Computer Science Literacy

Rushi Shah

7 November 2016

One of the courses every student at my high school was required to take was "Foundations of Computer Science". I ended up taking for granted that my peers had a certain level of computer science literacy that was surprisingly useful in explaining topics that seem orthogonal to computers in general. Now that I attend a school where no such requirement is imposed, I've noticed how useful general computer science literacy would be.

With that being said, in the coming years I doubt everyone will be expected to program their own solutions to problems. Instead of expecting everybody to be programmers, I expect everyone to have a base level of understanding of how the Computer Science field works and can be useful to their lives past the finished products they use. I equate this idea to how everyone knows a bit of math. Arithmetic corresponds to knowing how to use a mouse, type, and just in general get around computers. But a lot of people know a lot more math than just basic arithmetic. Here is an outline of the topics I think the general populus should know and a bit of why:

1 Types

I can't emphasize this enough, types are the core of Computer Science! Type intuition is the most applicable aspect of computer science to the non-tech community (which is probably a pretty unpopular opinion). I argue that every domain has an implicit type system. And recognizing this type system can clarify why some things that *are* wrong should intuitively *seem* wrong.

For example, in my introductory physics class, my teacher suggested checking that the units on your final answer correspond to the units the question is asking you for. For example, this dimensional analysis makes sure you don't give speed (distance per time) when the problem asks for just distance. Similarly, he suggested making sure that every piece of provided information has been "used up" [1]. In the speed example, this would imply that there is probably some time variable you forgot to take into account and multiply by. In physics, the units of your computations represent a type system!

Similarly, in the study of languages, some sentences just don't make sense. If you replace a noun with a verb, everything is just going to sound wrong. Reasoning about the types of speech can help you learn a new language, identify errors, etc. In grammar, parts of speech represent yet another type system!

[1] This approach is remarkably similar to type-directed program synthesis research being done at UT

The literacy I propose doesn't need to be perpetuated because of these examples (physicists and linguists don't necessarily need to understand the type system to understand what they're doing). But these examples demonstrate a larger point I'm trying to make: types are everywhere. Educating people about types in a more concrete sense will let them identify more abstract representations of types in their life.

What do I mean by educating people about concrete type systems? Every high school graduate should at least understand what a boolean and string are. They aren't difficult concepts to grasp once they've been explained, but without the requisite knowledge those two words seem intimidating and foreign. Its not a matter of destroying the jargon (which in this case I argue there is little of), but rather of introducing concepts. To return to the math literacy example: people know what "probability" is, but if they didn't, it would sound like a big scary word.

And the cool thing about types is you can demonstrate them in a REPL. The student doesn't even have to write programs to see why you can subtract two integers, but not necessarily subtract two boolean values. So I propose people learn about the basic types: booleans, numeric types like integers/floats/doubles/etc., and Strings.

Then, maybe learn a smattering of functions to combine these basic types together and show why types can help the computer reason about programs. I'm not sure if they should be introduced here or not, though. They are a topic in and of themselves, so they can't just be an add-on afterthought at the end of types. But, it would make sense to introduce them here to apply what we've learned about types. I'm torn and don't have a good answer.

2 Basic Data Structures

Okay great, people know what a String is. So like you can store your name in a variable. But what if you want to store everybody in your class's name? Would you need a variable for each person? Of course not, there has to be a better way to store lists of names. So everybody knows data is a thing, but perhaps they don't consider the fact that you can take a bunch of data and structure it. For example, you can take a list of names, and store them as (surprise, surprise) a *list* of Strings.

I would propose that the next step is at least teaching people what lists (arrays) are as an introduction to the concept of `data structures`. If I wanted to be optimistic, I would also recommend a few other data structures, namely trees and graphs. Trees represent hierarchical data that is overwhelmingly common and unique to other types of data structures in the way that you can consider subtrees at a time. Graphs will give the notion of having pieces of data point to other pieces of data, which is a valuable intution.

The emphasis here is not implementations of data structures, efficiency, or any such nonesense. Instead, we are trying to talk about data structures in a more abstract sense. We want to demonstrate why we need them and how they are useful.

3 Functions

What are computers good at that humans aren't? Doing the same repetitive task over and over (perhaps with slight tweaks) quickly, efficiently, and accurately. This is where functions come in, and they are fundamental to computer science. So much so, that I don't think anybody can be considered "literate" without at least being introduced to the idea of functions (or methods, or procedures, or whatever you want to call them).

This is especially easy because functions bear a striking resemblance to the functions everybody was introduced to in their high school math classes. They just take some input x and return some output y. The only difference is that x and y don't have to just be real numbers anymore, they can be any one of the types we outlined earlier. Functions will be presented as just bits of code that operate over specified inputs to give specified outputs and can be invoked as needed to execute that code over and over again.

4 Basic Control Flow

At this point, hopefully computer science will seem a little less daunting. Now that they know the atoms of programs, the student should start messing around with it a little bit to see that "hey this isn't so bad!". In the process of doing so, teach them if-statements to use those booleans they learned about. Teach them iteration to use those lists they learned about. Teach them recursion to use the functions they learned about. At this point, we might venture into the territory of teaching people how to program, rather than just about Computer Science, but it will be a worthy detour to tie everything together.

4.1 If Statements

Booleans can represent our world. From simple (is it past 5 o'clock) to complex (is a user logged in), often our world is binary. Since we've already shown the idea of boolean values, the student needs to know what they're good for. And the answer is naturally if-statements. The computer should do something different depending on what is going on in the world[2].

4.2 Iteration

After they understand boolean predicates, they can be introduced to while loops which use the same idea. The only difference is iteration allows you to tell a computer to just "keep doing something over and over". Since I'm guessing we're not going to just teach them functional programming, the ideas we want to introduce are while-loops and for-loops. Segway into this section by tying it to if-statements, and ultimately emphasize the importance of iterating over data structures.

4.3 Recursion

Recursion is another one of those things that just keeps popping up all over the place (like types). For example, conversations are recursive: you might discover something else to talk about in the process of talking about something, and when you exhaust your detour you should go back to the original topic. So what we're going to explain is the idea that sometimes

[2]Perhaps at this point we could take a detour to explain what "state" is. If we're doing so, we should explain why anything other than finite state is bad

when you finish doing something, you want to start over from the beginning. Recursion can be daunting to some students, but there's really nothing to be scared of. And since we're not tying ourselves down to teaching students how to program in one language specifically, we can teach recursion without the shackles of a language that is no good at recursion. We could use the fibonacci example here, because that's what people are used to, but I think it is a bad example. I would much prefer introducing trees in the Data Structures portion, and then introducing tree searching or something at this point.

5 Conclusion

I haven't hammered out the correct order yet, because should students learn functions first or data structures first? And should they start programming early or later? And should we integrate the iteration in more tightly with the lists and the trees with the recursion? There are just a lot of unanswered questions, and I think the biggest source of confusion for me is when should things be taught and what things should be taught together.

Regardless, these are the bases that should be covered for so called "computer-science literacy". Just having a general sense about the words and what they mean can help the lay-person get their bearings in the Computer Science field and figure out if it is something they want to pursue further. It would help demystify the little bubble we've created for ourselves to the non-tech community.