1 Thinking and Computation

Consider the following scenario:

> A professor enters a classroom where a group of undergraduates is sitting, and announces "There is free pizza in the hall!" Suddenly, the students stand up and stampede toward the classroom door.

The events described here seem so ordinary that it is easy to miss how truly remarkable they are. But step back from them for a moment and imagine that you are studying these students as a curious scientist from another world. You observe that a certain sound emanates from the professor and that this causes a flurry of activity in the students. Now, as a scientist, ask yourself this: What sort of *physics* would explain how acoustic energy can be transformed into kinetic energy in this way? In particular, note that a very small change in the acoustic energy (like the professor's saying "There is free pizza in Nepal.") can result in *no* kinetic energy being produced at all, except maybe for some puzzled head shaking.

This is the wonder of *intelligent behavior*, perhaps the single most complex natural phenomenon that we are aware of. As a sheer mystery, it easily overshadows topics like dark matter, the source of gravity, and the mechanics of cancer.

One striking aspect of intelligent behavior of this sort is that it is clearly conditioned by *knowledge*: for a very wide range of activities, people make decisions about what to do based on what they know (or believe) about the world, effortlessly and often unconsciously. It's certainly not the *sounds* themselves that cause the students to stand up like animals that have been trained to respond to a bell. This is easy enough to confirm. The professor could have brought in a *sign* with a pizza message on it written in big letters, and the effect would have been just the same. In fact, one can imagine a situation where the following is written on the whiteboard at the front of the classroom:

> As part of a psychology experiment, the professor will soon enter and tell you that free food is available. This is just a test. Please remain seated.

In this case, neither the sounds nor the sign would have any effect at all. One can try other small variations, and it will become clear that what makes the difference is whether the students come to *believe* there is free pizza to be had nearby.

Using what we believe or know in this way is so commonplace that we only really pay attention to it when it is not there. When we say that someone behaved *unintelligently*, for instance, when someone uses a lit match to see if there is any gas in a car's gas tank, what we usually mean is not that there is something the person did not know but rather that the person has failed to use what he or she *did* know. We might say: "You weren't thinking!" Indeed, it is *thinking* that is supposed to deliver what we know to the decisions we need to make. The students head toward the door because they *think* that is the way to free pizza. It is thinking, in the end, that makes human behavior intelligent.

But what is thinking, and how does it work? What exactly goes on in people's heads when they think about where to go for free pizza, or about who will win the Academy Award for Best Actor, or about whether a free market needs to be regulated?

The purpose of this book is to suggest where to look for an answer. It proposes that thinking is a form of computation. In the same way that digital computers perform calculations on representations of numbers, human brains perform calculations on representations of what is known.

This chapter is an introduction to this idea. The first section reviews very briefly the notion of thinking. The process of computation is somewhat less familiar, so more time is spent on it, in the second section. The third section introduces the (somewhat controversial) idea of thinking as a form of computation.

1.1 Thinking

Are brains like computers? In a word, *no*. It is true historically that in trying to understand the brain, people have proposed models that seem to mirror the most advanced technology of the time. Over the years, the brain has been described as clockwork, a steam engine, a telephone switchboard, and (these days) a computer. But in time, these descriptions are found to be much too simplistic to say anything useful about what is inside our heads. There is no reason to believe that the computer analogy will be any different in this regard.

In fact, this book has very little to say about the brain itself. Rather it focuses on thinking. But thinking is what the brain does. How can one study the relation between computers and thinking without studying the brain?

Here is a useful analogy. Consider the study of flight (in the days before airplanes). One might want to understand how certain animals like birds and bats are able to fly.

One might also want to try to build machines that are capable of flight. There are two ways to proceed:

- study flying animals like birds, looking very carefully at their wings, their feathers, their muscles, and then construct machines that emulate birds;
- study aerodynamics—how air flows above and below an airfoil, and how this provides lift—by using wind tunnels and varying the shapes of airfoils.

Both kinds of studies lead to insights, but of a different sort. The second strategy is the more general one: it seeks to discover the principles of flight that apply to anything, including birds.

It is this second strategy that is used here to study thinking. While there is a lot to be learned by studying the brain, this book focuses on the *thinking process* itself to determine general principles that will apply to brains and to anything else that needs to think.

1.1.1 What is thinking?

What exactly is thinking? For humans, it is clearly some sort of process that occurs in our heads over time. The easiest way to understand it is to observe it in action.

Read the following sentence:

> *The trophy would not fit into the brown suitcase because it was too small.*

Pause for a moment to make sure you understand it. Now answer this question: What was too small? What does *it* refer to? Clearly, *it* refers to the suitcase, not the trophy.

Now, how did you get the right answer?

Observe that there is nothing in the sentence itself that gives away the answer. This is easy to demonstrate. Simply replace the word *small* by *big*:

> *The trophy would not fit into the brown suitcase because it was too big.*

What was too big? Now the answer goes the other way: *it* now refers to the trophy, not the suitcase.

What this shows is that in making sense of the sentence, in particular, in determining what *it* refers to, you had to use what you already knew about the sizes of things, things fitting inside other things, and so on, even if you were unaware of doing so.

This is thinking.

Thinking is bringing what you know to bear on what you are doing. This is what you had to do to understand the sentence. The process happens in the brain, sometimes very quickly, and you may or may not be aware that it is taking place. (You may not have felt the change that happened in your thinking when the word was changed from *small* to *big*.)

Thinking is clearly a biological process, since we are biological creatures. Does this mean that thinking is like digestion? Or like mitosis in cells? Or is it different still?

The central conjecture of this book is this:

Thinking can be usefully understood as a computational process.

Thinking has perhaps more in common with multiplication or sorting a list of numbers than with digestion or mitosis.

This conjecture is *controversial*. Not all philosophers, psychologists, evolutionary biologists, and neuroscientists are lined up to support it. Some do, some don't, and some struggle with the concept of thinking in the first place.

Nonetheless this is the conjecture that this book pursues.

1.2 Computation

Computer science as a field of study has two branches: hardware and software. Hardware is the concern of engineers who study and build the physical machines (computers) themselves. Software is the written instructions—programs—specifying what those machines should do. And what the machines do is *computation*, a certain manipulation of symbols. Birds "produce" flight, musicians "produce" music, and computers "produce" computation according to a program.

This book focuses on computation itself as the primary subject of computer science. What (or who) is performing the symbol manipulation is a secondary concern. Modern electronic computers happen to provide a fast, cheap, and reliable way to do this computation, but other devices can do it as well. (The Dutch computer scientist Edsger Dijkstra once said that computer science is no more about the computers than astronomy is about telescopes.)

1.2.1 Symbols and symbolic structures

In their simplest form, *symbols* are just characters from some alphabet, like the following:

- digits: *3*, *7*, *V* (the last one is a Roman numeral)
- letters: *x*, *R*, β (the last one is a Greek letter)
- operators: +, $\leq$, $\cap$

They can be strung together into more complex forms:

- numerals: *5874, −3.75*
- words: *John, don't*

They can be grouped in certain ways:

- mathematical expressions: $247 + 4(x-1)^3$
- English phrases: *the woman John loved*

Finally, some of these groupings can be thought of as being true or false:

- mathematical inequalities: $247 + 4(x-1)^3 \leq \frac{n!}{4}$
- English sentences: *The woman John loved had brown hair.*

It is *symbolic structures*, arrangements of symbols like those just illustrated, that are the medium of computation.

1.2.2 What is computation?

For present purposes, *computation* is the process of taking symbolic structures, breaking them apart, comparing them, and reassembling them according to a precise recipe called a *procedure*. The symbols at the start of the procedure are called the *inputs*. The symbols at the end of the procedure are called the *outputs*. The procedure is *called* on the inputs and *returns* the outputs. It is important to keep track of where you are and to follow the instructions in the procedure *exactly*. (You may not be able to figure out *why* you are doing the steps involved. No matter.)

1.2.3 Some arithmetic procedures

Imagine explaining to someone (a young child) how to do subtraction:

$$\begin{array}{r} 53 \\ -\ 17 \\ \hline \end{array}$$

One might say something like this:

> First subtract the 7 from 3. But since since 7 is bigger than 3, borrow 10 from the 5 on the left. That changes the 3 to a 13 and changes the 5 to a 4. So subtract 7 not from 3 but from 13, which gives 6. Write the 6 as the first digit of the answer on the right, under the 7. Then subtract 1 not from 5 but from 4, which gives 3. Write the 3 as the second digit of the answer, under the 1. So the answer is 36.

Figure 1.1. Adding two single-digit numbers

Procedure PROC0:

You are given two digits as input and will return two digits as output.

To do so, use this table.

	0	1	2	3	4	5	6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
1	01	02	03	04	05	06	07	08	09	10
2	02	03	04	05	06	07	08	09	10	11
3	03	04	05	06	07	08	09	10	11	12
4	04	05	06	07	08	09	10	11	12	13
5	05	06	07	08	09	10	11	12	13	14
6	06	07	08	09	10	11	12	13	14	15
7	07	08	09	10	11	12	13	14	15	16
8	08	09	10	11	12	13	14	15	16	17
9	09	10	11	12	13	14	15	16	17	18

To add the two digits, find the *row* for the first digit in the table, and find the *column* for the second digit, and return as output the two digits that appear at their intersection in the table.

These are the kinds of detailed instructions one wants in a procedure.

PROC0

Maybe the simplest job in arithmetic, the one learned before any others, is the addition of two single-digit numbers. Figure 1.1 shows a procedure called PROC0 that gives explicit instructions for adding two digits.

Notice that it makes use of an *addition table*. (Typically, young children are not taught addition in this way, but most of us were taught multiplication using a table like this, which we had to memorize.) One can see that if PROC0 is called on `7` and `6`, it will return `13`, and if it is called on `3` and `5`, it will return `08`. (It will be handy for the answer returned here to be the two digits `08` and not simply `8`.)

Figure 1.2. Adding three single-digit numbers

```
Procedure PROC1:
    You are given three digits as input, $a$, $t$, and $b$.
    You will return two digits as output, $c$ and $s$.
 1. Call PROC0 on $t$ and $b$, and let $u$ and $v$ be the answers returned.
    (The $u$ is the left digit returned by PROC0, and the $v$ is the right one.)
 2. Call PROC0 on $a$ and $v$, and let $u'$ and $v'$ be the answers returned.
    (The $u'$ is the left digit returned, and the $v'$ is the right one.)
 3. If $u = u' = 0$, then return with $c = 0$ and $s = v'$.
    If $u = u' = 1$, then return with $c = 2$ and $s = v'$.
    Otherwise, return with $c = 1$ and $s = v'$.
```

PROC1

Now, building on this procedure, consider adding three digits. Intuitively, one should add the first two digits and then take the sum and add the last digit to it. However one needs to worry about the carry digits to get the answer right. The procedure PROC1 in figure 1.2 does this. It calls PROC0 twice and then decides whether the final carry digit to return should be 0, 1, or 2.

Trace the behavior of this procedure in detail to make sure you understand how it works. Suppose PROC1 is called with inputs 7, 4, and 5.

At the start, $a = 7$, $t = 4$, and $b = 5$. Determine the values of c and s to return.

1. Call PROC0 on 4 and 5, which returns 09. So $u = 0$ and $v = 9$.
2. Call PROC0 on 7 and 9, which returns 16. So $u' = 1$ and $v' = 6$.
3. Then $u \neq u'$, so return with $c = 1$ and $s = 6$.

The output returned by PROC1 in this case is 16, as desired.

Here are some other examples to make sure you understand how this works:

- If PROC1 is called on 3, 4, and 1, it will return 08: u and v will be 0 and 5, and u' and v' will be 0 and 8, so c will be 0 and s will be 8.
- If PROC1 is called on 8, 9, and 2, it will return 19: u and v will be 1 and 1, and u' and v' will be 0 and 9, so c will be 1 and s will be 9.

Figure 1.3. Adding two multidigit numbers

Procedure PROC2:

You are given two multidigit numbers as input, each with the same number of digits:

$$x_1\ x_2\ \ldots\ x_k$$
$$y_1\ y_2\ \cdots\ y_k$$

You will return a number with one additional digit as output:

$$z_0\ z_1\ z_2\ \cdots\ z_k$$

1. Start at the right-hand side of the inputs (looking at x_k and y_k).
   Call PROC1 with a as 0, t as x_k, and b as y_k.
   Let z_k be the s returned. (Keep the c returned for the next step.)
2. Move over one step to the left (looking at x_{k-1} and y_{k-1}).
   Call PROC1 with a as the c from the previous step, t as x_{k-1}, and b as y_{k-1}.
   Let z_{k-1} be the s returned. (Keep the c returned for the next step.)
3. Continue in this way through all the digits, from right to left, filling out in turn, $z_{k-2}, z_{k-3}, \ldots, z_3, z_2, z_1$.
4. Let z_0 be the final c returned by PROC1 (with x_1 and y_1).

- If PROC1 is called on 8, 9, and 6, it will return 23: u and v will be 1 and 5, and u' and v' will be 1 and 3, so c will be 2 and s will be 3.

Note that it is not the business of the procedure to explain *why* the operations are done. It needs to make a decision about a final carry digit c. But as far as the procedure is concerned, the only thing that matters is what the final answer should be.

PROC2

Now, building on the procedure PROC1, consider instructions for adding two multidigit numbers. In this case, you know that you have to go from right to left, keeping track of the carry digits along the way. A procedure PROC2 for doing this is shown in figure 1.3. Trace the behavior of this procedure when the inputs are 747 and 281 (so that $k = 3$):

1. Starting at the right side, call PROC1 with $a = 0$, $t = 7$, and $b = 1$.
   It will return $c = 0$ and $s = 8$. So z_3 will be 8.

Figure 1.4. Adding any list of numbers

Procedure PROC3:

You are given a list of numbers n_1, n_2, ...
You will return a single number *sum* as output.

1. Let *sum* start off being the single digit `0`.
2. Start with the first number n_1 and *sum*. Make sure they both have the same number of digits by inserting `0` symbols on the left as needed. Then call PROC2 on these two numbers, and let the new value of *sum* be the number it returns.
3. Do the same thing with n_2 and the current value of *sum*, to produce the next value for *sum*.
4. Continue in this way with the rest of the numbers, n_3, n_4, ...
5. Return as output the final value of *sum*.

---

2. Call PROC1 with $a = 0$ (the c from the previous step), $t = 4$, and $b = 8$.
   It will return $c = 1$ and $s = 2$. So z_2 will be `2`.
3. Call PROC1 with $a = 1$ (the c from the previous step), $t = 7$, and $b = 2$.
   It will return $c = 1$ and $s = 0$. So z_1 will be `0`.
4. Finally, z_0 will be `1` (the last c returned).

So the answer returned by PROC2 on `747` and `281` will be the four digits `1028`. And sure enough, 1028 is the sum of 747 and 281. Again observe that the procedure does not explain itself. Nowhere does it say that each digit stands for a power of ten (units, tens, hundreds, thousands, and so on) with the unit digits on the right. However, even if you do not know what the symbols are supposed to mean or why you are doing the operations, if you follow the directions in PROC2 exactly, keeping track of where you are at each step, you will add the two numbers correctly.

PROC3

Now consider a general addition procedure that will add any list of numbers, each with any number of digits. The procedure for this is in figure 1.4. It is easy to see that it does the right thing, repeatedly adding two numbers using PROC2 and keeping a running total. (PROC3 is not explicit about how to pad the shorter numbers with `0` symbols on the left. Imagine there is a PROC4 that does that.)

Going from here

Given a procedure that can do addition, it is not too hard to imagine doing subtraction the same way. Given those two, one can define procedures that do multiplication and division. Building on these, one can have a procedure to tell whether a number is prime. One can use pairs of numbers to represent fractions (rational numbers) and do arithmetic on them. One can arrange rational numbers into matrices and have procedures that operate on them to solve systems of equations. One can use systems of equations to model complex physical systems and have procedures that perform numerical simulations of these systems.

And on it goes.

So starting with simple symbolic operations (such as table lookup, putting together and taking apart sequences of symbols, comparing them, and so on), one can assemble the operations into ever larger procedures and develop an extremely wide range of behaviors as computational processes. This is what computer science is about.

1.2.4 The lesson

The key observation on these arithmetic procedures is this:

> *To produce meaningful answers, you do not have to understand what the symbols stand for or why the manipulations are correct.*

Although one can certainly understand the procedures as doing arithmetic, one does not need this understanding to actually carry out the procedures.

Here is a simple thought experiment to support this claim. Imagine replacing the symbols `0` through `9` everywhere by new symbols that do not look at all like digits, for example, a heart shape for `0`, a star for `1`, an anchor shape for `2`, and so on. Now give the procedures PROC0–PROC3, including the table for PROC0 with the new symbols, to a friend without saying what these new symbols mean or what the procedures are supposed to be doing. By following PROC2, the friend should still be able to do addition: take as input two sequences of new symbols representing numbers, and return as output the sequence of new symbols that represents their sum.

So symbols can be processed purely mechanically and still end up producing the right results. This might be called the trick of computation:

> *Computers can perform a wide variety of impressive activities precisely because those activities can be described as a type of symbol processing that can be carried out purely mechanically.*

This “trick” has turned out to be one of the major inventions of the twentieth century, allowing devices that perform computation to permeate almost all areas of our modern lives. And note: It has nothing to do with electronics or physics.

1.3 Thinking as computation

One might still ask, though, just what does computation have to do with ordinary thinking? Recall the central conjecture of this book:

Thinking can be usefully understood as a computational process.

What does this conjecture amount to?

- *Not* that the brain is something like an electronic computer (which it is in some ways perhaps, but in most ways is not).
- The process of thinking can be usefully understood as a form of *symbol processing* that can be carried out purely mechanically without having to know what the symbols stand for.

Why is this so controversial? Perhaps the idea that *some* types of thinking are computational is not so surprising. Consider activities like doing a homework problem in algebra, or filling out an income tax form, or estimating a grocery bill as you are shopping. These all involve thinking and are clearly computational.

The problem is that so much of our thinking seems to have very little to do with calculations or anything even remotely numerical. You can think about anything you want, not just numbers. Consider this example:

I know my keys are in my coat pocket or on the fridge.
That’s where I always leave them.
I felt in my coat pocket, and there’s nothing there.
So my keys must be on the fridge, and that’s where I should look.

This is an example of thinking that appears to have nothing to do with numbers. But it is about something: keys, coat pocket, refrigerator. In fact, thinking always seems to be about *something*. Computation, on the other hand, seems to be about *nothing*: it is the process of manipulating symbols in a mechanical way without taking into account what the symbols stand for. So there is certainly a conceptual gap between the two that needs to be bridged. Fortunately, the bulk of the groundwork was already done by Leibniz.

1.3.1 Leibniz and his idea

The idea that thinking can be seen as a kind of computation is one of the rare ideas in Western culture that does not go back to the ancient Greeks. The first person to take this idea seriously was the German philosopher Gottfried Leibniz (1646–1716).

Leibniz was an amazing thinker. Among many other ideas and discoveries, he invented the calculus at the same time as Isaac Newton did. While Newton was interested in problems in physics and chemistry, Leibniz was more interested in symbols and symbol manipulation. He only started doing mathematics seriously later in life. But intrigued by how symbols standing for variables and constants could be shuffled around to solve equations in algebra, he wondered whether there were symbolic solutions to problems involving tangents and areas. And the infinitesimal calculus (derivatives and integrals) came out of this.

When it came to arithmetic, Leibniz observed that it was sufficient to manipulate symbols on a piece of paper according to certain rules to be able to draw conclusions about otherwise abstract numbers. A number (like fourteen, say) might be a completely abstract notion, but the symbols used to represent it (like the symbols *14* or *XIV* or *1110*) are much more tangible: we can write them down, look at them, move them around. We can determine if a certain relation holds among these numbers (for example, determining whether a number is the sum of two others) just by manipulating the symbols. (The symbols *1110* represent the number fourteen in the binary number system invented by Leibniz that is used by digital computers.)

His idea then was this: *Ideas*, that is, the objects of ordinary thought, are like numbers. It will be sufficient to manipulate symbols standing for them according to certain rules. The ideas may be abstract, but the symbols are concrete. One will be able to go from one idea to the next just by doing symbolic manipulation.

In other words, he drew the following analogy:

- The rules of arithmetic allow us to deal with abstract numbers in terms of concrete symbols. The manipulation of those symbols mirrors the relations among the numbers being represented.
- The rules of *logic* allow us to deal with abstract ideas in terms of concrete symbols. The manipulation of those symbols mirrors the relations among the ideas being represented.

What a truly remarkable idea! It says that although the objects of human thought are formless and abstract, we can still deal with them concretely as a kind of arithmetic, by representing them symbolically and operating on the symbols.

In the case of arithmetic, we already know what numbers are and what symbols we should use to represent them. We have all been trained to do this symbolic processing starting at a very early age, without a second thought.

But what about ideas? What symbols should stand for them?

1.3.2 Propositions vs. sentences

A *proposition*, as the word is used in the philosophical literature, is an idea that can be expressed by a declarative sentence of English (or other language). So one can think of the sentence as a symbolic representation of the proposition. Consider these examples:

- *My keys are in my coat pocket.*
- *Dinosaurs were warm-blooded.*
- *The stock market composite index will rise to twice its current value within the next three years.*
- *Hate literature should not be tolerated, even if that impinges on free speech.*

These are all English sentences. The first one uses seven words of English, for example. But apart from being English sentences, they each express an *idea*, an idea that can be expressed in other languages with other sentences. So we have the *sentence*, on the one hand (like the first one with seven words), and the *proposition* it expresses, on the other (like the idea that my keys are located somewhere).

What can be said about the propositions themselves? They are abstract entities, like numbers, but they have some special properties:

- Propositions are considered to *hold* or to *not hold*. A sentence is *true* if the proposition it expresses holds, and *false* if that proposition does not hold.

  This does not mean that there will be no controversy about whether the proposition holds. It just means that it makes sense to ask *if* it holds (or if the corresponding sentence is true). A number is a very different sort of abstract object; we do not ask if a number holds in this sense.

- Propositions are considered to be related to people in certain ways: people may or may not believe them, fear them, regret them, wish for them, worry about them, and so on. These various relationships between people and propositions are what philosophers call *propositional attitudes*.

- Propositions are related to each other in certain ways: a proposition might imply, or provide evidence for, or contradict another proposition.

Uninterpreted sentences

A first clue that one might be able to understand thinking as computation is to look at a sentence of English as a purely symbolic structure made up of a sequence of words. Consider this, for example:

The snark was a boojum.

This is a line from the poem *The Hunting of the Snark*, by Lewis Carroll, that was intended to be nonsense. (What is this snark? What is a boojum?) Observe that if one assumes that the sentence is *true*, even without knowing what the words *snark* and *boojum* mean, one can answer certain questions:

- What kind of thing was the snark?
  (It was a boojum.)
- Is it true that the snark was either a beejum or a boojum?
  (Yes, because it was a boojum.)
- If no boojum is ever a beejum, was the snark a beejum?
  (No, it could not have been.)
- What is an example of something that was a boojum?
  (The snark, of course.)

The point is that one can provide appropriate answers to these questions *without having to know what the two symbols mean.* This is the first step toward linking thinking and computation. Some simple rules of logic make it possible to extract answers directly from the sentence itself (viewed as a symbolic structure) without having to determine first what the symbols *snark* and *boojum* stand for.

Now consider the following three examples:

1. *My keys are in my coat pocket or on the fridge.*
   Nothing is in my coat pocket.
   So: My keys are on the fridge.

2. *Henry is in the basement or in the garden.*
   Nobody is in the basement.
   So: Henry is in the garden.

3. *Jill is married to George or Jack.*
   Nobody is married to George.
   So: Jill is married to Jack.

Observe that in all these cases the thinking is the same. The pieces are put together in exactly the same way, even though the sentences are about quite different things. There could just as easily be a fourth example:

4. *The frumble is frimble or framble.*
   Nothing is frimble.
   So: The frumble is framble.

Again, one does not need to know what *frimble* means to get the correct conclusion. It does not really matter whether the subject is keys, Henry, Jill, or the frumble. What does matter is the *form* of the sentences in terms of the other connecting words, and the conclusion based on that form. For example, it would be wrong in the last case to conclude that the frumble was frimble.

Logical entailment

Telling us what to conclude in such examples is the job of logic. A collection of sentences $S_1, S_2, \ldots, S_n$ *logically entails* another sentence S if the truth of S is implicit in the truth of the S_i sentences. In other words, no matter what certain terms (like *boojum*, *garden*, *framble*) in the S_i sentences really mean, if they are all true, then the S sentence is also true. So, in determining if a collection of sentences logically entails another, it is not necessary to know what the terms in those sentences mean. (Certain keywords in sentences, such as *and*, do have specific functions.)

So, for example, the sentence

The snark was a boojum.

logically entails

Something was a boojum.

Similarly, the sentences

My keys are in my coat pocket or on the fridge.
Nothing is in my coat pocket.

logically entail

My keys are on the fridge.

The fact that these symbols can be used in an uninterpreted way is what allows the connection with computation.

Is thinking logic?

So in the end, is thinking just logic? For anyone who has studied logic, this is not a very plausible notion.

Suppose somebody at a party says,

> *George is a bachelor.*

Here are some of the sentences that this logically entails:

> *Somebody is a bachelor.*
> *George is either a bachelor or a pig farmer.*
> *Not everyone is not a bachelor.*
> *It is not the case that both George and Henry are not bachelors.*

Sure enough, these sentences will all be true if the given sentence is; that is what logical entailment does. But they are so very, very boring!

If you found out at a party that George was a bachelor, it is almost guaranteed that we would not spend time going through logical entailments like these. You might think about *George* (whom you might already know) or about what it means to be a *bachelor*. Thinking seems to be so much richer than just dry logical entailments because thinking seems to depend on what the words in a sentence *mean*.

In fact, the view that thinking is logic may seem so far off the mark that instead of asking what is *wrong* with it, one might be tempted to ask what is *right* with it.

1.3.3 Using what is known: The web of belief

To get a glimmer of what could be right with it, one has to go back to the idea of thinking: bringing knowledge to bear on an activity. In reaching the conclusions about George the bachelor, no other knowledge was used. The search for logical entailments is not from that one sentence alone but rather from that sentence together with everything else that is already known.

Figure 1.5 shows some of the relevant facts that may be known about George the bachelor. In this collection of sentences, the terms *George*, *bachelor*, *man*, and so on, appear in many places, linking the sentences together in the same way that the term *frimble* did in the sentences of the earlier example.

It is sometimes helpful to visualize the sentences as forming a kind of *network*, with nodes for each of the terms and links between them according to the sentences in which they appear. The network might look something like the one in figure 1.6. We may not know who George is, for example, but we can see that the node for *George* is connected to the node for *Mary* by way of the node for *son*. We may not know what

Figure 1.5. Some beliefs about George the bachelor

George was born in Boston, collects stamps.

George is the only son of Mary and Fred.

A son of someone is a child who is male.

A man is an adult male person.

A bachelor is a man who has never been married.

A (traditional) marriage is a contract between a man and a woman that is enacted by a wedding and dissolved by a divorce. While the contract is in effect, the man (called the husband) and the woman (called the wife) are said to be married.

A wedding is a ceremony where . . . bride . . . groom . . . bouquet . . .

and so on.

---

son is supposed to mean, but its node is connected to the nodes for *child* and *male*. Similarly, the *male* node is connected in a different way to the node for *man*. The node for *bachelor* is connected in a complex way to the node for *marriage* and from there, presumably, to *wedding* and *bride*. Although we may not know what any of these terms mean in isolation, the various sorts of links given by the sentences in the network provide a rich set of interdependencies among them.

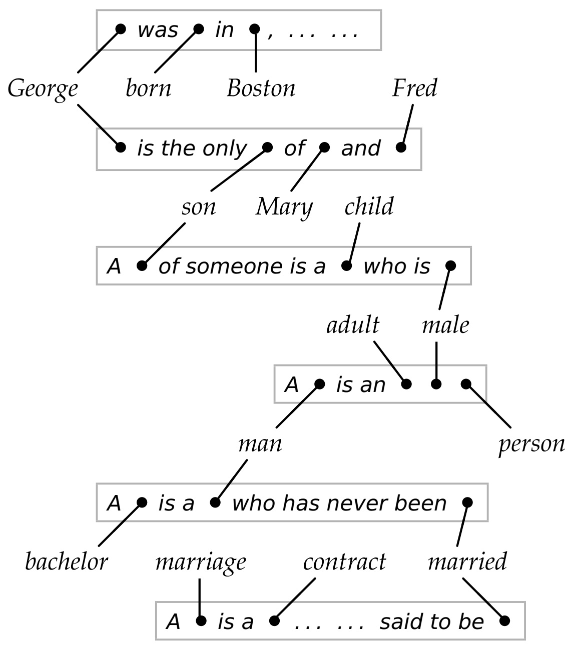
A network like this is sometimes called a *web of belief* to emphasize that the sentences do not stand alone but link to many others by virtue of the terms they use. The job of logical entailment is to crawl over this web looking for connections among the nodes, sensitive to the different types of links along the way. In figure 1.6 there is a certain path from *George* to *male*, for example, that can lead to the conclusion that George is male. If the fact that George is a bachelor is added to the web, a new set of pathways opens, including some connections from *George* to *marriage* that were not there before.

The logical entailments for the new sentence together with everything previously known gives some additional answers:

George has never been the groom at a wedding.
Mary has an unmarried son born in Boston.
No woman is the wife of any of Fred's children.

These are much more like the ordinary thoughts that people would think when learning that George was a bachelor. They are not exactly poignant, of course, but if some

Figure 1.6. Some beliefs as a network



additional facts were added about how parents hope their children end up happily married, one could go in that direction. (Or one might want to include facts about what a bachelor lifestyle is like and get additional entailments about George that would fit into a party setting.)

Observe that to get this richer set of conclusions, one does not need to know in advance what the symbols *George* and *bachelor* mean. What is needed, however, is a much richer collection of sentences over which to apply the rules of logic.

Knowledge bases

At this point, we have to be prepared to make a gigantic leap of the imagination:

> Imagine that we can draw conclusions from *millions* of such facts.

In other words, to make a plausible connection between thinking and computing, we have to imagine that we are considering the logical entailments of a potentially enormous collection of sentences, an entire web of belief. Such a collection is called a *knowledge base* (KB). The collection shown in figure 1.5 is just a very small sample.

So yes, there is a connection between thinking and logic, but it is misleading to think of it as Logic, the subject studied in philosophy. The way Logic is normally taught, one starts with a small set of premises and concentrates on ensuring that the conclusions from them are always correct:

> Socrates is a man.
> All men are mortal.
>
> Therefore, Socrates is mortal.

Thinking is very different. It starts with an enormous collection of premises (maybe millions of them) about a very wide array of subjects. There will be facts in the knowledge base about George and marriage, but also about barber shops, ferris wheels, Academy Awards, hate literature, and so on. The question then is, what are the logical entailments of *all* those sentences?

This leads to two hypotheses:

- Much of the subjective feeling of richness experienced in thinking might be explained as simple mechanical and logical operations, but applied to a very rich collection of sentences.
- To build computer systems with a number of desirable properties (versatile, flexible, extendible, easily maintained, and so on), one must
  - represent much of what the system needs to know as symbolic sentences of some sort, called its knowledge base;
  - perform processing over the knowledge base using the rules of logic to obtain new conclusions;
  - have the system act based on the conclusions it can derive.

  Systems built this way are called *knowledge-based systems*.

The big picture

In summary, thinking means bringing what one knows to bear on what one is doing. But how does this work? How do concrete, physical entities like people engage with something formless and abstract like knowledge? What is proposed in this chapter (via Leibniz) is that people engage with *symbolic representations* of that knowledge. In other words, knowledge is represented symbolically as a collection of sentences in a knowledge base, and then entailments of those sentences are computed as needed.

(Actually, there are good reasons to deviate somewhat from strict logical entailment for this. See chapter 3.)

So *computation over a knowledge base* is the direction pursued in this book, although it deals with only tiny knowledge bases. The next chapter studies a procedure that performs this computation for knowledge bases of a certain restricted form.

Want to read more?

This chapter introduced the connection between thinking and computation, an idea that is the subject matter of the rest of this book.

The starting point for this connection is the work of the philosopher Gottfried Leibniz. A good introduction to his thinking can be found in [59]. (His own work is scattered in the thousands of letters he wrote.) There is not much in terms of details, however, as Leibniz did not have the benefit of the modern notions of symbolic logic or computation. These came along in the 1930s, and it took until the 1950s, when John McCarthy, one of the founders of the field of artificial intelligence (AI) [11], became the first person to propose the approach followed in this book, representing what is known as a collection of sentences and computing their logical entailments [47].

An excellent place to begin to explore what psychologists have to say about human thinking is a book by Pinker [9] that is aimed at a broad audience. However, the whole topic of human thinking remains highly controversial, and even Pinker has his detractors [3]. Many find the notion of symbolic computation too limiting because it downplays the effect of the rest of the body on the thinking process [2]. Among the computationalists, many prefer symbolic structures other than sentences that can represent knowledge in a more concrete way [5]. There are also researchers who feel that a lot of thinking needs to be more pictorial or diagrammatic in nature [6].

The general notion of computation used here arose directly out of work on logic in the 1930s. This account of computation is due to Alan Turing, widely considered to be the father of computer science (John von Neumann is often called the father of electronic computers). In 1936, Turing proposed a formal definition of what is here called a procedure in terms of a very simple imaginary device: a *Turing machine* [12]. With this definition, one could now ask questions like whether it was even possible to compute a certain result. Turing was the first to prove that there was a *universal* Turing machine that could compute what any other one could, and remarkably enough, that some results could not be computed by *any* Turing machine.

Turing's carefully worked definition has withstood the test of time. Although different models of computation have been proposed (including a strikingly different one by Alonzo Church), they have all been shown to be equivalent to Turing's. Every computer built so far has been a special case of a Turing machine and subject to the restrictions proved by Turing. The claim that this will be true of *any* physical computer to be built is called the *Church-Turing thesis* [8].

Regarding thinking as computation, there is a superb though somewhat advanced philosophical book on this subject by Pylyshyn [10]. Much of the argument presented here about how actions are conditioned by what is known derives from this insightful book. (In the terminology he uses, those actions are said to be cognitively penetrable.)

Outside of philosophy, there is a subarea of AI called *knowledge representation and reasoning* that starts with the work of McCarthy and is concerned with the issue of representing knowledge in symbolic form and devising computational procedures to reason with it effectively. A graduate-level textbook on this area of research is by Brachman and Levesque [1]. The state of the art in this research area is reviewed in a (quite advanced) technical handbook [4].

Finally, although this book concentrates on small knowledge bases, the CYC project [7] is an example of a project whose goal is to build a large knowledge base along the lines discussed here.

Exercise

Consider the procedure PROCX in figure 1.7. It assumes there are other procedures elsewhere that will do some arithmetic (subtraction, multiplication, and less-than comparisons). It also assumes that one can *concatenate* strings of digits: $x\hat{}y$ means the string consisting of the digits in x followed by those in y. It works by repeatedly setting new values to u, v, *bot*, *top*, and *side* as the digits of the input are worked through in pairs from left to right.

Figure out what the procedure PROCX is doing. *Hint:* Trace its behavior when the input is `137641`.

Figure 1.7. A mystery procedure

```
Procedure PROCX:
    You are given a sequence of digits $x$ as input.
    You will return a sequence of digits as output.
  1. Group the digits in $x$ into pairs starting from the right. (If $x$ has an odd number
     of digits, the leftmost group will only have a single digit in it.)
  2. Start with $u$, $v$, $bot$, $top$, and $side$ all having an initial value of 0.
  3. Then, working your way from left to right on the groups in $x$,
     repeat the following:
      a. Set $bot$ to $(bot - u)$ ^(the next group from $x$).
      b. Set $side$ to $2 \times top$.
      c. Set $v$ to the largest single digit such that $v \times (side\hat{\ }v) \leq bot$.
      d. Set $u$ to $v \times (side\hat{\ }v)$.
      e. Set $top$ to $top\hat{\ }v$.
  4. The answer to return is the final value of $top$.
```