DON'T TEACH CODING

This edition first published 2020

Registered Office(s)
John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA

Editorial Office
John Wiley & Sons, Inc., River Street, Hoboken, NJ 07030, USA

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Library of Congress Cataloging-in-Publication Data

Names: Handley, Lindsey, author. | Foster, Stephen, 1985- author.
Title: Don't teach coding : until you read this book / Lindsey D. Handley, Stephen R. Foster.
Description: First edition. | Hoboken, NJ : Jossey-Bass/John Wiley & Sons, 2020. | Includes bibliographical references and index.
Identifiers: LCCN 2019055396 (print) | LCCN 2019055397 (ebook) | ISBN 9781119602620 (paperback) | ISBN 9781119602644 (adobe pdf) | ISBN 9781119602637 (epub)
Subjects: LCSH: Computer programming—Study and teaching.
Classification: LCC QA76.27 .H364 2020 (print) | LCC QA76.27 (ebook) | DDC 005.13—dc23
LC record available at https://lccn.loc.gov/2019055396
LC ebook record available at https://lccn.loc.gov/2019055397

Cover Design: Wiley

Set in 9/13pt, Ubuntu by SPi Global, Chennai, India.

10 9 8 7 6 5 4 3 2 1

Contents

About the Authors

Dr. Stephen R. Foster is a researcher, author, and co-founder of multiple social enterprises with a mission to teach teachers how to teach coding. A fierce advocate for the power of coding to bring about worldwide change, he has himself coded to generate peer-reviewed scientific results, coded to build educational technology solutions for teachers and students, and coded to bootstrap educational startups and non-profit organizations out of thin air. All in all, these countless lines of code have all been in service of a single vision: to establish coding education as a basic human right across the globe. In short, he codes to teach coding.

Dr. Lindsey D. Handley is a researcher, entrepreneur, teacher, and author. For the last 10 years, the National Science Foundation has funded the research, design work, and the social enterprises that she operates. As a skilled coder, data scientist, and biochemist, she envisions a world in which we no longer suffer from a worldwide shortage of scientific fluency. To this end, she fights for the unification of science and education on two fronts: the use of science to improve education; and the improved teaching of science worldwide. In short, she applies science to design better ways of teaching science.

Together, they are the co-founders and leaders of ThoughtSTEM and MetaCoders.org – two social enterprises that have touched the lives of hundreds of thousands of beginning coders worldwide.

Introduction

If there's one thing this book seeks to address, it is: What are programming languages?

It sounds simple, but answering this question deeply will require us to ask other questions: Why do we call them "languages"? Why are there so many? Why do people fluent in them get paid so much? How are they related to those other things we call "languages" (like English, Spanish, or American Sign Language)? Where do they come from? Where are they going? How do we learn them? What happens in your brain when you do?

And, above all:

How do we teach them?

Japan, Italy, England, and Finland are just a few of the countries that have begun to mandate coding education throughout K-12 public education. Computer science educational standards now exist in 22 U.S. states – two of which have passed legislation that requires coding education statewide from elementary to high school.

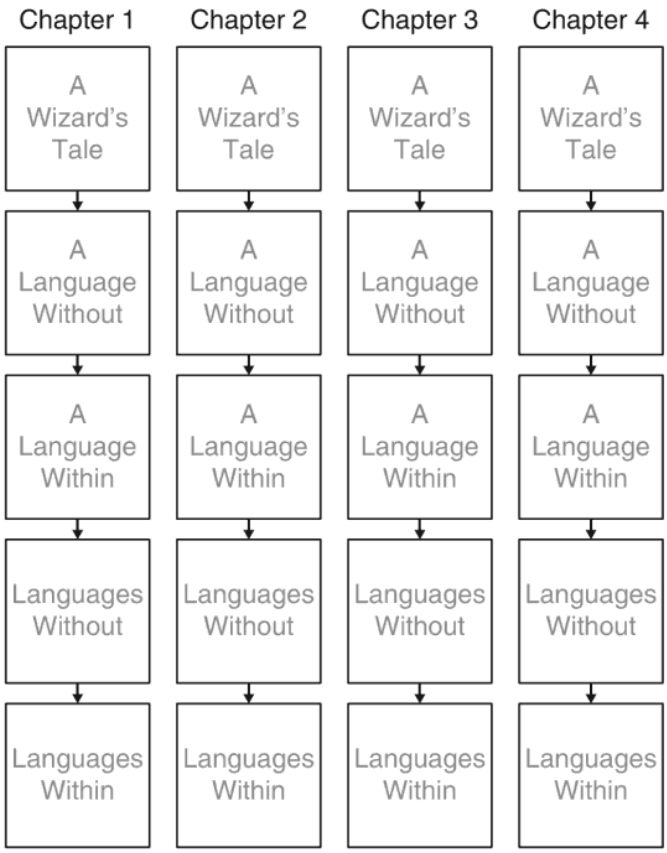
As the world embarks on a global change to its collective education systems, it is worth asking some basic questions.

Technically, this book is about what you should know *before* you start teaching (or learning) a programming language. But the book will also teach you a few simple languages in order to make headway on some of the deeper questions.

There is a structure that frames the four chapters in this book, each of which have five parts – five arcs that recur from chapter to chapter.

The **Wizard's Tale** is the only fictional arc of the book. At the beginning of each chapter, this narrative introduces the main ideas in a lighthearted way. Sometimes the truest things can only be said in fiction.

The pair of arcs called **A Language Without** and **A Language Within** are where you'll learn about coding – one language at a time. In **A Language Without**, we'll examine how the design of a language gives its users certain cognitive powers, and cognitive pitfalls. In **A Language Within**, you'll be given exercises to help you actually learn those languages (if you wish) – meditating on the gaining of those powers for yourself, while learning to avoid the pitfalls. The languages will increase in power and complexity as the book progresses – ending with the



most powerful kind of programming languages: what computer scientists call "the Turing-complete languages."

The arcs called **Languages Without** and **Languages Within** will "zoom out" – beyond you, beyond us, beyond this book, beyond the present day. In **Languages Without**, we'll piece together the epic story of language – literally, the story of stories themselves. It began before this book, indeed before the invention of writing and is still unfolding today. In **Languages Within**, we'll examine recent neuroscience about how the human brain processes language, how it acquires fluencies – and ultimately, how it earns the right to participate in that epic story of language that is unfolding all around us.

Human beings are linguistic creatures; and programming languages are one of the weirdest linguistic things we've done in the last few thousand years. The bigger our historical lens, the easier it is to see just how weirdly magical they are.

WHO IS THIS BOOK FOR?

Mainly, this book is for K-12 teachers of coding, or any educated adult with an interest in the teaching and learning of programming languages. We assume no prior coding knowledge on the part of the reader, however.

This is because, increasingly often in the coming years, teachers who once taught a different subject will find themselves suddenly teaching coding. So we wanted this book to be of use to teachers in those situations. As a rhetorical strategy, we'll often seem to be speaking to the reader as if they were a coding student. If you are a teacher who *is* also a student, feel free to assume we are speaking to you.

If you are a teacher who is not also a student – ask yourself, why *aren't* you a student? In this field, the learning never stops. There's always another language, another library, another framework, another tool-chain, another repository, another engine, another platform, another service, another environment, another paradigm, another sub-field, another beautiful idea.

The teaching and learning don't stop. The job titles just change.

Finally, because our goal is to teach coding teachers what all too many do not know before they begin teaching – even expert coders may find insights here that they were never taught (because their teachers did not know). Thus, your expertise in coding will not prevent you from enjoying this book. We expect the book to be readable by: industry veterans while their unit tests run, computer science grad students in between meetings with their advisors, and battle-tested hackers amidst contributions to open-source projects.

Many of us appreciate the power of K-12 education. The students of today will be our colleagues tomorrow.

LET'S DO IT!

We hope this book will empower teachers and students to write the future of education – one line of code at a time.

At any time, for any reason, join us.

dont-teach.com/coding

Chapter 1
Prologues

*"The programmers of tomorrow are the **wizards** of the future. You're going to look like you have magic powers compared to everyone else."*
Gabe Newell, founder, Valve

*"Any sufficiently advanced technology is indistinguishable from **magic**."*
Arthur C. Clarke

*"The programs we use to conjure processes are like a **sorcerer's spells**. They are carefully composed from symbolic expressions in arcane and esoteric programming languages."*
Harold Abelson and Gerald Jay Sussman,
Structure and Interpretation of Computer Programs

A WIZARD'S TALE

The Sorting of Wizards

"A sorting shall now commence!" an ancient wizard announced. "We must assign all of you into your various Houses. Each House at this prestigious school champions a slightly different way of learning how to become a coding wizard. I will now explain precisely how that works . . . "

Henry, who could not pay attention to lectures for very long, leaned over and asked his new friends, "How does it work? How many Houses are there?"

"My mom and dad say you shouldn't worry about getting a job," said Rob. "You should just learn to love magic."

Henry said, "I just want to be in a House with you two. But even you two can't agree."

Rob and Harmony exchanged a look. "Give us a moment," said Rob, pulling Harmony aside. They conferred in hushed tones.

Henry couldn't hear them over the constant drone of the sorting hat: "Prolog. Scratch. Algol. Perl. XML. Scratch. Haskell. CSS. Racket. Bash. Ruby. Python. TypeScript. Scratch." And so on.

When they came back, Rob said:

"Okay, we've decided. You'll go first, and whatever you get sorted into, we'll pick that too."

Harmony didn't seem happy about it, but she nodded. "Wizards work in teams," she said. "At the end of the day, what matters is that we stay together."

Henry was dumbfounded. He didn't deserve friends like these. They helped him to his feet, where he did his best to hide that his knees were shaky and weak. Arm in arm, they joined the end of the queue – the last of the young wizards to be sorted.

By the time Henry stepped up on the stage, the Great Hall was empty, save his two friends behind him, and the ancient wizard in front of him. Henry sat upon the stool and closed his eyes as the hat settled upon his head.

He could hear it talking through a speaker near his ear. "Well, well, well . . . what have we here?" it said. "Henry doesn't know what House he wants to be in . . . Hmmm . . . I suppose we could put you in HTML, and –" Henry stiffened. "No? What about Scratch?" Henry didn't know what to say. "Why am I asking you, anyway? I could put you anywhere, and you wouldn't know the difference." Henry shifted uncomfortably. "Still, I sense a great power within you – greater even than any of the cool kids who came onto the stage before you . . . " Henry wasn't sure whether he should feel complimented about his mysterious "great power" or worried that he was uncool. "Yes, the more information I gather, the more I'm certain of it. You're a very special young wizard. Much too special for the lesser Houses. Perhaps I could sort you into a venerable old House, such as C. Or perhaps an ancient House, such as Lisp. Or perhaps you'd excel in a hip, newer House, like Rust, or an obscure but powerful House like Prolog or Haskell. Or perhaps a solid, popular House, like Python or Java. Interesting . . . I've never had so much trouble sorting someone before," Henry's heart was beating so hard that he could barely hear the hat anymore. Was he really destined for greatness? The suspense was so painful that he wanted to just shout the name of a House at random in hopes

that the hat would put an end to it all. Somehow, he didn't. "Hmmm, well, if I can't tempt you by dropping the names of these Houses, I suppose I have no choice," said the hat, "but to place you into a House that I've only assigned a handful of young wizards before . . . " Henry tensed.

To his surprise, the ancient wizard took the hat off of him. The look on his face was grave.

"Henry," said the ancient wizard, "do you know what this means?"

Henry tensed. "I didn't hear it say anything."

"You're right," said the ancient wizard. "It has been many, many years since I've heard the hat say nothing at all. In fact, the last time this happened, I was the one sitting on that very stool." He scratched his beard. "Perhaps the three of you," he said, "have been chosen by fate."

Voice trembling, Henry asked, "What House did you get sorted into?"

The ancient wizard said, "This House has no name."

"The House of No Name!" gasped Rob. "My parents said it was just a myth."

The ancient wizard turned his attention to Rob and Harmony, still standing in the queue line, waiting to be sorted.

"No," said the ancient wizard. "If we called it the House of No Name, that would be a name, and therefore contradictory. When we refer to it, we must resort to 'This House has no name.' It's a sacrifice we must make to avoid the contradiction."

"I've never heard of a House with no name," said Harmony, skeptically. "There's no wizard language without a name."

"This House," said the ancient wizard, "is the only House that isn't named after a wizard language. That's because we don't subscribe to any particular wizard language."

Harmony scoffed. "One can't do magic unless one has a wizard language," she said, as if she were the authority on the matter.

"You're right," said the ancient wizard. "Focusing on a single language is not our main approach to learning magic. Rather, we study language itself."

As if to underscore that the ancient wizard had made his main point, the phrase "language itself" echoed throughout the now empty Great Hall.

"It definitely sounds way cooler . . . " said Rob. He and Henry both looked at Harmony.

"No way," she said. "I'm joining Python. I want to actually get a job."

The ancient wizard shrugged. "The sorting hat will ultimately respect your wishes. However, if I may impart just a small moment of wisdom . . . " The ancient wizard cleared his throat. "If a job is what you seek, many roads will take you

there. At the end of the day, though, when you're looking back on your life, don't you want to take comfort in the fact that you took the road that was way cooler?"

The words "way cooler" seemed to echo throughout the Great Hall.

Harmony waivered.

"Come on, Harmony," said Henry. "Wasn't it you who said that wizards always work in teams?"

With a sigh, she said, "Fine. I'll do it. I'll join the House of No Name – or this House which has no name, or whatever it is. But for the record, I think it sounds weird, and I don't like it."

The Call to Action

The ancient wizard motioned for them to follow him. "Come," he said, "I will personally teach you three the ways of this House which has no name."

He reached into his robe and pulled out three copies of a book, giving one to each of them. The title: *Don't Teach Coding*.

Henry glanced nervously at Harmony. She did not look pleased.

To be continued . . .

A LANGUAGE WITHOUT

Our Strange Protagonists

This book is about those languages that make computers do things.

Most people today call them "programming languages" – though they weren't always. These languages, oddly enough, are the protagonists of this book – and a mysterious set of heroes they are indeed. On the one hand, they are the tools with which programmers weave the software of the world. On the other hand, the act of learning these languages is what makes us into programmers. They are both tools and rites of passage.

As if that wasn't strange enough, once becoming programmers, we use programming languages to make other software – including, oddly enough, more programming languages. If this sounds like a loop, it is – one that affects everyone who has ever learned programming, and anyone who ever will.

Many of us can outline our personal histories as programmers by listing the languages we learned in different chapters of our lives. One of the authors first learned to program in a language called Applesoft BASIC, which came with his

parents' first PC. Back then, people were still calling Apple computers PCs, up until IBM-compatible PCs re-wrote that definition. These new "real" PCs also shipped with a version of BASIC called QuickBASIC – itself an evolution over earlier versions of BASIC. He learned Java, Logo, Visual BASIC, Perl, and Pascal in high school. In college, it was more Java and Haskell, with an additional helping of C, C++, Ruby, Python, and Lisp. When he went into industry, it was Ruby, PHP, SQL, Bash, XML, HTML, CSS, and JavaScript. For his master's degree, it was C#, more Java, more Haskell, and Racket. For his Ph.D. and beyond, it was more Racket, and–

You get the point.

And that's just *his* story. Ask any programmer what languages they've mastered in their lifetime, and you'll get a different story. Sometimes it will be a long story, sometimes short. The details will change depending on when and where they were born, which languages were in vogue when they were going through grade school, which ones were taught in college, which ones were used by the companies that offered them jobs, which ones they selected for personal projects.

As working programmers, we have many cognitive tools, yet our languages are truly special. They are what we use to magically convert the vague linguistic utterances of non-coders – that is, "Solve problem X for client Y" or "Make an app that makes money" or "Get us more users" or "Make this data comprehensible" – into precise programs that, when run, actually *do* those things that non-coders could only talk about. Our languages are what make us look like wizards to others.

The story of how a programmer's mind develops feels like a personal experience – yet every programmer's origin story is woven into that larger story of programming languages. There are common threads. There are patterns. The larger story knits us together as a community. Linguistic history is our history; linguistic future is our future. Languages are the tools that shape us; we are shaped by the programmers who shaped those languages.

Ironically, few of us know the larger stories before beginning to wield a language. It is a rare student indeed who picks up one of these sacred tools for the first time with full knowledge of its true history, or its true power. Rather, most of us made our first steps as programmers by pulling one of the many magic swords from its stone and proceeding to chop vegetables with it – unable to see the tool for what it truly was. Languages, after all, are strange things: tools of the mind. As such, they cannot be correctly seen until *after* they have been learned.

These cognitive tools also deeply affect the teaching arts. Their sheer number poses an Eternal Conundrum: Teachers and students must reckon with their

multitude year after year. The twin questions of the conundrum are: "Which one should I learn?" and "Which one should I teach?"

The Eternal Conundrum serves as the backdrop while our society embarks on a historic first: to install the first large-scale public infrastructures for teaching coding in a world that has finally seen that the light of the software dawn is only growing brighter. It took time, but the direction has become quite clear. K-12 computer science educational standards have been drawn up in 22 states (Lambert 2018). Iowa and Wyoming have passed legislation mandating coding in all elementary, middle, and high schools statewide (Iowa 2019) (Goldstein 2019). Non-profit advocacy groups like Code.org and CS For All continue to successfully drive the teaching of computing classes from Pre-K to 12th grade (Code.org 2019) (CSforALL 2019). The National Science Foundation has invested several million in the CS 10K initiative (Brown and Briggs 2015) – its mission: to produce 10,000 new high school computer science teachers across America. Even big tech companies like Google and Microsoft are spending money and labor on the effort – developing free or low-cost out-of-the-box curriculum and software to facilitate coding education.

England has already mandated computer science classes for all children between 5 and 16 years of age (United Kingdom 2013). Italy has launched an endeavor to introduce computing logic to over 40% of its primary schools (Passey 2017). Japan will mandate computing education starting in primary school by the year 2020 (Japan 2016). Finland introduces coding and computational thinking starting in 1st grade (Kwon and Schroderus 2017). One by one, the countries of the world join in this unified initiative.

When a society changes its public school systems, it is changing its very definition of "basic literacy" and therefore of "educated person." Let's take the current trend to its extreme and imagine, for a moment, a world in which coding fluency is acquired by all students throughout all grade levels and beyond. In other words, the average person walking down the street will have had 12 years of computer science education. It's safe to say, that if school systems do an even moderately good job, the average citizen will be fluent in one or more programming languages. For many, this fluency will start so early in life that they will have no recollection of *not* knowing how to code.

Because of the growing importance of these enigmatic things called programming languages, which we are eagerly welcoming into the minds of our children, this book examines a loop of linguistic ideas – each so interconnected with the others that they are best pondered together, in a single book.

Programmers design new programming languages. Teachers teach programming languages to non-programmers. Learning programming languages makes

languages in hopes of correcting damage caused by previous ones. Let it be stated for the record: This treatment appears to have been successful for the author in question.

For better or worse, the evolution of BASIC was once a strategic part of the early skirmishes between what would become two of the biggest software megagiants on the planet, Microsoft and Apple. Its existence sparked the origin stories of all programmers who completed their rite of passage on those machines. Yet today, the remnants of those original versions of BASIC remain alive only in the form of online JavaScript-based emulators that allow certain programmers to engage in nostalgic reconstructions of the programs we wrote as children. They are preserved: Software enshrined in software. As tools of the mind, though, they are not wielded as they once were.

Today, Microsoft champions many languages. Microsoft's TypeScript, a superset of JavaScript, is listed as the 41st most popular language in the world on the TIOBE index. Microsoft's C#, partially inspired by Java, is the 6th most popular language. Apple meanwhile champions languages like Objective-C and Swift, the 10th and 13th most popular languages. The linguistic ecosystem changes so quickly that these numbers will probably be out of date by the time you read them, which underscores the point. Languages are ever changing; what seem like mountains turn out to be tall waves in a shifting sea.

Whether brain damaging or enlightening, we learn these languages, and they make us who we are. Then we learn more, and continue to change.

For some, our first language may have been BASIC. For others, perhaps it was Logo. For others, Scratch. Regardless of our first language, the younger we are when we learn, the less likely it is that we are making an informed, rational decision about which language to learn.

It is not uncommon for everyone in a classroom (perhaps even the teacher) to be using a language without knowing where it came from and why. In that ahistorical context, students sit down to write their traditional first line, bidding their computer to say hello to a multilingual world they do not yet understand.

Tower of Babel

Our digital Tower of Babel is on the one hand quite beautiful, and on the other hand not. It's beautiful because unlike the biblical story of Babel, the legion of languages was not a curse cast upon humanity; it is an act of creation to which we have been willing participants. These languages didn't just happen. We created them – not by accident.

Programs written in these languages run our world – our planes, our cars, our governments, our militaries, our businesses, our charities, everything. An optimist might see it as the opposite of the Tower of Babel story: We *gave* ourselves the gift of tongues to write our edifices into existence.

On the other hand, there are less beautiful aspects of the polyglottic world – not the least of which is that newcomers face a bewildering array of choices the instant they enter the gate. Some of those choices are popular languages like Python and Java. Some are languages designed to make programming easier to learn – like Scratch, Hopscotch, and Snap. The Wikipedia page on "Educational Programming Languages" lists more than 50 languages that were either created for educational use or are used as such. Even the language BASIC (created in 1964) stands for Beginner All-purpose Symbolic Instruction Code – marking it as a language tailored for beginners, which is why it shipped in the 70s on "microcomputers," and then again in the 80s on "personal computers."

Being a beginner coder is a bit like being a hero embarking upon a quest, but then immediately being faced with a fork in the road that goes in more than 50 different directions (or 850 directions, if we look beyond specialized educational languages). It's like the quest to become a coder begins with a meta-quest: which quest to go on; which language to learn.

Confessions

This section is a disclaimer.

The authors of this book are not innocent when it comes to increasing the number of beginner languages in the world. As the architects of a coding education start-up (ThoughtSTEM), they've created a variety of languages with the purpose of making programming more accessible for beginners: *LearnToMod* is an environment and language for creating Minecraft mods; *CodeSpells* is a game where you program your own magic spells using an in-game version of JavaScript; `#lang vr-lang` is a Lisp-like language for constructing virtual reality scenes; `#lang game-engine` is for creating 2D RPG-style games. And that's not even all of them.

Once you've designed one new language, it becomes easier to design more.

While we designers mean well in creating these languages, it's a bit awkward to explain: "Hi, welcome to the land of programming. Sorry there are so many roads here at the entrance. But don't worry! We're making this part more user-friendly by paving these additional roads for you."

Um . . . Wait . . .

Alan Kay, the creator of object-oriented programming and the language Smalltalk, is often quoted:

> Every problem in computer science can be solved by another layer of indirection – except the problem of too many layers of indirection.

Similarly, problems in coding education can be solved with another language – except the problem of too many languages.

This book uses a different method.

Penances

Rather than paving yet another road, we decided to write a book *about* the roads – a book to be read before embarking on any of them.

One of the motivations was to show that the "problem of too many languages" is not a problem at all. It's what makes computer science the powerful and elegant field that it is.

Throughout this book, we'll examine a sequence of increasingly interesting languages, starting from basic ones and ending with ones as powerful as those in professional use today – as powerful as the ones that top the charts, as powerful as those mountains of our day.

There will be plenty of coding exercises – but never in any one language. We'll take the way cooler road.

To be continued . . .

A LANGUAGE WITHIN

Installing Languages

Because of the polyglottic nature of this book, we'll be using a special tool called Racket – a language for creating languages. If you want to run the programs in this book, all you have to do is 1) download Racket, and 2) download our languages. You only need to do these steps once.

If you're ready to do that now, here are the directions. If you're just reading the book cover-to-cover, you don't have to download Racket yet.

Even if you don't download, though, don't skip this section! That goes for any part of the book too: Don't skip. You won't get lost. Above all, we've written this

book to be read. Following along on the computer is for bonus points. Whenever the output of a program isn't obvious, we'll print it. This is so you can follow the main ideas whether or not you're following along on a computer.

Exercise

Step 0: Don't be scared to ask for help. If you get stuck installing, please feel free to ask for help at **dont-teach.com/coding/forum**

Step 1: Download and Install Racket. Go to *download.racket-lang.org*. Download the appropriate installer. Launch it and follow directions.

Step 2: Install the "Don't Teach Coding" Package. With Racket installed, you can now launch a program called DrRacket. Do so.

Next, click

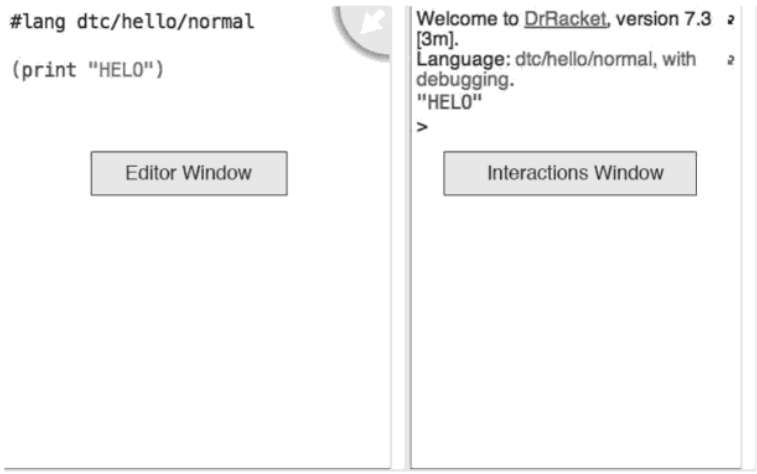
```
File > Install Package...
```

In the prompt, type `dtc` and press enter. Installing this package will take a few minutes.

Lastly, in the lower left-hand corner, DrRacket may say *No Language Selected*. Click that, and select *Determine language from source*. Racket is in polyglottic mode now, which we'll explain in a moment.

Step 3: Write some `"Hello, World"` programs. The point of such programs is less about printing `"Hello, World"` than it is about checking to see if everything got set up correctly. So feel free to print whatever you want.

Your DrRacket may look slightly different from the following figure. For example, the version number may be different (in ours it is 7.3). Do not be alarmed; the programs in this book will still work. To write your `"Hello, World"` program, you simply need to write it your Editor Window (the one that doesn't have the version number). In the following figure, we've labeled the two windows after running the program. Can you guess which window has the program and which has the output?



When you've written your program, you need to run it – which we think you will be able to figure out on your own (Hint: Look for the "Run" button). If something meaningful prints out, everything is set up correctly. If something went wrong, feel free to post on the forums.

dont-teach.com/coding/forum

Writing in Tongues

From now on, when there's a code example, we're not going to insert the entire screenshot as in the figure above. Instead, we'll show code examples like this:

```
#lang dtc/hello/normal

(print "HELO")
```

Notice the first line. This will always be there – the so-called "`#lang` line." It tells human readers *and* the computer the language under which to interpret what follows.

The key thing to notice about the last three examples is that we didn't change the code, only the language. The first time we said `"HELO"` to the world, we used `#lang dtc/hello/normal`. The second time, we used the language, `#lang dtc/hello/colors`. The third time, we used `#lang dtc/hello/animation`. What's interesting though, is that if you look at *just* the code (ignoring the language), you wouldn't be able to tell what language each is written in.

Here are all three programs together:

```
#lang dtc/hello/normal

(print "HELO")

#lang dtc/hello/colors

(print "HELO")

#lang dtc/hello/animation

(print "HELO")
```

They do different things because the word `print` means something different depending on the language.

This concept arises in human languages, not just programming languages. The English word "gift," in German, means "poison." The English word "kiss," in Swedish, means "pee." And flashing the peace sign on one's forehead, in American Sign Language, means "stupid."

The same words or signs, under different interpretations, are not the same words or signs.

Nova: Va o no va?

A famous (but surprisingly false) example of misinterpretation is the cautionary tale about the Chevy Nova. As the story is told in hundreds of marketing books and business seminars (Aisner 2000), (Colapinto 2011), this car sold poorly in Spanish-speaking countries because the word "Nova" is similar to the Spanish phrase "no va," which translates to "doesn't go." Supposedly, Spanish speakers were concerned that if they bought a Nova, it wouldn't go.

But language isn't always so simple. The Chevy Nova actually sold well in Spanish-speaking countries, meeting or exceeding expected sales numbers in

both of Chevrolet's primary Spanish-speaking markets: Mexico and Venezuela (Hammond 1993).

When you think about it, it makes sense. Just as in English, we don't say that a car "doesn't go," in Spanish, it would be more common to say something like "no funciona" ("it's not working") or "no camina" (literally "it's not walking," but a better English translation would be "it's not running"). Furthermore, until 2016, the word "Nova" was used for a kind of leaded gasoline provided by PEMEX – Mexico's state-owned petroleum company (Onursal and Gautam 1997). In terms of word associations, Nova gasoline is more connected to cars than the phrase "no va," which doesn't even apply to cars and isn't even pronounced like "Nova."

This story *is* a cautionary tale – but less about how to market cars than about how language works. It's a reminder not to make assumptions about how words, phrases, or pieces of syntax will be interpreted in a language you do not know. Languages are not simple.

The fact that the Chevy Nova story is so often repeated suggests that we English speakers don't need much convincing when it comes to how others are misinterpreting our words. The reality, of course, is that we are often the ones poorly interpreting the words of others: "No va" being unrelated to cars; and "Nova" being a leaded gasoline.

Hello, Hello, Hello

Programming students confuse syntax and semantics too. One of our favorite trick questions is, "In what language is the following code written?"

```
print("Hello world")
```

Students will raise hands and say things like Python, Ruby, or Lua – all of which *could* be correct. You *could* write this line in those languages. Something would happen.

However, the only truly correct answer is to note that it's backwards to ask, "In what language is this written?" – just as it would be backwards to ask in what language strings of symbols "gift," "kiss," or "Nova" are written. There isn't just one language in which a string of symbols like "gift" has meaning – just as there isn't just one in which `print("Hello World")` has meaning.

The reverse is also true: The string of symbols "gift" has *no* meaning in some languages (e.g., Chinese or Arabic, whose syntaxes use different symbols

entirely). Likewise, there are languages in which `print("Hello World")` has no meaning – in Node.js (a non-browser-based implementation of JavaScript), for example. Running it will trigger an error that basically translates to: "I don't know what `print` means." In browser-based versions of JavaScript (yes, there are many JavaScripts), `print` does have meaning. It will ignore everything inside the parentheses and open a print window. Running `print("HELO")` and `print("Goodbye")` do the same thing for this JavaScript.

In other words: The same string of characters that causes one language to say `"hello"` will cause others to tell you that you've made a mistake, and still others will do something else entirely.

Furthermore (as if it weren't confusing enough), any JavaScript programmer can make *their* JavaScript understand that `print` should display words, rather than triggering an error or opening the print window. Whereas the following doesn't *normally* function in JavaScript the same way it does in Python, Ruby, and Lua – one could *make* it do so:

```
print("Hello world")
```

It's not difficult at all – and we'll cover this very basic form of language extension ("defining a function") later in this book. Many languages allow you to change the basic functionality of certain words, like `print`. Every programming language in modern use allows you to add vocabulary that wasn't there before.

Summing up:

- On the one hand, a program can act differently in different languages.
- On the other hand, two programs that act the same may look different in different languages.
- To make matters more confusing, some programs that don't work at all in a language can be made to work in that language by extending it with new vocabulary.

Syntax even changes between versions. This works in Python 2.7 . . .

```
print "Hello, World!"
```

. . . but in Python 3 it must be . . .

```
print("Hello, World!")
```

Yes, it's true. There are many Pythons. We say this because it often comes as a surprise to our students – even the ones who love Python, even the ones to

whom we have already explained that there are many JavaScripts. Python was created in the 90s, and there have been many implementations since: Python, Jython, PyPy, and so on. And of course, there have been many versions *within* each of these Pythons (2.7 and 3.0, for example).

Don't despair, though! The process of becoming a coder is, in part, the process of becoming better at learning new languages – and becoming better at keeping them separate in your mind. It might seem impossible at first. It might seem like programming requires the world's longest cheatsheet. But we humans have an incredible capacity for learning languages and switching among them based on context.

If you're truly bilingual in English and Spanish you will have little trouble switching between them. Likewise, as you become a skilled coder, you'll have no more trouble keeping track of which programming language you're using than you would keeping track of the difference between poker and solitaire. Our brains have tremendous capacity for absorbing new sets of rules – whether they are game rules (like how the knight moves in chess), or grammatical rules (like "Put a comma between salutations, like 'Hello,' and recipients, like 'World"'), or syntactical rules (like "Put a parenthesis after symbols like `print`"), or even meta-linguistic rules (like "Speak Spanish with your mom and English with your dad," or "Use Python for the company's web app but use Bash for your personal shell scripts").

In any event, with our newfound ability to write in tongues, we are ready to begin our journey through a fascinating land – fraught with syntactic perils and semantic adventures.

To be continued . . .

LANGUAGES WITHOUT

As C-3PO put it:

> I'm fluent in over 6,000,000 forms of communication.

The authors have taught many students and teachers throughout the years. We overhear things.

When we had first launched our company, one 12-year-old student boasted on his first day of a coding summer camp that he was "fluent in six different languages." When asked which ones, he said: "English, Scratch, Java, JavaScript, and a little bit of Python."

"That's only five," one of the camp counselors pointed out.

"Oh, I forgot," he said. "My mom also taught me sign language." He started signing the alphabet with the kind of total confidence that "fluent" 12-year-olds sometimes have.

One of the other kids in the camp inadvertently asked a deep philosophical question, "What does 'fluent' mean?"

"It means you know it," the first kid said, rolling his eyes, as if the question was offensively trivial.

As the coding camp went on, it became clear that his only fluency was in English. His "fluency" in Scratch came from having done an "Hour of Code" at his elementary school. His Java/JavaScript "fluency" came from his father (a full stack web developer) having explained the difference between "Java" and "JavaScript" while showing him some of the JavaScript code he had written for work. His Python "fluency" came from a few hours of online Python lessons with his dad, who had decided that Python was a better first language than JavaScript. His American Sign Language "fluency" boiled down to knowing the words for "hungry" and "milk," and about 15 of the signs for letters of the alphabet.

Still, we learned a lot from this kid. This was the first time we'd heard someone claim to be "fluent" in a programming language – and to so deeply conflate this "fluency" with fluency in natural languages like English and American Sign Language.

As it turned out, this kid had his finger on the pulse of the times. A few years later, from 2015 through 2016, Senate Bill 468 (Florida Senate 2016) was percolating through the Florida state government. The bill was to require the Florida College System to allow high school coding classes to count as foreign language credits. That is, high school students could take a Java class instead of a Spanish class – and the two would be equivalent for the purposes of college admission and college credit.

The League of United Latin American Citizens (LULAC) and the Spanish American League Against Discrimination (SALAD) objected (Clark 2016):

> Our children need skills in both technology and in foreign languages to compete in today's global economy. However, to define coding and computer science as a foreign language is a misleading and mischievous misnomer that deceives our students, jeopardizes their eligibility to admission to universities, and will result in many losing out on the foreign language skills they desperately need even for entry-level jobs in South Florida.

Although the bill was stopped in the House after passing the Senate 35 to 5, the whole attempt suggests how easy it is for 12-year-olds and/or lawmakers to


image
not
available

something like "gorblesnop" if, indeed, he wanted a senseless word. However, to do so for all operations, across all the symbolic languages one might design, leaves the mind juggling many words that have no sense in *any* language.

Programming language designers over the decades have imported other words from English rather than inventing new words: "if," "class," "object," "graph," "interpret," "compile," "execute," "tree," "branch," "library," etc. These do not retain their ambiguous English meanings when used by programmers. They take on a much more technical meaning in "programmer English."

Babbage's Calculus Club

Even before this, as early as 1813, when Charles Babbage was merely a college student at Cambridge – long before working on his Difference Engine or his Analytical Engine – he wrote of what he called "symbolic language," describing such mathematical notations as tools for unburdening the mind:

> It is the spirit of this symbolic language . . . (so much in unison with all our faculties,) which carries the eye . . . to condense pages into lines, and volumes into pages; shortening the road to discovery, and preserving the mind unfatigued by continued efforts of attention to the minor parts, that may exert its whole vigor on those which are more important.

This was written in the preface to a manifesto authored by a small group of revolutionary Cambridge students called the Analytical Society – of which Babbage was a founding member. The group's goal was to bring about a linguistic change in their education system: getting Cambridge to abandon the use of Newton's calculus notation in textbooks and exams, and rather to use the more popular notation in continental Europe – Leibniz's notation.

This might sound trivial or geeky, but it was actually a surprisingly forward-thinking idea. Recall that the debate about who invented calculus (Newton or Leibniz) still smolders to this day. Newton was the Englishman, so one can imagine which side the schools of England were on. Babbage's group was championing Leibniz – the non-Englishman who had invented a notation that many considered superior to Newton's. In spite of their controversial position, however, they were ultimately successful, with the momentum continuing even after the group had been disbanded. Cambridge began to include Leibniz's notation on exams; textbooks were translated; and by 1830, Leibniz's notation was commonplace in England, alongside Newton's. Today, pick up any calculus textbook, and you will find Leibniz's notation for derivatives and integrals, not Newton's.

It is perhaps the first example of an education system's wholesale adoption of one symbolic language only to replace it with another, more popular one. In computing, this is commonplace with the symbolic languages of our day: Before universities taught Java they taught C++ and Pascal (Guzdial 2011); today, more and more are shifting to Python, which is now the most common language taught (Guo 2014). Tomorrow? Who knows?

Babbage's Analytical Society was less about calculus and more about a psychological idea:

that one language of symbols can be a better tool for the mind than another.

Later, he would go on to do what he is most famous for: inventing the Analytical Engine – a machine that could manipulate symbols. This machine would be programmable in a Notation of his invention – a symbolic language. In other words, the career of the father of computing was, from his university years onward, intertwined with symbolic languages, what mathematicians today would call "formal languages." This is a broad linguistic category that includes mathematical notations like the ones of De Morgan and Babbage, as well as modern programming languages like Java and Python.

Between 1837 and 1845, Babbage and Lovelace would pen (using the Notation) the first things that historians consider to be "computer programs" – a full century before the first thing that historians consider to be "computer hardware" would actually be built (Konrad Zuse's electronic Z3 computer, in 1941).

In the 1950s, the designers of early programming languages (many of them mathematicians) would begin to gravitate toward the already-accepted L-word for describing the notations of mathematics. It had been that way since before they were born. Though this fact is probably lost on the average elementary school student learning the Scratch language, the L-word gravitation was less about connecting programming languages to English or Spanish, and more about connecting it to the mathematical languages of people like Newton, Leibniz, De Morgan, Babbage, Russel, and Whitehead.

Today, the programming languages we teach and learn in schools are instantiations of a long and ongoing tradition of language design – one that predates the earliest of the electrical machines on which they run. The machines are by no means the origin of these languages; and to think so would be unfair to them.

Indeed, there isn't even a chicken/egg ambiguity here: Programming languages were built a full century before programmable machines.

Babbage never built the Analytical Engine, in fact. Yes, that machine he is most famous for – the thing that launched the digital age – was never fully constructed. The notation he and Lovelace wielded came from a long tradition of notational design that predated even his *drawings* of that machine – inherited from predecessors like Newton and Leibniz. Babbage's and Lovelace's early programs were comprehensible to others in spite of the Analytical Engine being the stuff of dreams.

Diffs

It's easy to find differences between Java and English, or Scratch and ASL. Let's get most of that out of our system here, so that the rest of the book can deal with what is actually much more interesting: *what these very different kinds of languages have in common.*

In the next chapter, we'll see that, different though they are, programming languages and natural languages actually do have one surprising thing in common: In an fMRI machine, a medical device used in brain imaging, the parts of a coder's brain that light up when they are reading code are the same ones that light up for all human beings when we comprehend natural language.

This was a groundbreaking scientific discovery in 2014 that will be all the more exciting and delightfully puzzling after we have spent the rest of this chapter examining how *different* computer languages and people languages are.

Finite Descriptions of the Infinite

The 3rd edition of the American Heritage Dictionary of the English Language has more than 350,000 words (Soukhanov 1992). The Academic Dictionary of Lithuanian has about half a million (Academic Dictionary of Lithuanian 2005). The online dictionary of the Turkish Language Institute contains more than 600,000 words (Turkish Language Institute 2019). And the online dictionary for the northern and southern dialects of Korean tops out at more than 1.1 million words (National Institute of Korean Language 2019).

Natural languages are big things.

That said, one might argue that the number of words in a dictionary doesn't really denote the size of a language. That's true. The average American high school graduate knows about 45,000 words (Pinker 1994). And although they may know and recognize 45,000 words, the active vocabulary of the average American adult is closer to 20,000 words (Jackson 2011). That's far short of the