CHAPTER 5 Computers vs. Cells (and Minds)

There have been many attempts to understand natural symbol systems that draw upon our knowledge of artificially constructed digital computing systems. Digital computers are the latest in a long line of human artifacts that have been used as analogical source domains for the human mind. They certainly have a more variegated and adaptable architecture than steam engines or water fountain piping. However, computers share with other products of human manual and mental skill a dependence on humans for their birth, growth, maintenance, and reproduction. Because they did not have to emerge 'by themselves' from the non-mental (or non-biological) world, they can take advantage of modes of organization that are different from those employed by naturally-occurring systems like cells and people. When a disk drive fails, a human can replace it; when a program or an operating system crashes, a human can edit, recompile, and restart it (or more commonly, curse, restart, and remember to avoid). The much more stringent conditions on biological and cultural evolution don't allow 'crashes'--at least among the survivors--of which there always have to be some. Modern life can be traced back through an unbroken reticulum of cell generations that haven't crashed since the origin of life. Similarly, cultural evolution depends absolutely on an unbroken stream of new individuals who are capable of and who actually did develop peculiarly human activity patterns in their brains.

Having a smart maintainer, though, is hardly an impediment to making an artificial cognitive system (or artificial life); instead, it surely must open up new routes to intelligence. And many different routes have been tried. So far, it has not yet been possible to construct artificial systems with the same degree of resilient common sense as the natural ones have; and larger systems approaching a fraction of the size of a human cognitive system become harder and harder to maintain. Finally, none of these systems are even vaguely close to self-reproduction. On the other hand, it *has* been possible in a few short decades to construct and debug systems that have far surpassed humans abilities in many different specialized areas. And here, we have arrived at the main point--obvious, but often neglected--that modern digital computers were designed to be programmable for human ends, not to maintain themselves and reproduce. Our goal here is to describe the architecture that makes this possible, explain why it is that way, and then show how it differs materially from the architecture introduced in the previous chapter.

The attempt to model intelligence with computers has generated a large amount of peevish controversy (e.g., Searle (1980), Fodor and Pylyshyn (1988), and responses to them) as the focus of interest and funding has see-sawed back and forth between symbolic artificial intelligence and more neural-like approaches. Despite continual improvements, symbolic artificial intelligence has lost the self-assuredness it had in the 1970's. As neuroscience began to grow explosively, some of AI's excitement, funding, and even researchers were captured by the rebirth of connectionism and neural modeling in the mid 1980's. After a decade of connectionism, though, some of the complaints originally directed against AI--particularly the difficulty of generalizing beyond restricted 'microworlds'--have re-surfaced. And some of the hard problems tackled by symbolic AI were not necessarily solved better, but often temporarily put aside. This is particularly true of tasks that require establishing variable-sized temporary representations--like problem solving, scene understanding, and language comprehension. The point of the present analogy, in fact, is to try to come up with more neural-like solutions to postponed, but very real problems. Symbol-use in computers will be our last diversion. The comparison with cellular symbols provides a fresh perspective on these perennially rehearsed debates.

The idea and realization of a digital computer

In returning to the earliest modern description of the idea of a computer--before it was possible to actually make one--it is interesting to find that the *source* domain for the human/computer analogy was human thought; Turing envisaged his abstract computer as a mechanized, idealized version of a human consciously following rules with pencil and paper. The analogy was turned around only later, after the development of higher level programming languages in the late 1950's. We will start with the original Turing idea and then see how most of its key aspects have been retained fifty years later.

The abstract Turing machine has a linear tape with individual cells on it each capable of holding one symbol, a head that can hold a symbol of its own

and that can read and write tape symbols, and then the operating instructions--a table that associates each possible input pair (current head symbol, current tape symbol) with an output triple (new head symbol, new tape symbol, left or right move). The original Turing machine is local and serial, only executing one action (reading then writing/moving) at a time. It reads and writes symbols with digital effects (noiseless, ideal symbols). Finally, it assumes that the tape is inert and one-dimensional--that is, if the head writes a symbol at a particular tape cell at one time, it is guaranteed to find the same symbol there the next time the head gets back to that cell. The tape symbols don't interact with the world or with each other. Because the symbols in a Turing machine live completely in their own world, they will have an arbitrary relationship (of the symbol-object kind) with anything they might be taken to stand for in the world or in the mind of the user.

The idea of detaching the symbols and symbol-processor from anything in the world was not new; it was a thread running through early 20th century analytic philosophy and philosophy of science. But the deceptively simple local serial symbol processing machinery turned out to be quite powerful and general. A Universal Turning machine is one that can simulate any other Turing machine given a description of the other one on its tape--in essence, it is a programmable Turing machine. Universal Turing machines can be very small; Marvin Minsky found one with only 28 entries in its input/output table (see Haugeland, 1985). A Universal Turing machine turned out to be exactly as powerful as any computer in terms of the stepwise solvable problems that it could solve (though certainly nowhere as efficient as a modern computer). Adding additional parallel heads and tapes doesn't change what can be solved either (Hopcroft and Ullman, 1979).

The Turing machine is an abstract idea. To make a real physical computer, some additional key engineering ideas were needed. One key requirement is a mechanism for generating the digital effects assumed by Turing in the reading and writing of symbols. This is no more (or less) than a simple categorization whereby a class of similar tokens (examples) are taken to be exactly equivalent to the category exemplar. A particular token of a tape symbol (or head state symbol) must be recognized or written just as surely as any other token of that symbol (or head state). In a real microprocessor or disk drive, symbols are coded as binary patterns (each bit is OFF or ON--e.g., 0 or 5 volts). But real voltages in the millions of gates of a microprocessor never measure exactly 0 or 5 volts; they have to be actively classified as one or the other. When a voltage on a particular line is a little off--say, 4.97 volts--it must nevertheless have exactly the same effect as 5 volts; similarly, voltages a little above or below 0 must act exactly like true 0. Elaborate mechanisms have been developed to maintain this digital idealization despite conduction losses, electrical noise, imperfect transistors, stray capacitance which rounds pulses, disk imperfections, and so on. Stochastic computation is possible; but it is fair to say that virtually all real work (even on stochastic computation!) is done with non-stochastic computers that are characterized by an absolutely strict maintenance of the digital idealization. Modern microprocessors can perform billions of these perfect classifications per second for weeks at a time and reliably store and retrieve trillions of ideal symbols from a disk without a single error.

The other key physical ingredient of a computer is the provision for long one-dimensional chains of symbols that can serve as the substrate for a digital-effects recognition and writing device--namely RAM and disk space. As in the original Turing idea, the crucial property of these symbol chains is that the symbols in them do not directly interact with each other. The recognition device in the CPU can count on finding a symbol where it was last put; it doesn't have to worry about between-symbol interactions occurring while it's 'not looking'. There are, of course, some transitional cases at the boundaries of a computational system. For example, an analog-to-digital (A/D) sound card may turn smoothly changing voltages from a microphone into digital bits that it inserts into specialized registers, which may then be addressed as memory locations by the CPU. A mouse or keyboard has similar effects. Or a disk drive may be intelligent enough to move some bits to a new location when the old location gets flaky, while making it appear to the CPU that nothing happened. But the main point is that in the great bulk of conventional computation, the string of symbols in memory is one-dimensional; it does not routinely 'fold up' and interact with itself. Symbols only interact with each other via the CPU. All of the convenient higher level language abstractions depend on fundamentally on an architecture like this.

Modern computers typically have a slightly different, more efficient architecture than a Turing machine. Instead of a head that can only move one cell at a time, the CPU 'head' can go directly to any location on the memory 'tape' in a small

number of steps. Program and data are then easier to separate. There is a specialized part of the CPU (stack, registers) where data is temporarily put when it is actively being worked on, and high speed, temporary cache memory. Finally, there are a variety of things that are done in parallel to speed things up (for example, at an if-A-do-X-if-B-do-Y branch point, both X *and* Y may be done concurrently with the A vs. B test, and then the unneeded one of X and Y discarded). But none of these things fundamentally changes the idea of a computer as initially outlined by Turing; this more practical von Neumann architecture is not capable of computing things that a Turing machine can't compute. And in the best of all programmer's worlds, memory still appears to be one (really) long 1-D string.

Computers vs. cells as symbol-using systems

When we turn to cells ('life machines'), we can point quite uncontroversially to long symbol strings that are accessed with digital effects, and that behave as inert one-dimensional objects. But these symbols are used in a remarkably different way than they are in computers (Alberts et al., 1994). First, let's see why the symbol strings are similar. Each living cell contains somewhere between 1 and 200 megabytes of DNA code. Messenger RNA sequences are transcribed from this permanent DNA store and are then recognized by the cell during the process of protein synthesis to contain three-nucleotide codons (words). Each of the 4 nucleotide bases--adenosine (A), thymine (T), guanine (G), and cytosine (C)--can be described as having two binary features or bits--"long/short" and "strong/weak". A and G (purines) are longer than C and T (pyrimidines); but then G and C form a stronger bond (3 hydrogen bonds) than A and T (two hydrogen bonds). Thus, A, for example, is [+long] [-strong]. There are therefore $4^3 = 64$ possible three-nucleotide words. After excluding three stop codons, the remaining 61 codons are grouped into 20 different sets that are actually distinguished (thus, 5 out of the 6 bits per 3-nucleotide codon would have more than sufficed since 5 bits can already code for 32 different things if there is no redundancy).

Thought of as static structures, it is easy to forget the underlying continuity that is being categorized to generate the ideal digital effects that occur when the cellular symbols are copied or used. For any actual recognition event in which a nucleotide base is being paired with its complement (e.g., a G recognizing a C), the binary features that direct this pairing will only be approximately present. The units in a long chain of a polynucleotides are in constant *relative* motion so that the long crevice presented by the short C in a strand-to-be-read is only approximately 'long'--any given C will be bulging, receding, and rotating a bit during the recognition event, which will nevertheless result in a full G being matched to it. The recognition accuracy is not quite as high as a computer. But but it still has to be quite good; there are several hundred megabytes of code in many eukaryotic cells, and a single bit error can sometimes disable a protein.

These symbol strings also have the property of being inert and one-dimensional. This means that they don't generally interact with themselves at a distance. It is true that eukaryotic cells, which have a lot of DNA, have developed archival strategies to help them manage their extremely long DNA molecules. For example, eukaryotic DNA is often wrapped around histone cores to form a beads-on-a-string structure which then condenses into a 6-bead-thick strand, which is itself then condensed into even higher order loops (Alberts et al., 1994, pp. 342-346). And there are DNA twisting and untwisting and untangling enzymes (more on all this later). But this uniform condensation for the most part ignores the sequence; and DNA can even be read while still wrapped around the histones. Barring various mutational disasters, the recognition apparatus can count on finding a particular DNA base just where it was in the sequence the next time it comes by.

As with symbol chains in a computer, we find exceptions around the edges; for example, the tertiary structure of mRNA's may control whether or not they are interpreted (for now, we will postpone discussion of the important structural RNA's--tRNA, rRNA, 7S RNA, cutting and splicing RNA's, nucleolar RNAs). But, in general, most mRNA's are disentangled and fed through the ribosome, one codon at a time, and the particular amino acid chosen to add to the chain on the basis of the current codon is not affected by codons at a distance.

But when we turn to how cells actually *use* these code strings, we find that they have put them to work in a remarkably different fashion than computers. Instead of reading code for the purpose of *operating on other code*, cells use the code to make proteins (especially enzymes), which they then use to maintain a metabolism (see Fig. 13). Proteins are constructed by simply bonding amino

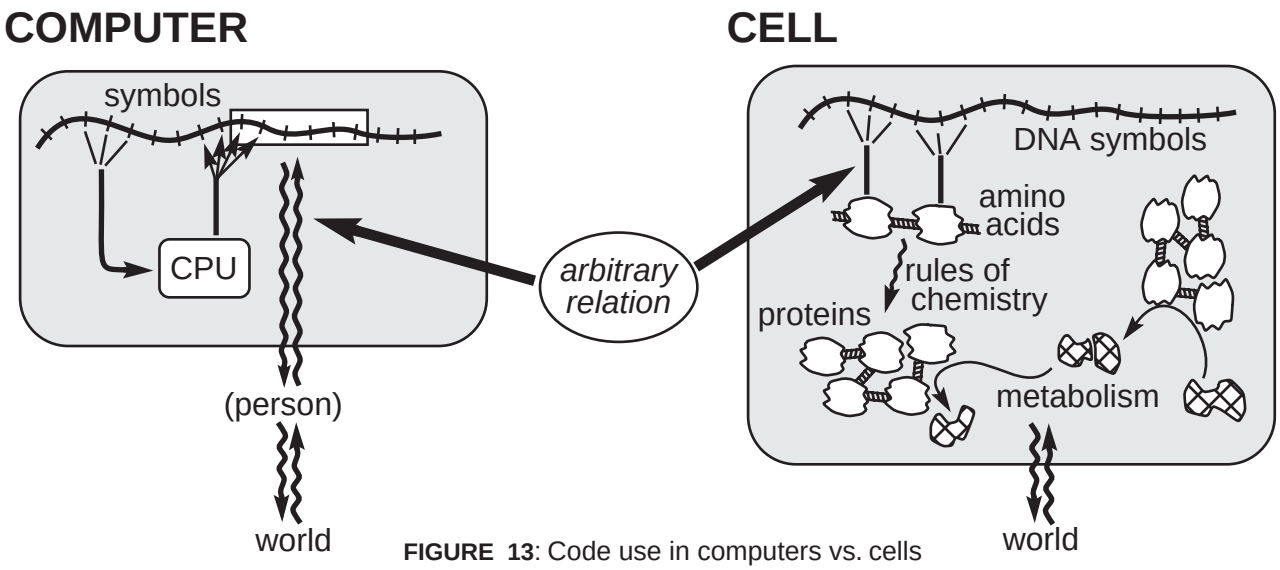

FIGURE 13: Code use in computers vs. cells

acids into an initially 1-D chain that is parallel to the recognized codons (words) in the messenger RNA chain and then allowing it to fold up in 3-D. Protein folding is a particularly intimate kind of folding that brings specific units into tight contact with each other to form a unique, densely packed 3-D structure for each particular sequence (quite different from the uniform warehousing of DNA described above). Despite the initial linearity of the output string, the result of reading the code can in no way be construed as writing symbol chains of the same kind that were read, as occurs in the operation of a computer; in fact, there is no code-writing in the cellular 'life machine' at all, as we have previously noted.

Proteins are complex molecules, each containing thousands of atoms in a precise 3-D arrangement that is crucial to their function. The DNA sequences in the genome, however, constitute only a trivial portion of what would be required to explicitly specify the 3-D structure of a protein; a single gene typically contains only a few hundred bytes of information. This information goes such a long way because it depends for its interpretation on the existence of elaborate geometrical constraints due to covalent chemical bonding, weak electronic interactions, the hydrophobic effect, the structural details of the 20 amino acids, and so on--a large set of 'hard-wired' effects that the cell harnesses, but cannot change. Once the amino acid chain has been synthesized, its self-assembly (folding) is directed by these prebiotic, non-symbolic chemical constraints.

There are of course, some switch-like digital effects in proteins; for example, an enzyme may be switched on and off by a particular cofactor. But then isolated digital effects are found all over, at many levels. There is a kind of digital effect when an ant falls into the conical sand well of an ant lion--once the ant gets 'close enough', it always ends up in the jaws. This is not a computer. And neither is a cell. There is no sense in which the basic function of a protein can be said to depend on it being a chain of non-interacting digital subunits. The subunit amino acids of a protein must directly interact in a non-digital way *without* the intervention of a CPU to realize the intended function of the whole protein.

Computer-like cells and cell-like computers

A brief inspection of Figure 13 reveals that there is a fundamentally different locus of symbol-object arbitrariness in the computer and the cell. In the computer, there is an arbitrary relation between particular segments of code and their referents in the world. This has been widely trumpeted as the fundamental advance brought about by the computational metaphor (e.g., Pylyshyn, 1984). It doesn't matter whether a particular symbol in a memory location stands for a hamburger or some part of a sub-sub-lemma in a computerized proof of the four-color theorem. The hamburger program will make the computer do the right thing for hamburgers just as the four color proving program will make it do the right thing in that much more abstract domain.

In the cell, by contrast, the main symbol-object arbitrariness is between code-like DNA and RNA, on one hand, and object-like amino acids on the other. There is much less representational flexibility in the rest of the system. For example, in protein assembly, the cell can't control what a

particular amino acid is going to do in any given protein. It is stuck with the hardwired chemistry of the peptide linkage, van der Waals forces, the hydrophobic effect, and so on; it can only vary the input sequence and see what comes out. Similarly, there is little flexibility in defining the relationship between a folded protein and the rest of the world. If a folded protein doesn't have the right shape to stick to (some molecular component of) a hamburger, then there is nothing the cell can do except try a different sequence. In fact, at the cellular level, the ability to establish different maps across the arbitrary connection has not been 'exercised' much at all--the genetic code is almost universal.

As in the previous chapter, it helps to imagine what each system would be like if it used symbols more like the other one. First we can ask about a cell that was made more like a computer. This would be a cell that was actually able to write DNA code as a result of reading DNA according to a rule table. This is not nearly as far-fetched as our last chapter scenarios; in fact, there has been some progress toward trying to solve hard computational problems with various artificial DNA systems (Adleman, 1994; Ouyang et al., 1997). Our project here, by contrast, is to turn a cell into a stand-alone computer. The most straightforward version of a computing cell would be a DNA-based Turing machine. Cells can already 'write DNA' in the minimal sense of DNA replication or reverse transcription (RNA => DNA); but this makes for a particularly boring Turing machine. To compute something useful, there would minimally have to be the ability to read one DNA symbol and then write a *different* one. There are a class of DNA-maintenance enzymes that fix up various aspects of DNA after the slings, arrows, and mutagens of a hard day (e.g., tacking lost purines back on, unsticking thymine dimers, and others). Overly aggressive forms of these can actually introduce substitutions into a DNA strand while attempting to 'repair' it. There are also a set of 'DNA-editing' enzymes with similar effects. It might be possible to generate a fuller set of these by directed mutation. The modified enzymes would also have to maintain some minimal state themselves. With a large enough set, it might be possible to generate a transition table for a useful, or even a Universal Turing machine. With the latter, the program could be generated with a DNA synthesizer. It is not clear that it would ever be possible to make a useful computer this way. Turing machines are slow, and DNA repair and editing enzymes operate so much more slowly than silicon gates do that it seems unlikely that even the truly massive parallelism possible with replicating cells would ever allow us to catch up to the ever faster and easier-to-program CPU's that make it to the desktop each year. Nevertheless, it is pretty clear what would have to be done to turn a cell into a computer. Perhaps the most interesting unanswered question this brings up is why cells haven't come up with this themselves.

Conversely, we might imagine turning a computer into something more like a cell. In this case, we would have to add on an elaborate 'chemical soup'--imagine a large electronic network into which activity patterns could be injected and where they would propagate around and interact with each other. Instead of the computer reading code and writing code, it would only read code and then use it to assemble patterns in this large add-on device. The hamstrung 'programmer' would be unable to directly control the many properties of the add-on (e.g., how the injected activity patterns interacted with each other) and could only use code to control the sequence of patterns that were introduced--a greatly reduced job description.

From the perspective of these two scenarios, it is pretty clear why computers have been designed as they have been. For the purpose of helping out humans solve mathematical proofs, control ignition timing, simulate weather, model neural circuits, transcribe speech to text, simulate the sound of an overdriven vacuum tube guitar amplifier, reformat office memos, and many other amazingly diverse things, you *certainly* wouldn't want a computer to use its code more like a cell does. A computer whose code was completely entangled with a large, recalcitrant dynamical system would be *very* difficult to program. The idea of making a clean break with the real world and letting symbols operate on other symbols without any outside interference is a wonderful and useful idea. These kinds of systems have turned out to be tremendously more practical than early twentieth century philosophers could ever have conceived (imagine explaining to Frege that even toasters are now controlled by formal systems that process symbols at a rate of millions per second).

Computers vs. cells as models of the mind

But I hope to eventually convince you that those philosophers and Turing were probably wrong about disconnected symbol processing as a model of human thought (despite, ironically, thought

being the source domain for the original analogy). Figure 14 is similar to the previous figure but

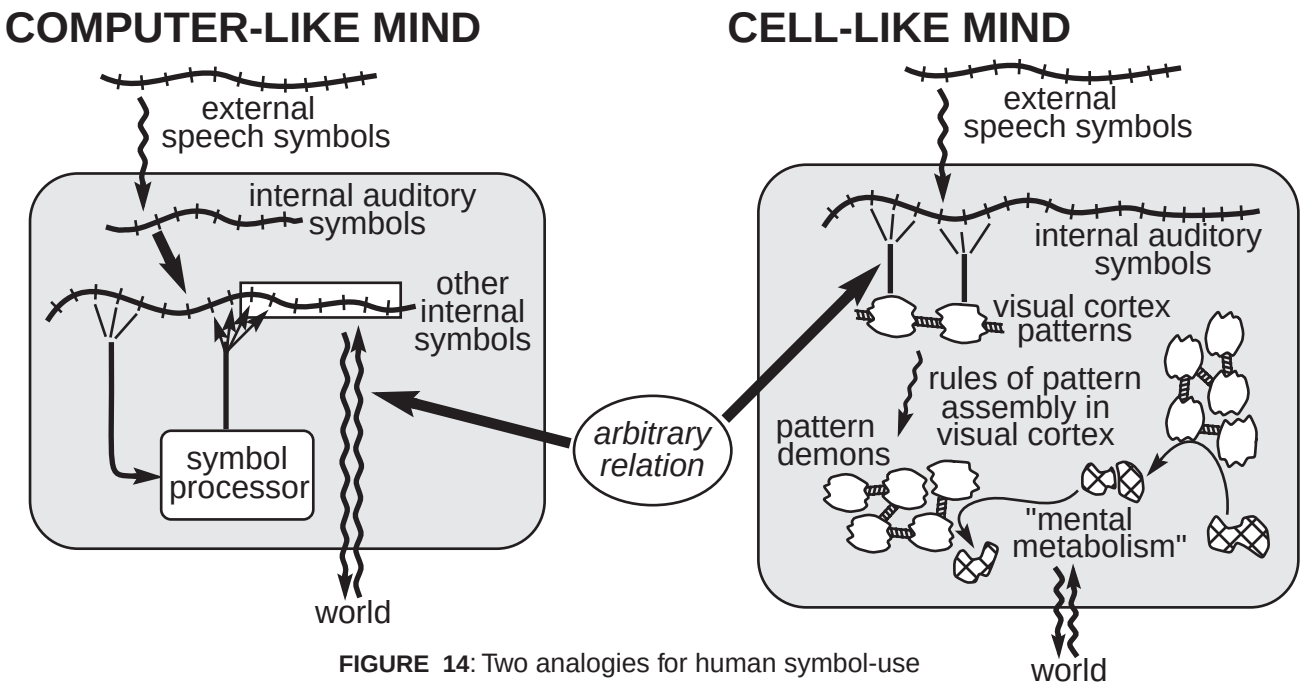

FIGURE 14: Two analogies for human symbol-use

instead of comparing a computer to a cell, it more explicitly lays out a comparison between the computer and cell *analogies* for language understanding in the human mind. The two analogies agree on the existence of external speech sounds as well as on internal categorized copies of them. After that, the models are completely different. The computer analogy postulates a much larger set of internal symbols not having anything directly to do with speech sounds, and, just beyond specialized sensory modules, a symbol processing core that operates on the non-speech sound symbols (e.g., Fodor, 1983). This is all, of course, supposed to be implemented in neural circuits. But the neural symbols are supposed to retain certain crucial computer-like characteristics; the digital aspects (usually called "causal properties") of the symbols that are actually used by the internal symbol processing device are supposed to be causally detached from the world, just as in the original Turing idea (Fodor, 1983; Pylyshyn, 1984).

Simulation, production systems, and cells

At this point, those with some knowledge of artificial intelligence are likely to be surprised or outraged that we have hardly said anything so far about higher level virtual machines or high level program architectures--the things that are usually singled out for as models for a new brain operating system. They would complain that no one today tries to compare the grungy low level details and fast clock speed of a von Neumann CPU to a brain. Turing did initially say his machine was like conscious rule-following by a human (a mathematician). But after the development of compilers and higher level languages, the Turing-machine-like lower level of a modern computer running, say, a production system, is now merely an implementational detail; the computational architecture of the *production system* (e.g., Anderson, 1983; Newell, 1990) is what is supposed to be important.

One problem with these arguments is that they forget just how good computers are at simulating other systems (cf. Haugeland, 1985). Computers can be used to prove mathematical theorems, which does look quite a lot like symbol processing; but they can also be used to simulate the weather. Most people would not want grant symbol processing to the actual physical weather. There is a smooth gradation of intermediate examples. Computers can simulate a serial production system; but they can also simulate a parallel production system (Rosenbloom et al., 1987). They can simulate an abstract neural net with simple units (Hopfield, 1982); but then they can also simulate a realistic neural net with neurons that have Hodgkin-Huxley conductances and branched dendrites with hundreds of compartments (Koch and Segev, 1998). They can simulate an abstract neural net simulating a parallel noise tolerant

production system (Touretzky and Hinton, 1986). All this flexibility critically depends on being able to read and write non-interacting symbols--something that is often *not* true of the system being simulated.

The problem is that since computers are such good simulation machines, it seems hard to argue that what is being done in the *system being simulated* is computation without being able to see the only kind of physical hardware we know for sure is actually capable of computation--namely, something like a von Neuman machine with symbol reading and writing, and non-interacting symbol chains. As with cells, the argument is that the 'bottom level' shouldn't be ignored. Most of what happens in the world isn't computation (arguments about hypothetical computers made out of pecking pigeons aside). If it was, we wouldn't have had to work to make usable computers. Computers have a very specific architecture. It seems different than the architecture of the brain.

But what about production systems--even if they *are* simulated, and therefore not positively a pure form of computation? Don't they have the right architecture anyway, philosophical disputes about what precisely computation is aside? Leaving aside also, for the moment, the brain, let's compare a *production system* to a *cell*. A number of aspects of the architecture of cellular metabolism are, in fact, very much like a production system. The enzymes (productions) of metabolism operate on their chemical substrates (objects in working memory) in a cytoplasm (shared memory space), which requires that the enzymes have a great deal of specificity to avoid inappropriate interactions. Different enzymes see many of the objects in working memory in parallel, but only operate on the ones that they 'match'. As in some kinds of AI production systems, enzymes can take other enzymes as substrates (productions modifying other productions). There is even a certain amount of conflict resolution since a substrate can only be in one place at a time. There can be multiple copies of particular objects, again like in a production system. To help with scheduling, the cell compartmentalizes the cytoplasm with some areas reserved for active synthesis and other areas with more long term storage of objects (working memory vs. long-term memory).

There is a key difference, though, between a standard production system model and a cell. The code in the cellular system is used strictly to make the enzyme 'productions' (see Fig. 15). Once the

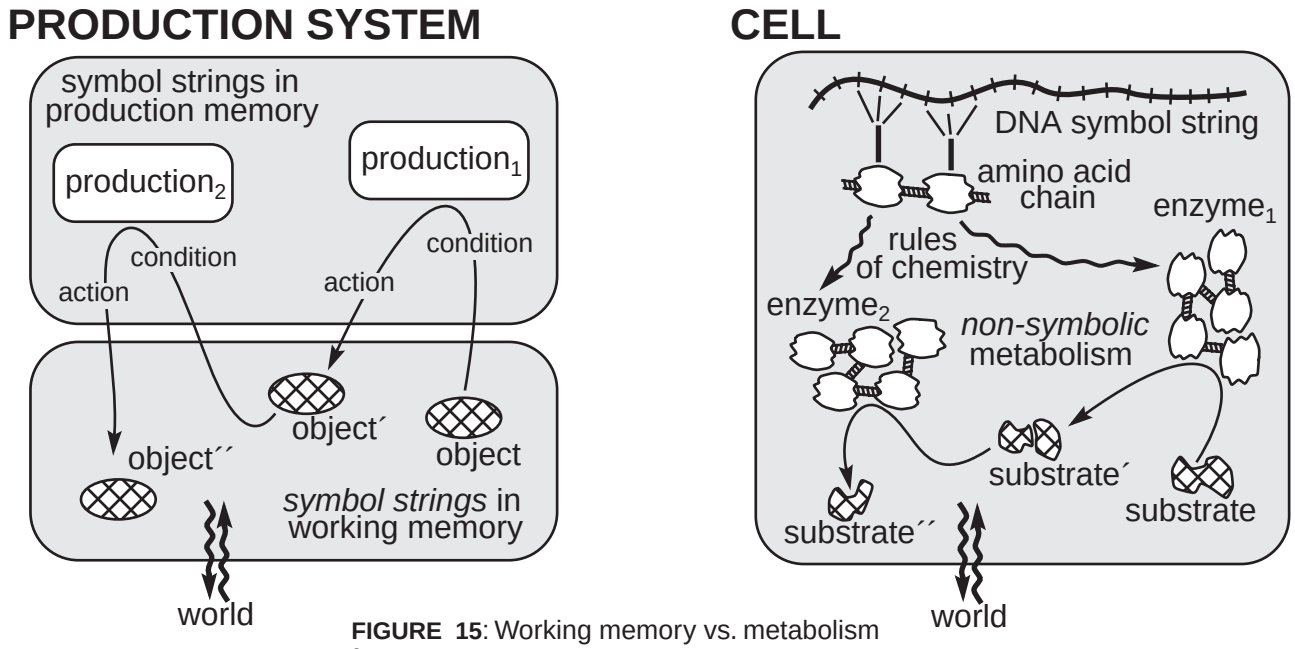

FIGURE 15: Working memory vs. metabolism

'productions' are made, they fold up and operate primarily in a non-symbolic milieu and on non-symbolic things in the cytoplasm. Instead of recognizing code-like features of their substrates, enzymes recognize the distributed surface shape of their substrates. Most of the substrates themselves--for example, small molecules, polysaccharides, polypeptides--are not code strings. The cell has no access to the convenient symbolic names we have given these things; they have to do hard work to recognize and distinguish their three-dimensional surfaces. It is true that some proteins bind to DNA code and can recognize code sequences on it; but the purpose is almost entirely to decide which part of the DNA

should be turned into non-code-like folded enzymes.

A persistent and extremely cooperative AI supporter might say, well then, we could model the key recognition events between the enzyme-like productions and the working memory objects with a more complex non-binary-like surface-shape recognition mechanism rather than with productions that simply look for a few binary features. At this point, I would grant that we would finally be getting close to the beginnings of a *simulated* model of a cell (and, we will soon argue, a simulated human brain). Once the complex constraints on how enzymes actually get their shapes and how they actually modify their substrates begin to be added in, however, the system would immediately become much less code-like, and much less programmable--just like a real cell. All these additional constraints are what makes the cellular code go so far; the DNA code for an entire living, reproducing, self-maintaining *E. coli* bacterium is a fraction of the size of the code for Microsoft Word. Most of us would agree that life is a better bang for the byte.

So Turing was a little wrong about his initial idea of what a computer was. It is actually a general purpose simulation machine to help humans rather than a human thought simulation machine. Most thoughts (even of mathematicians) are far too attached by non-arbitrary connections to the world to be very good for general purpose, computer-style simulation. Humans are very bad at most of the things that computers are used for. Instead of trying to defend outposts of human superiority against computers that are using obviously different methods than humans (chess playing computers are an excellent example), we should recognize that the human-thought-like parts *omitted* from computers are exactly the things that make them better than humans for simulation.

Why aren't there more naturally-occurring cell-like symbol-using systems?

Living cells are all based on the same kind of symbol-using system that, as far as we can tell, came into existence soon after the earth cooled enough for there to be sedimentary rocks. The basic ideas are:

- use mostly pre-existing, pre-biotic amino acid meaning units
- use 1-D symbol strings to control only the order of assembly of meaning units
- bond the pre-systemic meanings into chains to exploit the rules of chemistry via chain folding (non-adjacent interactions)
- arrange a production system-like metabolism controlled by thousands of bonded-together meaning chains
- use material (mRNA, tRNA, rRNA) halfway between a symbols (DNA) and meanings (protein) to help decode symbols

At first glance (especially to someone without detailed knowledge of molecular biology), this doesn't seem that hard. An immediate question is, why, if it was successful enough to coat the entire earth's surface with megabytes of DNA code per square millimeter hasn't a parallel system of this kind naturally appeared again and again?

One answer is that once a cellular living system came into existence, it was able to eat up the early stages of all the other ones that ever 'tried' to come into existence afterward, at least at the single-cellular level. But, what about symbol-using systems at other, higher levels of organization? (lower levels are possible, but perhaps less likely since cellular symbols are already single molecules with each symbol segment containing only a handful of atoms). We can consider long symbol-chains and symbol-use in both biological and geological contexts--e.g., non-human-brain tissues, animal social groups including ants, the geology and hydrology of streams, the slow convective currents in the earth's mantle, volcanos, and so on.

A moment's thought brings us to the conclusion that these other systems don't have the proper connectivity or interrelatedness, or crowdedness to make something like a cell work, process the code chains fast enough to keep everything assembled (proteins are assembled at the rate of a couple of amino acids per second--but they also break down relatively rapidly), and in general, prevent attack by dissipative forces of the prebiotic soup. Certainly it *is* possible to dissect out many of the different reactions of cellular metabolism and run them each individually in a test tube. Like the brain-in-multiple-vats question of Dan Dennett (could brain function survive neurons being separated into many separate but intercommunicating vats?), the question is whether we could implement "cell-in-multiple-tiny-vats". This, of course, is exactly how biochemists and molecular biologists figured out how cells work. But, in a real cell, these things are all crowded together in an amazingly intimate fashion; codon (word) recognition for cellular mRNA code streams takes place with thousands of irrelevant

constituents of the cytoplasm constantly crashing into the ribosomal apparatus, the code chain, and the amino acid meanings (Goodsell, 1991). The crucial point, however, is that is it not possible to 'uncrowd' all these reactions and reaction-controllers into separate compartments and still get the thing to work right, at least with enzymes the way they are now. For example, time constants of reactions are intimately interwoven into the mechanism. The cell in multiple vats (or perhaps, the "really small Chinese gymnasium/room" argument) won't work for seemingly trivial reasons.

Now this might seem a mere cavil; wouldn't it work if we just got all the reactions right and made different stable intermediates that could sit around longer while we more leisurely transferred them between bins? Perhaps, but remember that this thing has to actually live in the world without a biochemist if we really wanted it to be freely-living and -reproducing like a real cell. As we noted, even the relatively stable parts of the cell, like DNA, are actively maintained--millions of base pairs are repaired every day. There is no magic barrier here--just a complex set of constraints on a dynamical system made out of a soup of reasonably stable covalently-bonded molecules. We don't really have an explicit, large-scale theory of how the dynamics of cells work, or exactly what it is about that dynamics that is lacking from streams or other geological systems. But we have very little difficulty distinguishing living cells from other non-living stuff in the world. For now, it seems reasonable to think that making such a system demands a certain connectedness and crowdedness, for lack of better terms, that the most dynamical regimes just don't have. [5,700]