visual person and in order to write a tree in tree language I would have to first draw it out on paper. Does this mean that the tree language wouldn't be very useful for people like me and that using a GUI would be better?

it's efficient if you're trying to visually represent your hand-drawn tree on paper. imagine how else you'd be sending the document to a friend without a scanner at hand. you'd probably use word or ppt to draw boxes and draw lines between them. or you could use this concise program to do the same thing with no drawing, just spaces and words

I didn't understand hardly any of this section. The example gets too caught up in the programming details without referring back to the abstraction concept often enough.

I agree- I don't feel like the idea of "abstractions" was represented- although that might be my own confusion

Although I can see this example as a demonstration of abstraction, I agree and think there was a little too much coding and specific references to unfamiliar languages... at least for an introduction. While it still greatly applies the principles of abstraction, it might be better suited for a subsequent chapter, where we get into a little more detail explaining abstraction.

The main idea is that programs one can write a program to take an input and do something with it independently of your main objective. Hence your solving two seperate problems.

this sentence is awkward...use 'diagrams are a powerful form of abstraction.' ?