One Hundred Billion Lines of C++

Brian Cantwell Smith[1]

The year is 2073. You have a job working for General Electric, designing fuel cells. Martian have landed. One stands over your desk, demanding to see what you are working on. You sketch a complex combustion chamber. Next to an input port, on the left side, is the word 'oxygen,' with an arrow pointing inwards. On the right is a similar port, with the word 'hydrogen.' "Amazing!," says the Martian to a conspecific, later that day. "Earthlings build symbol combustion machines! I saw an engineer designing one. He showed me how the word 'oxygen' would be combined with the word 'hydrogen' in a weird kind of mixing chamber."

The Martian is confused. That was a *diagram* for a fuel cell, not a fuel cell itself. The word 'oxygen' was a *label*. What you meant to funnel into the chamber—to belabour the obvious—was *oxygen gas*, not (a token of) the word 'oxygen.' Words entering chambers makes no sense.

Far-fetched? Perhaps. But in this paper I argue that the debate between symbolists and connectionists founders over a troublingly similar error. Perhaps not quite as egregious—but a misunderstanding, nonetheless. It is as if John Searle had wandered into a hacker's office, looked over her shoulder at the program she was writing, seen lots of symbols arranged on the screen, and concluded that the resulting system must be symbolic. Searle's inference, I claim, is no more valid than the Martian's.

1. Background

A glimmer of trouble is evident in the way the debate is framed. Both positions consider only two kinds of architecture. On one side are traditional von Neumann architectures, of the sort imagined in "good old fashioned ai" ('gofai,' to use Haugeland's term). These systems are assumed to be constructed out of a set of atomic symbols, combined in countless ways by rules of composition, in the way that is paradigmati-

---

 This version appeared in *Cog Sci News*, Lehigh University, 1997; an expanded version will be submitted to *Minds and Machines*. Thanks to Irene Appelbaum and Güven Güzeldere for comments on an early draft.

Cognitive and Computer Science Lindley Hall, Indiana University
Bloomington, Indiana 47405 usa smithbc@indiana.edu

cally exemplified by the axioms of a first-order theorem prover. On the other side are connectionist (or dynamic) systems, composed instead of a web of interconnected nodes, each dynamically assigned a numerical weight. For purposes of this debate, it seems as if that is all there is. Some writers[2] even take the first, symbolic, model, to be synonymous with computation *tout court*. So they frame the argument this way: that cognition is (should be understood as, will best succumb to analysis as, etc.) a *dynamical* system, not a *computational* system.

What happens to real-world programming in this scheme—the uncountably many network routers and video games and disk compression schemes and e-mail programs and operating systems and so on and so forth, that are the stock and trade of practicing programmers? Which side of the debate are they on? Most people, I take it, assume that they fall on the symbolic side. But is that so? And if so, why are such systems never mentioned?

It cannot be because such programs are rare. As they say on npr, "let's do the numbers."[3] Sure enough, some combinatorial symbolic systems have been constructed, over the years, of just the sort envisaged (and defended) by Fodor, Pylyshyn, and others on the symbolic side of the debate.[4] Logic-based programs, theorem provers, and knowledge-representation systems were early examples. Soar is a more modern instance, as is the cyc project of Lenat and Feigenbaum. Perhaps the category should even be taken to include the bulk of expert systems, case-based reasoners, truth-maintenance systems, and diagnosis programs. What does this come to, overall? Perhaps 10,000 programs, at a maximum? Comprised of ten million lines of code?

But now consider the bulk of real-world programming. Think of e-mail clients, of network routers, of word processors and spreadsheets and calendar programs, of operating systems and just-in-time compilers, of Java applets and network agents, of embedded programs that run the brakes in our cars, control traffic lights, and hand your cellular telephone call from one zone to the next, invisibly, as you drive down the interstate. Think, that is, of *commercial software*. Such programs constitute far

---

[2]See e.g. Port, Robert and van Gelder, Timothy (eds.), *Mind as Motion*, Cambridge, Mass.: MIT Press (1995), or van Gelder, Timothy "Computation and Dynamics," *Journal of Philosophy*, …

[3]On the program "Marketplace" produced at usc.

[4]See for example Pinker, Steve, and Mehler, Jacques (eds.), *Connections and Symbols*, Cambridge, Mass.: MIT Press, 1988.

and away the mainstay of computing. Again, it is impossible to make an exact estimate, but there are probably something on the order of 10^{11}—one hundred billion—lines of C++ in the world.[5] And we are barely started. In sum: symbolic AI systems constitute approximately 0.01% of written software.

By themselves, numbers don't matter. What I want to do is to use these facts to support the following claims:

1. Within the overall space of possible computational architectures, the vast majority of commercial software—which is to say, the vast majority of software, period—is neither "symbolic," in the sense defended by Fodor and Pylyshyn, nor "connectionist," in the sense defended by Smolensky, but rather some third kind entirely;
2. The only reason for thinking that commercial software is symbolic is a confusion between a *program* with the *process* or *computation* that it specifies (something of a use/mention error, not unlike that made by the Martian); and
3. In order to understand how such a confusion could be so endemic in the literature (and have remain so unremarked), one needs to understand that the word "semantics" is used differently in computer science from how it is used in logic, philosophy, and cognitive science.

In a sense the moral comes to this: the "design space" of possible representational / computational systems is enormous—far larger than non-computer-scientists may realize. Both the traditional "symbolic" variety of system, as imagined in gofai, and the currently-popular connectionist architectures, are just two points, of almost vanishingly small area, within this vast space. It may indeed be interesting to ask how much our human cognitive faculties are like each (to what extent, in what circumstances, with respect to what sorts of capacities, etc.). But to assume that the two represent the entire space, or a very large fraction of the space—even to assume that they

---

[5]Nor is it clear how to individuate programs—or lines of code. When does one line turn into another one? How long does a line have to exist (e.g., in a rough-draft of a program, in a throw-away implementation) in order to count? What about multiple copies? Moreover, since C++ is already passé, what about Java? Or the language that will be invented after that?

I haven't a clue how to answer such questions. Maybe this is a better estimate: $10^{9\pm(3\pm2)}$.

are especially important anchor points in terms of which to dimension the space—is a mistake. Our imaginations need to run much freer than that. And commercial software shows us the way.

2. Compositionality

What it is that defines the symbolic model is itself a matter of debate. But as Fodor and Pylyshyn make clear, there are several strands to the basic picture. First, it is assumed that there exist a relatively small (perhaps finite) stock of basic representational ingredients: something like words, atoms, or other *simplexes*. Second, there are grammatical formation rules, specifying how two or more representational structures can be put together to make *complexes*. (Words of English are good examples of simplexes, and sentences and other complex phrases of natural language are good examples of complexes. But words have various additional properties—such as having spellings, being formulable in a consensual medium between and among people so as to serve as vehicles for communication, etc.—that aren't taken to be essential to the symbolic paradigm.) Third, it is assumed that the simplexes have some *meaning* or *semantic content*: something in the world that they mean, denote, represent, or signify. Fourth, and crucially, the meanings of the complexes are assumed to be built up, in a systematic way, from the meanings of the constituents.

The picture is thus somewhat algebraic or molecular: you have a stock of ingredients of various basic types, which can be put together in an almost limitless variety of ways, in order to mean or represent whatever you please. This "compositional" structure[6] underwrites two properties that Fodor identifies as critical aspects of human thinking: *productivity* (the fact that we can produce and understand an enormous variety of sentences, including examples that have never before occurred) and *systematicity* (the fact that the meaning of large complexes is systematically related to the meanings of their parts). Much the same structure is taken by such writers as Evans

[6]Compositionality is a complex notion, but is typically understood to consist of two aspects: first, a syntactic or structural aspect, consisting of a form of "composition" whereby representational symbols or vehicles are put together in a systematic way (according to what are often known as formation rules), and a semantic aspect, whereby the meaning or interpretation or content of the resulting complex is systematically formed out of the meanings or interpretations or contents of its constituents, in systematic way (in a way, furthermore, associated with the particular formation rule the complex instantiates).

and Cussins[7] to underlie what is called *conceptual* representation. The basic idea is that your concepts come in a variety of kinds: some for individual objects, some for properties or types, some for collections, etc.; and that they, too, can similarly be rearranged and composed essentially at will. So a representation with the content **P(x)** is said to be conceptual, for agent **A**, just in case: for every other object **x'**, **x''**, etc. that **A** can represent, **A** can also represent **P(x')**, **P(x'')**, etc., and for every other property **P'**, **P''**, etc. that **A** can represent, **A** can also represent **P'(x)**, **P''(x),** etc.

Thus suppose we can say (or entertain the thought) that a table is 29" high, and that a book is stolen. So too, it is claimed—given that thought at this level is conceptual—we can also say (or entertain the thought) that the table is stolen and the book is 29" high (even if the latter doesn't make a whole lot of sense). This condition, which Evans calls a *generality condition*, is taken to underwrite the productive power of natural language and rational thought. Furthermore, it is clearly a property that holds of the paradigmatic instances of "symbolic" AI—i.e., of logical axiomatisations, knowledge representation systems, and the like. Whether being compositional and productive is considered to be a *feature*, as Fodor suggests, or a *non-feature*, as various defenders of non-conceptual content suggest, there is widespread agreement that it is an important property of some representation schemes—one paradigmatically exemplified by ordinary logic. Indeed, the converse, while too strong, is not far from the truth: some people believe that connectionist, "subsymbolic," "non-symbolic" and other forms of dynamical system are recommended exactly in virtue of being *non-compositional* or *non-conceptual*.

3. Programs

What about those billions of lines of C++ code? Are they conceptual in this sense?

We need a distinction. Sure enough, the *programming language* C++ is a perfect example of a symbolic system. There is an indefinite stock of atomic symbols, called *identifiers*, some of which are primitive, others of which can be defined. There are some (rather complex) syntactic formation rules, which show how to make complex structures, such as conditionals, assignment statements, procedure definitions, etc.,

---

[7]Evans, Gareth, *Varieties of Reference*, Oxford: Clarendon Press (1982); Cussins, Adrian, "On the Connectionist Construction of Concepts," in Boden, Margaret. (ed.), *The Philosophy of Artificial Intelligence*, New York: Oxford University Press (1990).

out of simpler ones. Any arrangement of identifiers and keywords that matches the formation rules is considered to be a well-formed C++ program—and will thus, one can presume, be compiled and run. By far the majority of the resulting programs won't *do* anything of interest, of course (just as by far the majority of syntactically legal arrangements of English words make no sense). But it is important that all these possible combinations are legal. That's exactly what makes programming languages so powerful.

But it does not follow that most commercial software is symbolic. For consider the language used in that last paragraph. What is compositional—and hence symbolic—is the *programming language*, taken as a whole, not any *specific program* that one writes in that language. It follows that the *activity of programming*—the activity engaged in by people, for which they often get well paid—is a symbolic process. That may be an important fact. It might be usable as an early indicator of what children will grow up into good programmers, for example, or represent a limitation on how we construct computers. But it is irrelevant to the computational theory of mind. It is not *programming* that mentation is supposed to be like, after all, according cognitivism's fundamental thesis.[8] It is the *running of a (single) program*.

If you write a network control program, and I write a hyperbolic browser, and a friend writes a just-in-time compiler, all in C++, each of us uses the compositional power of the language to *specify a particular computational program or process or architecture*. There is no reason to suppose—indeed, good reason not to suppose—that those programs, those resulting specific, concrete active loci of behavior, will retain the compositional power of the language we used to specify them. To think so is, like the Martian, to make something of a use/mention mistake.

To make this precise, we need to be more careful with our language. As is standard, I will call C++ and its ilk (Fortran, Basic, Java, etc.) *programming languages*. Programming languages, I admit, are compositional representational systems, and hence symbolic. They are used, by people, to specify or construct individual *programs*. Programs are static, or at least passive, roughly textual, entities, of the sort that you read, edit, print out, etc.—i.e., of the sort that exists in your emacs buffer.[9]

---

[8]It is by no means clear that programming is a computational activity. Chances are, it can be argued to be computational only if cognitivism is true.

[9]Technically, a distinction needs to be made between the program at the level of abstraction

What programs are *for* is to produce behavior. That's why we write them. Behavior is derived from programs by *executing* or *running* them. Programs can be executed directly, which is called *interpretation.*[10] Or they can first be translated, by a process called *compilation*, into another language more appropriate for direct execution by a machine. That resulting behavior I will call a *process.* When a program is interpreted, therefore (in the computer scientist's sense of that term), what results is behavior or a process. But when a program is compiled, what results is not behavior, but another program, written in a different language (typically machine language). When that machine language program is executed, however, once again a process (or behavior) will result.

For our purposes, having to do with what is and is not symbolic, what matters is that once a program is created, *its structure is fixed*. Except in esoteric cases of reflective and self-modifying behavior—which is to say, except in a vanishingly small fraction of those 10^{11} lines of code—the entire productive, systematic, compositional power of the programming language is set aside when the program is complete. The process that results from running that program is…well, whatever the program specifies. But, at least to a first order of approximation, the compositional power of the programming language is as irrelevant to the resulting process as the compositional and productive power of a cad system is irrelevant to the thereby-specified fuel cell.

Consider an example. Suppose we are writing a driver for a print server, and need to represent the information as to whether the printer we are currently servicing is powered up. It would be ordinary programming practice to define a variable called current-printer[11] to represent whatever printer is currently being serviced, and a predicate called PoweredUp? to be the Boolean test. This would support the following sort of code:[12]

(and internal implementation) that a compiler can see—the one that gets "written" on a computer's hard disk, etc.,—and the strictly "print representation" in ascii letters, that people can read. For purposes of this paper, however, this distinction does not matter. As is common parlance, therefore, I will refer to both, interchangeable, as "the program."

[10]Why this is called interpretation will emerge in the next section, on semantics.

[11]Some mathematicians find it curious that, as here, programmers use multi-letter variable names. Then again, many programmers find it curious that mathematicians can get away with single-letter variable names.

[12]This code fragment is ridiculously skeletal—by design.


```
if PoweredUp?(current-printer)
    then … print out the file
    else TellUser("Your printer is not powered on.  Sorry.")
```

But now consider what happens when this program is compiled. Since the question of whether or not a printer is powered up is a Boolean matter, the compiler can allocate a single bit in the machine (per printer) to represent it. That will work so long as the hardware is arranged to ensure that whenever the printer is powered up, the bit is set to (say) 1; otherwise, it should be set to 0. Instances of calls to PoweredUp? can then be translated into simple and direct accesses of that bit. In the code fragment above, for example, if that bit is 1, the file will be printed; if it is a 0, the user will be given an error message. And so all the compiler needs to produce is a machine whose behavior is functionally dependent on the state of that bit in some way or other.

This is all straightforward—even elementary. But think of its significance. Consider Evans' generality condition, described above. What was required, in order for a system to be compositional in the requisite way, was the following: that the system be able to "entertain" a thought—construct a representation, say—whose content is that any property it knows about hold of any object it knows about. Suppose, for argument, that we say that the print driver "knows about" the current printer, and also "knows about" the user—the person who has requested the print job, to whom the potential error message will be directed. Suppose, further, that we say that the driver, as written, can "entertain the thought" that the printer is powered up. Does that imply that it can entertain a thought (or construct a representation) whose content is that the *user* is powered up?

Of course not. In fact the print driver process cannot entertain *a single "thought"* that does not occur in the program. That shows that it is not really "entertaining" the thought at all. For the issue of whether the printer is powered up is not a proposition that can figure, arbitrarily, in the print driver's deliberations. In a sense, the print driver doesn't "deliberate" at all. It is a machine, designed for a single purpose. And that is why the representation of whether a given printer is powered up can be reduced to a single bit. It can be reduced to a single bit because *the program has absolutely no flexibility in using it*. Sure, the original programmer could have written any of an unlimited set of other programs, rather than the program they wrote. But given that they wrote the particular one that they did, that extrinsic flexibility is essentially irrelevant. Indeed, there is a sense in which that is exactly why we compile programs:

to get rid of the overhead that is required in the original programming language to keep open (for the programmer) the vast combinatoric space of possible programs. Once a particular program is written, this space of other possibilities is no longer of interest. In fact it is in the way. It is part of the compiler's task to wash away as many traces of that original flexibility as possible, in order to produce a sleeker, more efficient machine.

Another numerical point will help drive the point home. Programs to control high-end networked printers are several million lines long. The long-promised but as-yet unreleased Windows nt 5.0 is rumoured to contain 35 million lines of code. It is not unreasonable to suppose that such programs contain a new identifier every four or five lines. That suggests that the number of identifiers used in a printer control program can approach a million, and that Windows nt will contain as many as 7 million identifiers. Suppose a person's conceptual repertoire is approximately the same size as their linguistic vocabulary. Educated people typically know something like 40,000 to 80,000 words. That implies that people have on the order of 100,000 concepts. Is it possible—as seems to be entailed by the symbolists' position—that a Xerox printer has a conceptual repertoire ten times larger than you do, or a Microsoft operating system, seventy times larger?

I think not.[13]

4. Processes

A way to understand what is going on is given in figure 1. The box at the top left is (a label for!) the program: the passive textual entity selected out of the vast space of possible programs implicitly provided by the background programming language. The cloud at the middle right is intended to signify the process or behavior that results from running the program.[14] The scene at the bottom is a picture of the program's task domain or subject matter. For example in this case the process might be an architectural system dealing with house design.

---

[13]Indeed, no program—at least not any we currently know how to build—could possibly cope with millions of differently-signifying identifiers, if all those identifiers could be mixed and matched, in a compositional way, as envisaged in the symbolists' imagination.

[14]Whether the cloud represents a single run (execution) of the process, or a more general abstract type, of which individual runs are instances, is an orthogonal issue—important in general, but immaterial to the current argument.

Given these three entities, two relations are most important: that labelled α, from program to process, and that labelled β, from resulting process to task domain. Moreover, what is perhaps the single most confusing fact in cognitive science's use of computation is this: the word 'semantics' is used by different people for *both of these relations*. In computer science, the phrase "the semantics of a program" refers to the program-behavior (process) relation α, whereas the relation considered semantic in the philosophy of mind is the process-world relation β. For discussion, in order not to confuse them, I will refer to α as *program semantics*, and to β as *process semantics*. It is essential to realize that they are not the same.[15]

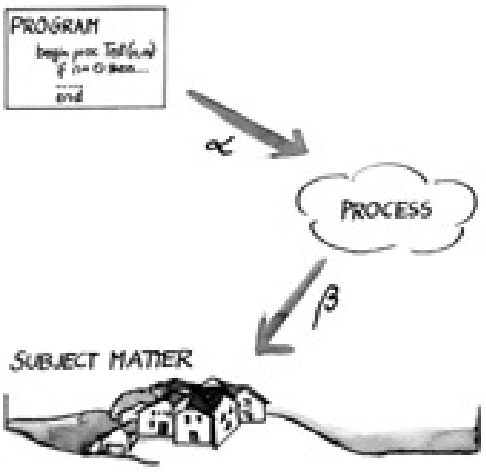

FIGURE 1 PROCESS VS. PROGRAM SEMANTICS

All sorts of confusion can be cleared up with just this one distinction. But a cautionary note is needed first. Given that processes and behaviours are computer science's primary subject matter, you might think that there would be a standard way to describe them. Curiously enough, however, that is not so. Rather, the practice is instead to *model* processes in various ways. The most common way to model a process is with a (mathematical) function mapping the inputs onto the outputs. A second way is to treat the computer as a state machine, and then to view the process or behaviour as a sequence of state changes. A third is to have the process produce a linear record of everything that it does (called a "dribble" or "log" file), and to model the process in its terms. A fourth (called operational semantics) is to model the process in terms of a different program in a different

---

[15]Many years ago, at Stanford's Center for the Study of Language and Information, I, with a background in AI and philosophy of mind, tried in vain to communicate about semantics with Gordon Plotkin, one of the most preëminent theoretical semanticists in all of computer science. Finally, a glimmer of genuine communication transpired when I came to understand the picure painted in the text, and realised that we were using the term 'semantics' differently. "What I am studying," I said, trying to put it in his language, "is *the semantics of the semantics of programs*."

Plotkin smiled.

language that would, if run, generate the same behavior as the original. A fifth and particularly important one (called denotational semantics) is to model the concrete activity that the program actually produces with various abstract mathematical structures (such as lattices), rather in the way that physicists model concrete reality with similarly abstract mathematical structures. Especially because of the use of these mathematical models, outsiders are sometimes tempted to think that computer science's notion of semantics is similar or equivalent to that used in logic and model theory, missing the fact that although the relation is studied in a familiar way, what relation it is that is so studied may not be the one they think.

Once these modelling issues are sorted out, one can use these basic distinctions to make the following points:

1. (Discussed above) It is programs, not processes, that, in standard computational practice, are symbolic (compositional, productive, etc.).
2. It is again programs, not processes, that computer scientists take to be *syntactic*. It strikes the ear of a computer scientist oddly to say that a process or behavior is syntactic. But when Fodor talks about the language of thought, and argues that thinking is formal, what he means, of course, is that *human thought processes* are syntactic.
3. Searle's analogy of the mind to a *program* is misleading.[16] What is analogous to mind, if anything, is *process*.
4. There is no reason to suppose—indeed, I know of no one who ever has proposed—that there should be a *program* for the human mind, in the sense we are using here: a syntactic, static entity, which specifies, out of a vast combinatoric realm of possibilities, the one particular architecture that the mind in fact instantiates. Perhaps cognitive scientists will ultimately devise such a program. But it is unimaginable that evolution constructed us by writing one.
5. For simple engineering reasons, the program-process relation α, in the figure, must be constrained to being effective (how else would the program run?). There is no reason to suppose that the process-world relation

[16]Searle, John, *Minds, Brains, and Science,* Cambridge: Harvard University Press (1984).

β need be effective, however—unless for some reason one were *metaphysically* committed to such a world view.

6. It is because computational semanticists study the program-process relation α, not the process-world relation β, that theoretical computer science makes such heavy use of intuitionistic logic (type theory, Girard's linear logic, etc.) and constructive mathematics.

5. Conclusion

What, in sum, can we say about the cognitive case? Two things, one negative, one positive. On the negative side, it must be recognized that it is a mistake to assume that modern commercial programming gives rise to processes that satisfy anything like the defining characteristics of the "symbolic" paradigm. Perhaps someone could argue that most—even all—of present-day computational processes are symbolic on some much more generalized notion of symbol.[17] But the more focused moral remains: the vast majority of extant computer systems are not symbolic in the sense of "symbol" that figures in the "symbolic vs. connectionist" debate.

What are the computer systems we use, then? Are they connectionist? No, of course not. Rather—this is the positive moral—they spread out across a extraordinarily wide space of possibilities. With respect to the full range of computational possibility, moreover, present practice may not amount to much. Computation is still in its infancy, after all; we have presumably explored only a tiny subset of the space—perhaps not even a very theoretically interesting subset, at that. But even the space that has already been explored is far wider than debates in the cognitive sciences seem to recognize.

—— *end of file* ——

[17]If it were enough, in order to be a symbol, to be discrete and to carry information, then (at least arguably) most modern computational processes would count as symbolic. Or at least that would be true if computation were discrete—another myth, I believe (see my "Indiscrete Affairs"). But the symbolic vs. connectionist debate is not simply a debate about discrete vs. continuous systems.

