One Hundred Billion Lines of C++[†]

Brian Cantwell Smith[*]
University of Toronto

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. On the large CAD display surface forming your desk, you are sketching a complex combustion chamber for a new eco-engine you and some colleagues are designing. 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 some engineers designing one. They showed me how the word 'oxygen' would be combined with the word 'hydrogen' in a wondrous kind of symbol 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. Map is not territory. What will be funneled into the input chamber—to belabour the obvious—is 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 that has been conducted, over the last decade or so, between symbolists and connectionists founders over a troublingly similar er-

[†]Published in *Cog Sci News*, Lehigh University, 1997. Thanks to Irene Appelbaum and Güven Güzeldere for comments on an early draft.
[*]Coach House Institute, Faculty of Information, University of Toronto
90 Wellesley St W, Toronto, Ontario M5S 1C5 Canada

Last edited: November 13, 2009
Comments welcome
brian.cantwell.smith@utoronto.ca

ror. Perhaps not quite as egregious—but a misunderstanding, nonetheless. Moreover, the confusion goes far beyond that particular debate, infecting (mis)understandings of the computational theory of mind throughout philosophy—including, to take just one example, the debate about Searle's notorious Chinese Room. 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.

For discussion, I will focus on the connectionist debate, but the points can easily be extended to other contexts.

1 Background

A glimmer of trouble is evident in the way the connectionist 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 paradigmatically 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[1] 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 peo-

[1]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*, …

ple, 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 that they are not mentioned because such programs are rare. In National Public Radio's famous phrase, "let's do the numbers."[2] 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.[3] Logic-based programs, theorem provers, and knowledge representation systems were early examples. SOAR[4] 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 somewhere between 1,000 and 10,000 programs? Suppose each comprises an average of 10,000 lines of code (a couple of hundred pages, in normal formatting). That would come to ten million lines of code, overall.

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 and away the mainstay of computing. Again, it is impossible to make even much of a rough estimate, but it will not be too misleading if we assume that there are probably something on the order of 10^{11}—i.e., one hundred billion—lines of C++ code in the world.[5]

---

[2]«Ref 'Marketplace'»

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

[4]'«ref»

[5]It is not even clear how one would individuate programs—or, for that matters, 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 have no clue as to how to answer such questions. Maybe this is a bet-

And we are barely started.

In sum: symbolic AI systems constitute approximately 0.01% of written software.

By themselves, the numbers do not 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, nor "dynamic," in the sense advocated by van Gelder, but rather some fourth kind entirely;

2. The only reason for thinking that commercial software is symbolic, as we will see, stems from a confusion between a **program** and 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—a requirement that in turn will require us to understand something about the history of the technical vocabulary used in computer science.

In a sense, the ultimate 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 and dynamic architectures, are *just two tiny regions*, of almost vanishingly small total extent, within this vast space.

Within the hugely important project of exploring how human

---

ter estimate: $10^{9\pm(3\pm2)}$. Whatever; the answers do not matter to any of the points being made in the text.

cognition works, it may be important, or anyway of moderate interest, to ask whether and how much human cognition fits within these regions—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 even a very large fraction of the space—even to assume that they 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:

1. It is assumed that there exist a relatively small (perhaps finite) stock of basic representational ingredients: something like words, atoms, or other entities we can call **simplexes**.
2. There are **grammatical formation rules**, specifying how two or more representational structures can be put together to make **complexes**.[6]
3. It is assumed that the simplexes have some **meaning** or **semantic content**: something in the world that they mean, denote, represent, or signify.
4. Finally—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

[6]Words of English—or anyway their morphological stems—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 are not taken to be essential to the symbolic paradigm.

represent whatever you please. This "compositional" structure[7] 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 and Cussins[8] 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.[9]

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 does not make a whole lot of sense). This condition, called the "**Generality Condition**" by Evans, is taken to underwrite the productive power of natural language and rational thought. It is also clearly a property taken to hold of the paradig-

[7]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).

[8]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).

[9]Evans says 'entertain the judgment' that *a* is *F*, that *b* is *G*, etc., rather than 'represent'; I use the representational phrasing here since the subject matter is symbolic computation.

matic 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—i.e., whether it is viewed positively or negatively—there is widespread agreement that it is an important property of some representation schemes, and 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 compositional sense?

We need a distinction. Sure enough, the programming language C++ is a perfect example of a symbolic system. An indefinite stock of atomic symbols is made available, called *identifiers*, some of which are primitive, others of which can be defined. There are (rather complex) syntactic formation rules, which show how to make complex structures, such as conditionals, assignment statements, procedure definitions, etc., 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 will do nothing 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 these possible combinations are all *legal*. That is exactly what makes programming languages so powerful.

But—and this matters—it does not follow that most commercial *software* is symbolic. For consider the language used in that last paragraph. What is compositional—and hence is 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* is a symbolic process—i.e., the activity engaged in by people, for which they are often well paid. That may be an important fact, for a variety of reasons: it might be usable as an early

indicator of what children will grow up to be good programmers, or represent an insight into or limitation on how we construct computers. But it is irrelevant to the computational theory of mind, since it is not *programming* that mentation is supposed to be like, according cognitivism's fundamental thesis.[10] Rather, the claim of the computational theory of mind is that thought or cognition or mentation is like (or even: is) *the running of a (single) program*.

Thus 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 C++ programming language to specify a particular computational program or process or architecture. There is no reason to suppose—good reason not to suppose, in fact—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 entirely standard, I will call C++ and its ilk (Fortran, Basic, Java, JavaScript, etc.) **programming languages**. As stated above, I admit that programming languages 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.[11]

What programs are for is to produce behavior. That is why we write them. Behavior is derived from programs by *executing* or *running* them. Programs can be executed directly in one of two ways: (i) they can be executed by the underlying hardware of the

[10]It is by no means clear that programming is a *computational* activity. Chances are, programming will turn out to be to be computational if and only if cognitivism is true.

[11]Technically, a distinction needs to be made between the program at the level of abstraction (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, too, does not matter. As is common parlance, therefore, I will refer to both, interchangeable, as "the program."

machine, if they are written in the lowest level language (called 'machine language'), in which case the term 'execution' is the most common one used; or (ii) they can be executed by another computational process, which itself results (directly or indirectly) form the execution of a machine language program, in which case the execution of the (higher-level) program is typically called **interpretation**, and the process that does the execution, the **interpreter**.[12] Of the two, the notion of interpretation is more general; and since most machines, these days, are micro-coded, even (so-called' machine language programs are typically interpreted, but a process resulting from a still-further lower level program, written in what is called 'microcode,' which in turn is directly executed by the microcode hardware.

Commonly, however, programs are not directly executed. Instead, they are first *translated*, by a process called **compilation**, into another language more appropriate for direct execution by a machine. That is, if program P_1 is written in C++, instead of being run or executed directly, by a C++ interpreter, it will instead be translated into another program P_2, perhaps in machine language, such that the execution of P_2 results in the "same" behaviour as would have resulted by the direct execution of P_1 by a C++ interpreter.

However it comes into existence, the ultimately resulting behavior—the whole point of the exercise—is what I will call a **process**. When (in the computer scientist's sense of that term) a program is *interpreted*, therefore, to put this all simply, what results is *behavior* or a *process*. But when a program is *compiled*, what results is not behavior, but another program, 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

[12]Why this is called *interpretation* will be discussed in the next section.

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 computer-aided design system (CAD) 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-printer11 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:[13]

```
if PoweredUp?(current-printer)
        then ... print out the file ...
        else TellUser ("Printer 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 is free to 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 (say) to '1'; otherwise, it should be set to '0'. Instances of calls to `PoweredUp?` can then be translated into simple and direct accesses of that single 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. In particular Consider Evans' Generality Condition, described above. In order for a system to be compositional in the requisite way, what was required, 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

[13]By design, this code fragment is ridiculously skeletal.

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, given that C++ is incontestably symbolic, productive, and so forth, 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.

From one point if view, in fact, 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. Operating systems are 100s of millions of lines of code.[14] It is not unreasonable to suppose that such programs contain a new identifier every four or five lines. That suggests that

[14]Microsoft Windows NT 5.0, the release of which was thought to be imminent at the point when this paper was first written, was rumoured to contain 35 million lines of code. (It was eventually released on February 17, 2000.)

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. Suppose we therefore assume 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.[15]

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.[16] 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

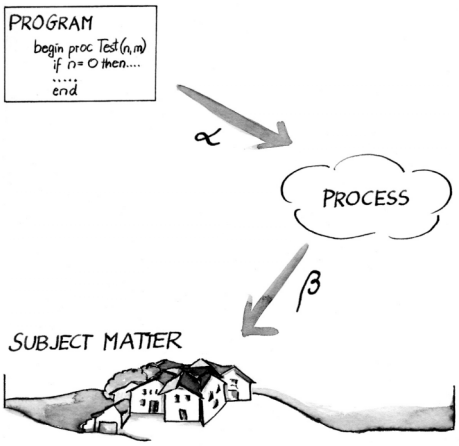

Figure 1 — Program and Process

[15]Indeed, no program—at least none 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.

[16]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.

dealing with house design.

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.[17] Not only do they relate different things, but they are subject to vastly different constraints—and are of distinct metaphysical kinds.

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, professional practice instead *models* processes in various ways:

… In the final version it will probably be helpful to devote more than a sentence to each of these; perhaps even worth constructing a target program P that does something (a bit more complex than the printer example above), and then actually presenting the five different models of the processes that result. …

1. The most common way to talk about processes is to model them with (mathematical) functions mapping their inputs onto their outputs.

[17]Many years ago, at Stanford's Center for the Study of Language and Information (CLSI), 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 picture sketched in figure 1, 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.

2. 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.

3. 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.

4. A fourth (called "operational semantics") is to model the process in terms of a different program in a different language that would, if run, generate the same behavior as the original.

5. A fifth and particularly important one—called **denotational semantics**—models the concrete activity that the program actually produces (i.e., the behaviour Q) with various abstract mathematical structures (such as lattices), rather in the way that physicists model concrete reality with similarly abstract mathematical structures (tensors, vector fields, etc.).

Especially because of the common use of mathematical models in several of these approaches (#s 1 and 5 especially, though they can all be mathematized), 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. But that assumption is misleading. Although the relation is studied in a familiar way, what relation it is that is so studied may differ substantially from what is supposed.

5 Discussion

Once these modelling issues are sorted out, we can use these basic distinctions they are defined in terms of to make the following points:

… This section has not really been written; the six points identified below should be amplified enough to communicate the essential moral, in each case, to someone who does not "already know it," as it were—in particular, enough detail both to motivate and to convey it to a philosophical reader, even one without computational experience …

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.[18] What is analogous to mind, if anything (i.e., if the computational theory of mind is true) is *process*.

4. Not only is there no reason to suppose, but in fact 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 provided for by the programming language, the one particular architecture that the mind in fact instantiates. Perhaps cognitive scientists will ultimately devise such a program. But it seems relatively unimaginable that evolution constructed us by writing one.[19]

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 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

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

[19]Of course one could call DNA a programming language in this sense…«talk about how it is subject to some of the same efficacy constraints»

intuitionistic logic (type theory, Girard's linear logic, etc.) and constructive mathematics.

6 Conclusion

… Once §5 is properly written, this § will deserve a rewrite …

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.[20] 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" or "computational vs. dynamic" debates.

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; we have presumably explored only a tiny subset of the space—perhaps not even a very theoretically interesting subset, at that. But this much we can know, already; the space that has already been explored is far wider than debates in the cognitive sciences have so far recognized.

[20]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 chapter ■■). But the symbolic vs. connectionist and/or dynamicist debate is not simply a debate about discrete vs. continuous systems.