Proposed high-level overview of CS curriculum for the Student Preparedness Seminar

Jonathan Lam

2022/04/27

This is a proposed lesson outline intended for three fifty-minute teaching sessions. None of this is final, and represents the (very biased) views of the author.

Contents

1 Lesson 1: The essence of computing

The purpose of this section is to free ourselves from preconceptions about computing, since computing is not inherently any one thing. It is not typing in front of a computer, or programming games, or walls of falling green ones and zeros.

> The stuff we call "software" is not like anything that human society is used to thinking about. Software is something like a machine, and something like mathematics, and something like language, and something like thought, and art, and information... But software is not in fact any of those other things.
> – *The Hacker Crackdown*, Bruce Sterling

Our treatment of computing will be at a fairly abstract level. The goal of this three-lecture sequence is to provide you with the basic logical skills useful to approach any programming or related problem.

We will not be doing any "real" programming here, in the sense of running code on a computer. That is because, while producing and testing real code is an essential part of the software development cycle, it is not inherently necessary to learn about programming. Moreover, there are too many intricacies of real languages to get confused with with in a three hour sequence. Many talented mathematicians existed before the advent of the computer, and the computer greatly augmented their ability. We do the same as them, simulating programs using pen and paper. We will be operating using toy programming languages, equally expressive but much simpler than "real" languages[1].

1.1 What is computing?

A computer performs computations, and nothing more. It can do this very fast, but it can't do anything that you couldn't do by hand. You can simulate what any computer could do in a finite amount of time, given a large but finite amount of time, paper, and patience.

This is true of every piece of software you know. While much of software seems like magic, we may decompose a complex piece of software into its simpler parts, and those parts into simpler parts, and so on *recursively* until we reach the hardware level. *Abstraction* is the process of building complexity out of simpler parts, and we will study this in Lesson 2.

TODO: motivate why you want to learn computing: subfields of cs and active research, e.g.: machine learning (including NLP, deep learning); scientific computing, modeling for engineering and science, and numerical analysis; formal program verification; cybersecurity; systems programming and embedded programming; software engineering (probably most "industrial" and large scale,

[1]Not dealing with real code has two additional benefits other than teaching fundamentals: firstly, that CS curricula in high schools is highly unstandardized, so we cannot expect a prior knowledge in any particular language. Additionally, even if everyone has a background in the same language (e.g., Python or Java), the number of technical issues with setup are prohibitive, especially within such a short lecture sequence.

includes web development and cloud computing); (probably missing many fields here)

1.2 What is a programming language?

(Natural) languages are *interfaces* which allow for efficient and precise communication between people. In the same way, programming languages are interfaces between a human and a computer. Unlike natural language, we need to be very careful about specifying a programming language to avoid ambiguity.

When we specify a programming language, we specify its *syntax* and *semantics*. The syntax is the grammar of a programming language, in the same way English has a grammar. The semantics are the meaning of a programming language.

Usually we distinguish between two types of semantics. *Static semantics* usually refers to *compile-time type-checking*. *Dynamic semantics* refers to the behavior of *evaluation*. Evaluation is the process of reducing an expression down to an irreducible expression, or *value*.

1.2.1 Typing systems

types are used to (statically) predict the behavior of programs

classifications of typing systems: static vs. dynamic vs. gradual typing, manifest vs. inferred types, algebraic datatypes

more powerful typing systems allow you to do type-driven programming and even express proofs using types (e.g., Agda); types can carry a lot of meaning

1.3 Exercises in specifications

TODO: give several word problems to translate between different specifications (including word problems)

1.4 Sample (programming) languages

a.k.a. logic systems, a.k.a., specifications, a.k.a., interfaces

the math language: the one you've learned for a long time (easy to have recursion (and loops) but need to add conditionals to make this interesting)

excel programming: macros are functions

an imperative flowchart language

LISP (a "real" programming language!)

the lambda calculus (pretty much any functional language!)

the WHILE language (maybe?)

formal logic, inference rules

2 Lesson 2: Abstraction makes programming powerful

2.1 The importance of interfaces

abstraction promotes the use of black-box implementations, we care only about the interface

define the term "API" – very generic term

data and functional abstraction give common interfaces to data structures

again, usefulness when communicating/collaborating with others

2.2 Data abstraction

introduce pairs, linked lists, record types

introduce algebraic data types (sums and product types)

2.3 Functional abstraction

show how functions allow you to not repeat yourself

built into the lambda calculus, so we can use that for our notation

2.3.1 Higher-order functions

generic operations when there is a shared type

2.3.2 The `map`, `filter`, and `fold` operations

TODO: show how you can build up substantial algorithms from these

2.4 Modular abstraction

keeping code dry, encapsulation, interfaces between modules written in the same language

a.k.a. packages, crates, classes, etc.

2.5 Exercises in abstraction

quicksort example from constituent pieces

examples using map, filter, reduce

factoring out common pieces of code using functions

3 Lesson 3: Standard debugging techniques

3.1 Motivation

common complaint from students: “I don’t know how to start” or “this code doesn’t work”; e.g., see many questions on stack overflow

also will be useful in other engineering disciplines, you’re not really taught this

useful skill for interviews, people will ask you to do things out on a whiteboard, you won’t have a computer out to help you and you have to walk through your code logically, to check whether it will work or not: “compile in your head”

3.2 Overview of techniques

syntactic vs. semantic errors

to handle semantic errors, you first have to understand yourself what your code does; many students I have say their code doesn’t work but don’t know what it means for their code to work

3.2.1 Note error messages

this is the first thing to do

keep track of error messages. these are the first sign that a problem has occurred, and may happen if your code doesn’t compile or has a run-time error (e.g., a dynamic type error)

an error message is crucial if you end up asking someone else about the error. it is also probably the quickest way to find the solution (look up the error message/code)

note that sometimes, you may have a problem that doesn’t throw a error; in these cases, it may be because the program is silently failing, or otherwise producing incorrect output. these tend to be harder to debug, and you would start by narrowing down the problem

3.2.2 Narrow down the problem

this is probably the most generic advice

if possible, find out what file the error comes from. if possible, the function or even the line of code. this is the first step of any debugging after noticing an error

this may mean reducing down your code to a small example in which you can reproduce the error, which will make it easier to single out the bug. comment pieces of code, run your code one line at a time, etc. until you narrow down to a place where the code’s output doesn’t match what you expect. use a lot of print statements or a debugger

3.2.3 Step through your code on paper

one of the best ways to make sure you really understand what your code does, or what someone else's code does (in the case of reverse engineering the code from class or someone else on the web)

also reinforces the fact that anything a computer can do, you can do (albeit a lot slower most of the time)

start with some interesting test case and go from there. if you have the intuition that a particular test case is faulty, then start with that

3.2.4 Learn your tools

debuggers, ides, repls, etc. will make your programming experience more fun, efficient, visually appealing, etc. find one that works for you; learn its keyboard shortcuts and the debugger functions so you know how to step through your code

give a brief introduction on a debugger (e.g., gdb in intellij)

also github for version control, great for collaboration and version control (i won't get into details for sake of time)

3.2.5 Test your code (often!)

testing is a huge part of modern software development; it (and also security checks) are deeply integrated into the industrial setting, but are often poorly integrated into an academic environment

unit testing is useful, write tests for each functional block/module of your program; we built up abstractions, now we need to check that each one does what it should do

test-driven development: build your test cases before you even write your code, similar to walking through code before writing any code; shows that you really understand what you're looking for and understand the edge cases

how to formulate test cases; some examples and edge cases

integration testing builds a particular block into a larger system, probably not as relevant

3.2.6 Using your resources

the web is a resource! Stack Overflow, Reddit, etc. documentation pages. of course don't look up your questions verbatim, but for syntax/language-specific issues, people online will be happy to help, as well as teachers

in order to save the time of other people and yourself, make sure to ask a good question. this means that you have followed the previous instructions:

- provide as much relevant information as you can, especially error messages and other debugging output
- narrow down the problem.

- provide a minimum working example (MWE). this will show that you've narrowed down the problem, and also allow give other people a small example with which they can reproduce the problem and debug on their own machine
- be polite

3.3 Debugging exercises

TODO: simple debugging examples using the specification language(s) from the first week, ideally using real bugs

4 Useful references

- *Practical Foundations for Programming Languages, Second Edition.* Harper. (Useful for foundations of programming language theory, such as syntax, semantics, typing, and the λ-calculus.)
- *The Structure and Interpretation of Computer Programs, Second Edition.* Abelson and Sussman. (Useful for discussions of abstraction, and the LISP programming language.)