ERIC B. BAUM

WHAT IS THOUGHT?

What Is Thought?

Eric B. Baum

A Bradford Book
The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times New Roman on 3B2 by Asco Typesetters, Hong Kong, and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Baum, Eric B., 1957–
What is thought? / Eric B. Baum.
p. cm.
"A Bradford book."
Includes bibliographical references (p.) and index.
ISBN 0-262-02548-5 (hc. : alk. paper)
1. Philosophy of mind. 2. Cognitive science. 3. Thought and thinking. 4. Semantics (Philosophy) I. Title.
BD418.3.B38 2004
128′.2—dc22 2003059544

10 9 8 7 6 5 4 3 2

Contents

Acknowledgments

I want to thank and acknowledge all the scientists and scholars whose ideas and teachings have influenced my thoughts, but particularly to thank by name the many individuals who have read and commented on portions of the text, including Elise Baum, Stefi Baum, Gary Flake, Dan Gindikin, David Heckerman, Elliot Justin, Charles Markley, Lee Neuwirth, Chris O'Dea, Barak Pearlmutter, Herman Tull, several anonymous referees, and especially Peter Neuwirth. This book was greatly improved by their comments, but of course any errors that remain are my own. I also thank Small World Coffee for stimulation and a fine working environment.

1 Introduction

Over half a century ago, Erwin Schrödinger, the co-inventor of quantum mechanics, wrote a short book called *What Is Life?* (1944). He began as follows:

> How can the events *in space and time* which take place within the spatial boundaries of a living organism be accounted for by physics and chemistry?
>
> The preliminary answer which this little book will endeavor to expound and establish can be summarized as follows:
>
> The obvious inability of present-day physics and chemistry to account for such events is no reason at all for doubting that they can be accounted for by those sciences. (3; italics in original)

Schrödinger was writing ten years before the discovery of the structure of DNA by James Watson and Francis Crick, but the main thesis of his book was that genetic information was carried by a molecule, which he incorrectly thought was a protein. The reason why the physics and chemistry of Schrödinger's day could not understand life, he remarked, was that ordinary physics and chemistry arise from statistics, involving the interaction of vast numbers of atoms. Statistics assumes that the system will be found in a random configuration and thus will have properties characteristic of likely configurations. But life is the result of the evolution of genetic information, which has selected for very complex processes that by ordinary considerations would be enormously unlikely. Thus, most of the ideas, intuitions, and methods of classical physics are inappropriate for understanding life.

It was unusual for Schrödinger, a physicist not a biologist, to be writing on life. However, he believed that, short of an appeal to mysticism, life must be explainable at a fundamental level by physics and chemistry. Yet life seemed to violate the normal behavior of entropy, which is central to physics. The chemicals in the body continue a long cascade of intricately organized reactions for 70-plus years, contrary to usual statistics that expect organization to be dissipated, as a pot scatters into shards when it falls. The fact that life had ultimately to be explainable by physics, yet seemed inconsistent with physics, seemed like a fruitful avenue to explore.

Today I believe we are at a stage where it is productive to write a book called *What Is Thought?* asking how the computational events that take place within the spatial boundaries of your brain can be accounted for by computer science. I argue that the obvious inability of present-day computer science to account for such events is no reason at all for doubting that they can be accounted for by computer science. The situation we have is in fact parallel to that facing Schrödinger: the mind is complex because it is the outcome of evolution. Evolution has built thought processes that act unlike the standard algorithms understood by computer scientists. To understand the mind we need to understand these thought processes, and the evolutionary process that produced them, at a computational level.

My goal is to lay out a plausible picture of mind consistent with all we know, and in fact to lay out what I argue is the most straightforward, simplest picture of mind. I accept no mysticism; assume that we are just the result of mechanical processes explainable by physics; accept that we were created by evolution; accept some unproven hypotheses for which there is near-consensus among the computer science community on the basis of strong evidence (such as "the Church-Turing thesis" and "P $\neq$ NP," both of which I explain); and bring to bear whatever seem like hard results from a variety of fields, including molecular biology, linguistics, ethology, evolutionary psychology, neuroscience, and computational experimentation. As much as seems warranted, the discussion in this book follows what I perceive to be folk wisdom among computer scientists interested in cognition, but the attempt to pull ideas together and see whether a fully coherent picture emerges will lead us in directions that have been underexplored. I am confident that the picture herein will not convince all readers, because at the present level of knowledge it is impossible to offer a proof that the mind in fact works in such and such a way, especially on subjects like "the nature of experience," but I hope to offer a principled proposal to meet all concerns.

A nice feature of Schrödinger's book is that it was written at a level accessible and interesting to both scientists and scientifically literate laypeople. This was possible because Schrödinger was a great writer but also because the understanding of life in the literature was hazy at the big picture level and missing or wrong in all the details, so that what was important was to explain the essence of some big ideas. In the present task also, I think we are sufficiently far from understanding mind that the details of published work on the computational approach to intelligence would probably be irrelevant. My hope is to extract key ideas from the computational and other literatures, to fit them into a big picture, and to explain everything essential about that picture—indeed everything I think I know about the mind—to a mixed audience. Of course, once a suitable picture exists, it would be important to fill in the details with as much mathematical rigor as possible, but I am treating that largely as a matter for future work. I have sketched for the general reader the intuitions behind mathematical arguments where they are crucial. At some places I could have filled in more details but chose not to, so as not to lose the train of thought. But there are many places where the mathematics remains to be worked out.

1.1 Meaning, Understanding, and Thought

There is an underlying theme to almost everything this book says, which can be expressed in a single summary sentence. Here it is.

Semantics is equivalent to capturing and exploiting the compact structure of the world, and thought is all about semantics.

Let's focus on the first half of this summary sentence first. The book explains in some detail why computer scientists are confident that thought, and for that matter life, arises from the execution of a computer program. The execution of a computer program is always equivalent to pure syntax—the juggling of 1s and 0s according to simple rules. The key question, which has been posed primarily by philosophers, is how syntax comes to correspond to semantics, or real meaning in the world.

The answer this book suggests is that semantics arises from the principle, roughly speaking, that a sufficiently compact program explaining and exploiting a complex world essentially captures reality. The point is that the only way one can find an extremely short computer program that makes a huge number of decisions correctly in a vast and complex world is if the world actually has a compact underlying structure and the program essentially captures that structure.

Physics is a good analogy here. Physicists have written down a short list of laws that allow them to predict the outcomes of many experiments. Thus, they believe that the world really does have an underlying simplicity described by these simple laws. I argue here that mind is a complex but still compact program that captures and exploits the underlying compact structure of the world.

I refer to this principle as *Occam's razor*. Simpler versions of Occam's razor date back to at least William of Occam's fourteenth-century dictum *Pluralitas non est ponenda sine necessitate*, "Entities should not be multiplied unnecessarily." Occam's razor has been formalized over the last few decades by computer scientists to describe the training of compact programs that predict many observations, and it is this work that first spurred my investigations. But the reason for using the term *exploiting* in the summary sentence is that this book discusses Occam's razor in a somewhat more general context. Finding a compact underlying structure and finding algorithms to exploit it are two separate hard computational problems. The mind exploits its understanding of the world in order to reason. The programs trained by computer scientists typically output a single classification, positive or negative, to a single instance of some problem. But mind typically produces a computer program capable of behaving, that is, of doing elaborate computation leading to an appropriate series of actions, and in fact of behaving in the face of a whole class of problems. I argue that this additional power is a result of the evolutionary programming that led to mind and implies, or rather is equivalent to, understanding the semantics.

If we look for a compact description underlying mind, one stands out like Venus on a moonless night. With its myriad of neurons and connections the brain is huge, but its DNA program is much smaller—at first glance quite surprisingly small when

one analyzes its function. So I argue that, counter to some of the folk wisdom in the computational and cognitive science communities, mind is essentially inherent in the DNA in some detail. There is no doubt that learning during life is important, but because the DNA is the compact program that is the core, learning during life is essentially guided and programmed by the DNA—a phenomenon called inductive bias. We learn extremely rapidly, much more rapidly than computer scientists have been able to explain, because our learning is entirely based on and guided by semantics. The reason we learn so fast, the reason our learning is guided by semantics, is that the compact DNA code has already extracted the semantics and constrains our reasoning and learning to deal only with meaningful quantities.

The second half of the summary sentence is that mind is all about semantics. This is true on many levels. What distinguishes mind from artificial intelligence programs is that mind understands—it exploits semantics, or meaning, for computation. The way the mind reasons about things, which is different from the way human-written computer programs do, is by manipulating semantic chunks and exploiting the compact structure.

How does mind solve problems as fast as it does, and how did evolution solve the problem of producing us as fast as it did? Evolution had 4 billion years and vast resources, but computer science tells us to expect that these problems are so hard to solve (because there are so many possible answers that must be searched through) that even that amount of time and those resources should not have been enough. I discuss several answers, but the main one is that mind is so fast because it exploits semantics (as evolution did). Evolution discovered semantically meaningful chunks such as the subroutine "build an eye here" and then was able to reason how to construct new creatures by manipulating these semantically meaningful chunks. Mind understands the world in terms of meaningful concepts and is able to reason so fast because it only searches through meaningful possibilities.

The flip side of dealing with an apparently very complex world that however has a very compact underlying structure is that the complexities are often highly constrained, indeed overconstrained. Once the semantics are understood, one can often reason straightforwardly by exploiting the constraints. While there are myriad possibilities, only one or a few make sense.

When people write computer programs, they organize them into small, mostly self-contained units called subroutines or modules, each addressing some particular subcomputation. Evolving a very compact code that deals with the world leads to a code that is highly modular. By producing a program with many subroutines corresponding to meaningful concepts, evolution produced a program that is compact because it reuses these subroutines in multiple different contexts. This is why, I believe, thought

is so often based on analogy and metaphor—mind invokes one of these subroutines to understand a new context.

The brain does vast computations of which we are unaware in order to compute semantically meaningful quantities. What reaches our awareness is only the outcome of these processes—meaningful quantities. Mind is an evolved program that exploits the compact structure of the world to make decisions advancing the interest of the genes. Once one looks at it in these terms, it is straightforward to explain the qualitative nature of experience, the meaning of self, the nature of awareness, free will, and all that. Once one makes the ansatz[1] that every thought is simply the execution of computer code, and understands how that code is evolved to deal with semantics, a self-consistent, compact, and meaningful picture of consciousness and soul will follow as naturally as thoughts follow from the constraints of meaning.

In short, we're going to go back and forth between two closely related concepts and one process: compactness, meaning, and evolutionary programming. This book proposes that meaning arises from evolving a very compact program and that understanding is equivalent to exploiting the compact structure of the world. Thought and learning as well as the evolution of this program are as fast as they are because they exploit meaning and understanding. Awareness is awareness of meaningful quantities. The structure and nature of thought as well as consciousness naturally arise from the dynamics of evolution of programs that exploit the compact structure of the world.

1.2 A Road Map

The previous section painted some conclusions with a very broad brush. The rest of this chapter surveys the paths that led me to these conclusions, which I detail in subsequent chapters to justify them and make their meaning more explicit and concrete.

Chapter 2 begins by describing what an algorithm is, what a program is, and hence why computer scientists are so confident that the mind is equivalent to a computer program. To this end, it reviews Alan Turing's 1936 construction of the Turing machine, the intellectual model on which the computer is based. Turing addressed the question, What is an algorithm? At the time, before the invention of electronic computers, this question was much less settled than it is now. His approach was to analyze the process of thought, breaking it down into simple steps in a very general way. He asked, What is the most general thing that the mind could possibly be doing, and how can I analyze that into small, simple pieces? Since he could capture the most general possible thing in a computer program, he was able to show that

last 20 years or so point to the answers to these challenges and thus elucidate the nature of the program of mind and of how it came to exist.

The picture presented here is that mind relies fundamentally on Occam's razor. Occam's razor is the well-known and intuitive prescription that, given any set of facts, the simplest explanation is the best. Occam's razor underlies all of science. It is, for example, the way in which physicists come to their small collection of simple laws that fundamentally explain all physical phenomena, how chemists arrive at the periodic table, why biologists believe in heredity. Newton's laws, for example, are simple in the sense that they can be written down on a single page, yet they explain a vast number of physical experiments and phenomena.

Occam's razor further underlies all of human reasoning. It is why, for example, jurors do not reach for some Rube Goldberg hypothesis that is consistent with any evidence that could possibly be presented and also exonerates the accused. An explanation that is too complex is judged to be "beyond reasonable doubt."

This book claims that Occam's razor (as generalized and extended) is the basis of mind itself.

The examples given of Occam's razor in scientific and ordinary reasoning are intuitively appealing, but examined further the intuition does not make clear exactly in what sense an explanation is simple nor why Occam's razor should hold. Computer scientists have formalized Occam's razor in several related ways, three of which are discussed in chapter 4. Roughly speaking, for computer scientists an explanation is a computer program, and the simplest explanation is just the shortest computer program.

The simplest of the three formal versions of Occam's razor is just a sophisticated version of curve fitting. If you have a large collection of appropriately gathered data points and you succeed in finding a straight line that fits them well, you can be pretty confident that the line has really captured some truth about the world. Its slope is not just a symbol in the curve fitter; rather, it corresponds to reality. If you go out and gather more data points, you expect that they also will lie on the line—the line makes predictions that generally come true in the world. This is why statisticians, social scientists, salespeople, and politicians all like to hold up charts showing straight lines fitting data.

Roughly speaking, the reason a line that agrees with data is believed to have predictive power is that there are very few ways to draw lines. Each data point that the line is required to agree with is another constraint on the line. If the line agrees with a lot of points, that's unlikely to be an accident.

Computer scientists sometimes study a model of brain circuits called neural networks. These contain a collection of objects that represent neurons (the objects are really just simple mathematical operators). The "neurons" are wired together, with

the output of some neurons feeding into the input of others. So all a neuron is (in this model) is something that looks at numerical inputs and produces a numerical output according to a simple mathematical rule. Associated with the connections between neurons are weights that determine how strongly the neurons are connected. Some numbers are fed into the inputs of the whole network. Some neurons compute their outputs and pass them to other neurons, which then compute outputs, until finally the whole net produces its output.

Such a neural network is trained by adjusting the connection weights so that the net learns to do the right thing on many training examples. You might, for example, show it pictures of faces, some smiling and some frowning. The neurons would input a numerical representation of the pictures and produce output numbers. You would adjust the weights so that whenever you show the network a smiling face from a set of training pictures, it outputs a 1, which we take to indicate "smiling," and similarly when you show it a frowning face from the set, it outputs a 0, which we take to indicate "frowning." Now, the interesting thing is that once a network has been trained on a sufficiently large collection of examples—many more than there are weights in the network[2]—it learns to generalize and will correctly distinguish smiling from frowning faces in most pictures it has never seen before. To some limited extent, it "understands" enough to distinguish smiling and frowning.

This is essentially just a more complex example of curve fitting. Neural networks are just a complex class of curve, that is, they are mathematical functions parameterized by weights, and once one has constrained them enough with examples, they are forced, so to speak, to represent the underlying structure in the process classifying the data. That is why they then get the right answers on new examples they have not seen before.

More generally, there are only so many small computer programs of a given type. Again, each data point with which you require a small computer program to agree is another constraint on that program. If the number of constraints vastly outweighs the flexibility in writing the program, you would naively not expect to be able to find a program agreeing with the data. When you do, it is in a sense because the syntax of the program fundamentally reflects the process producing the data. Finding such a compact program thus demonstrates that the data are actually produced by some simple process and also that the program you found reflects the simple process in some way. So, this is my first answer to how and why the syntax of the computer program of mind corresponds to reality in the world: it is based on a program so compact it has no choice but to do so.

The claim is that if one somehow finds a sufficiently compact program agreeing with sufficiently many data points, the mere fact of the existence of the program

more or less guarantees that the data are in fact generated by a simple process and, further, that the syntax of the program reflects that process. Another interesting question is how such a compact program agreeing with the data could be found. Computer scientists train neural networks by a slow process closely related to evolution in that they make a long series of small changes, each of which improves the agreement of the net with the data a little bit. In this way, the whole net slowly settles into a configuration where its syntax reflects the process creating the data. Each weight becomes tuned so that it cooperates with the others in a representation of the world. Such a training process is computationally intensive. The mind was created by an even more computationally intensive evolutionary process over 4 billion years, which yielded a vastly more impressive understanding of the world than any artificial neural network can. The characteristics and effectiveness of such hillclimbing procedures are discussed in chapter 5.

From this point of view one can easily understand why the AI programs critiqued by Dreyfus and others were so clueless and why computer programs still show no sign of "understanding." The answer is that the creation of these programs involved essentially no compression at all. One can readily get a computer program to parrot answers to a fixed set of questions by simply programming in the list of answers. Tell the computer if it receives question A, give answer *a*, and if it receives question B, give answer *b*. But a list of answers is not compressed; it is a program as long as the number of answers it can give. In contrast to the process of training a neural net, this storing of answers requires extremely little computation. And it is not at all constrained: one can program in any list of answers one chooses. A parrot may impress for a few seconds by speaking some complicated phrase, but it has no understanding of what it is saying. A computer program that is not compressed is not much better than a parrot and will be tripped up when it gets to a novel question.

Of course, many AI programs are more sophisticated than simple lists, but even so, they have not been produced by a computationally intensive optimization process. Rather, they have simply been written down by people. People are smart, but they don't seem to have the capability of doing enough hard optimization to produce truly compressed programs. Constructing extremely compressed programs that extract the structure of complex data is a very hard optimization problem requiring extensive computation such as is done in training neural networks or by evolution. We are no match for our computers at solving hard optimization problems and very far indeed from the computational capabilities that evolutionary history brought to bear on the problem. Human-created programs, for example, the AI programs called expert systems, thus do not reflect Occam's razor in the way human thought does, and so do not display understanding.

Another problem with standard AI and neural net approaches is that they typically throw out much of the structure of the world before they start. To understand language, for example, one must understand how language is about the world, but the academic divisions within computer science treat language as divorced from vision, as divorced from planning, as divorced from the world generally. Language is usually treated as pure syntax before one begins, so there is little hope of recovering the semantics. Similarly, planning is often treated in the scientific literature as independent of the specific knowledge about a particular domain for which one wants to plan, and in particular as independent of human knowledge about topology and geometry. By dividing up the world into academic subdisciplines such as language, planning, and vision, computer scientists are throwing out the relationships that the compressed program of the human mind exploits. I believe that any approach aiming to achieve understanding must be much more holistic and must evolve a compact computer code that compresses much experience about the world.

This picture of compression as understanding also readily explains why the guts of heavily compressed programs, for example, the internal representations of some trained neural networks, are inherently hard to understand. The argument, as further elaborated in chapter 6, is that understanding comes from having a very compressed description. But an understanding of the guts of a compressed program would then be an even more compressed description. Such will not generally exist.

Now, it is a huge step to go from simple curves, or even complex curves such as neural networks, to the kind of thought, understanding, and consciousness that we observe in human beings. In fact, neither neural networks nor any other function class that only classifies presented examples can model the mind well. Rather, we need to talk in terms of powerful computer programs. The program of mind does not simply represent or mirror the world; rather, it knows how to do complex computations about the world. It can do things like output algorithms to address whole new problem classes it has never seen before. It does not just mirror the compact description of the world; it exploits this structure to plan and to compute.

Finding a compact description of the world is already a hard task. But given a compact description of some computational problem, computing how to solve the problem is a separate impressive feat. Nonetheless, this exploitation of structure has arisen through program evolution. Evolution has produced minds that not only mirror the structure of the world but do amazing calculations about it.

What does it mean to exploit compact structure, and how can programs evolve to exploit compact structure? A first comment is that the mind does not arise from the kind of input-output classification training discussed, for example, in the neural network literature but rather from a process more like what the computer science

literature calls *reinforcement learning*. In reinforcement learning a robot interacts with its environment and is rewarded when it behaves correctly. Thus, the robot must learn to sense the appropriate features of the world, to compute what to do, and then to act. In a complex world it cannot simply sense and react through a simple function. It is rewarded only when it correctly decides what to do, so it is trained directly not only to reflect structure but to exploit it. I argue that our ancestors were trained by evolution to sense and think the right thoughts and take the right actions. We are not just reactive systems but have learned to do the right computations as well.

Three approaches to reinforcement learning are discussed in chapter 7. The first approach, which is the most widely studied in the literature, essentially consists of memorizing what to do in every state; it serves in this book as another example of a program that does *not* shed much light on the mind. Such memorization does not involve any compression and hence does not invoke Occam's razor. Memorization of this type can work only in very simple regimes because memory by itself can never tell you what to do for situations you have never encountered before. Memorizing and understanding are at opposite extremes. My point of view throughout is that it is Occam's razor—the finding of very compressed representations—that leads to understanding.

The next approach is using neural nets to do compression in reinforcement learning and to achieve generalization to environments never before seen. This builds on the formal Occam's razor results, and hence, for simple enough classes of nets, is relatively well understood from a formal perspective and does succeed empirically to a certain extent in learning and generalization. However, I critique this well-studied approach, arguing that in many environments it is conceptually flawed, that it will never get sufficient compression in complex environments, and that it will never handle reflective thought because the simple kinds of neural nets that computer scientists know how to train are essentially reactive. The neural representations people typically study are just not powerful enough to represent the kind of processes that are going on in the mind.

It seems very likely that the kinds of neural circuits studied provide a good model of certain brain functions such as early vision, but it seems unlikely that they are a good model of higher mental processes. Although it is true that more general classes of neural nets could simulate the actual neural circuitry of the brain, it does not follow that this is a fruitful way to talk about thought. To talk about thought fruitfully, at least along the line of attack in this book, one must be able to discuss the compactness in the algorithm. The compactness in the program of mind lies in the DNA and in the process by which the neural circuitry is constructed. The neural circuitry

To develop more intuition about what it means to exploit structure for computation as well as about program evolution and how an evolved program might come to understand, I discuss the solution by people, by AI programs, and by evolutionary programming of some standard benchmark problems, such as Rubik's cube and the simple stacking of children's blocks. Such problems are of course much more limited than a general analysis of concepts, but for this reason they can be discussed in concrete terms to help us get a firmer grasp on these concepts.

Evolving a program to understand is a hard problem, and in order to make progress on this I had to take some inspiration from observations regarding the qualitative nature of the program of mind. So before talking about structure exploitation and program evolution further, I pause to make some remarks regarding the nature of the program of mind.

For a large number of reasons, compact code for dealing with complex phenomena invariably has a modular structure. One doesn't sit down and write a program as one integral whole. Rather, one divides the problem up into subproblems and writes *subroutines*, sometimes called *objects*, *modules*, or *functions*, to solve the pieces. The whole program then is a complex society, composed of multiple modules that refer to each other, with each module charged with some particular task, achieving a division of labor.

It is impossible to construct code, or to evolve it, or to debug it, unless it is modular. If every part of the program interacts with every other part, then any change breaks so many things that it is impossible to make progress in constructing the code. On the other hand, if the program can be constructed module by module, it may be relatively straightforward to continue to improve it and add new functionality because at each step it is necessary to modify or produce only a relatively well-contained module. This can be expected to be true not only for human authors but for evolution as well.

Modularity is a powerful way to get compact code that exploits structure. If, as I walk, a module in my mind positioning my ankle is independent of a module contemplating my future plans, then each of these modules can be compact. In other words, code dealing with the world will be modular because the world is modular, and exploiting the modularity gives a compact program that can calculate how to interact with the world. By having modularity, one gains combinatorially in dealing with the myriad possible states of the world. A relatively small number of modules can interact to represent and deal with an exponentially huge number of world states. Moreover, the code is compact to the extent that modules can be reused many times and preferably in many contexts. A large part of the reason that standard neural

networks are often an inappropriate model for mind is that they don't readily lend themselves to describing such a modular structure.

Researchers in many different disciplines—computer science, cognitive science, neuroscience, evolutionary psychology, and others—have reached a separate consensus within their several fields that the mind is modular. The different viewpoints give interesting aspects of what the modules look like and how they interact and are coordinated. A number of these different pictures are examined in chapter 9—cognitive deficits coming from localized damage, psychophysical experiments that indicate the existence of a dedicated module for reasoning about social obligations, and so on. But as always, I take the point of view that the mind is a program. Hence these modules represent modules in the program. The upshot is that you have, I believe, modules representing different concepts, representing different objects, representing how the objects act, what their properties are, how you deal with them, for example, how you go about lifting a cup to your lips and what to expect when it gets there. These modules call submodules that are reused in many different ways. Many of the submodules that you use for dealing with cups are also used for dealing with forks or cars or computers.

I want to mention here two additional points about modules. The first is that the metaphoric structure of language gives a window on the modular structure of the mind's code. Lakoff and Johnson (1980) have pointed out that our language is deeply metaphoric. For example, time is money, in that we spend time, borrow time, waste time, lose time, save time, and so on. Such built-in metaphors are incredibly pervasive in language. I believe this provides a picture of code reuse. Time is money because we have a module for valuable resource management that we reuse to deal with time. From this point of view we have many modules that call one another in complex ways. This massive code reuse yields a very compact program that understands and deals with the world.

A second point is that we can learn whole new modules. The main example, discussed in chapters 8 and 9, comes from computer science. I've said repeatedly that the mind exploits the compact structure of the world to solve problems rapidly. In fact, exploiting structure to solve problems rapidly is the central problem of computer science. The mind is in many ways better at this than computer scientists because the mind has access to many modules discovered over evolutionary time that exploit the structure of the world. Computer scientists have, however, developed a whole bag of tricks for exploiting structure. Examples of such tricks are methods called divide and conquer, recursion, branch-and-bound, dynamic programming, gradient descent, and many others. Each of these is a trick for exploiting structure

in problems. Using these tricks, computer scientists can program computers to solve much bigger problems than could be solved without them, problems that are vastly too hard for the unaided mind to solve. A typical research paper in computer science consists of applying one or more of these tricks to a novel problem. A breakthrough research paper finds a new trick.

I suggest that when you learn at university about one of these tricks, say, recursion, you are really building a module in your mind that knows how to search for recursive solutions to new problems. This is a fairly sophisticated module, calling a lot of previous modules you have in your mind, but it is pretty evidently distinct from what you knew before. It clearly is something that you explicitly learn. In fact, I hope readers who do not already know what recursion is will gain some idea of what it is from my description in chapter 8. If the mind is a program, which after all is the central premise of this book, then this recursion ability is presumably a new module added to that program.

I expect that a large part of the way our powerful minds have been formed is by accretion of new modules built on the existing structure, giving it new abilities. Each module may call previous ones. Constructing a program as complex as mind is an incredibly complex task, but it is much easier if it is done incrementally. Most of the discovery of such new tricks for exploiting structure has, I expect, happened over evolutionary time. But the example of these new modules shows that some of it happens in our lifetimes as well. Human thought is so powerful because we can discover new modules and pass them on. It only took one computer scientist to discover a trick like recursion, and he could add it to the human repertoire. I discuss in later paragraphs the relation between modules built by evolution and modules discovered or learned during life.

To recap the argument to this point, I have proposed that the mind is an evolutionary program. Because it has a very compact description, the syntax of the program corresponds to real semantics in the world. Because the world has structure, and because the program of mind has evolved to exploit that structure, it is able rapidly to compute and output algorithms for addressing problems in the world. Understanding comes from the compactness and the ability to exploit structure for computation. The program has a modular structure, with modules corresponding to concepts calling other such modules. I say concepts because they can be seen as having semantic meaning, which has arisen during the production of compact code capable of dealing with vast numbers of situations.

To illustrate what it means to exploit structure for computation, I discuss the solution of some standard AI benchmark problems, such as Rubik's cube and the

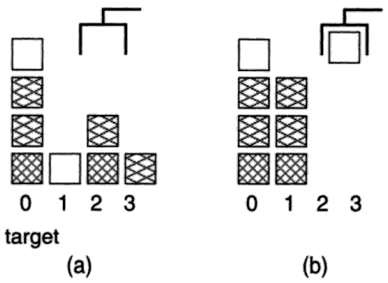

Figure 1.1
In block-stacking problems one is presented with four stacks of colored blocks. The last three stacks taken together contain the same number of blocks of each color as the first stack. The solver controls a hand that can lift blocks off the last three stacks and place them on the last three stacks but that cannot disturb the first stack. The hand can hold at most one block at a time. The goal is to pile the blocks in the last three stacks onto stack 1 in the same order as the blocks are presented in stack 0. The picture shows a particular instance of this problem with four blocks and three colors: *a*, the initial state; *b*, the position just before solution. When the hand drops the white block on stack 1, the instance will be solved.

simple stacking of children's blocks (figure 1.1). Such problems are more limited than a general analysis of concepts, but for this reason they can be discussed in more concrete terms.

Block stacking as presented in this book is not a problem but rather a class of problems. The class contains infinitely many problems because infinitely many blocks can be arranged in infinitely many ways, but the class has a very compact overall description, namely, a paragraph of text that describes what the block-stacking problems all look like (see the caption of figure 1.1). Block-stacking problems are thus a little like the integers: there are an infinite number of them, but a short description shows they have much structure in common that can be exploited to do computations of various sorts.

A person looking at these problems can in short order produce an algorithm capable of solving arbitrary block-stacking problems rapidly, by exploiting the structure of the problem. That is, you can describe an algorithm that works for problems having this particular short description, just as you can rapidly tell whether any long number is divisible by 2. Think about how you would solve any problem in this class, if you haven't already.

By contrast, standard AI planning approaches don't exploit the structure of the problem at all; instead, they do a vast brute-force search. I mentioned before that the planning literature more or less abstracts the planning problem to the point of planning without using special knowledge about the particular domain, such as the domain of stacking blocks. Instead, standard planning algorithms simply search over

all possible configurations, in this case, all possible block stackings, looking for one that works. They thus treat the problem as simply a huge list of all possible action sequences, ignoring the fact that the problem domain has a short description, an exploitable structure. Since there are a vast number of possible block stackings, this approach fails for any problem of interesting size.

You, a human being, bring to bear understanding of topology, geometry, and goals. You had, I expect, modules in your mind that know about these concepts before I presented you with these block-stacking problems. So, you don't search at all over block stackings; you simply output an effective procedure that solves the problem. If you search at all, it is over different approaches, different algorithms. You are thus working in an entirely different space than the AI approaches, and on a very different problem.

A trained computer scientist has yet another module in her mind: a module for recursion. This allows her to construct a yet more efficient solution to the problem. That is, this module exploits the structure of these block-stacking problems to output an algorithm that solves them very rapidly.

Chapter 10 discusses computational experiments in which programs are evolved, starting from pure random computer code, to solve a few well-known problems such as Rubik's cube (which is only partially solved) and the class of block-stacking problems.

Evolving a compact program is extremely difficult for at least two reasons. First, the space of possible programs is enormous and thus hard to search. Second, small changes in a program, even single-line changes, can completely disrupt its behavior. This makes it hard to figure out how to improve a candidate solution. It leads evolutionary approaches to get stuck in local optima where any small change doesn't improve performance and where one has to somehow discover a larger change in order to progress. In fact, it is a fundamental mystery how evolution has succeeded in evolving the program for building us, even given the 4 billion years it has been working. In order to evolve these compact programs using only a few days on a desktop computer, my colleagues and I employed the following ideas.

A powerful way to exploit structure and produce a compact program is to produce a modular program that exploits natural modular structure in the world. Ideally, small interacting modules can be found that can be reused for different problems, contributing to compactness. Moreover, if the problem can be factored, that is, if a product structure can be identified, one can hope to search only for modules rather than having to find the whole program at once, which makes the search much smaller. Our picture of the mind is of a complex adaptive system composed of modules that interact in complex ways, forming a very compact program in total.

the problem. But the space of possible algorithms is so immense that it is amazing you can do this rapidly.

I do not offer a definitive answer to this puzzle but discuss a number of interesting possibilities. Our evolutionary computer experiments did succeed in outputting some interesting algorithms, albeit in a relatively restricted example. Computer scientists have typically thrown out the useful structure that the mind employs before formulating hard questions. Computer scientists apply their understanding to real-world problems, extracting already compressed problems with too little remaining structure to be rapidly solved. For example, the fact that subproblems of intelligence are hard is not in itself alarming but may simply arise because the academic division of problems into subproblems throws out the useful structure.

But the main suggestion I make, in keeping with the overall thrust of this book, is that the overconstrained nature of the world again comes to the rescue. Occam's razor says that when there are a lot of data to explain, and thus many constraints on a compact solution, the compact solution must reflect the underlying simple structure of the world. Something similar turns out to happen in at least some instances of NP-hard problems. When there are many constraints on the solution to a problem, there is usually only one way to extend a partial solution. Then you can extend it rapidly. I believe this may describe a critical and common feature of thought. Our understanding of the world is so compact and thus so constrained, that only one or a few possible ways exist to extend it that make sense at any given time. Thus, thought flows forward, each step following more or less inexorably from the last. We can understand speech because, applying all the constraints that come from our understanding of the world and from grammar, only one or a few ways of understanding each sentence make sense. We can play chess because, given our understanding of the position, only a few lines of play must be searched. These human thought processes are in stark contrast to standard computer science methods for analyzing speech or playing chess, which are not based on such a compact understanding and thus must search a vast number of possible extensions.

To state this in a slightly different way, human thought is fast because we search only possibilities that make sense. That is, our thought is organized to search semantically meaningful possibilities only, as units. Semantics arises because the world is highly overconstrained, and by finding code that knows how to deal with the world but is extremely compact, evolution has captured and learned to exploit those constraints.

Evolution built creatures that had to do the right thing and do it fast. It built from simple behaviors toward more complex computations but at each step stuck to a domain where each step followed from the last without much search. We are built

with an understanding based on an enormously complex collection of modules computing different concepts, and we evolved to coordinate their actions purposefully. Many of these modules predate human beings: we are built on concepts that have been useful to understand the world for eons just as we are built out of enzymes that have been useful for eons. The structure of our program is so constrained by the world and so similar from person to person that we can understand each other and even communicate whole new concepts. Evolution has discovered these modules one step at a time, at each step figuring out how to add some new, useful concept. And people have added to the store of program modules over time, at each step discovering a new module by making a small enough step so that it can be discovered without impossibly hard search. The reason we are able to discover and add new useful modules to the program is perhaps that the understanding of the world we inherited from evolution already contained so many useful modules, interacting in such a coordinated fashion, that we are quite constrained in how we can add new modules, new thoughts.

Evolution itself has learned to search in semantically meaningful directions, for example, using computational units that have acquired semantics. Evolution long ago discovered genetic regulatory circuits coding for development that involve a small set of toolkit genes, such as Hox genes. These genes came to have semantically sensible meanings, so that, for example, by expressing one specific gene on a fly's wing it is possible to grow a well-formed eye there. Evolution then proceeded to experiment with new body designs by building programs out of such semantically meaningful modules. Evolution is manipulating a highly compressed representation of the world, and thus in a sense it too is applying understanding to the process of discovery. I suggest that this in large measure is how evolution has produced such amazingly powerful programs in only 4 billion years.

Understanding the evolution of compact programs is critical to understanding mind because it is evident as a matter of historical record that a process of training a compact program did in fact generate our minds. The compact program was encoded as DNA, and the training process was evolution. This process is analogous to the reinforcement learning model by which we trained our artificial economy. Evolution trained on data for 4 billion years, involving (as I estimate) perhaps 10^{35} or more separate learning trials, and in a sense distilled all this learning into a compact expression in the DNA. The DNA program is quite compact compared to this massive training.

A bit is the minimal unit of information, the amount communicated by a single symbol that can take the value 0 or 1, in other words, the amount of information in a single choice between two alternatives. A byte is a word of eight bits and thus can

specify a choice among 2^8, or 256, alternatives. I estimate that the DNA program has effective information content of roughly 10 million bytes. To put this in perspective, this book is a string of text (omitting pictures) long enough to allow specification of about 1 million bytes. Thus, the 10 megabyte estimate for the information in the DNA is approximately equivalent to what could be specified in ten volumes of text the size of this one. For all its massive capabilities and extensive training, the effective content of the DNA program is a fraction of the size of the source code for programs like Microsoft Office. But the computing machine it specifies (for instance, you) looks like it understands the world and can be expected to act as a human being would in new situations.

Schrödinger (1944) considered the question of the nature of mind but felt it to be unsolvable. He wrote,

> [The fact that life is produced by simple mechanical interactions of proteins reflecting genetically stored information] is a marvel—than which only one is greater; one that, if intimately connected with it, yet lies on a different plane. I mean the fact that we, whose total being is entirely based on a marvelous interplay of this very kind, yet possess the power of acquiring considerable knowledge about it. I think it possible that knowledge may advance to little short of a complete understanding—of the first marvel [life]. The second [mind] may well be beyond human understanding. (31)

Yet I am proposing that the analogy to Schrödinger's explanation of life is exact. The answer to the mystery of life and the answer to the mystery of mind (thought) are one and the same. It is given by the information flow, which in each case is provided (largely) by the genome. In the case relevant here, understanding thought, the genome encodes a compact expression that gives rise to understanding in the mind. This genome is quite a compact expression, which grows out (interacting with the world) into an immense flowering—the mind—much as the genome grows out (interacting with the world) into the body.

Since Schrödinger's day, our understanding of life has improved considerably. We have unraveled the structure of DNA and are beginning to understand the complex series of reactions, regulatory networks, and chemical cascades by which it goes through the algorithmic process that leads to the body. Few scientists today would suggest that there is anything mystical, anything not ultimately explainable by science, in life. It is a goal of this book to further the process of understanding the algorithmic process that leads to the mind.

The preliminary sequencing of the human genome has produced a surprise: human beings are now thought to have only 30,000 or so genes, perhaps one third as many as had been once expected. The 100,000 previously predicted had already been thought to be a small number. Many creatures seemingly less impressive than ourselves have

as many genes as we do. Yet, from the point of view of this book, the low number of genes making up the human genome should not be surprising. Our focus is on compression inducing Occam's razor. A more impressive intellect can result from a more compressed genome, not necessarily just a longer one.

Many computer and cognitive scientists, particularly neural net practitioners, believe that the key factor in producing the mind is learning during life, not structure imparted in our DNA. They argue that we are born in a state that is largely *tabula rasa*, knowing nothing but possessing a general learning algorithm capable of learning about the world (maybe any conceivable world). The computational learning literature is largely oblivious to the interaction of learning during evolution and learning during life, and many in the field find the notion of special-purpose modules built into the mind by evolution to be extremely controversial.

Nonetheless, in chapter 12, I discuss how modern computational learning theory has shown that inductive bias, that is, the built-in predisposition to learn one thing rather than another, is crucial in being able to learn in complex environments. I have emphasized from the beginning that Occam's razor—finding a short explanation—is crucial in learning. But finding a short explanation involves first fixing on a language for describing explanations (a way of describing programs) and then being able to seek an explanation rapidly, that is, a learning algorithm. These things constitute inductive bias. Without these, you cannot learn. Once you have specified a language and a learning algorithm, however, what you can learn is not fully general—you will learn some things much more readily than others. And the explanation you will come to is more or less predetermined. The same algorithm and language will repeatably lead to a similar explanation when confronted with the same world. Your DNA can reliably be counted on to produce something with a mind rather like yours in the important ways if it is executed in contact with any roughly similar environment. The guts of learning during life are thus in the DNA.

There are a number of arguments to make clear the importance of the DNA code in predisposing learning. Some of these are the following. First, the DNA is where the compactness is: the DNA code is compact, but the brain is not. For example, the brain has at least 10 million times as many neural connections as there are bits of information in the DNA code. So, if one believes in Occam's razor, if one believes that it is the compact specification that leads to the mind's power, one should focus on the DNA. Second, finding the DNA is where almost all the computation has gone. Learning is a computationally hard problem, requiring vast computational resources. Vast computation has gone into evolving the genome, and only a relatively small amount goes into learning during life, which we mostly do in real time. We are predisposed to learn so rapidly that there is no time for computation; how we

fit new facts into our improving program is so constrained by what we already know and by built-in algorithms that it is fast and largely predetermined. Third, if we examine the learning abilities of various creatures, we see numerous examples of explicit inductive bias: animals are predisposed to learn certain types of things. They are not general-purpose learners at all. People, too, are predisposed to learn certain types of things. Chapter 12 describes a case study providing compelling evidence that we are wired to learn grammar rapidly.

Fourth, the nature of evolutionary computing is such that we could not have helped but evolve to be predisposed to learn. We often think about development (the growth of an embryo into an adult) as distinct from learning, but actually development takes place in contact with the world and must generalize over variations in its environment in a way quite analogous to how learning must generalize flexibly to unseen data. Development of the brain takes place in contact with sensory input, and the nature of evolution is such that the brain naturally comes to depend on this sensory input to develop correctly. The DNA codes for a program that is evaluated in the presence of an environment, resulting in a human being. In evolution the DNA is tweaked, the resulting creature develops in contact with the environment, and successful creatures propagate. This process leads to DNA coding for development, in contact with the environment, of the right structure and the right behavior.

This same evolutionary process, however, has led to a large expansion of the complexity of program execution through use of external storage in the form of culture. Learning the right things during life, that is, having an algorithm for development that interacts with the environment and develops the right behavior, is a crucial part of behaving the right way and thus greatly affects evolutionary fitness. Once parents became a part of the environment in which the child developed, the development process naturally evolved (as the DNA was tweaked and what worked was kept) to exploit passage of knowledge (programs) from parent to child. Culture came to interact with development and to affect the evolution of the genome. Such an effect began long before there were human beings, but it became more important once the development of language permitted them to pass on much more programming.

This book argues at some length that the learning and thought we do during life are heavily biased in that they are built on top of semantically meaningful modules that evolution long ago wired into the DNA. But human beings have built a major structure on top of these modules, much as mathematicians have discovered many theorems of number theory by working out the implications of a small set of axioms. The theorems are implied by the axioms, but it required massive computation over generations of mathematicians to work them out. Similarly, humankind discovered a

covered that it was important to do computation in real time as the creature behaves. If everything were programmed purely reactively, a creature could not adjust its behavior to any change in its environment over time scales much less than a generation. It is far more powerful and thus evolutionarily fit to create a creature that can plan and reflect, and to effectively program this creature to try to maximize the propagation of its genes. Chapter 13 describes a succession of increasingly sophisticated programs: the minds of *E. coli*, the wasp, the bee hive, and the dog.

Evolution thus designed the mind for the purpose of making decisions leading to propagation of the DNA. But to accomplish this effectively, the DNA has to program the mind to make decisions, both short-range decisions and long-range plans, that favor the DNA's interest. The DNA has to build a program that makes sophisticated decisions based on local conditions and sensory information, but equally important, it has to control this system so that it does the right thing, that is, favors the interests of the genes rather than wandering off to some other calculation. In other words, it has to build in decision-making capability and, effectively, build in an internal reward function that the creature will seek to maximize (see chapter 14). Evolution thus leads to creatures that are essentially reinforcement learners with an innate, programmed-in reward system: avoid pain, eat when hungry but not when full, desire parental approval, and react to stop whatever causes your children to cry. These urges are detailed and complex—not all orgasms are identical—and moreover are not automatic—creatures must weigh one against the other, weigh long-range payoff versus short-range payoff in a sophisticated fashion. Reinforcement learning is all about calculating how to maximize reward over the long term, disdaining short-term rewards where necessary to achieve greater long-term ones. Even very simple creatures are no doubt vastly more sophisticated than the reinforcement learners in computer science simulations, which already weigh long-term versus short-term gain.

Thus, the minds of creatures have evolved to be something I'll call *sovereign agents*, goal-driven decision makers that strive purposefully toward internally generated goals. We look around us, and we look at ourselves, and this is what we see: sovereign agents. We need modules in our minds for interacting with all these creatures, including ourselves, in order to make decisions about what actions to take. To maximize long-term rewards we have to predict what other creatures will do and what we ourselves will do in the future. The way we understand this is by a computational module or modules that we call consciousness—that is, we attribute free will to ourselves and others, we attribute consciousness to ourselves and others.

After all, what it means for us to have any thought, for example, the thought that we are conscious, is that we are executing computational modules in our mind. The

attribution-of-consciousness module is a very simple, compact way of understanding the world. It is a very useful module, naturally arising through Occam's razor, applying widely to a host of phenomena.

Looking at mind from this evolutionary point of view makes natural something that some authors defending strong AI seem skeptical about: the unity of self. Daniel Dennett and Marvin Minsky, for example, have emphasized that the mind is a huge, multimodule program with lots of stuff going on in parallel. They doubt there is any single individual, any single interest; rather, they see a cacophony of competing agents. But, as we found in our economic simulations, the coordination of agents is crucial in exploiting structure. The program of mind was designed for one end: to propagate the genome. The mind is indeed a complex parallel program for reinforcement learning, but it comes equipped with a single internal reward function: representing the interest of the genes. Thus, the mind is like a huge law office with hundreds of attorneys running around and filing briefs but with a single client, the self. Because we are designed for complex and long-range planning—representing our genes' interests over generations and in widely different circumstances—exactly what the interests of the self are differs from individual to individual and over time. Suicide bombers, mothers, and capitalists are all striving to advance the interests of their genes as their respective minds compute those interests. But for all the modules in the mind and all the many computations going on in parallel, there is one central self focusing all the computation—one central reward being optimized—the resultant of the interests of the genes.

To make decisions that effectively favor our genes' interests involves an enormous computation. We run this evolved program, and it analyzes the world using its compressed description, as discussed throughout the book. For example, it understands the world in terms of interacting objects like cups and fluids and modules for valuable resource management. Analyzing the world like this involves discarding vast amounts of information. When we appreciate a collection of atoms as a cup, we abstract away little details about the shape and color that we could otherwise attend to. But evolution has told us these details are less important to making decisions, and so we engage in massive calculations of which we are unaware that discard all the unimportant information and pass on a processed picture of the world to other modules that output actions, for example, that speak appropriate words.

The portions of our minds that directly control our mouths have no access to all these earlier computations. That is what it means that we are unaware of these computations. We cannot report them. But, at the same time, the later modules do evidently understand the world in terms of the processed picture. At the upper levels of this complex computation, we have modules looking at the outputs of lower-level

modules and effectively understanding the world in terms of the processed picture. We can report on this because these modules directly control our words. This is what it means that we are aware of events in the world. We can report these events, we report them to others or to ourselves. Our awareness is thus nothing more or less than the higher portions of this evolved computation.

I suggest that this picture will, when we are done examining it, qualitatively explain everything about our consciousness and our experience of living. It will explain the nature of our words and of our thoughts. Nonetheless, I'm sure some readers will be unsatisfied with this straightforward mechanistic view of the nature of experience. How a physical system made of meat can possibly give rise to a subjective experience of being is "the hard problem," according to the philosopher David Chalmers. The philosopher Frank C. Jackson (1982), arguing that the way experience feels to us cannot possibly be explained from a physicalist perspective, wrote,

> Tell me everything physical there is to tell about what is going on in a living brain, and ... you won't have told me about the hurtfulness of pains, the itchiness of itches, pangs of jealousy, or about the characteristic experience of tasting a lemon, smelling a rose, hearing a loud noise or seeing the sky. (127)

But I argue that the view advanced in this book explains everything, and does so economically. I specifically take on each of the challenges quoted from Jackson. He picked these, no doubt, because they are intense experiences, but they are intense experiences because it was important to evolution to build them into mind strongly, long before human beings evolved, and for that very reason it is straightforward to explain them qualitatively. It is hard to imagine, for example, how the subjective experience of the "hurtfulness of pains" could possibly be changed to make any clearer that it was built in by evolution to guide the behavior of the program that will advance the interests of the genes.

This will still not convince all readers because the nature of experience is so gripping. However, these remaining doubts are a failure of the imagination analogous to many that have occurred in the history of science. I believe I have presented in this book a simple, straightforward, mechanistic, self-consistent theory that explains all observations, including introspective "experience." It starts by positing (and justifying) the axiom that any thought is an execution of computer code and from this compactly explains every observation, subjective or objective. It explains what you say about your experience to me, and what you think about it to yourself, and what it means to think about it to yourself. To those who still say they don't understand this explanation of their sensation of experience, I reply, Our explanation addresses directly what it means to understand and can explicitly explain why you don't

understand. Thus, we have a self-consistent theory that explains everything, including the doubts of the dubious. By Occam's razor, it should be accepted unless a simpler alternative can be found.

Finally, in chapter 15, I consider the prospects for producing intelligence and understanding in a computer. If the exponential increase in hardware speed of computers continues for a few more decades, we will have machines that can compete in raw computational power with the human brain. Many authors thus predict that we will have smart machines. However, we will never in the foreseeable future have the computational resources to compete directly with evolution, and I argue throughout this book that most of the computation that went into producing our intelligence was done by evolution. Thus, I am a pessimist on the possibility of producing mind. On the other hand, chapter 10 gives some arguments that indicate evolution is very far from optimal at evolving intelligence. If this is so, then better algorithms, together with improvements in hardware speed over the coming decades, might plausibly allow the development of smart machines. One reason why I have engaged in this project is to think through the directions that might lead us there.

2 The Mind Is a Computer Program

The goal of this chapter is to describe why and in what sense computer scientists are confident that the mind is equivalent to a computer program. Then, in the rest of the book, I describe what that computer program looks like, how it works, and how it came to its present form. This section does not deal with new material; it reviews arguments published by Turing in 1936. Readers familiar with the Turing machine model of computation are invited to skip to section 2.1.

The modern field of computer science began in the 1930s with studies by Alonzo Church and Alan Turing on the nature of an algorithm, also known as an effective procedure. The basic idea of an algorithm is very simple. An algorithm is simply a recipe for computing, like a recipe for cooking, but written without any ambiguity. It is a list of instructions, with each instruction stating exactly what to do next. It may not necessarily be deterministic: at step 348 the algorithm may say "flip a coin and if it comes up heads add a tablespoonful of sugar and if it comes up tails add a teaspoonful of salt." But if it always says what to do next, how explicitly to determine what happens next, then it is an algorithm.

We are mostly used to thinking about algorithms that have been crafted to do something specific, like an algorithm to prepare duck à l'orange or an algorithm to compute the digits of π. But, more generally, we may have an algorithm that just evolves. In a sense, then, the evolution of any system under a well-defined set of laws of physics that specify what happens next at all times is an algorithm.

Church and Turing were trying to answer a famous open question posed decades earlier by David Hilbert: Can one write down an algorithm that would prove all the true theorems of mathematics? Essentially: could you replace a mathematician with a computer program. They were, of course, working before electronic computers existed, so the notion of a computer program had not yet been exemplified. Church and Turing recognized that to address this problem they would first have to formalize what was meant by an algorithm.

Turing addressed this problem by trying to formalize the notion of a mathematician. He would see if he could reduce everything the mathematician did to a series of steps that were simple enough to be explicitly written down.

Turing visualized a mathematician sitting and working on proving theorems, writing on a big pad of paper. The mathematician might do three things. He might read what is on the paper, he might erase it and write something else, and he might think. Paper is ordinarily two-dimensional, but to simplify the discussion Turing suggested considering the paper as one-dimensional, so that the mathematician would really just write on a long paper tape, which Turing assumed was divided into squares. It will hopefully become clear, if it is not already, that the restriction to one-dimensional paper does not affect the generality of the argument, i.e. does not

with the rest of the world, to which state it next transits. Since Turing couldn't specify the physics of the brain as it proves some arbitrary theorem, he allowed the brain to do anything it could conceivably do, constrained only by the fact that its action was determined by its state and what was written on the tape with which it was interacting. In this way, he could bound all the possible things the mathematician could possibly think or prove.

So, Turing allowed that thinking could possibly take the mathematician's brain from whichever state it was in into any of the other vast but finite number of states available to it. A step of the proof process now consists of reading the symbols written on some squares of the paper, the mind transiting from one state to another, writing some symbols on some squares of the paper, and shifting one's gaze some number of squares to another square on the paper. Exactly what happens, what gets written on the paper, and what state of mind one winds up in after such a step is determined by what one's state of mind was before the step and what one read on the paper. So, there is effectively a huge lookup table that says, if the mathematician reads symbol j, and his mind is in state a, then he will write symbol k, and his mind will subsequently be in state b, and he will shift his gaze some number of squares to the right or left. Since Turing allowed for all possibilities, this lookup table is arbitrary: any possible assignment to each possible entry is allowed. There are thus a truly huge number of possible lookup tables—a number in fact exponential in the huge number of possible states of the brain. So there might be $10^{10^{10^{30}}}$ possible tables. But, in this view, any mathematician's mind is perforce equivalent to one such lookup table.

Inspired by this physical model, Turing wrote down a simple model of a computer, now known as a Turing machine. This model was only slightly simplified from the description just given. Instead of being able to look at squares of paper near the one being glanced at, the computer could only look at one square at a time. Instead of being able to shift his gaze some distance, the computer could shift his gaze only one square.

A Turing machine thus formally consists of a *read-write head*, a finite *lookup table*, a finite set of *states* including a *start state* and a *stop state*, and an infinitely long *tape* on which is written a finite string of symbols from some finite alphabet (with blank space extending infinitely far in both directions beyond where the string of symbols is written). Figure 2.1 shows a Turing machine about to begin computation. Table 2.1 shows an example of a lookup table.

The machine starts with its head reading the *start location* on the tape and in the start state. The set of symbols initially written on the tape is called its *program*. At each step, the head reads the symbol it is looking at. It consults the lookup table for

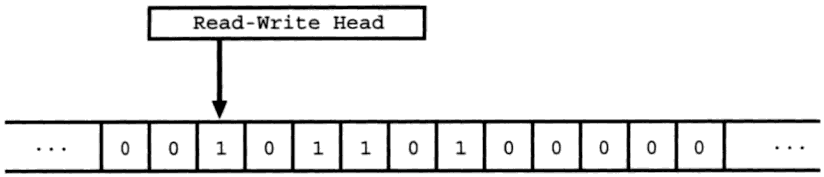

Figure 2.1
A Turing machine. Blank squares are indicated by 0. An infinite number of blank squares lie to the left and the right. The read-write head is scanning the start state, on which 1 is written. Consulting a lookup table like table 2.1 next to the entry for S_0 (the start state, which it is currently in), and since it is reading 1, the read-write head will write a 0, stay in state S_0, and move one square to the right.

	0	**1**
S_0	$1, S_1, \mathrm{R}$	$0, S_0, \mathrm{R}$
S_1	$0, S_1, \mathrm{L}$	$1, S_2, \mathrm{R}$
S_2	Halt	Halt

Table 2.1
A lookup table with three states, S_0 (the start state), S_1, and S_2 (the stop state), and two symbols, 0 and 1. Each entry in the table tells what to do next if the machine is in the state indicated by the row and reading the symbol indicated by the column. A blank square is indicated by 0. Thus, if the machine is in state S_0 and the read-write head is reading a square containing symbol 0, it will write a 1 in the square, move to the right, and change state to state S_1.

what to do when it sees that symbol, given that it is in its current state. The lookup table tells it three things. It says what symbol to write on the tape replacing the one it just read, which of the states it should transit to next, and in which direction, right or left, it is to move its read-write head. Since the number of states and the number of symbols are each finite, and since the lookup table says exactly what to do for each state and each symbol being read, this prescription gives an algorithm: it is clearly an effective procedure that always tells the machine what to do next. The computation proceeds until the machine enters the stop state (or forever, if it never enters the stop state). When it stops, if the program did something interesting, it may have written the answer to some question on the tape.

Turing was able to prove mathematically that this simple model had all the power of the more general model first presented. Moreover, he was able to show that for some particular choices of lookup tables, one would get a Turing machine A that was *universal* in the sense that it could *simulate* any other Turing machine program. Given any other Turing machine, say Turing machine B, there would be a program, i.e. a particular string of 1s and 0s written on the tape of the universal Turing

machine—so that the universal machine simulated *B*'s operation step-by-step. The way this works is simple. *A*'s tape is simply initiated with a description of *B*'s lookup table and *B*'s program. As the machine *A* reads the tape and follows the instructions in its lookup table, it goes through a sequence of operations. For every configuration that machine *B* goes through, machine *A* has an exactly corresponding configuration that it goes through. When eventually *B* halts (assuming it does), *A* halts in its corresponding configuration with an exactly corresponding message written on its tape. Machine *A*'s computation is in logical one-to-one correspondence with machine *B*'s.

The proof of this is not hard but is not given here. The reader interested in seeing it in detail is referred to Turing's (1937) original paper or to one of a number of books (e.g., Minsky 1967). The proof is constructive. One can describe an explicit lookup table for Turing machine *A* and show exactly how it simulates *B*'s action, whatever *B*'s lookup table may be. For each step of *B*'s action, *A* keeps explicit records on its tape of what the state of *B* is, what is written on *B*'s tape, where on *B*'s tape the read-write head is pointing. Turing machine *A*'s read-write head just runs back and forth along its tape keeping these records, mirroring exactly the performance of *B*. As it is simulating each step of *B*'s computation, *A* simply consults its tape, on which *B*'s lookup table is enumerated, to figure out what *B* would do next, and modifies its tape to keep the appropriate records. In this process, *A* goes through a number of steps for each step that *B* does, but every step *B* goes through is faithfully represented in *A*'s action, and when *B* halts, *A* does, and whatever string of symbols is written on *B*'s tape is copied on *A*'s as well.

The proof that this restricted model is equivalent to the less restricted model that was described first is essentially identical. One simply shows that the restricted model can simulate exactly the less restricted model. Again, one gives an explicit construction where for each state of the less restricted model there is a corresponding state of the more restricted model, and shows explicitly that the more restricted model halts in the state corresponding to the less restricted model's when it halts, with a corresponding message written on its tape.

Moreover, it is possible to show that universal Turing machines can have remarkably compact descriptions. Claude Shannon (1956), who is also famous as the inventor of information theory, showed that one can have a universal Turing machine that uses only two symbols. The two-symbol universal Turing machine can simply simulate having many more symbols by encoding the states using binary notation, that is, using a number of symbols to represent what would be represented by a single symbol in a Turing machine with more symbols. Marvin Minsky (1967), one of the founders of Artificial Intelligence, wrote down a universal Turing machine with only seven states and four symbols. This is a lookup table with only 28 entries

that specifies a Turing machine that, with an appropriate program on its tape, can simulate the working of any other Turing machine and program whatsoever.

Thus, although this discussion started by describing a mathematician with an enormous number of states in his mind, perhaps $2^{10^{30}}$ states, everything that he could do can be *exactly* done by a Turing machine with only seven states and four symbols that is fed an appropriate computer program in the form of an appropriate, finite string of symbols on its tape.

I have presented the preceding argument as if it were a derivation. I gave it in a form that implies that a Turing machine could effectively simulate the computation performed by any physical device that could be built, whether built by people or by evolution. This is a modern, rather strong view of the Church-Turing thesis that asserts that anything *computable* can be computed by a Turing machine. Some readers may reasonably complain that the discussion arguing that the brain can have only finitely many states was not mathematically rigorous.

Indeed, it has been pointed out more recently (Vergis, Steiglitz, and Dickinson 1986; Smith 1993; 1999) that if one assumes some formal set of physical laws, one can rigorously address the question of whether a physical device could in principle be built that could not be effectively simulated by a Turing machine. Unfortunately, we cannot answer this question rigorously for our universe, because we don't know for sure the exact physical laws of our universe, and the laws we believe are true are too complicated to analyze. However, if one studies a "toy universe" in which some set of simplified, formal laws are hypothesized, one can in principle answer the question. Physicists commonly analyze toy sets of laws in order to get insights. For example, anyone who has taken high school physics has answered questions "assuming ideal conditions in which there is no friction." Smith (1993; 1999) has studied several sets of laws of considerable realism and complexity, and has proved that the Church-Turing thesis holds for some sets of laws and not for others. Also, other authors (e.g., Siegelmann 1995) have discussed, without worrying much about the possibility of implementation, various super-Turing models of computation, that is, models which could compute functions that are not computable on a Turing machine. Typically these invoke some model of a chaotic analog system.

All the super-Turing models studied by Smith and others exploit the use of real numbers of arbitrary precision. Typically they are very unnatural. One essentially stores the answer to some computation so complex it provably cannot be computed on a Turing machine, storing the answer in the increasingly high-order bits of real numbers. If the super-Turing computer only accessed any finite number of bits of precision, say 1,000, a Turing machine could simulate the program by placing the 1,000 bits on different memory locations (different squares on the tape). So, to do

something super-Turing, to compute some function that could not be computed at all on a Turing machine, one must use infinite precision, compute with infinitely precise real numbers, and make fundamental use of the infinitely least significant bits of these numbers. In fact, these super-Turing machines must make use of a number not even computable by a Turing machine, so in a strong sense the non-Turing nature is inserted before the computation even begins.

All these approaches are thus hopelessly delicate and nonrobust, and in the presence of even the tiniest bit of noise, would compute nothing more than can be computed by standard computers. Since noise and friction are omnipresent in the real world, it seems highly unlikely that such computers could ever be built, even in principle. Since biological brains are excessively noisy, stochastic environments, and since they have to have arisen by evolution, it seems very far-fetched that such models have anything to do with intelligence or mind. The upshot of all these investigations, then, is that it seems apparent that the brain as well as any reasonable physical device can be simulated by a Turing machine.

Turing was of course unaware of these later investigations into mathematical physics, but neither did he suggest that his analysis of the proof process of the mathematician was mathematically rigorous. Indeed, he offered it only for intuition. Turing (1937) wrote,

> All arguments which can be given [as to the nature of what algorithms might be] are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is "What are the possible processes which can be carried out in computing a number?" (249)

Turing offered three arguments in support of his view that a Turing machine could compute any function computable in any "natural model of computation." The first is the analysis of the physics of the mathematician, which I have described. He did not claim it was rigorous but offered it as "a direct appeal to intuition." Turing's third argument, which I do not discuss here, was a version of the first, slightly modified to "avoid introducing the 'state of mind' by considering a more physical and definite counterpart of it."

Turing's second argument, which is often judged the most compelling, was a direct proof that many natural models of computation, proposed by different people, were equivalent and could all be computed by Turing machines. I discuss this next.

Can there be other kinds of computers, not Turing machines, that compute other things? Researchers have written down various other models of what it might mean to "compute" a function, that is, to execute an algorithm. One model is the class of *general recursive functions*, which are all the functions that can be formed by com-

Generally when someone writes a program for a PC or a Mac or any other modern computer, they don't write it in 1s and 0s but rather in high-level instructions. One can still think of what's going on as a simple random access machine. The computer has just had coded into it a lookup table that looks at the high-level instructions and translates them into 1s and 0s. Using a high-level language can make it much easier for people to program, but it doesn't speed up the computation. The computation still essentially consists of looking at the contents of a register in memory, accessing a relatively small lookup table to see what that instruction means to the computer, and executing that instruction. The instruction still just writes on some location in memory, shifts to another location in memory, and/or shifts the state of the computer.

A random access machine has a single processor, but other computers are multiprocessors with many independent processors that can read and write and change state independently. The brain, composed of many neurons, seems rather like a huge multiprocessor with many little weak computers hooked together in an interesting way. Highly parallel computers do not have more computing power than a single appropriately faster computer. A single computer can simulate 100 computers working in parallel by simulating the computation of each of the computers in turn. It can simulate the first computer straightforwardly until it tries to read some information written by another of the computers, say computer 2. Then it has to back up and simulate whatever computer 2 was doing to figure out what the message was, before it can go back and resume work simulating computer 1. But it is always possible to simulate all the computations of all 100 computers if one has a computer that is 100 times as fast, or if one runs a single computer 100 times as long. Using one hundred different computers may speed up one's calculation by at most a factor of 100, but never more.

The converse, that a multiprocessor consisting of 100 computers can compute 100 times as fast as a single processor, is not true. Sometimes it can, and sometimes it can't. It depends on what one wants the computer to do, on what the program is. If each step of a calculation depends on the last step, having 100 computers will not allow the calculation to be speeded up at all. Say the task is to compute the ten-thousandth Fibonacci number in the most straightforward manner. The Fibonacci numbers are defined as follows. The first two Fibonacci numbers are 1 and 1, and thereafter the next Fibonacci number is generated at each step by adding the last two. So the third Fibonacci number is 2, the fourth is 3, the fifth is 5, and so on. What is the ten-thousandth Fibonacci number? Let's assume this will be computed by simply calculating each successive Fibonacci number as the sum of the previous two. A single processor can compute the ten-thousandth Fibonacci number after

10,000 steps. At each step the last two numbers are added until one obtains the ten-thousandth number.[3] If this calculation is done on a multiprocessor, how could it be speeded up? At any given time, to do any further calculation, one would need to know the last two numbers. There is no obvious way to jump ahead and break up the calculation so that one processor can work on part of it while another processor works on another part. The multiprocessor, no matter how many processors it has, will be no faster than a single processor at this program.[4] Thus, for this computation, one processor that is ten times as fast is better than ten processors. Using multiple processors can only speed up some problems, and only when the programmer correctly exploits the structure of the problem.

It is important to understand that although the brain is a highly parallel computer, we can reasonably think about how we would simulate it on an ordinary computer. There may be advantages to the structure of the brain in that it is designed to compute the exact kinds of functions it needs rapidly. Indeed, it would be shocking if evolution had not designed the wiring diagram of the brain efficiently to compute precisely whatever the brain needs to compute. The question is, rather, why evolution has provided human beings with brains capable of doing many things that do not, at first glance, seem evolutionarily important, such as proving mathematical theorems. But whatever the brain is computing, we can equally well compute it with the appropriate program on an ordinary computer of sufficient speed. The ordinary computer will need to do only as many logical operations as the brain did, in total, just as we can simulate the actions of a parallel computer with a serial computer.

Moreover, because we can in principle map from one sufficiently powerful model of computer to another with no loss of power, we can switch back and forth from model to model. If we want to talk about using artificial neural nets (a model of computing discussed in chapter 4), we can talk about programs on an ordinary computer. Indeed, the artificial neural nets modeled by intelligence researchers are almost always programmed onto ordinary computers. On the other hand, many artificial neural net models are not nearly as powerful as random access machines, or equivalently as Turing machines, because they lack a memory on which to write. Simple counting is a very hard problem for most artificial neural nets. These neural nets do not have the ability to perform recursive functions. When they are augmented with the ability to perform recursive functions, the techniques known for engineering artificial neural networks break, and not much is known about how to program them to accomplish anything, much less the astonishing range of tasks people accomplish rapidly.

Recently there have been proposals of *quantum computers*. Quantum computers are not claimed to be super-Turing because nobody claims they can compute any

functions not computable on standard computers. It is accepted that standard computers can simulate, state by state and line by line, the operation of quantum computers. However, it is plausibly claimed that quantum computers might violate the strong Church-Turing thesis by computing some functions faster.

In quantum mechanics, systems can be not in one state or another but rather in a superposition of states. For example, one can prepare a spinning particle that has half a chance of spinning to the left and half a chance of spinning forward. The classic example of the consequences of this is *Schrödinger's cat*. Say you arrange a box with a radioactive source in it that emits a particle with probability 1/2 per hour. Within the box you place a live cat and a device that if it detects an emitted particle will release poison gas, killing the cat. You close the box. After an hour, the box is in a superposition with half a chance of holding a live cat and half a chance of holding a dead cat. Until you open the box and observe, the cat is in a superposition of states. When you observe, the superposition collapses and is in one state or the other, either dead or alive. While the case of the cat may seem a bit paradoxical, it is not hard to observe superposition effects in very small-scale experiments, involving quantum-size objects such as one or a few electrons. A quantum computer exploits the possibility of matter to be in a superposition of states to create a computer that is not in one state or another. Rather, it is possible to write into its memory many states at once. By searching over all the multiple states in its memory, the quantum computer may be able to compute faster than a standard computer.

If a quantum computer could be built, it could not compute any function not computable by a Turing machine. Rather, it is clear that a Turing machine could simulate the computation of a quantum computer by carefully enumerating all the possible states. However, it is possible that a quantum computer could compute somewhat faster than a standard random access computer. Intuitively, the magical ability to search over a potentially exponential number of memory superpositions seems like it could plausibly lead to more power. More concretely, an algorithm for factoring numbers rapidly is known for a quantum computer (Shor 1994). No such algorithm is known for a standard computer. On the other hand, neither is it known whether it is impossible to factor numbers rapidly on a standard computer. The factoring problem is not even NP-complete, a class of problems believed hard (see chapter 11). But it is generally believed that factoring cannot be done rapidly on a standard computer. Computer scientists have invested considerable effort in trying to figure out how to factor rapidly, without success. Standard cryptographic systems could be readily deciphered if it turns out that factoring could be done rapidly. So, the computer science community has a considerable investment in the belief (hope?) that factoring is hard. If factoring is in fact hard on standard random access

machines, then quantum computers, while they can only compute the same functions as a standard machine, could compute at least some of them faster.

I noted that a Turing machine can simulate in time at most kT^3 any computation done by a random access machine program in time T. This is a polynomial slowdown, because the time taken by the Turing machine grows only as a power function (in this case, the cube) of the time taken by the random access machine. The strong Church-Turing thesis says that not only can a Turing machine simulate any computation done by any other computer whatsoever but can do so with only a polynomial slowdown.

An example of a slowdown that would be nonpolynomial is an exponential slowdown. If a tape has T squares, and if each of these can have a 1 or a 0 written on it, there are 2^T possible ways of writing a collection of 1s and 0s on the T squares of the tape. If a quantum computer could hold all these 2^T states in a superposition and somehow search over them all, it might be able to do in time T what on an ordinary Turing machine, which would have to successively enumerate all the 2^T possibilities, would take an exponential amount of time.

The extent to which quantum computers are physically realizable is unclear. They depend on keeping a very delicate *quantum coherence* among large-scale structures. Not just one or a few electrons but many objects have to be kept in a complex superposition. Large-scale objects (such as cats) are extremely hard to keep in superposed states. If a quantum computer can be built, its functioning would depend critically on enormously careful isolation from outside sources of interference, on being held at near absolute zero temperatures, and on exceedingly delicate algorithms that seem hopelessly nonrobust and unevolvable. It is not clear whether a quantum computer of any interesting size is constructable even in principle, and it is generally accepted that it would be very unlikely to occur in nature, much less in nervous systems. Thus, it seems highly unlikely that quantum computation is relevant to the mind. Even if it were to turn out that the brain is a quantum computer, it would still be true that its operation could be exactly simulated on an ordinary Turing machine.

To summarize this discussion: Turing formulated the notion of an algorithm as a Turing machine, which is a simple model of a computer involving a finite lookup table and an infinite tape. Universal Turing machines can be constructed with as few as four symbols and seven states, that can simulate exactly the performance of any other Turing machine. These can also simulate exactly the computation performed by other models of computation, many seemingly very different, including every model that people have had the imagination to write down except a few that make

critical use of infinite precision numbers. All the latter, super-Turing models, do not seem buildable and are generally not regarded as natural models of computation. Computer scientists thus generally accept the Church-Turing thesis that any computable function can be computed by a Turing machine. This thesis is unproved inasmuch as it essentially rests on an intuitive definition of what is meant by computable. In spite of the fact that there is intuitive support for this definition, computer scientists are open to the possibility that they have overlooked something. However, it seems highly plausible that Turing machines could compute any function that could be computed by any physical device, whether made by people or by evolution. So, the consensus of computer scientists is that, in principle, a computer program running on an ordinary computer could simulate exactly the performance of the brain. In this sense, if in no other, the mind is equivalent to a computer program. Our quest to understand the actions of the human brain within computer science thus comes down to a quest to understand how the actions of the human brain can be computed on an ordinary computer.

A skeptic could still validly assert that it has not been mathematically proved that the mind is equivalent to a computer program. I hope, however, that the preceding discussion has made it clear why the straightforward picture, the overwhelming consensus of the field, is that the mind must be equivalent to a computer program. If it turns out that the mind cannot be so explained, we may be driven to a whole new vision of computer science. My hope is to show in this book that no such extremes are necessary and that it is entirely plausible that a straightforward and elegant explanation will emerge within computer science. This is where we should look first.

2.1 Evolution as Computation

In 1948, five years before Watson and Crick discovered the structure of DNA, John von Neumann (1966) more or less constructed it, answering the question of how one could construct a machine that could reproduce itself. Figure 2.2 is a copy of a figure drawn by von Neumann. It consists of a rigid backbone, with a side chain present or not present at each position in the backbone. The presence of a side chain represents a 1 and its absence a 0. Readers may recognize this as logically isomorphic to DNA.

Von Neumann didn't seek to model actual human chemistry; rather, he abstracted the computational question of how complex structure might reproduce. He imagined the reproducing automata floating in a sea of parts. The parts were to be simple (for his first cut at the problem), that is, not at the level of accurate depictions of proteins, lest he bog down in the details of chemistry. He imagined instead about a dozen

This answered Hilbert's open question (mentioned earlier in the chapter) in the negative: there is no effective procedure that can prove all the true theorems of mathematics. But it left open a revised form of Hilbert's question: Can one find an effective procedure that, given any mathematical assertion, would decide whether or not the assertion had a proof? Hilbert believed that the answer would be yes.

Turing answered this also in the negative. Turing showed that it is impossible to decide, given a Turing machine and a program, whether the computation of running the program will ever halt. If it will halt, then that fact is provable—the proof consists of simply running the program and verifying that it halts. But one can not in general establish that it will not halt. One's inability to decide whether a given Turing machine computation will halt or not is an example showing one can not decide whether arbitrary mathematical assertions are provable.

One can have a small Turing machine and a short program such that when one runs the program on the machine, computation never halts. If one tried to verify whether or not it would halt, one would wait forever and it would still be running. One might run another program for an arbitrary length of time, after which it might halt in the next second. Moreover, there is no algorithm to shortcut the process of deciding halting. If there was such an algorithm, Turing showed that applying it in a self-referential way to decide if it, itself, will halt can lead to a contradiction. Although Turing machines have short descriptions and proceed according to simple, well-defined, syntactic rules, their actual computation can be arbitrarily complex and unpredictable. Chapter 14 returns to this point in the context of discussing free will and determinism.

Given the examples used in Gödel's proof and Turing's proof, it seems intuitive that paradoxes are inherent in self-referential systems of sufficient complexity.[5] Von Neumann was worried that similar paradoxes might befall an automaton that attempted to examine its own structure and reproduce it. Such an automaton would perforce be both self-referential and complex.

Von Neumann proposed a very simple approach that avoids all these difficulties. If you were going to construct a car from parts, it would help if you started with a plan telling you what to construct and where to put it, that is, detailed instructions:

1. Start with a part of type 1.
2. Attach to the end of part 1 a part of type 2 using soldering gun 3.
3. ...

In other words, what is needed is an algorithm, a computer program, that tells you how to construct the car. Then all you would have to do is follow instructions.

So, von Neumann proposed that the machine he would copy would be analogous not only to the car but to the car *and* the algorithm for constructing it. This is a much easier thing to duplicate than just the car. You begin by copying the algorithm. This is easy—it's a linear sequence of instructions. In fact, it is reducible to a Turing machine program, since I've argued that every algorithm is equivalent to a Turing machine program. Thus it is a sequence of 1s and 0s that can be copied straightforwardly. Then you follow the algorithm to copy the car. The already constructed car is superfluous, just a distraction. All you have to do is follow the instructions to get a copy of everything you began with.

And since you want the whole thing to be a single machine, all you have to do is to write the computer program in hardware and attach it to the car. Writing it in hardware is easy because it's logically just a sequence of 0s and 1s. In fact, writing it in von Neumann's simplified formal model is easy: you can build the whole thing out of the rigid skeletal elements. That is what figure 2.2 shows: the picture von Neumann drew of a chain of rigid elements with at each joint either a side element pointing off, representing a 1, or no side element pointing off, indicating a 0.

Here is how von Neumann (1966) described the whole construction:

> There is no great difficulty in giving a complete axiomatic account of how to describe any conceivable automaton in binary code. Any such description can then be represented by a chain of rigid elements like that [of figure 2.2]. Given any automaton X, let $\phi(X)$ designate the chain which represents X. Once you have done this, you can design a universal machine tool A which, when furnished with such a chain $\phi(X)$ will take it and gradually consume it, at the same time building up the automaton X from parts floating around freely in the surrounding milieu. All this design is laborious, but it is not difficult in principle, for it's a succession of steps in formal logics. It is not qualitatively different from the type of argumentation with which Turing constructed his universal automaton. (84)

Now, it is not hard to design a second machine, B, that can copy linear chains of rigid elements. Fed a description of anything, $\phi(X)$, B consumes the description and produces two copies of it. For example, given the structure in figure 2.2, B produces two exact copies of it.

"Now," von Neumann writes,

> we can do the following thing. We can add a certain amount of control equipment C to the automaton $A + B$. The automaton C dominates both A and B, actuating them alternately according to the following pattern. The control C will first cause B to make two copies of $\phi(X)$. The control will next cause A to construct X at the price of destroying one copy of $\phi(X)$. Finally, the control C will tie X and the remaining copy of $\phi(X)$ together and cut them loose from the complex $(A + B + C)$. At the end the entity $X + \phi(X)$ has been produced. Now choose the aggregate $(A + B + C)$ for X. The automaton $(A + B + C) + \phi(A + B + C)$ will produce $(A + B + C) + \phi(A + B + C)$. Hence auto-reproduction has taken place. (85)

Having thus designed the reproducing automaton (denoted here by $A + B + C$), von Neumann realized immediately one other thing: his theory immediately gives rise to a microscopic description of evolution. Again, I don't think I can do better than to close this section with another quotation from von Neumann:

> You can do one more thing. Let X be $A + B + C + D$, where D is any automaton. Then $(A + B + C) + \phi(A + B + C + D)$ produces $(A + B + C + D) + \phi(A + B + C + D)$. In other words, our constructing automaton is now of such a nature that in its normal operation it produces another object D as well as making a copy of itself.... The system $(A + B + C + D) + \phi(A + B + C + D)$ can undergo processes similar to the process of mutation.... By mutation I simply mean a random change of one element anywhere.... If there is a change in the description $\phi(A + B + C + D)$ the system will produce, not itself, but a modification of itself. Whether the next generation can produce anything or not depends on where the mutation is. If the change is in A, B, or C the next generation will be sterile. If the change occurs in D, the system with the mutation is exactly like the original system except that D has been replaced by D'. This system can reproduce itself, but its by-product will be D' rather than D. This is the normal pattern of an inheritable mutation. (86)

2.2 The Program of Life

This final section of chapter 2 briefly outlines the computational process that is life, that produces and maintains our bodies. The goals here are to appreciate the fact that we are nothing but a huge computation, and to marvel at the intricate working of this machine. A priori, the computation of life and the computation of mind might be expected to be rather different. The computation of mind must ultimately sit atop the computations of chemistry, but many would argue that reduction to chemistry is the wrong way to go—they conjecture instead that mind is some type of emergent phenomenon, that is, a phenomenon qualitatively different in the aggregate than the simpler processes that compose it. But I will ultimately describe mind in terms not so different from the computation of life. I see both as complex, evolved computations largely programmed in the DNA, both exploiting semantics in related ways. In any case, it provides worthwhile background to review another example of evolved, natural computation.

You were conceived when DNA from your mother and DNA from your father came together in an egg. These two separate DNA programs merged to form a single program. This joint program was then executed, producing you.

In von Neumann's day, describing this as a program might have been imaginative or controversial, but with the understanding we have achieved over the last 50 years, it has become mundane. The DNA is information: a sequence of bits that is read by

chemical machinery and that causes a sequence of mechanical interactions to transpire, processing the information in the DNA.

The machinery is not exactly identical to the read-write head in a Turing machine, but as I've said, any algorithmic machine can be logically mapped into a universal Turing machine, and conversely, a Turing machine program is logically isomorphic to any other model of universal computation. Thus, we can consider Turing machines, parallel processors, Lambda calculi, Post machines, and many other models to all equivalently be computers.

The mechanisms of life are most similar to Post machines, described earlier in this chapter. Recall that Post machines consist of a collection of productions. Each production has (one or more) input sequences of symbols and an output sequence. When a production matches its input sequence in an existing string of symbols, it can substitute its output sequence in its place. As mentioned, one can think of the proof processes of mathematicians as starting with some axioms written as strings of symbols and acting on them with productions to derive lemmas and theorems, which are other strings of symbols. Life is reasonably well described as a giant Post production system. Again and again in life, computation proceeds by matching a pattern and thus invoking the next computational step, which is typically similar to invoking a Post production by posting some new sequence for later patterns to match.

The DNA program is arranged in submodules called genes, organized into collections called chromosomes. Each chromosome is a strand of DNA. This strand is a long molecule that is explicitly analogous to a computer program: its organization conveys logical information encoded in a sequence of discrete symbols that controls the operation. Alternatively, it is a long sequence of symbols as operated on in Post machines. The DNA molecule consists of a string of bases, where each base can be one of four possible molecules represented, respectively, by the four symbols A, C, G, T. Thus, a strand of DNA might be ... A A T G A A C T T G ....

A gene is a small subroutine of this program containing the instructions for producing a protein. As is usual with subroutines, the execution of a particular subroutine corresponding to a given gene begins when the subroutine is called by the execution of other parts of the program.

This is done when molecules called transcription factors (analogous to productions in a Post machine) are produced elsewhere in the program. The transcription factors recognize the beginning of the gene by finding specific short sequences of nucleotides nearby, each transcription factor recognizing certain specific patterns of nucleotides and attaching itself to the DNA sequence where it recognizes this pattern. Thus, as with Post systems, execution of the gene begins by matching one or more patterns in

a string. The combination of such a pattern on the DNA sequence and the transcription factor that recognizes it is appropriately called a control module (Branden and Tooze 1999, 151).

The matching of the transcription factors effectively posts other patterns, which are matched by a molecule called polymerase that attaches to the DNA at a point specified by the pattern matching. The polymerase transcribes the information in the DNA base by base into the related molecule RNA in a one-to-one mapping, each A in the DNA sequence being mapped into a U in the RNA sequence, each C into a G, each T into an A, and each G into a C. Thus the strand ... AATGAACTTG ... would be mapped into ... UUACUUGAAC.... So, at the end of this map, an RNA sequence with the identical information content as was in the gene is prepared. The transcription proceeds along the DNA, copying the whole gene (or often a sequence of genes) into RNA until it is stopped upon encountering a specific stop pattern.

Why is the information copied into RNA before further processing? One theory is that this structure arose as an artifact of evolutionary history. Because RNA can serve as a catalyst as well as carry information, it is believed that first life was purely RNA-based, involving no DNA. Thus, the mechanisms of life evolved using RNA. As creatures became more complex, evolution discovered that it could use DNA, which is chemically more stable than RNA, as long-term storage for the valuable information. But it continues to translate this information back into RNA to employ descendants of the previously evolved mechanisms for manipulating information in RNA.

Large portions of the RNA sequence called introns are then carefully snipped out (see the box What Is Junk DNA?). This is done by small particles (made of a combination of proteins and nucleic acids) that contain RNA sequences matching the sequences at the junctions between the introns and the adjacent coding regions. Such particles bind to a junction end, twist its intron into a loop, excise the loop, and splice the coding regions (the exons) back together.

The RNA sequence that remains when the exons are stitched back together is the program for producing a protein. It is translated into a protein as follows. The sequence of letters is parsed into words by reading the bases sequentially three at a time. Three sequential bases are called a codon because they code for an amino acid. So AAA is a codon, as is AAC, and so on, through all the 64 possible combinations of three letters from the set {A, C, G, U}. The codons are translated into amino acids using a simple lookup table called the genetic code. The RNA is read starting at a position on the sequence where the codon AUG is found, indicating the start of the

How Big Is the DNA Program?

How much effective information content is there in the DNA program? Neglecting the "junk," I estimate this to be 10 megabytes. The human genome comprises roughly 30,000 genes, each coding for a protein roughly 300 amino acids long on average. Each amino acid requires less than 5 bits to specify (since there are 20 alternative amino acids). Thus, one arrives at 45 million bits, or since there are 8 bits to the byte, roughly 5 megabytes. The estimate of 10 megabytes thus allows for a fudge factor of 2 to take account of additional regulatory information.

It's possible that this 10-megabyte estimate is overgenerous. The true information content of the DNA program might be quite a bit smaller because the functionality of the program would be identical under many changes in the amino acids. Proteins form three-dimensional structures by folding up linear molecules made of sequences of amino acids. The function of the protein, how it interacts in the program, depends on its folded shape, but the fold is not sensitive to many changes in the amino acids. Different proteins evolve having the same fold and function that have only 30 percent or less of their amino acids in common. The proteins fold in such a way that amino acids that are hydrophobic are on the inside of the fold and amino acids that are hydrophilic are on the outside. It is an exaggeration to claim that this is all that counts about amino acids, but in many cases this a reasonable approximation. This approximation would lead to the estimate that the true information content per amino acid is actually closer to 1 bit than 5. Support for this picture comes from experiments that led Keefe and Szostak (2001) to the estimate that about 1 in 10^{11} sequences of 80 amino acids has a given, specified chemical functionality. This figure suggests that the true information in a gene is only $\log_2 10^{11} \sim 36$ bits. Such arguments suggest that the total information content in the program for a human being might be closer to 1 megabyte than 10.

There are other reasons why the effective information content of the DNA program might be smaller than our estimate, potentially even smaller than 1 megabyte. A naive count of the weights in a neural net can overcount its effective number of parameters, especially when it is trained using a procedure called early stopping (see section 6.2). Similar effects may well apply to counting the effective number of bits in the DNA program.

The 10-megabyte figure could be too low if we are underestimating regulatory information. Current understanding does not allow a confident calculation of the useful information content outside of the genes. However, the full genome, including all "junk," is only about 3 billion base pairs long, and thus contains less than 6 billion bits. This absolute upper bound is still quite compact compared to the naive complexity of the world and to the length of the training data. If information from 10^{35} or so evolutionary learning trials has affected the program, then even with all the "junk" included, the length of the DNA program is vastly smaller than the information that went into its evolution (see chapter 5). Why the relevant comparison is to the useful information content rather than to the total length is discussed in chapter 6.

into tight molecules. In fact, additional molecular machinery has evolved to help the proteins fold into their most compact, lowest-energy state even in the crowded molecular environment found within cells.[6]

After the sequence has folded, the shape of the folded structure determines the subsequent chemical reactions it will undergo. Some regions on the surface of the folded protein can react with regions on the surface of other proteins. This occurs, roughly speaking, because the shape of the surface and the pattern of positive and negative charges exposed there fit the pattern on another protein. One fits into another like a key fits into a lock—the two patterns match. As with a key and a lock,

the proteins in the body are evolved to be very specific about what fits where. Each key can fit only in locks designed to be a close match.

If we understood how to predict the folding, and better yet, how to design sequences that would fold into different shapes, we could design drugs for many purposes. Whatever key we needed for a given lock, such as a virus or an enzyme, we would presumably design. The massive pharmaceutical market has seen to it that this is one of the most heavily studied fields in science. Nonetheless, our computers are not yet powerful enough to reliably simulate the folding of proteins, and scientists do not yet have a shortcut allowing them reliably to predict how proteins will fold and thus cannot generally create proteins that fold nearly as tightly as the ones coded for in DNA. The proteins, in deciding how to fold, do an analog computation of great complexity.

The proteins within the body react when they find an appropriate mate. In other words, we again have a Post system–like pattern-matching.[7] The proteins float around, looking to match a pattern, and create other patterns when they do.

Often the proteins react catalytically, which means that they trap some other proteins long enough, and in an appropriate way, to get them to react together. An enzyme may have patterns that match patterns on two different specific proteins well. It will then hold them in place for a while, and perhaps influence their shapes, so that they react with each other. The enzyme then emerges, free to catalyze other reactions.

Three-party interactions like this, where one entity gates a reaction between two others, are quite useful for building universal computers. Indeed, that is essentially what a transistor does. Like transistors, the interactions between proteins allow universal computation.

Enzymes, for example, serve to gate reactions. Consider the reaction shown in figure 2.3. Here A, B, C, and D are compounds and 1, 2, and 3 are enzymes catalyzing the reactions, respectively, A $\rightarrow$ B, $\rightarrow$ C, and $\rightarrow$ D. In the presence of enzyme 1, compound B will be produced from A. In the absence of enzyme 1, very little of B will be produced. Similarly, the presence or absence of enzyme 2 or 3 determines whether the reaction forks to produce compound C or compound D. To a computer scientist, this is nothing more than a simple flowchart showing the results of some if-then instructions.

The usual first step in writing a computer program is to produce a flowchart. A flowchart is a graphical representation of what the program will do that consists of nodes representing the instructions in the program connected by arrows showing the flow of control through the program. So, an arrow will connect one node to another if the program will first execute the one instruction followed by the other. Because

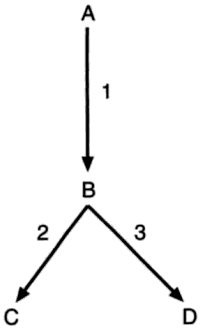

Figure 2.3
A simple enzymatic pathway. Compounds B, C, and D are produced from A depending on the presence of enzymes 1, 2, and 3.

programs branch depending on the partial results as the computation proceeds, a given node may have several arrows leading from it to other nodes. Typically the nodes are represented in a flowchart by ovals, squares, or diamonds depending on the type of instruction the node represents, and the instruction is written within the figure (e.g., within the oval). Figure 2.3 can be thought of as a very simple flowchart showing flow from state A to state B if 1 is present, and flow from state B to state C or to state D depending on whether enzyme 2 or 3, respectively, is present.

The human metabolism is an enormously complex program with an enormously complex flowchart. It is broken down into modules called *pathways*, tightly organized metabolic systems in which certain enzymes react to produce products necessary to other reactions. The whole reaction runs in an organized fashion, all programmed originally in the genes, to do a huge logical calculation that causes a person to develop in precise fashion, behave properly, and keep on going for 70-odd years. Thousands of proteins are created at the right times, in the right concentrations. Which products are created next depend on what molecules are present, in a way analogous to the pathway represented in figure 2.3. The pathways feed back on themselves in complex ways to control the metabolism so that it works precisely.

An awe-inspiring wall poster, about 3′ ft × 10′ ft, titled "Biochemical Pathways" (Michal 1998), shows the entire flowchart of the metabolism (or, at least, all the portions that have been worked out). The poster is filled with a flowchart of fine lines labeled by fine print and would be completely illegible if I tried to reproduce it here on a book-size page. When one views this poster, one's first impression[8] is awe at the complexity of the program and respect for the extensive scientific efforts that have gone into unraveling it.

To some extent there is a random element in this machinery. Molecules move somewhat randomly within the body (which is composed largely of water) and react when they run into the molecules they are designed to interact with. This does not detract from the nature of the system as a computer: it is a somewhat randomized computation that, however, is designed (by evolution) to work well in the presence of this randomness.[9]

But there are also carefully constructed molecular scaffolds for conveying information in precise ways and to precise places to cause precise reactions. For example, there are long molecules called receptor tyrosine kinases that serve to bring information into cells. These stick out through the surface of the cell, and extend into the cell into molecular complexes. When a hormone molecule docks at the sensor portion of the molecular complex that sticks out of the cell wall (which it does upon recognizing an appropriate pattern there), molecular changes in the complex occur, enabling enzymatic modules in the protein complex to pass along the information that a specific hormone has been received. The information is thus conveyed to specific molecules within the cell that act in a deterministic way. DNA within the cell may be activated to produce a certain protein, for example, insulin if the cell is a pancreatic cell and has been appropriately stimulated. Similar molecular complexes serve as the brain of the *E. coli*, causing the bacterium to swim toward nutrients and away from poisons (see section 13.1.1).

2.2.1 Genetic Regulatory Networks

I have described how a gene is transcribed into a protein but have yet to speak of the complex structure that controls which genes are transcribed when. This control structure again is machinelike, indeed Post system–like. Each cell in the body contains a full copy of that body's DNA. What determines whether it is a muscle cell or a brain cell is which genes are executed, that is, turned into proteins. (The usual term is that a gene is *expressed*. I occasionally use *executed* to emphasize that gene is basically a small subroutine.) To keep the metabolism working, specific genes have to be expressed at specific times. That is, a thread of the execution of the full program of life has to call the subroutine of that specific gene at the correct times. This subroutine calling is done by a vast genomic regulatory network, which interacts with the metabolic flowchart.

The details of the regulatory network are far from worked out, but pieces of it are being elucidated. For example, the gene regulatory network that controls substantial portions of the development of the sea urchin embryo is surveyed by Davidson et al. (2002). The regulatory networks that control development in bilaterally symmetric animals, and the evolution of these networks from the earliest such animals to more

complex vertebrates, are surveyed by Carroll, Grenier, and Weatherbee (2001). Gene regulation proceeds through a complex network where the presence or absence of a given product determines the branching of a program. Genes are turned off or on depending on whether repressors or inducers are present, which in turn depends on whether other products are present. The system explicitly realizes a logic: production of one product may depend on the conjunction or the disjunction of other products, and on more complex logical circuits.

As development proceeds, it utilizes memory to control the program flow—memory, for example, in the form of DNA rearrangements and molecular modifications such as methylation patterns. Methylation occurs when a molecule called a methyl is attached to a protein or a nucleic acid. For example, some of the cytosines in a gene may be methylated, and which particular cytosines are methylated influence whether the gene is active. When a liver cell divides, it produces new liver cells, which implies that it remembers which genes should be active and which inactive in a liver cell. Other genes would be active in a neuron or a skin cell. An important way that this memory is stored is in the methylation patterns, which are carefully conserved when the DNA is replicated. Stem cells, which famously can develop into any type of cell, have all these memory mechanisms initiated blank (Maynard Smith and Szathmary 1999, 113–114).

All this logic is implemented in a Post-like production system. For example, a repressor, which is a protein that binds to DNA, represses the expression of a gene by matching a specific pattern on the DNA and attaching itself where it is matched. This then has some effect that suppresses expression of the DNA, such as covering up a nearby location on the DNA where a promoter might otherwise match to induce expression.

Development proceeds as a program execution, starting with a single cell. Molecular computations of the kinds just outlined determine which genes are expressed, and the cell then duplicates. More molecular computations occur, and each of the two cells duplicate again. At precise duplication points, the cells begin to specialize, to become liver and brain and muscle cells. Everything proceeds according to logic, as specified in the DNA program. The DNA program continues its clockwork execution for 70 or more years of a person's life, the liver cells, for instance, remembering all that time how they should act to be liver cells. They act correctly by executing a complex program of molecular interactions. The working memory of the program (effectively the tape of the Turing machine, or more aptly, the collection of strings and productions in the Post machine) is stored in the proteins present in the cells, in nucleotide sequences present in the cells, and in modifications of these proteins and nucleotides.

"This book is the deepest, and at the same time the most commonsensical, approach to the problem of mind and thought that I have read. The approach is from the point of view of computer science, yet Baum has no illusions about the progress which has been made within that field. He presents the many technical advances which have been made—the book will be enormously useful for this aspect alone—but refuses to play down their glaring inadequacies. He also presents a road map for getting further and makes the case that many of the apparently 'deep' philosophical problems such as free will may simply evaporate when one gets closer to real understanding."
—Philip W. Anderson, Joseph Henry Professor of Physics, Princeton University, 1977 Nobel Laureate in Physics

"Eric Baum's book is a remarkable achievement. He presents a novel thesis—that the mind is a program whose components are semantically meaningful modules—and explores it with a rich array of evidence drawn from a variety of fields. Baum's argument depends on much of the intellectual core of computer science, and as a result the book can also serve as a short course in computer science for nonspecialists. To top it off, *What Is Thought?* is beautifully written and will be at least as clear and accessible to the intelligent lay public as *Scientific American*."
—David Waltz, Director, Center for Computational Learning Systems, Columbia University

"What's great about this book is the detailed way in which Baum shows the explanatory power of a few ideas, such as compression of information, the mind and DNA as computer programs, and various concepts in computer science and learning theory such as simplicity, recursion, and position evaluation. *What Is Thought?* is a terrific book, and I hope it gets the wide readership it deserves."
—Gilbert Harman, Department of Philosophy, Princeton University

"There is no problem more important, or more daunting, than discovering the structure and processes behind human thought. *What Is Thought?* is an important step toward finding the answer. A concise summary of the progress and pitfalls to date gives the reader the context necessary to appreciate Baum's important insights into the nature of cognition."
—Nathan Myhrvold, Managing Director, Intellectual Ventures, and former Chief Technology Officer, Microsoft

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142
http://mitpress.mit.edu

0-262-02548-5