fundamentals of computer science

Tiffany Madruga

Open SoARce

THE FUNDAMENTALS OF COMPUTER SCIENCE

A CRASH COURSE

Open SoARce

Tiffany M. Madruga
Harvey Mudd College
Pitzer College

Created using
AR technology.

Acknowledgements

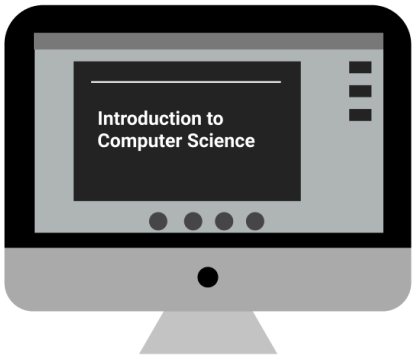
This project would not have been possible without the support of my advisors and peers. I would like to extend my sincerest thanks to Professor Carlin Wing, my thesis advisor, for providing advice and support surrounding the project as it evolved. I could not have completed the project to the state that it is in without her consistent support and reviews.

I would also like to thank Professor Ming-Yuen S. Ma, and my peer reviewers, Akari Ishida, Zayn Singh, Suki Liang, and Eric Culhane, for providing guidance throughout the process. The feedback received helped guide me through the direction that I wanted to follow throughout the process. In addition, my computer science professors at Harvey Mudd College, specifically Professors Ran Libeskind-Hadas, Colleen Lewis, and Zach Dodds, were all immensely helpful during the initial stages of my project.

In addition, much of the content used in this demo textbook is heavily borrowed from the textbook *Structure and Interpretation of Computer Program* by Harold Abelson and Gerald Jay Sussman, which is licensed under a Creative Commons Attribution-ShareAlike 4.0 International. This textbook uses material from this text and adds some additional information. In addition, some of the content for this textbook was inspired by the CS5 Gold: Introduction to Computer Science course taught at Harvey Mudd College.

Finally, I would like to thank my friends and family for providing support and feedback during this process. Without their continued encouragement, I would not have been able to complete the project.

1 Introduction to Computer Science



1.1 What is Computer Science?

Computer Science is officially defined as "the science that deals with the theory and methods of processing information in digital computers, the design of computer hardware and software, and the applications of computers."[1] In other words, it encompasses many disciplines that ultimately come together to create the automation that we take for granted today. It primarily uses algorithms to manipulate, store, and communicate digital information which can then be used in several different contexts. The goal of this text is to teach the basics of the field and provide a crash course for an overview of some of the most important topics in the field.

1.2 Is Computer Science Programming?

Usually, the phrase Computer Science can immediately conjure the idea of programming and software engineering. These subsets of computer science refer to the physical developments of programs, which will be discussed later, and software that can be used on any electronic device. Although programming is a major aspect of the field, it is just one element of the science. Computer Scientists solve problems that range from the abstract — determining what problems can be solved with computers and the complexity

of the algorithms that solve them – to the tangible – designing applications that perform well on handheld devices, that are easy to use, and that uphold security measures.[2] So, in short, Computer Science is not necessarily just programming. In reading more of the textbook, you will come to realize the truth of this statement

The major subjects that will be covered in this text include programming, computer architecture, algorithms and data structures, operating systems, computer networking, databases, languages and compilers, and distributed systems. Each of these subjects will be covered briefly in a crash-course format so that upon finishing the text, the reader will have a basic understanding of each of the major topics in Computer Science.

1.3 What is a Computer?

Figure 1.1: Image of an older computer, a Macintosh SE30.
Photo Credit: Cornellanense. Wikimedia Commons.

Before learning about the components that make up Computer Science, it is important to understand what a computer is. According to definition, a computer is "an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program."[3] In short, a computer is a machine that is able to be programmed. It consists of both hardware and software. The hardware portions of the

computer are the physical parts of the device – the wires, circuitry, etc. Meanwhile, the software portion is the digital portion — what people generally think Computer Scientists deal with. This is primarily the code, instructions, and data.

However, when thinking of the oldest computers, one may think of the image on the previous page. A larger desktop that controls programs. This is mostly correct. However, the oldest computers were not actually digital at all. "Computers" existed as early as in the 18th century. However, these 'computers' were just humans who performed the intensive calculations that machines do today. Since human computers were used to literally compute calculations in order to obtain results, it makes sense that computers now do the same thing, and much, much more.

The basics of Computer Science lie in the realm of math. In order to fully understand the components of the field, it is important to have a strong foundation in math. There is also a somewhat significant difference between a Computer Scientist and a Software Engineer. This textbook will start to prepare the reader to embark on a journey to become a Computer Scientist rather than simply a Software Engineer.

Software Engineering refers to a specific subsection of computer science that will be mentioned throughout the text. Software engineering mostly refers to building and maintaining software. On the other hand, Computer Science focuses on building the skills and breadth of knowledge necessary for creating the next generation of people who can shape the software and hardware of the future. Nowadays, several online crash courses are available for students to learn how to code quickly. These courses will allow students to learn the quick and dirty approach to becoming a software engineer, without building the breadth and depth of knowledge necessary to become the next great thinkers in the field.

1.4 Why study Computer Science?

The world is primarily shifting from an analog to a digital world. Newspapers are becoming obsolete as people are turning to the internet to find news. Computer Science is the basis of the digital world. Everything from websites, to cell phones, to machinery can use some form of it to enhance technology. Furthermore, many other fields can be automated and advanced with the addition of Computer Science. However, not only is studying Computer Science important for understanding how to manage software and other forms of computers, it provides a foundation for problem solving.

Figure 1.2: Seniors holding graduation caps.
Photo Credit: fotoinfot, Stock Images.

Furthermore, the job market is incredibly versatile. Skills acquired from learning Computer Science can be applied to any field. For example, computational skills are useful in fields of biology, entertainment, or even English. Some of the biggest advancements have been made possible because of the field. As the world is becoming more automated, it is important to understand the basics behind how the world works. This will ultimately manifest in learning each aspect of Computer Science which we will cover in this textbook.

As the job market continues to require Computer Scientists to help develop the software that can help save lives, learning the foundational materials will help anyone understand how to contribute to society in the new age. This digital language will become a necessary skill for everyone to know.

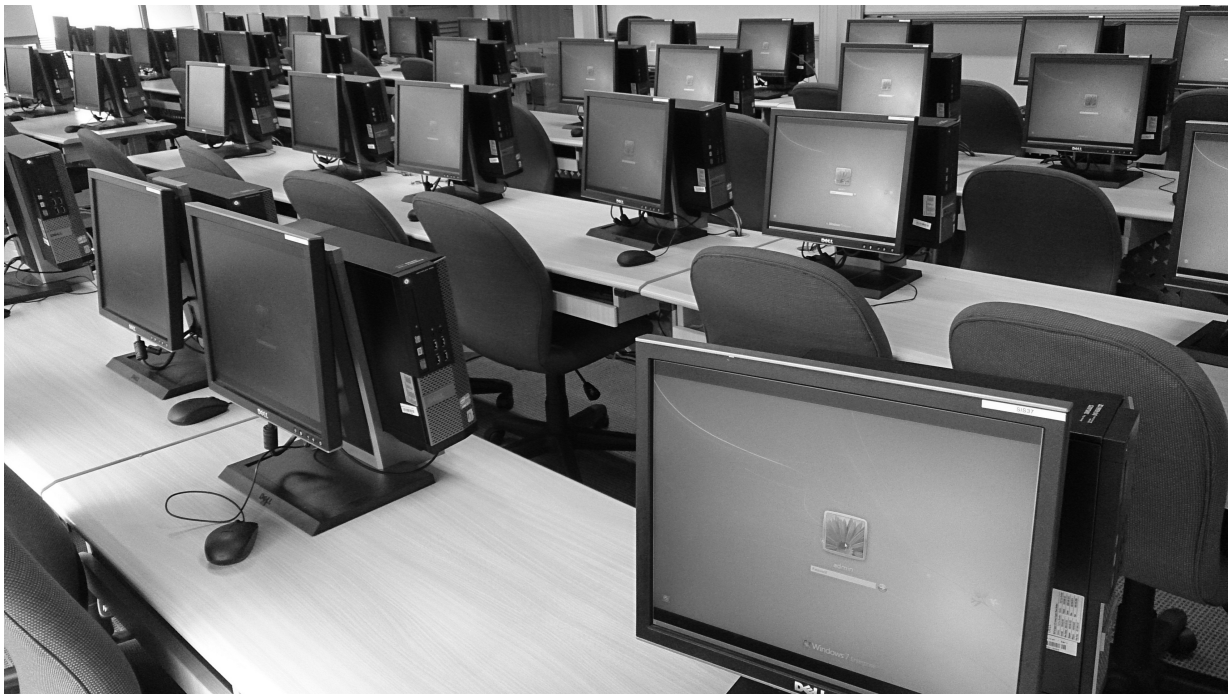


Figure 1.3: A computer lab.
Photo Credit: Wikimedia Commons.

1.5 Summary

This section covered what this textbook hopes to explain as well as give a brief background into what Computer Science really is.

- Computer Science is a large field that encompasses many disciplines
- Major sections of computer science include programming, computer architecture, algorithms and data structures, operating systems, computer networking, databases, languages and compilers, and distributed systems.
- A computer is a machine that is able to be programmed.
- Computer Science is an up-and-coming field that is worth studying because of the myriad opportunities available for Computer Scientists to make an impact.

2 PROGRAMMING

2.1 What is a Computer Program?

What is a computer program? A computer program is a collection of instructions that performs a specific task when executed by a computer. [4] Most computer devices require programs to function properly. A computer program is usually written by a computer programmer in a programming language, which will be discussed further in the text. This program is a human-readable form of source code which will then be interpreted by a computer through a compiler that converts the code to machine level code. This machine level code is in the form that a computer can execute.

2.2 What makes up a programming language?

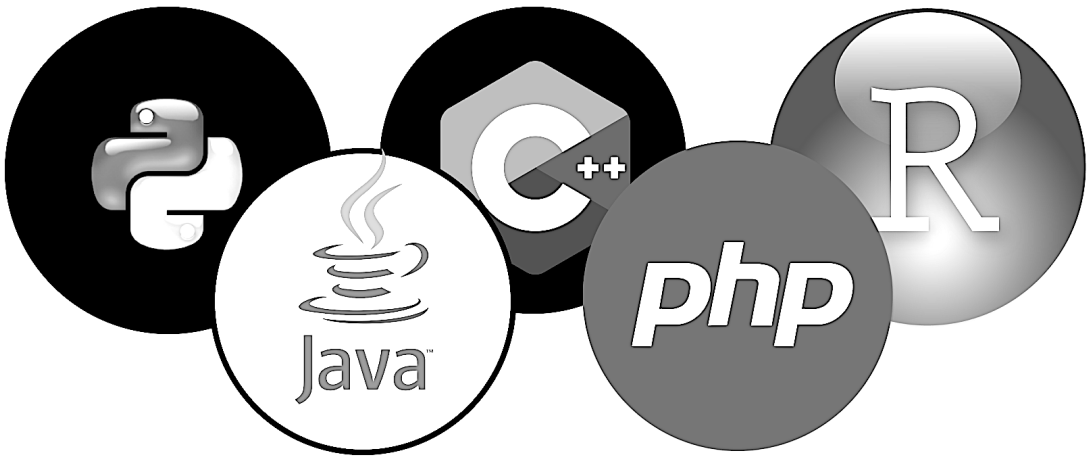


Figure 2.1: Examples of programming languages. From left to right: Python, Java, C++, PHP, and R.

A programming language helps organize the ideas behind a process. These languages therefore need a way to create more complicated ideas from simpler ones. As mentioned by the *Structure and Interpretation of Computer Programs*, every powerful language has three mechanisms for accomplishing this:

- primitive expressions, which represent the simplest entities the language is concerned with,
- means of combination, by which compound elements are built from simpler ones, and
- means of abstraction, by which compound elements can be named and manipulated as units.[5]

Programming primarily consists of data and procedures. We need procedures to manipulate data. Data is the smaller building blocks that make up a program while procedures are what we can use to combine them. In order to achieve this, all programming languages will need something to describe the smallest versions of these building blocks and procedures and ways to combine them to become more complex.

2.3 Expressions

The easiest way to start programming is to jump into a sample program. Since there are so many programming languages out there, it can be difficult to figure out how to start. We can begin with a programming language called Lisp, one of the earliest programming languages.

The most basic part of a program is a primitive expression. One kind of primitive expression you might type is a number. One easy way to get started at programming is to examine some typical interactions with an interpreter for the Scheme dialect of Lisp. Imagine that you are sitting at a computer terminal. You type an *expression*, and the interpreter responds by displaying the result of its *evaluating* that expression. If you present Lisp with a number

```
486
```

the interpreter will respond by printing

```
486
```

Expressions representing numbers may be combined with an expression representing a primitive procedure (such as + or *) to form a compound expression that represents the application of the procedure to those numbers. For example:

```
(+ 137 349)
486
(- 1000 334)
666
(* 5 99)
495
(/ 10 5)
2
(+ 2.7 10)
12.7
```

Expressions such as these, formed by delimiting a list of expressions within parentheses in order to denote procedure application, are called *combinations*. The leftmost element in the list is called the *operator*, and the other elements are called *operands*. The value of a combination is obtained by applying the procedure specified by the operator to the *arguments* that are the values of the operands.

Most, if not all, programming languages start with the very basic forms of variables. So, even though we are beginning to learn programming using Lisp, the many different programming languages operate under the same format, although they may look a bit different. Now that we have the basic building blocks of a program, we can start to make more complex calculations.

2.4 More Complex Programs

We have identified in Lisp some of the elements that must appear in any powerful programming language:

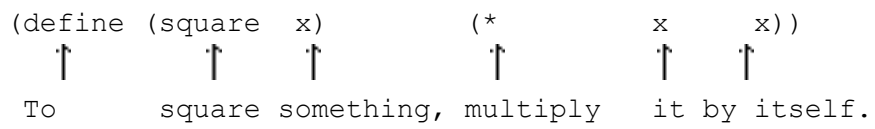
- Numbers and arithmetic operations are primitive data and procedures.
- Nesting of combinations provides a means of combining operations.
- Definitions that associate names with values provide a limited means of abstraction.

Now we will learn about *procedure definitions*, a much more powerful abstraction technique by which a compound operation can be given a name and then referred to as a unit.

We begin by examining how to express the idea of "squaring." We might say, "To square something, multiply it by itself." This is expressed in our language as

```
(define (square x) (* x x))
```

We can understand this in the following way:



We have here a *compound procedure*, which has been given the name square. The procedure represents the operation of multiplying something by itself. The thing to be multiplied is given a local name, x, which plays the same role that a pronoun plays in natural language. Evaluating the definition creates this compound procedure and associates it with the name square.

The general form of a procedure definition is

```
(define (<name> <formal parameters>) <body>)
```

The <*name*> is a symbol to be associated with the procedure definition in the environment. The <*formal parameters*> are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The <*body*> is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied. The <*name*> and the <*formal parameters*> are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

Having defined `square`, we can now use it:

```
(square 21)
441

(square (+ 2 5))
49
```

We can also use `square` as a building block in defining other procedures. For example, $x^2 + y^2$ can be expressed as

```
(+ (square x) (square y))
```

We can easily define a procedure `sum-of-squares` that, given any two numbers as arguments, produces the sum of their squares:

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
25
```

Now we can use `sum-of-squares` as a building block in constructing further procedures:

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

Compound procedures are used in exactly the same way as primitive procedures. Indeed, one could not tell by looking at the definition of sum-of-squares given above whether square was built into the interpreter, like + and *, or defined as a compound procedure.

2.5 Conditionals

The expressive power of the class of procedures that we can define at this point is very limited, because we have no way to make tests and to perform different operations depending on the result of a test. For instance, if we wanted to do one thing if $x \geq 0$ and another thing if $x > 0$. Lisp, and many programming languages have a way to ensure that these cases are addressed.

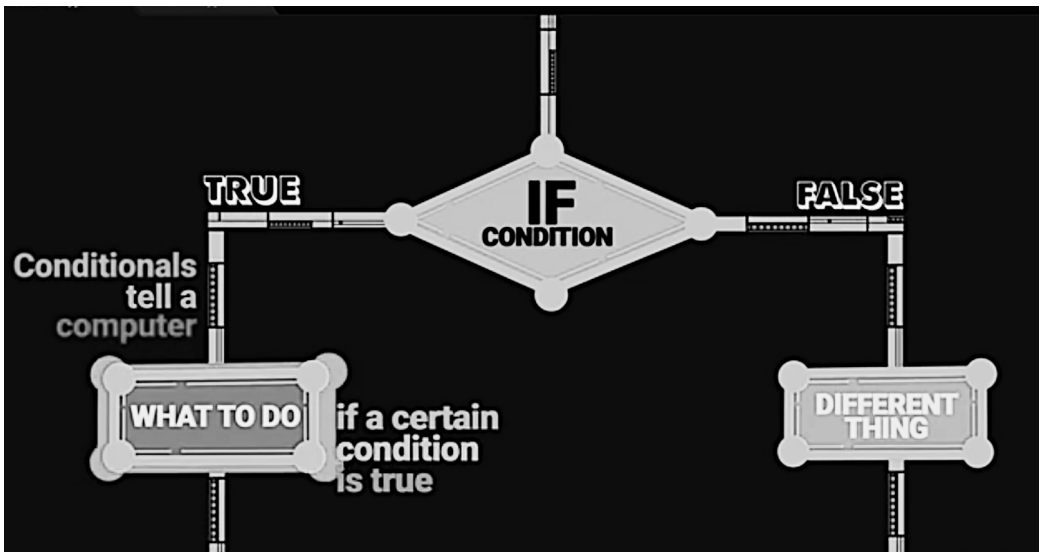


Figure 2.2: A flowchart describing conditionals.
Image Credit: Flocabulary.com.

Conditional expressions are evaluated as follows. The predicate <p_1> is evaluated first. If its value is false, then <p_2> is evaluated. If <p_2>'s value is also false, then <p_3> is evaluated. This process continues until a predicate is found whose value is true, in which case the interpreter returns the value of the corresponding *consequent expression* <*e*> of the clause as the value of the conditional expression. If none of the <*p*>'s is found to be true, the value of the `cond` is undefined.

In addition to primitive predicates such as <, =, and >, there are logical composition operations, which enable us to construct compound predicates. The three most frequently used are these:

- `(and` <e_1> `...` <e_n>`)`

  The interpreter evaluates the expressions <*e*> one at a time, in left-to-right order. If any <*e*> evaluates to false, the value of the and expression is false, and the rest of the <*e*>'s are not evaluated. If all <*e*>'s evaluate to true values, the value of the and expression is the value of the last one.

- `(or` <e_1> `...` <e_n>`)`

  The interpreter evaluates the expressions <*e*> one at a time, in left-to-right order. If any <*e*> evaluates to a true value, that value is returned as the value of the or expression, and the rest of the <*e*>'s are not evaluated. If all <*e*>'s evaluate to false, the value of the or expression is false.

- `(not <e>)`

  The value of a `not` expression is true when the expression <*e*> evaluates to false, and false otherwise.

Notice that `and` and `or` are special forms, not procedures, because the subexpressions are not necessarily all evaluated. `Not` is an ordinary procedure. Now that you know how to use conditionals in programming to set various cases, we can now move onto seeing how we can use this to operate a program.

2.6 Recursion

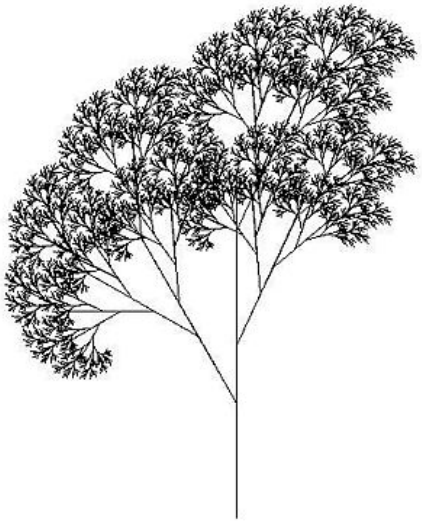


Figure 2.3: A tree representing recursion.
Photo Credit: Wikimedia Commons.

We have now considered the elements of programming: We have used primitive arithmetic operations, we have combined these operations, and we have abstracted these composite operations by defining them as compound procedures. But that is not enough to enable us to say that we know how to program. Our situation is analogous to that of someone who has learned the rules for how the pieces move in chess but knows nothing of typical openings, tactics, or strategy. Like the novice chess player, we don't yet know the common patterns of usage in the domain. We lack the knowledge of which moves are worth making (which procedures are worth defining). We lack the experience to predict the consequences of making a move (executing a procedure).

In this section we will examine some common “shapes” for processes generated by simple procedures. We will also investigate the rates at which these processes consume the important computational resources of time and space. The procedures we will consider are very simple. Their role is like that played by test patterns in photography: as oversimplified prototypical patterns, rather than practical examples in their own right.

Recursion occurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of disciplines ranging from linguistics to logic. The most common application of recursion is in mathematics and computer science, where a function being defined is applied within its own definition.

We can use recursion to solve several problems in programming. In fact it is a very powerful tool that can be used to solve many of the most challenging problems in computer programming.

We begin by considering the factorial function, defined by

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

There are many ways to compute factorials. One way is to make use of the observation that $n!$ is equal to n times $(n - 1)!$ for any positive integer n:

$$n! = n \cdot [(n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n-1)!$$

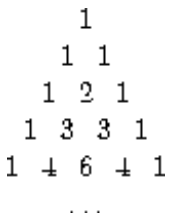
Thus, we can compute $n!$ by computing $(n - 1)!$ and multiplying the result by n. If we add the stipulation that 1! is equal to 1, this observation translates directly into a procedure:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

We must be careful not to confuse the notion of a recursive *process* with the notion of a recursive *procedure*. When we describe a procedure as recursive, we are referring to the syntactic fact that the procedure definition refers (either directly or indirectly) to the procedure itself. But when we describe a process as following a pattern that is, say, linearly recursive, we are speaking about how the process evolves, not about the syntax of how a procedure is written.

We can solidify our understanding of recursive processes through some sample practice problems.

Practice 2.1: The following pattern of numbers is called *Pascal's triangle*.



The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

Practice 2.2: The following pattern of numbers is known as the Fibonacci Sequence. Create a recursive function that will generate n number of elements in the Fibonacci Sequence.

1 1 2 3 5 8 13 21 …

References

[1] "Computer Science." *Dictionary.com*, Dictionary.com, www.dictionary.com/browse/computer-science.

[2] "What Is Computer Science?" *What Is Computer Science? | Undergraduate Computer Science at UMD*, University of Maryland, undergrad.cs.umd.edu/what-computer-science.

[3] "Computer: Definition of Computer by Lexico." *Lexico Dictionaries | English*, Lexico Dictionaries, www.lexico.com/en/definition/computer.

[4] *Program*s, every powerful language has three mechanisms for accomplishing this:

[5] Abelson, Harold, and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed., MIT Press, 1996.

"CS 5:" *CS5 - WebHome*, www.cs.hmc.edu/twiki/bin/view/CS5.