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?"

His better-informed friend Harmony replied, “There are over a thousand, with more being added every day.”

“Over a thousand!” hissed Henry. There was no *way* he was going to end up in the same House as his new friends.

“But,” his goofy yet loyal friend Rob said, “I’ve heard that the sorting algorithm takes your preferences into account. So you can basically choose which House you start in.”

Henry sighed with relief. “Good, which one are you both picking?”

“Definitely Python,” said his friend Harmony, as if there were no doubt in the matter.

“Really?” says his friend Rob, doubtfully. “My dad said I should pick Scratch.”

This sparked a debate between Harmony, whose position was that none of the great wizards actually use Scratch in their day-to-day work, and Rob, whose position was that Scratch was a better House for beginners to start in, and that they could always switch Houses later.

“You can switch?” asked Henry.

But Rob and Harmony didn’t hear him. They were busy debating.

Henry’s heart was starting to pound. The ancient wizard had already begun directing the students standing at the front of the Great Hall to start queuing up for the sorting process. One by one, they went onto the stage and sat on a stool. One by one, the ancient wizard placed upon them a weird cap with blinking lights. Each time, after a few moments of making beeps and boops, the cap’s lights flashed green. Each time, it announced with a mechanical voice the House into which the candidate had been sorted:

“Java,” it said for one. “C plus plus,” it said for another. “Javascript,” it said for another.

“Is Java part of the JavaScript House?” Henry tried to ask his friends. But they were too deep in conversation.

Henry noticed that he was being watched by a nearby group of “cool kids.” One boy said to his friends, “This kid doesn’t even know what JavaScript is.”

Henry tried to ignore them, but their snickering hurt.

Meanwhile the hat continued to drone out the names of Houses. “C sharp,” it said for one. “Ruby,” it said for another. “C,” it said for another. After receiving the hat’s proclamation, each student grinned and exited the Great Hall through a door in the rear of the stage, presumably to meet the other students in their Houses. The number of students left in the Great Hall was beginning to thin.

Henry began to make his way through the crowd of students, toward the back of the Great Hall. Other kids gave him strange looks as he squeezed between them.

Finally, at the rear of the hall, just as he was about to curl into a fetal position between a suit of armor and a damp stone corner, he discovered that Harmony and Rob had followed him.

"What's wrong?" said Rob. "Where are you going?"

"It's just . . . " said Henry.

"You don't know which House to pick, do you?" said Harmony. "Not to worry. I'll help. The top 10 Houses right now are JavaScript, Python, Java, C++, PHP, Swift, C#, Ruby, Objective-C, and SQL. But you obviously wouldn't want to go into the SQL House – because they don't do general purpose magic, just database magic. And you might want to be careful if you go into the Swift or Objective-C Houses – because their magic is proprietary. With C#, some people think that's proprietary magic too, but there's an open source –"

Henry covered his ears to stop the flood of words he didn't understand. "I wasn't born into a coding family. Maybe I'm just not cut out for –"

"Neither was Harmony," his friend Rob interrupted. "She just spends a lot of time on Google. My parents, on the other hand, were both wizards." He said this proudly. "My father sorted into Java, and my mother sorted into Ruby. But both of them always say that it doesn't really matter where you start." He gave Harmony a pointed look. "It's where you end up that matters."

"But," said Henry, frantic, "they can't ALL be equally good places to start. If they were all equally good, why are there so many different ones?"

"Each House does magic," said Harmony. "They just use a different language to express their magic. At the end of the day, though, magic is magic."

Meanwhile, over half of the students had already been sorted. Still the cap droned on. "Objective-C," it announced. "FORTRAN," it said. "HTML," it said.

Harmony and Rob both winced at the "HTML" announcement.

"See?" exclaimed Henry. "They aren't all equal. What is HTML? What was that face you both made?"

"Okay," said Harmony, "maybe they aren't ALL created equally. HTML is a kind of magic, don't get me wrong. But it's a less powerful kind of magic. You wouldn't want to spend your whole time here just studying HTML. If you did, you might not be able to get a job as a wizard afterward."

Henry sank to the ground and rested his head against the cold metal of the suit of armor.

"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

non-programmers into programmers. This means, some of the programmers we teach today will design languages that the teachers of tomorrow will use. These languages will help shape the minds of students the day after tomorrow. So the wheel turns.

This means that some of the programmers of today will directly affect the teaching and learning of programming tomorrow. And by corollary: All of the teaching and learning of programming today has already been affected by programmers who created the languages of yesterday. As schools across the world engage in their individual experiments to teach coding, let us not lose sight of what connects us all.

Our programming languages can be learned, wielded, taught, and created. They are many-faced and multifaceted. In this book, we will sometimes speak of them all as a unified thing – a single entity named "the programming languages of the world." Sometimes we will speak of them as individual things: like "Java," "BASIC," or "Racket."

We said these languages are "the protagonists." And that is true: Sometimes we will study them as protagonists. But languages are tricky things.

Sometimes they may seem to shrink down and become objects of interest in some other protagonist's story. That is how we as students first encounter a language: They pop into the story of our lives, perhaps helping us, perhaps frustrating us, perhaps intriguing us, perhaps boring us. There is no shame in any of these.

Even for a single student, the metaphors may shift over time: A language is the road beneath your feet, the magic sword you are wielding, the mentor you meet along the way, the mountain you are climbing, or even (at times) your archnemesis.

Languages will go by many metaphors throughout this book too: sometimes a fabric that connects us, sometimes a sword, sometimes a tool, sometimes an object of study, sometimes a mountain, sometimes a wave, sometimes magic. It is inevitable. Languages are complex characters.

They were not created by simple creatures.

`(cons 'Apple 'Soft)`

The word "Applesoft" in "Applesoft BASIC" is a combination of two company names: "Apple" and "Microsoft." Originally licensed by Apple from Microsoft in 1977, the Applesoft license only cost Apple $31,000 (McCracken 2012), but allowed them to ship their Apple II computers when Apple was unable to finalize their own version of BASIC (Isaacson 2011). Eight years later, when the

Applesoft license needed renewing, Microsoft renewed on the condition that Apple would kill MacBASIC (Manes and Andrews 1994) (Hertzfeld 1985), Apple's up-and-coming BASIC programming language, which everyone expected to be more efficient. (Allen 1984)

Indeed, MacBASIC had better benchmarks and more features than Microsoft's BASIC – yet it scarcely saw the light of day and vanished from the world when Apple dropped it. Programmers went on using and learning the old BASIC, oblivious to the fact that language designers they had never met had made a decision they never knew about, quashing language features they had never learned.

These language wars (whose version of BASIC would ship on whose computers) were largely overshadowed by what was seen as the dominating historical force at the time: the hardware wars (whose machines would sit on the desktops of the world). The fact that improving BASIC might improve the lives of the programmers of the time wasn't enough to drive the evolution of the language.

Famous computer scientist Edsger Dijkstra claimed in 1975 that BASIC caused brain damage (Dijkstra 1982):

> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.

It does seem odd *not* to improve something that might cause brain damage. But economic pressures are often stronger than educational ones. For better or worse, this was the first program one author remembers typing into Applesoft BASIC, a few years after MacBASIC's secret death:

```
PRINT "HELO"
```

He was too young to know how to spell `"Hello"`, and too inexperienced of a typist to type out the full traditional `"HELLO, WORLD"`. Nor did he know, at the time, how to turn off the caps lock.

These were his struggles. Far from his mind were the warnings that this very language might cause brain damage. Far from his mind was the story of how Applesoft BASIC had come to be installed on that computer. So it is with most of us. We cannot see languages for what they are until after we have learned them; and we often cannot see the bigger historical picture until long after that.

At some point, brain damage or not, it is too late to unlearn a language. If Dijkstra is right and some languages *can* cause setbacks in one's ability to become a programmer, the only recourse is to hope he is wrong about being "beyond hope of regeneration." Our only choice at that point is to learn *more*

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

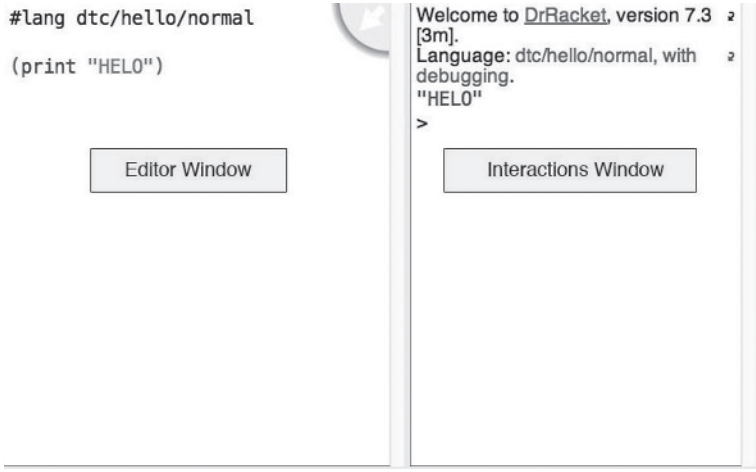
```
File > Install Package...
```

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.

This is how we'll accomplish writing in tongues. Any time you see a `#lang` line at the top of a snippet of code, you know three things:

- That's a piece of code you could type into DrRacket yourself.
- The computer will use that language when it interprets what follows.
- *You* should use that language to interpret what follows.

All code we write will be written for you to read – only incidentally for the machines you type it into. This is not a book about machines. It is about language.

Often, after a snippet of code, we'll show the output. Though it may *look* like code, it will never have a `#lang` line.

Here's the output of the code above.

```
"HELO"
```

The reason for abandoning screenshots is obvious – they take up a lot of room. If we used screenshots for every example in this book, it would either become twice as long or have half as many examples.

Efficient use of space is key.

Kiss, Gift, Poison

Let's try another language.

```
#lang dtc/hello/colors

(print "HELO")
```

Here's its output:

If you run it again, you might get:

This is a language with some trickiness to it. Whereas in `#lang dtc/hello/normal`, the `print` word simply printed the thing that came after it, the `print` word in `#lang dtc/hello/colors` places the thing that came after it onto a randomly colored shape.

Language designers would say:

> The syntax is the same; but the semantics are different.

Most books about coding deal with one language, and have just one "Hello, World" example. But this book is about language itself, so we'll see a few "Hello, World" programs in different languages. The point is to give you an intuition for syntax and semantics. We'll define these more formally in the next chapter. When we do, you'll already have absorbed their meanings via linguistic immersion.

Let's look at one more example that has the same syntax as the previous two – but different semantics again.

```
#lang dtc/hello/animation

(print "HELO")
```

Now, when you run the code, something different happens again. The word `"HELO"` is displayed with an animation every time you run it. We can't print a dynamic picture in a static book, but the basic story is:

And so on.

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

notice that the L-word in "programming language" and the L-word in "foreign language" are, in fact, the same L-word, and to draw conclusions about the two being the same.

Lawmakers in Georgia and Rhode Island have both also recently discussed allowing coding credits to substitute for foreign language credit, as have legislators in Washington, Kentucky, and Virginia. Texas actually *does* allow students to substitute coding for foreign language, provided that the student has attempted the required foreign language class and failed (Galvin 2016).

Tongueless Languages

If you're "in the know" – that is, already a coder – it's easy to laugh. But it's not entirely people's fault. That pesky L-word is literally sitting there, proclaiming that these things – Java, Scratch, Python, Racket, etc. – are all "languages." It's worth asking how that L-word got there.

They weren't always universally called languages. Early computing pioneer Charles Babbage labeled it the "Notation" in the 1840s. In the 1940s Konrad Zuse invented what he simply called "the calculus of plans" (Plankalkül, in German), which we recognize in retrospect as the world's first high-level programming language. Some of the other earliest things we recognize today as languages were simply called "autocodes" by their creators. There were times when people were more reserved about dubbing these pieces of technology "languages."

But in the 1950s, the L-word caught on (Nofre et al. 2014). It began to be included in the names of influential early languages – like ALGOL, developed in the 1950s: ***ALGO****rithmic* ***L****anguage*.

Today, we sometimes include it in the names of particular languages – like HTML ("Hypertext Markup Language") or SQL ("Structured Query Language"). But largely we don't bother – referring to the entire category of technologies as languages.

There's nothing wrong with that.

We just need to be aware of the confusion this can cause. Students are often unaware of how programming languages differ from those other things that we call "languages" – the ones we are much more familiar with, the ones that have likely been around since the dawn of our species. Likewise, teachers sometimes struggle to articulate their subtle relationships: either overstating their similarities or overstating their differences.

Naming something is a powerful linguistic act. To name something a *language* is a powerful meta-linguistic act. When we do such a thing, it is worth asking why.

The attribution of the word "language" isn't always given as easily as it was to programming languages. In the 1960s, American Sign Language (also called by other names, "Ameslan" and "manual communication") was *finally* formally accepted as a language by the scientific community. Sadly, it took almost a century. People had to fight for it. Powerful people fought against it, as we will see later in the book.

The word, "language," of course, comes from the Latin word for "tongue" (*lingua*). No surprise then that the concept is anchored to speech, sometimes limiting our understanding of other things that might qualify.

Yet, those technologies that are far less "language-y" than American Sign Language or English got the L-word appended without much fuss, pomp, or circumstance. They slipped into the club for free.

Who paid their membership?

As it turns out, mathematicians had adopted the L-word before modern electric computers had been built and before modern programming languages had been designed. The *Principia Mathematica* routinely distinguishes between "ordinary language" (Russel and Whitehead's native tongue: English) and things like "logical language" (their mathematical notations) (Whitehead and Russell 1925). These twentieth century mathematicians were not the first to make such metalinguistic distinctions. It goes back much farther.

Mathematicians had, for centuries, been cultivating their notations while simultaneously and responsibly separating their newly constructed "languages" (as they called them) from the other kind – their native tongues. Augustus De Morgan – famous for his contributions to mathematical logic and algebra, and for being a calculus tutor to Ada Lovelace – wrote of this separation (De Morgan 1849):

> In abandoning the meanings of [mathematical] symbols, we also abandon those of the [ordinary language] words which describe them. Thus addition is to be, for the present, **a sound void of sense**. It is a mode of combination represented by +; when + receives its meaning, so also will the word addition. . . . [N]o word nor sign of arithmetic or algebra has one atom of meaning throughout this chapter . . . If any one were to assert that + and - might mean reward and punishment, and A, B, C, etc. might stand for virtues and vices, the reader might believe him, or contradict him, as he pleases . . .

He is saying that the word "addition" is merely imported from English – but when it crosses the threshold between English and algebra, it becomes a "sound void of sense." Even the sense of "addition operates on numbers" is – as he emphasizes – too much to assume. At first, one wonders why he didn't use

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

350,000 words in the English dictionary. And that's okay. Knowing all of the words in a dictionary is never a requisite for fluency in any natural language. Still, whether you look at the number of words in a dictionary, or something more modest, you see the same thing when you put that number next to the tiny numbers that describe the sizes of programming languages.

Java, for example, has about 50 keywords (Gosling et al. 2019). The thing is, no one learns Java simply by learning the definitions for those keywords. They are merely symbols void of sense – until they are combined with other symbols according to certain rules.

Even these rules, though, are few in number compared to the many grammatical rules of natural languages. Java's rules fit on a single webpage (Gosling et al. 2019). Though their number grows as the language evolves, there have never been more than a few hundred.

Even when we include all the vocabulary and all the grammar rules of Java, we'd be in the low hundreds, whereas our *most conservative* estimation of English vocabulary size is 20,000 ambiguously defined words, plus many (often subtle) grammatical rules. This difference should serve to illustrate that when it comes to natural languages, there's simply more there. A lot more. At minimum, two hundred times more. At maximum, it's hard to even estimate.

Scheme, famous for being perhaps the smallest programming language that human minds can effectively use, has only 5 keywords and 8 syntactic forms. Peter Norvig (current director of research at Google), wrote on the subject, showing in a single blog post that you can re-create Scheme with fewer than 120 lines of Python code (Norvig 2010). This alone should illustrate the smallness of programming languages. You can re-create one inside the other.

The idea of "re-creating Chinese within English" is nonsensical. But how much English would need to be spent to fully articulate the subtleties of the Chinese language – from word definitions to grammatical usages? Definitely more than 120 lines.

Although programmers routinely add new vocabulary to programming languages in the form of "libraries," the small number of built-in keywords are the ones that you get when you download a language from the internet and sit down to write with it for the first time. Indeed, that's what a programming language is: A program that understands *other* programs – provided they are written with a certain set of keywords that are arranged according to a certain set of grammatical rules.

Downloaded Scheme? You now have a program that reads programs. What programs does it read? Any, as long as they use those 5 keywords, combined appropriately with those 8 syntactic forms. That's what it reads, and that's *all* it reads (at least at first).

Let's not underestimate the complexity of programming languages, though. After all, there are only 88 keys on a piano too (and really only 12 tones, repeated across various octaves). Yet the "language" of the piano takes many years to master. Indeed, the number of pleasing ways those 12 tones can be arranged is, for all practical purposes, endless.

Endless too are the number of ways even those 5 keywords of Scheme can be combined. Not to mention, one of those keywords is `define` – which lets you add new keywords. So it's a bit like a piano that has a meta-key that lets you add new keys. Let's ignore that, for now, though.

Languages start small. Yet, in those small spaces, the seeds of infinity are already there. This is a vital quality of all languages – they are "generative." Given sufficient time, fluent individuals can generate an infinite number of new expressions, all grammatically correct, all unique, all meaningful. This applies to both natural languages and programming languages. This connection between the two kinds of languages was articulated by linguist Noam Chomsky.

However, when it comes to programming languages, mathematical notations, and music – their infinite potential proceeds from something undeniably smaller, something small enough to be immediately perceivable as a tool. Sometimes 88 keys. Sometimes 5 keywords and 8 rules.

Bottling the Human Will

The channel of a language is how that particular language is transmitted to other human beings. English and other spoken languages can make use of the vocal-auditory channel – for example, you voice, they hear. American Sign Languages and other signed languages use a physical-visual channel – that is, you sign, they see. Written English makes use of a physical-visual channel too – that is, you write, they see.

In some sense, programming languages are akin to written languages – in so far as they share the same channel. You type, they see.

Furthermore, we can say that they rarely (if ever) make use of the vocal-auditory channel. If you don't believe us, try reading some programs aloud and

see if people understand you. We're not saying they'll understand nothing; but it's a guarantee that, very quickly, they'll tell you to shut up and just let them read the source code.

That's certainly one difference – the channel – and a key one. No doubt, the visual nature of programming must belong to any serious discussion about how students learn it. This will come up again later, as we discuss the training of "vision" for skills such as chess.

But before we leave behind the idea of channels, doesn't it feel a bit strange to talk about the "channel" of a programming language? When it comes to programming, there need not necessarily *be* a channel between two human beings. Programs need not be means of communication at all.

True, there is that famous adage:

> Programs should be written for people to read, and only incidentally for machines to execute.

Should. Not must.

We *can* write programs in secret; we can even run them. If the programs we write are never seen by eyes other than our own, this does nothing to remove their ability to move matter in the world. We type programs into a computer, and their story unfolds. The presence of a human mind besides the author's is never strictly required. Once it is running, the laws of physics have already taken over. Such laws operate in spite of our awareness or attention. It is like setting a boulder rolling down a hill – or rather, trillions of boulders, called "electrons."

Indeed, a program can be written and forgotten about, yet still be running and doing things in the world. Thank goodness for that; our digital society rests upon it. Coding wizards write programs that run our airplanes, power plants, cars, systems of commerce, stock markets, and systems of communication. The vast majority of what these programs do is hidden from users – uncommunicated. Only the very tip of the iceberg is made manifest to most – for example, that the planes do not crash, that the stock market keeps humming, that our text messages are sent.

To put it dramatically, program authors may die, yet the programs will not.

When you turn the steering wheel on many modern cars, you cause your car's computer to execute code that helps turn the car's wheels. What that code looks like and what it might have communicated if you read the source code are mysteries. Yet, the wheels do turn. And our society rolls on.

What an incredible difference between programming languages and natural ones. As philosophers are fond of asking: If a tree falls in the forest and no one is

around to hear it, does it transmit any information across the auditory channel? Certainly not. If a man speaks his dying words alone in the forest, are his words lost forever? Certainly so. If a dying man writes a message, places it in a bottle and casts it into the sea, do not those words lie inert and dormant until the bottle is discovered? Certainly so.

Yet, if a dying person writes their final program, presses "Run" and perishes, what can we say? Nothing so concrete. We can only shrug our shoulders and say, I need to know more about this story. What is the text of this program that survived, and what computer was it running on, and what network, and was there was any other technology involved? A robotic arm? A self-driving car? A nuclear missile? A great many things might happen after this person's death – their will living on, encoded into a program. It's not a message bottled up; it's a message uncorked and in motion. Motion can be hard to predict.

Still, on real software teams: Programs *are* shared between people. What you write today will be read tomorrow – by you and by others. Small programs, if they're lucky, become big programs that require many people working together, all reading and understanding the same growing piece of text. By one report, in 2016, Google had approximately 30,000 software developers working on a multitude of codebases (Security and Exchange Commission 2014). In these large endeavors, being able to write code that communicates to computers *and* to human beings, who are writing and editing the same textual artifact, is critical.

On the entryway to his school in Athens, Plato is said to have written, "Let no one ignorant of geometry enter." Perhaps upon the school for coding wizards there should be written, "Let none enter here who write programs solely for machines."

Wise coders know this. But the reason we must specify it up front is that we can all fail to do so. It must, therefore, be a practice and a discipline. Not just to write, but to write well, to write clearly, to write for others, to write better over time.

Natural languages are similar in this way. But the difference is still clear: In natural languages, the option of writing them "incidentally for machines" or otherwise never comes up. Freed from that option, we are freed from that temptation.

Machines Anchor Language

Both programming languages and people languages change and grow over time. However, people languages change slowly and seem to have a mind of their own.

Programming languages, on the other hand, change immediately, and only when we allow them to.

Looking back in time at an earlier English can be an odd experience.

```
And gladly wolde he lerne, and gladly teche.
The lyf so short, the craft so long to lerne
For hym was levere have at his beddes heed
Twenty bookes, clad in blak or reed,
Of Aristotle and his philosophie...
```

English didn't always look or sound the way it does, as anyone who has read Chaucer – our "Father of English literature" – can see. Certainly, the English language does change. Still, the above passages are *relatively* intelligible, in spite of being from the 1300s.

When it comes to English, the fact that it *can't* change suddenly is one thing that makes it so useful. We can still look back more than 700 years and understand the messages of the past. We can observe that, today, life is still as short and the craft is still as long as when those words were written. (These words were as true in Chaucer's day as they were in the days of Hippocritus, whom he was probably quoting).

Although we occasionally have to worry about new words like "selfie" popping into our lives or words like "levere" dropping out of it, we do not have to worry that we'll wake up one day, drive to Starbucks, and say "Can I have coffee?" and be met with blank stares. The basic words and grammatical structures we rely on are, thankfully, hard to change in dramatic ways on a large scale. To do so immediately is impossible. Our language is distributed among too many.

Consider for a moment what you would have to do if you *did* want to make a minor but definitive change to English. For example, suppose you wanted to add a word. As a first stab, maybe you could get dictionary writers to insert an entry for your word. That's probably not enough, though. People don't wake up in the morning and consult a dictionary for recent updates and then proceed through the day with the linguistic "patch" installed in their brains. You could try to use the internet to make your word "go viral" – but that is easier said than done. Barring that, you may have to launch a costly worldwide campaign.

Most people who are not giant corporations are unable to launch such campaigns. We mentioned Starbucks previously without having to define it. Starbucks already defined itself for us. Whereas Starbucks successfully added

the word "Starbucks" to our language, the quests of the rest of us will be harder. The rest of us sometimes have to band together to make the same kind of society-wide linguistic patches.

By comparison, programming languages are trivial to change. Python was first released in 1991. A version 2.0 was released just nine years later – in 2000. This version went through seven minor version changes, up to version 2.7 – at which point the language was given an end-of-life date of 2015 (which was later bumped to 2020). Although Python 2.7 was scheduled to die, the 3.0 (released in December 2008) is alive and well. And so on. If history is any indication, this version will not be the last.

Suppose you had been an ordinary Python programmer on that cold day in December 2008 when version 3.0 had been released. Suppose you got your morning Starbucks, and sat down at your computer to write some Python code. Would you have been met with a blank stare from a computer that yesterday understood version 2.7 and today understands only 3.0?

Luckily, not. It's perfectly normal for one computer to understand Python 2.7, even though other computers understand 3.0. You can even get your computer to understand either one – whichever you happen to want to use at a given time. This should come as no surprise: Earlier in this chapter, we installed Racket and then wrote "Hello, World" programs in several different languages. Python 2.7 and Python 3.0 are simply different languages – and it's quite easy to have different languages installed on the same computer.

The fact is that programming languages are just software – and as such, they get changed. New versions get released, and old ones die. Whereas people languages are not governed by an authority that legislates top-down changes – programming languages generally *are* governed in this way. Python's creator Guido van Rossum was given by the Python community the title "Benevolent Dictator for Life." It was a joke – but with a grain of truth. Although he voluntarily relieved himself of this title in 2018, he was a major player in the ongoing design and redesign of the Python language for more than a quarter of a century. Although he can't reach into your computer and legislate the version of Python that's installed there – he can (like you) install his favorite version on his own computer, release it to others, and then talk about all the cool features that it has. Others can follow suit and switch. Or they can decide not to – even on a project-by-project basis.

This is exactly what you *can't* do with people languages. You can't just start saying "I'd like a large gorblesnop with milk, please" at Starbucks without immediate backlash – even if you go on to clarify that by "large" you meant "venti." But you

can create your own programming language and use it all you like without anyone even knowing. It is, therefore, much easier to experiment with new programming languages than it is to experiment with new people languages. A corollary of this is that it is much easier to create multiple versions of the same language, or new languages entirely. We need only create them for ourselves and use them long enough to convince ourselves that they are better. Then, when the time comes to convince others, they can dip their toe into the new language without fully committing – that is, by installing it alongside their other languages so that they can use the new without abandoning the old.

Reading Chaucer shows us that people languages change and drift. But the changes are rarely driven by a single person, or even traceable back to a single person. Shakespeare coined many new English words – like "articulate" and "invulnerable" – but that kind of single-person linguistic contribution is the exception, not the rule.

When linguistic construction by a centralized authority *is* the rule, it is the stuff of nightmares – or dystopian novels like *1984*. The Party's language of Newspeak is designed to limit free thought by redesigning free speech – removing politically undesirable words. Negative words like "bad" are removed in favor of modified positives like "ungood"; new words like "goodthink" are given definitions like "conformance with the Party's ideals." The book meditates on the Sapir-Whorf hypothesis – a hypothesis that suggests the structure of our native tongue limits the space of thoughts we are able to independently explore. If the Sapir-Whorf hypothesis is true, then such a dystopian nightmare is nothing short of state control over people's ability to think.

With programming languages, the situation is reversed: Changes never happen without someone initiating them, and we can almost always trace those changes back to the small number of individuals who initiated them. The whole process of changing a computer language is consciously managed, by small groups or by large ones. The process is not unconscious. In some open source projects, the changes can be publicly discussed by thousands (as was the case with Mozilla's Rust programming language, with more than 2,000 people contributing to the project). All are willing participants in changes to this tool of the mind.

In coding, one reason so many can collaborate is that it's easier to agree on the meanings of things. In a programming language, we can fall back on defining the "meaning" of some word or phrase as "what it makes the computer do." In a natural language, we define the "meaning" of a word or phrase as "what other people will understand."

When it come to meaning, software tends to be more consistent than brainware (or as it is commonly called, "wetware"). In practice this means that if two programmers disagree about what a program means, they don't need to argue about it. They *can*. . . But they can often just run the program and see who is right and who is wrong. Whoever disagrees with the computer is wrong.

What's more, programmers fundamentally agree about this method of settling disagreements. It's part of becoming a programmer – realizing that a program's meaning is defined by what the computer actually does, not by what we thought we told it to do.

With this empirical test to anchor the meaning of programs, we are free to experiment. For example, we can create new languages – knowing that if we create an abomination, we can return to the safety of our last working version. And that last version will have the semantics we all agreed upon at the time.

Consider a phrase like "life is short; but art is long":

Does it ring to you as a lament about the shortness of life? Or is it a celebration about the staying power of art? Does "art" apply to software artifacts? Is life an art? Is art life? The questions of meaning are potentially endless – a single phrase, an infinity of questions.

With a program, the buck stops – computers having only one interpretation. Programmers know this, which cuts down on a certain amount of philosophical discussion from the get-go. A finite number of questions will answer any meaningful question – that is, any question about meaning.

Now That It's Out of Our System

We have just spent a lot of natural language to articulate the differences between computer languages and natural languages. But the truth is that the two are intertwined in ways that are truly fascinating. These are the main motivations for this book.

Yes, computer languages are smaller (in terms of raw vocabulary), and, yes, they use a different channel, and, yes, they don't even need a channel at all necessarily, and, yes, they have precise meanings and do not drift when we create new versions.

But, as we will see, computer languages and natural languages are locked together in a cyclical way. The construction of new programming languages often requires the construction of highly specialized people languages. And the construction of new, highly specialized people languages often arises from the construction of new programming languages.

Furthermore, as we have already foreshadowed, programmer brains process programming languages with the same parts that process natural language. So, perhaps the two kinds of languages are more similar than they appear at first glance.

Their similarities might not always be clear "out there." But the science is clear: "In here," as neurocognitive activities, they are interwoven.

To be continued . . .

LANGUAGES WITHIN

It is easier to talk about language "out there" than to talk about what happens when our brains process language. Much of that inner process is unconscious, unknown even to ourselves. Many secrets of that gray organ have remained locked inside our skulls, hidden from scientists for as long as science has existed, hidden from all of us for as long as we have existed.

Our attempts to measure something often changes the thing we are measuring – the brain being particularly guilty of this. Throughout history, perhaps more so than any other object that we have yearned to study, the healthy functioning brain has resisted the crude instruments of science: our surgical knives and our microscopes. Such an inconveniently fragile thing – it tends to stop being definable as "healthy or functioning" the moment we start probing it with our instruments.

In 1892, an antique dealer named Edwin Smith bought an ancient Egyptian papyrus more than 15 feet long. Although it was unintelligible to him, it was later discovered to be a medical text from circa 1600 BCE, containing the world's earliest recorded observations about the brain. This "message in a bottle" revealed that the author, an ancient Egyptian surgeon, had made observations related to brain injuries and their adverse linguistic effects – for example, aphasia, the inability to speak.

It's true. The earliest record we have about the brain contains observations about language.

For many centuries, the knife was indeed the scientific instrument of necessity. The Romans used it to cut up the brains of animals. Andreas Vesalius, in the 1500s, used the knife to dissect human cadavers, including their brains – positing that the brain makes use of the nervous system for transmitting both sensation and motion (Vesalius 1543).

In 1861, French anatomist Pierre Paul Broca used the knife to uncover the first evidence that the brains have substructures responsible for different activities of the mind. One area of the brain still bears his name today: Broca's area. It was given to a portion of the frontal lobe of a patient who had, over a period of 21 years, progressively lost the ability to speak and move, but never lost the ability to think or comprehend language. Of course, Broca did not know why until, post-mortem, his knife revealed a lesion at what we now call "Broca's area." Until then, it had not been shown that one region might control language production, whereas others might control comprehension.

As we linguistically wired creatures so often do when we discover something new: We named it. Indeed, as we so often do: We gave it a name that had already been given out. As a result, there is now a region within all of our skulls that bears the name of a French anatomist from the 1800s (for reference: born about a decade after Ada Lovelace).

Knives and microscopes gave rise to many of our named discoveries over the years: lobes, hemispheres, neurons, axons, dendrites, synapses, and so on. More cutting led to the discovery of more pieces worth giving names to. It wasn't until 1964, however, that the first department of neuroscience was founded at the University of California, Irvine. Harvard followed suit a few years later. Today, the field is firmly established in our scientific institutions, and our tools have evolved considerably beyond just knives and microscopes (though it should be mentioned that these tools are by no means obsolete!).

One tool is functional magnetic resonance imaging (fMRI). First used on humans in 1992, fMRI allows us, without the use of knives or saws or metal probes, to peer into the skull, into the brain's inner workings – all with virtually no discomfort to human subjects – save that they must lie still inside of a machine for the duration of the experiment. The subject, however, may speak, listen, read, and perform a range of tasks. All the while, scientists may gather precise spatial data about which parts of the brain are active.

Finally, we could ask questions about how we process language, without worry that our instruments of inquiry would change, damage, or destroy the object of inquiry. And furthermore, we could do it as the subject was in the midst of linguistic acts (actually reading words!) – rather than, as Broca was forced to do, wait for 21 years until a subject had died.

Within the decade, fMRI was being used to study language. If Broca had magically lived more than a century longer, he would have seen the region bearing his name light up on a screen, confirming his life's work. The "lighting

up" is a computer's rendering of the fact that the blood near Broca's area is deploying oxygen to the neurons there more quickly than to other areas. The effect can be detected by a magnetic field because oxygenated blood has different magnetic properties than deoxygenated blood. This oxygenation response literally fuels thought: Neurons need the oxygen to fire, and thoughts need neurons to fire.

The combination of knives and modern instruments has allowed humanity to discover or confirm the role of several areas in the brain, further refining our "atlas" of the brain.

We now know of areas that play roles in a range of other linguistic or language-related activities: working memory, word retrieval, object naming, planning, decision making, and so on.

In 2014, scientists showed that areas such as these – language areas – were the ones lighting up when programmers read programs. In 2017, it was confirmed.

Interestingly, this wasn't the first visual-spatial language discovered to do so.

Signed Languages

An obvious question: Do signed languages activate the same parts of the brain as spoken language?

Broca's area is fairly close to parts that control the face (including the mouth), whereas another linguistic region called Wernicke's area is close to the auditory cortex. One might then hypothesize that both regions are specialized for speech. Furthermore, the left hemisphere had often been considered to be for verbal processing, whereas the right was for spatial processing – which raised questions about where sign language processing might occur.

However, studies of deaf signers who have sustained brain injuries have shown that damage to the left hemisphere does inhibit signing ability, whereas damage to the right hemisphere does not (Hickok et al. 2002). This would suggest that sign language is at least processed in the same hemisphere as spoken language.

Localizing it further, damage to the left frontal regions (near Broca's area) results in difficulties with producing signs – just as damage to Broca's area results in difficulty producing words. Damage to the left hemisphere in the temporal regions (near Wernike's area) results in difficulties with comprehension of signs – just as damage to Wernike's area results in difficulties understanding spoken words. Thus, despite the modality difference, each form of language uses the same wetware for similar tasks.

Studies across a variety of techniques (including fMRI) seem to support this, with one meta-analysis (Campbell et al. 2007) summing up the current scientific understanding:

> . . . [P]art of Wernicke's region appears to be particularly interested in signed language processing, just as it has been shown to be for spoken language processing using similar experimental paradigms (see Scott, Blank, Rosen, & Wise, 2000; Narain et al., 2003). Thus, this appears to be a language-specific region that is not sensitive to the modality of the language it encounters. . . . Every neuroimaging study to date reports activation in the left inferior frontal regions for signed language production and planning. Broca's area is always involved, and left hemisphere dominance is always observed.

Regions like Broca's area are now conclusively linked to spoken languages, signed languages, and programming languages.

As is the case in many stories, protagonists occasionally pick up new companions along the way. Our main protagonists are, of course, programming languages. And spoken languages are an old friend. But let us briefly fill in the surprisingly tragic back-story of signed languages, the latest addition to the party. If they share a neurocognitive boat with our protagonists, it makes sense to get to know them.

Silent Battles

Gallaudet University, whose charter was signed by Abraham Lincoln, is a thriving deaf university with over 60 bachelor and graduate degree programs, and even a Ph.D. program in educational neuroscience. Many American high schools offer ASL, counting it as a foreign language (though it sprang up on American soil). As of 2016, ASL was the third most studied language in U.S. institutions of higher education, with approximately 31% of American colleges offering ASL classes (Looney and Lusin 2019). It is becoming more mainstream to see deaf actors signing on television – for example, *Switched at Birth* (2011-2017) – and in blockbuster movies – for example, *A Quiet Place* (2018).

However, there was a time when neither the American public, nor the American scientific establishment, nor American educational institutions considered ASL to be a true language. It was a century of educational darkness.

But let us start at the beginning – a time of growth and prosperity.

For reference, the story starts roughly between the years that Babbage was in university (the 1810s) and ends in the year that the Lisp programming language

was released (1960). In other words, while the rest of the world was inventing the computer and the second high level programming language after Fortran, another very different kind of visual-spatial language experienced an arc of its own: It was born, grew into something beautiful, was almost erased from the earth by dystopian educational reform, and was finally saved (in part) by a formal language called Stokoe notation – with just a few keywords and grammatical structures.

ASL students are required to learn the beginning of this story by heart, an educational oral tradition – much like our `"Hello, World"` tradition. You can watch students and experts alike signing it on YouTube by searching for "ASL Gallaudet Story."

As it begins: In the early 1800s, a hearing man named Thomas Hopkins Gallaudet, from Connecticut, traveled first to England and then to France in search of a teacher for his neighbor's daughter, who was deaf. In France, he met a deaf educator named Laurent Clerc, fluent in what is now known as Old French Sign Language. The two of them sailed back to America and, in 1817, founded a school that used sign language for instruction – the first of its kind on American soil. A group of deaf students came from a place called Martha's Vineyard (near Cape Cod) – where unusually high rates of congenital deafness had led to a community of signers – some hearing, some deaf, all signers. Others came from nearby villages in Maine and New Hampshire, bringing their own, independently developed sign languages. Others brought their own "home-signs," developed solely to communicate with family members within their own home. Thus began the evolution of American Sign Language – a melting pot of Laurent Clerc's Old French Sign Language and all of these various indigenous sign languages that had sprung up organically in America prior to his arrival.

All seemed well for half a century. However, after the end of the American Civil War, a counter-movement in deaf education called "oralism" began to emerge. The key players in this movement were neither deaf nor fluent in sign language. Although their reasons were diverse, they were unified in their anti-signing philosophy. Some believed simply that the deaf should learn to speak and lip-read to become more tightly integrated with society – whereas others believed that sign language was an inferior language, a relic of our ape ancestors and "savage races" like the Native Americans and other tribal people (Baynton 1998).

It would be a century before science would show the grammatical structures of signed languages to be as rich and complex as any spoken language.

There would be no fMRI studies to save the day either – not for almost a century and a half.

No, at this time, science was on the dystopian side: The emerging vocabulary of Darwinism had been co-opted by eugenicists, who believed the human population should be improved by discouraging the reproduction of certain individuals with undesirable qualities, such as disabilities or certain skin colors. Many oralists used this same line of reasoning to make their case against sign language. A key leader in the oralist movement – none other than scientist and inventor Alexander Graham Bell – wrote an article called "Upon the Formation of a Deaf Variety of the Human Race" in which he wished to call attention to the problems of the deaf people intermarrying, reproducing, and thus passing their deafness on to the next generation. He ends the essay with the sentence:

> Having shown the tendency to the formation of a deaf variety of the human race in America, and some of the means that should be taken to counteract it, I commend the whole subject to the attention of scientific men.

As it turned out, for Bell, the recommended way to prevent "the deaf race" was to stop using sign language in deaf education. You can't erase a language from a brain without changing the wetware; but you can erase it from the world simply by changing education. And so, Bell – the father of the telephone – also influenced educational establishments to begin silencing that already-silent language. In 1880, there began what is referred to by deaf historians as "the dark ages for deaf education in America" (Winefield 1987).

The dusk began at what sounds, at first glance, like a relatively innocent place: a meeting of the International Congress on the Education of the Deaf. Here, however, it was decreed that the best means of instruction for the deaf was oralism – the teaching of speech and lipreading in classrooms, and the active suppression of sign language. Of the 164 educators who attended that meeting, only one of them was deaf. One of their resolutions was:

> Considering the incontestable superiority of speech over signs in restoring the deaf-mute to society, and in giving him a more perfect knowledge of language. . . . That the Oral method ought to be preferred to that of signs for the education and instruction of the deaf and dumb.

For three days at that conference, there were presentations from oralism's champion – Alexander Graham Bell. He was deeply affected by his mother's gradual progression into deafness, which began when Bell was age 12. Around this time his father, a famous phonetician, invented a writing system called Visible

Speech: a collection of symbols that described how speech organs should be positioned in order to produce sounds. This was a notation for pronunciation that Bell believed could be used in deaf education, and in which he became fluent at the tutelage of his father. Later, his growing wealth from inventions gave him a platform from which to promote Visible Speech and oralism.

The proponents of sign language – who were not in attendance – could do nothing to stop the conclusions reached. Nor could they stop the worldwide damage that followed. After the 1880 decision, the dystopian conversion of deaf education began: One-by-one, hearing teachers replaced deaf educators such that by 1919 over 80% of educators were neither deaf nor fluent in sign language. Entire schools switched from the manual method to the oral method. Sign language itself was banned in classrooms and other formal educational settings. Students caught using it were punished. Some were forced to wear gloves tied together to prevent the use of signs.

The language was preserved in boarding school dormitories, in Deaf social circles, and through the intentional use of film technology, which arrived just in time. Here is an English-interpreted message in a bottle from a 1913 American film called "The Preservation of the Sign Language" (Veditz 1913):

> We American deaf are now facing bad times for our schools. False prophets are now appearing, announcing to the public that our American means of teaching the deaf are all wrong. These men have tried to educate the public and make them believe that the oral method is really the one best means of educating the deaf. But we American deaf know, the French deaf know, the German deaf know that in truth, the oral method is the worst. A new race of pharaohs that knew not Joseph is taking over the land and many of our American schools. They do not understand signs for they cannot sign. They proclaim that signs are worthless and of no help to the deaf. Enemies of the sign language, they are enemies of the true welfare of the deaf. We must use our films to pass on the beauty of the signs we have now. As long as we have deaf people on earth, we will have signs. And as long as we have our films, we can preserve signs in their old purity. It is my hope that we will all love and guard our beautiful sign language as the noblest gift God has given to deaf people.

And so film technology was used for signed language in the same way that writing systems had been used for spoken languages for centuries: to encode words and thoughts for retrieval by others in the future – messages in bottles. To make words live beyond the ephemeral moment of their birth.

So the battle of language was waged. On the one side, the oralist's Visible Speech writing system was used as a tool to dislodge sign language from deaf

education – although this was later abandoned. On the other side, sign language retreated to the shadows while signers turned to the technology of the time to make their linguistic acts less ephemeral – and harder to dislodge. All the while, the battleground was the American education of deaf children.

Visible Speech, lipreading, and phonetic education were not the only linguistic tools designed by non-signers to replace American Sign Language. Many such tools were developed. Manually Coded English is a signed language that borrows English grammatical structures, while mapping spoken words to signs borrowed from ASL. Another system, Signing Exact English has a similar motivation, as does Seeing Essential English.

Oddly enough, these linguistic tools were designed in service of erasing a language that wasn't designed at all: one that evolved naturally – by Americans, for Americans. Whether you saw these new languages as tools or weapons, depended very much on which side of the linguistic trenches you were standing.

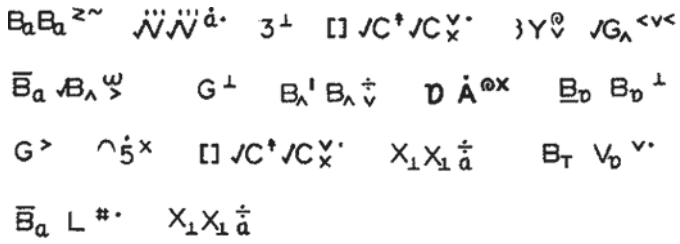
The dawn began to break in the 1960s, when linguist William Stokoe revolutionized his field by studying American Sign Language. He was the first to explain what had always been true: American Sign Language was a rich and thriving linguistic system, complete with its own grammar and syntax, worthy of study in its own right, of deep value to those who know it, and of tremendous potential value to any who choose to learn it. He was able to bridge a linguistic gap between ASL and the meta-language of linguistics. His book *A Dictionary of American Sign Language on Linguistic Principles* popularized the name "American Sign Language," permanently displacing other names like Ameslan, manual communication, and sign language.

David Armstrong, anthropologist and former Gallaudet University administrator, said a few days after Stokoe's death in the year 2000 (Armstrong 2000):

> At the time of his arrival at Gallaudet, the sign language used by deaf Americans and now known as American Sign Language (ASL) was generally believed to be a corrupt visual code for spoken English or elaborate pantomime. It and other national sign languages were widely suppressed in educational programs for deaf students, in favor of instruction in articulation and lip-reading. Stokoe proposed instead that ASL was, in fact, a fully formed human language in the same sense as spoken languages like English. He set about devising a descriptive system for the language that could be used to demonstrate this point to other linguists and the general public.

Note this final sentence. Stokoe's contribution wasn't just to *observe* that ASL was a language – not just to proclaim this fact in English. His contribution was

to devise a way of explaining that fact with new, different language. In order to do so, he had to invent a new notation.



Above is the beginning of the classic Goldilocks fairytale, with the ASL written in Stokoe notation. Each "word" is a sign in ASL. Each symbol within each word denotes how that sign is performed (how you hold your hand, the movement of your hand, whether you use one hand or two, etc.)

Consider Stokoe notation, which participated in the elevation of ASL, in relation to Bell's notation, which participated in its suppression. If you ignore this opposition, both are surprisingly similar: One is a notation for how speech organs should be arranged to produce English sounds; the other is a notation for how the hands and fingers should be arranged to produce ASL signs. Both are simply new ways of writing down things that were already known to many.

Both made their impacts. Both have faded into disuse, as if they only needed to exist for a brief moment in history, just long enough to cause change.

Although the fight for Deaf rights and better education for the Deaf is far from over, the dystopian dark ages have ended. Armstrong goes on to say:

> . . . Stokoe's other published works won wide acceptance in the linguistic community and ultimately among educators of the deaf, such that ASL is now widely recognized as an appropriate language of instruction for deaf students and even as an appropriate second language for hearing students in high schools and universities in the United States. Stokoe was also a tireless personal advocate for the linguistic and educational rights of deaf people, often in the face of skepticism or even outright hostility.

Stokoe didn't just elevate ASL, though. He also paved the way for ASL to elevate the study of linguistics itself. After his demonstration that ASL was indeed a language, any serious investigation of "language in general" from that point forward would need to consider signed languages.

Our Strange Citizens of Broca's Area

fMRIs show that a great many things are processed by the parts of the brain that were first identified in the context of processing speech: programming languages, signed languages, spoken languages. Studies have also found that the processing of musical syntax (Kunert et al. 2015) and the processing of simple formal languages known as "artificial grammars" are also processed in Broca's area (Fitch and Friederici 2012).

Teachers and students of *any* language: the science about other languages matters. Each sheds light on the other. They're all in us together.

To be continued . . .