Can Computers Carry Content 'Inexplicitly'?[1]

PAUL G. SKOKOWSKI
Department of Philosophy, Stanford University, Stanford, CA, 94305, U.S.A.
(paulsko@csli.stanford.edu)

Abstract. I examine whether it is possible for content relevant to a computer's behavior to be carried without an explicit internal representation. I consider three approaches. First, an example of a chess playing computer carrying 'emergent' content is offered from Dennett. Next I examine Cummins' response to this example. Cummins says Dennett's computer executes a rule which is inexplicitly represented. Cummins describes a process wherein a computer interprets explicit rules in its program, implements them to form a chess-playing device, then this device executes the rules in a way that exhibits them inexplicitly. Though this approach is intriguing, I argue that the chess-playing device cannot exist as imagined. The processes of interpretation and implementation produce explicit representations of the content claimed to be inexplicit. Finally, the Chinese Room argument is examined and shown not to save the notion of inexplicit information. This means the strategy of attributing inexplicit content to a computer which is executing a rule, fails.

Introduction

Is it possible for representational content that is relevant to the explanation of a computer's behavior to be carried by the computer without explicit internal

[1]I wish to thank Fred Dretske and John Perry for helpful comments and suggestions. I would also like to thank an anonymous reviewer for suggesting I take a look at the implications of the Chinese Room argument for the notion of inexplicit information. Earlier versions of this paper were read at the American Philosophical Association Pacific Division Meeting in San Francisco in March, 1993, and at the 7th International Conference on Computing and Philosophy in Orlando in August, 1992.

representation of that content? In this paper I will examine some answers to this question. One answer is to hold that the content is an emergent property of the system. It is content that is convenient for describing behavior of the system, but the content plays no causal role for the system. This is the view of Daniel Dennett. Another, more interesting answer is that the content *is* somehow carried and used by the system in a way relevant to its performance, but that there is no representational state in the system which can be singled out as carrying that content. Since there is no state carrying the content, the content is said to be carried by the computer *inexplicitly*. This is an intriguing view defended recently by Robert Cummins (1986) in his paper *Inexplicit Information*. Dennett's view in effect poses a problem to someone like Cummins who seeks to attribute a *bona fide* role to content which Dennett describes as emergent. I start with an example by Dennett.

1. The Problem of Emergent Content in Digital Computers

In *Brainstorms*, Dennett describes the behavior of a certain chess-playing program in the following way:

> In a recent conversation with the designer of a chess-playing program I heard the following criticism of a rival program: "It thinks it should get its queen out early." This ascribes a propositional attitude to the program in a very useful and predictive way, for as the designer went on to say, one can usually count on chasing that queen around the board. But for all the many levels of explicit representation to be found in that program, nowhere is anything roughly synonymous with "I should get my queen out early" explicitly tokened. The level of analysis to which the designer's remark belongs describes features of the program that are, in an entirely innocent way, emergent properties of the computational processes that have "engineering reality". (Dennett, 1978, p. 107)

The point Dennett is making here is that though it *appears* that the chess-playing computer behaves as if it had a representational state with the content "I should get my queen out early," there is *no state* in the computer with that content. Or, on the programming level, there is no line in the computer program to the effect that the computer should get its queen out early. Since there is no such explicit state to be found, the content attributed to the computer appears, on his view, to be 'emergent'.

The explicit representational states, *qua* lines in the program that *run* the computer, play a causal role in the life of the computer. Emergent contents, on the other hand, may help us explain the behavior of the computer, but play no causal role for the machine.

Dennett's view on content goes beyond this, of course. He ultimately claims that all attributions of content to systems are dependent on the explainer's stance. (Dennett, 1987) So, in a perverse way, it doesn't even matter whether contents are carried by 'explicitly tokened' representations (his terminology) or not, since all contents fall in the same boat: if a particular set of contents work for explaining the behavior of a system then we needn't worry about details like attributing the particular contents used in the explanation to particular states in the system being explained. 'Emergent' contents are just as good as determinate contents for explaining behavior.

But the above quote shows that Dennett is feeling generous about internal content-carrying states; he allows that *some* explicit representations are to be found in the chess-playing program. Therefore, we can presumably take him to mean that the computer has certain states which *do* function as representational states with content. Surely chess-playing computers have some internal representation of the positions of the pieces on the chess board during every move. They may even have certain 'goals' such as 'try to take the highest-ranking white piece at the earliest opportunity.' Such states might be understood as the computer's analogues of belief and desire states. Furthermore, these explicit tokens, or states, are what *cause* the computer to move its pieces on the board. These are states, then, which appear to have representational content; states which cause the computer to behave for *reasons*.

But again, there is no state with the content “I should get my queen out early” in the computer. And this poses a problem for anyone who prefers to explain the behavior of computers entirely by reference to the causal powers of their internal representational states. How can one holding such a view deal with the claim that certain intentional descriptions we apply to systems at best describe ‘emergent’ properties of the system? Dennett’s example appears to show that although *many* intentional ascriptions appear to apply to physical states which are causally efficacious and have content in the right sorts of ways for explaining behavior, not all intentional ascriptions do. Our chess-playing computer exhibits queen-out-early behavior, and predictably. But there is no queeen-out-early internal state playing a causal role in the computer. If intentional ascriptions such as these do not apply to distinct physical states, then how do whatever it is they *do* apply to *cause* anything? And perhaps even more disturbing, how would such a thing (if it *is* a thing) carry content?

2. Cummins’ Solution: Inexplicit Content

An intriguing answer to this question is offered by Robert Cummins (1986). In *Inexplicit Information* Cummins deliberates upon Dennett’s example of the content exhibited by the chess-playing computer:

> If we turn back to Dennett’s parable now, we find something of a muddle. Evidently, the case in which we have a rule explicitly tokened in memory is not the one at issue. In the case lately imagined, however, although we do have an instruction in the program, nothing like DEPLOY THE QUEEN EARLY is tokened by the system. This is the case we are interested in, but here it seems plainly incorrect to say that the system thinks or believes that it should deploy the queen early, though this does apply to the programmer. Instead, the correct description seems to be that in the system there is a kind of inexplicit information, information lodged, as it were, in whatever physical facts underwrite the capacity to execute the program. Explanatory appeals to this sort of information evidently differ radically from explanatory appeals to the system’s representations -- e.g., representations of current position. (Cummins, 1986, p. 124)

This quote suggests that the machine carries a determinate content while running, a content Cummins calls *inexplicit information*.[2] For Cummins, this content is in the form of what he calls *rules*. These rules are not explicitly represented in the machine as, say, knowledge or beliefs or desires would be. Rather, the rules are inexplicitly represented in virtue of being '*executed*'.

The rule Cummins says Dennett's computer seems to be executing is: IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN. Let us follow Cummins in calling this rule "(R)". He says,

> There are two ways in which a system can "have" a rule like (R): (R) might be a rule that is represented in the system's memory, or (R) might be an *instruction*, i.e., a rule in the program that the system *executes*. (Cummins, 1986, p. 122)

If a system has (R) in the *first* way described in the quote, then (R) is *explicitly* represented. Indeed, a system with (R) represented in memory "knows what to do." (p. 121) Such a representation in memory therefore plays the role of a knowledge or belief state with the content (R). For Cummins, an explicit representation "is available as a premise in reasoning, and is subject to evidential assessment." (p. 125) As Cummins recognizes, this way of representing a rule is fairly straightforward, and so isn't all that interesting. However, if a system has (R) in the *second* way, where (R) is a rule *executed* by the system, then he claims (R) is *inexplicitly* represented. Here is where things get a little more interesting.

[2]Note that Cummins uses the term "information" more in the way others use "content". That is, for Cummins, information may be false. (See p. 118 of *Inexplicit Information*.) I will follow Cummins and use both terms interchangeably in this paper.

Let us look at what Cummins calls the second way in which a system can "have" a rule (R). Cummins asks us to consider a device called CHESS, which plays chess in the way which Dennett describes.[3] Cummins says,

> What about the case in which (R) is a rule in the program that the system executes -- i.e., an instruction? Let's move back to the original example derived from Dennett. Suppose the *program* executed by the *device* CHESS contains the following rule:
>
> IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN.
>
> Does CHESS -- the *device* executing the program -- believe that it should deploy its queen early? The programmer certainly believed it. And CHESS will behave as if it believed it too: hence the characterization Dennett reports. But CHESS (as we are now imagining it) simply executes the rule without representing it at all . . . (Cummins, 1986, p. 122)

Here Cummins suggests a contrast between: (1) a rule (R) written explicitly as an instruction in a program, and (2) the subsequent *execution* of that rule by a *device* running that program -- where the *device*, even though running the program, doesn't represent the rule (R). Let me explain these in order.

First, what does it mean to have a *program* with the rule (R) contained as an instruction within it? An instruction is a line in the computer program which expresses, in a high-level computer language, the rule (R). Of course, programming languages are never written in plain English. So instructions representing the rule (R) will be written differently from one computer language to the next. For example, in the programming language **C+**, the rule (R) might be expressed by a line in the program written:

[3]Throughout this paper, Cummins vacillates between calling CHESS a *system* and a *device*. The key to discerning the difference between when a system *is a device* and when a system *is something else* appears to be this: systems which are devices *execute* programs and rules, whereas systems *simpliciter represent* programs and rules. This, I take it, is Cummins' distinction. I will use the following conventions for the rest of this discussion on Cummins. I will use *device* when and only when talking of Cummins' *device* CHESS. I won't use the term *system* for this. When I talk about the entire computer system, or a part thereof, I will make it clear what I mean -- but it *won't* mean device.

```
if n ≤ 10 {q1 = x };
```

which says something like "if the current move is one of the first 10 move the queen from Q1 to position x (presumably already calculated)" -- in other words, IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN, where "EARLY" is for simplicity taken to be the first opportunity in the initial 10 moves of the game. Instructions such as these therefore act as representations of the rule (R), since (R) cannot be directly expressed in English in the program that the computer runs. These instructions serve as explicit representational states within the *program*.

Second, let me explain a device executing a rule in a program. Note that even though (R) may be explicitly represented by an instruction in the program, the *device* CHESS, according to Cummins, *does not* represent this rule. Rather, he says, CHESS *executes* the rule. This *execution* of a rule (R) by a *device* is what he claims leads to the device carrying the inexplicit information *that* (R). The idea is that the *device* CHESS -- not the *program* containing CHESS's instructions -- has no representational state with the content *that* (R), and yet behaves as if it did. Cummins says of this inexplicit information

> When we formulate the content of this information and attribute it to the system, we intentionally characterize the system, and rightly so, even though the propositional contents of our characterizations are not represented by the system we characterize. (Cummins, 1986, p. 125)

So at one level, the level of the *program* that CHESS runs, the rule (R) is represented by an instruction in the program. At the level of the *device* CHESS, on the other hand, the rule (R) is *not* represented, but is *executed*.

In this way, Cummins makes a distinction between rules represented in a program and rules executed by a device. Cummins intends to drive this distinction home by showing how he thinks CHESS, though implemented on a computer, will not represent a rule (R) -- even though (R) is a rule in the very program "interpreted" by the computer in order to implement CHESS in the first place. He says,

> . . . if CHESS is implemented on a general purpose computer, rather than hardwired, the program itself will also be represented "in" the system: it may be on a disk, for example. But these are not representations to CHESS, they are representations to the system that implements CHESS, typically an interpreter and an operating system. The interpreter "reads" these rules, not CHESS... CHESS doesn't represent its program, it executes it, and this is made possible by the fact that a quite different system does represent CHESS's program. (Cummins, 1986, p. 123)

Certainly on the face of it this seems reasonable, since programs written in a high-level language must be 'interpreted' or 'compiled' in order for the computer to run them. So the *device* CHESS won't come to exist until such an interpretation or compilation is done. Think of it this way. You own a brand new Mac II, and you want to play chess against it. Well, it does no good to just turn it on and try to play chess against it. You must write a program, preferably in a high-level language (to save the tedium of writing the 1's and 0's required of machine language), to do this. Once you have the program in the high level language, you use a 'compiler' or 'interpreter' program to 'implement' your new program on the machine. The *product* of this implementation of a high-level program is what Cummins means by the *device* CHESS. It is this product, the *device* CHESS, that you play your games of chess against. Again, the *device* CHESS is *not* the high-level program; it is the implementation of the program. So, returning to the original point, Cummins claims that though rules like IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN are represented in the high-level program as it sits 'uninterpreted' on the disk, they are *not* represented in the *device* CHESS implemented on the machine.

Cummins' notion of the device CHESS *executing*, without *representing*, the rule (R) relies heavily on understanding how a computer 'compiles' or 'interprets' and then executes a program. If we can clearly grasp these notions of compiling, interpreting and executing a high-level program, we will discover a problem with this analysis.

3. Why Cummins' Solution Won't Work for Digital Computers

Compilers and interpreters are examples of programs called *translators*. Imagine for a moment the sort of job that a human translator does -- expressions written or spoken in one language are translated, with the goal of preserving meaning, into another language. Translator programs operate on similar principles:

> A *translator* is a program that takes as input a program written in one programming language (the *source language*) and produces as output a program in another language (the *object* or *target language*). If the source language is a high-level language such as FORTRAN, PL/I, or COBOL, and the object language is a low level language such as an assembly language or machine language, then such a translator is called a *compiler*. (Aho & Ullman, 1979, p. 1)

Let's look at a simple example of a translation. Suppose we denote a statement in the high level language **C+** by, say A_C, where the subscript "C" indicates that the statement is in **C+**. A compiler given this statement will produce a statement, call it A_M, in machine language (the object language) which the computer can run directly. A_M will be made up entirely of 0's and 1's, but it is a sequence which represents the same rule in machine language as A_C did in **C+**. Thus if A_C is the statement X=1+1, for example, then A_M will be a direct translation of that statement in the 0's and 1's of machine language.

Note that interpreters are also translators, that is, they take high-level program languages and translate them into lower-level languages, but they operate slightly differently from compilers. (Other examples of translators are assemblers and preprocessors.) For the purposes of understanding the example of a chess-playing computer, just about any type of translator will do. I will stick with the most common type of translator, a compiler, in this discussion.

Now that we have a handle on compiling a high-level programming language, we can understand how such a program is *executed*:

> Executing a program written in a high-level programming language is basically a two-step process, as illustrated in Figure 1. The source program must first be compiled, that is, translated into the object program. Then the resulting object program is loaded into memory and executed. (Aho & Ullman, 1979, p. 1,2)

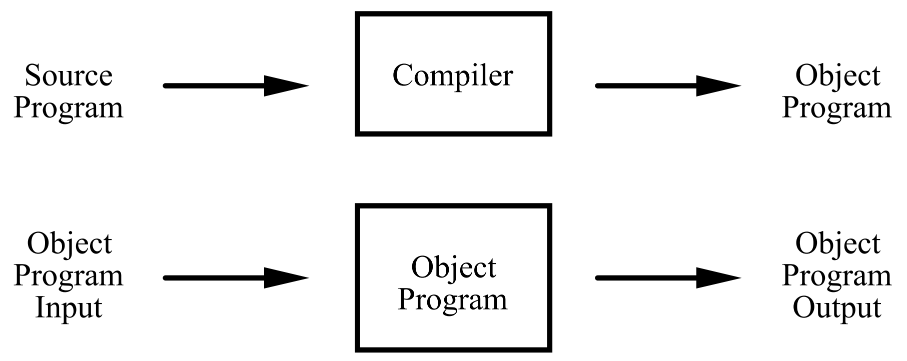

Fig. 1. Program Compilation and Execution

The second step in executing a program is important. It involves the object program (written in machine language) residing in memory -- this memory is local to the Central Processing Unit, or CPU -- and then being executed by the CPU. As the second part of Figure 1 shows, the object program takes input and produces output. That is, it is the object program, written in machine language, which is *executed*.

But now that we have these notions of programming language, object language, translation, compilation, and execution in hand, we can see a problem with Cummins' analysis of inexplicit information. Recall that, at the level of the *program* (written in a high-level language) that CHESS runs, the rule (R), IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN, is represented by an instruction in the program. At the level of the *device* CHESS, on the other hand, the rule (R) is not represented, but is *executed*. This is because CHESS is not itself a program, rather it is the *implementation* of the program -- a product of interpreting or compiling the high-level program -- which Cummins calls the *device* CHESS. Thus, though there is an instruction representing the rule (R) in the high-level program, there will not, according to Cummins, be a representation of (R) in the *device* CHESS which executes the program.

Here is where the problem surfaces. We now know that when a high-level program is compiled, or interpreted, the result will be a second program, the object program, written in machine language. This object program is a *translation* of the high-level program. Thus if the high-level program contained a representation of a rule (R) written in it, then the object program will contain a *translation* of (R), written in it. Therefore the object program also has a representation in it -- a representation of the rule (R). To illustrate this point, let's return to an earlier example. Suppose our chess playing program is written in the language **C+**. Recall that the statement "if n ≤ 10 {q1 = x };" is an instruction in the high level language **C+** that represented the rule IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN. Now, according to Cummins, this is a representation in the program, but not in the *device* CHESS. But to get the device CHESS, we must implement it on the machine. In order to implement it we must compile the high-level program. We do this. Now we have the object program residing in memory local to the CPU. But now note the following. We look in the object program, and lo and behold, there is an instruction, say "0110010001110001", which is a direct translation of the statement “if n ≤ 10 {q1 = x };”. This is because our compiler directly translated the instruction “if n ≤ 10 {q1 = x };” which was written in the high-level language **C+**. *And this translated instruction is a representation of the rule (R).*

Now recall Figure 1. Note that the product of compilation (or interpretation) is not just an abstract device -- it is an object *program*. Furthermore, this program, by virtue of the compiler, contains instructions which represent the same rules that the high-level program being translated did. Thus, what is *implemented* on the computer is *not* an abstract device without representations, instead, what is *implemented* is a new program, complete with lots of representations. In particular, if rule (R) is represented in our high-level program as “if n ≤ 10 {q1 = x };”, then it will be implemented in our new object program as the instruction “0110010001110001”, where this instruction also represents the rule (R). Thus the *device* CHESS, what is *implemented* on our computer, has representations of the rule (R) after all, contrary to Cummins' claims that it did not.

Cummins reply to this surely must be that the *device* CHESS *executes* the rule (R), it doesn’t represent it. Executing and representing are entirely different matters. But this reply hinges entirely on what a device *is*. For if a device *contains* a representation of a rule *within* it, as *part* of the device, then it is false to say that by executing a rule the device doesn’t represent the rule. For the rule is still represented by the device by the instruction residing *within* the device.

To see this, note that there are exactly three possibilities for Cummins as to what the *device* CHESS is. Upon compilation of the high-level program, only two entities are required for execution of a program: the object program and the CPU. And these two entities considered alone, or considered together, add up to the following three possible *devices*: (1) the object program by itself is the *device* CHESS; (2) the CPU by itself is the *device* CHESS; or, (3) the object program *together with* the CPU is the *device* CHESS.[4] Let's consider each of these in turn.

First, let us consider possibility (1). Is the object program by itself the *device* CHESS? Suppose it is. But then there are two problems. First, the object program contains a representation of the rule (R), and as we know by now, the *device* CHESS should not contain any representations of this rule. Second, the object program by itself can't *execute* anything. It needs a CPU to run it. Since we know from Cummins that the *device* CHESS *executes*, then the object program by itself won't do, since it can't execute anything.

Second, let us consider possibility (2). Is the CPU by itself the *device* CHESS? Suppose it is. But then there is a problem. For a CPU that sits by itself, with no program written in machine language in local memory for it to run, will sit there idly *forever*. CPU's are entirely deterministic; they don't do anything until told to, and what they *are* told to do is precisely what is contained in the program written in machine language residing in their local memory -- a program chock full of rules like (R). But we are considering the possibility that the device CHESS is the CPU alone, without such a program. Of course, if this is the case, then the CPU will never execute *anything*. Still further, suppose again that the CPU by itself is the *device* CHESS. But then it would have been the *device* CHESS even before we ever loaded a chess-playing program onto our machine. And this conclusion is absurd.

This leaves us with possibility (3). Is the object program *together with* the CPU the *device* CHESS? Suppose it is. Then it can execute the rule (R), as required. So far, so good. Indeed, it can execute the object program just as the definition of execution given separately above requires. Even better. But now also note that the *device* CHESS contains, as a part, the object program. Thus the object program contains within it a

[4]Note that nothing else can be the device CHESS. The operating system, interpreter/compiler and the high-level program have already been ruled out by Cummins. See pp. 122 and 123.

representation of the rule (R). But then the *device* CHESS contains within it a representation of the rule (R). And this, as we by now know fully well, contradicts the claim that "CHESS simply executes the rule without representing it at all".

4. A Visit to the Chinese Room

It is worth visiting Searle's famous Chinese Room for a moment to see if the key to inexplicit information resides within. The Chinese Room merits a visit because, for Searle, it is a room completely devoid of meaning, or representational content. On the face of it, this is a similar situation to that we've been considering: namely, that the rules in a computer's program are inexplicitly represented when executed. The reader may recall that the occupant of the Chinese Room is given a batch of Chinese writing, and after checking some rules, hands another batch of Chinese writing out of the room. The occupant knows no Chinese. For the purposes of this discussion I replace the occupant of this room with one who receives expressions in the computer language **C+** and sends out (after consulting a table), appropriately translated lines of code in machine language. The new occupant is thoroughly computer illiterate -- to him the expressions he receives in **C+** are so many meaningless squiggles and the machine language instructions he passes on out of the room are so many meaningless squoggles.[5] Thus the new occupant plays the role of a compiler as discussed above, though he doesn't know it.

Our occupant also understands nothing regarding the computer program he is translating -- that it is a chess-playing program, that it was written in **C+**, and so on. Perhaps then, this room can now behave as the *device* CHESS described above was intended to behave. That is, the Chinese Room will execute the rule (R), IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN, without representing it. Even though there is an instruction representing the rule (R) in the high-level program handed

[5]Here I use the technical terminology of 'squiggles' and 'squoggles' introduced in (Searle, 1980).

into the room, there will not be a representation of (R) in the room when the program executes. Hence, semantic properties vanish in the room after the compilation takes place, perhaps in a manner amenable to Cummins' formulation of inexplicit information.

After a thorough walk-through and inspection of the Chinese Room however, I find it has little to offer in saving the notion of inexplicit information that we have been examining. Let me make two points to justify this claim.

The first point is that for Searle, the occupant of the room understands nothing about the computing job he is performing for his (external) programmers. Searle uses this fact as a way to show that a computer has no beliefs, or other attitudes we normally associate with intentional beings. But this is no different from the compiler described earlier, which blindly (but nomically) translated statements from **C+** to machine language. As far as I know, no extant computer understands its operations in the way Searle requires. Certainly my compiler doesn't understand (or need to) that is is compiling instructions for a chess-playing program, or that there is a rule in the program telling it to get its queen out early. This, I take it, is not an issue, and so the Chinese Room offers no help for inexplicit information here.

The second point is that representation in a computer can take other forms than those Searle concentrates on. For example, Searle will insist that the statement "if $n \leq 10$ {q1 = x};" that is handed to the occupant in the room will not represent anything -- *to the occupant*. But this does not mean the written statement, the actual physical token manipulated by the occupant, does not represent anything. Indeed, both Cummins and I have accepted throughout the above that this statement written in **C+** did represent something to the programmer, namely the rule (R): IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN. This is a representational content we have assigned by convention to this statement given our goal to write a chess-playing program that gets its queen out early.[6] The statement was assigned this representational content given the rest of the program, and our goal. Of course this same line could occur in numerous other computer programs and not represent anything like the same thing.

But what about the statements in machine language that the inhabitant passes out of the room after translating them? Do these represent anything? Before I answer this question, consider the following story. I call up my wife at her workplace and leave a message on her voice mail, saying “I mailed the package to your brother this morning.” Later in the day she listens to the message and gets the information.[7] Now, in this exchange, informational content flowed through various media before it reached the intended recipient, and translation occurred at many stages along the way. Acoustic signals generated by my speaking into the phone were picked up by the phone, translated into analogue electrical signals and transmitted to a phone exchange substation; these signals were then transmitted to my wife’s voice mail, where they were compressed, converted to digital form, and stored; and when she listened to the mail, these digital signals were translated back to analogue form which drove the speaker in her phone to play back my spoken message.

In this example I maintain that the original informational content has been preserved throughout the various stages of transmission and translation, and despite the various physical media transmitting and storing the states carrying this information. The information is transmitted, translated, and stored due to nomic relations between the various operating parts of the system. Neither I nor my wife know the exact details of

[6]A clear discussion of representational categories may be found in (Dretske, 1988). Representations according to agreed conventions are called Type I representations, according to the hierarchy of representations Dretske develops.
[7]Though 'representation' and 'information' are used interchangeably in this article (see footnote 2), I prefer to use 'information' when the content is true, as I'm assuming in this example. Besides, it just seems more natural to say that phone lines carry information rather than representations, though frankly, I'm sure just the opposite is true.

this transmission, translation and storage, but we don't need to.[8] As long as the nomic relations are as they should be for phone lines, exchange stations, digital storage media, phone speakers, etc. in the phone system, the information will be delivered. It is not only the acoustic signals (which *sound* like my voice) which carry the information “I mailed the package to your brother this morning”, it is also the electrical impulses and digital disk patterns which carry this information.[9] And again, they carry this information because of nomic relations between the various parts of the phone system which were designed to transmit phone conversations.

In the same fashion, the statements in machine language that the occupant passes out of the Chinese Room after translating them will represent what the statements handed to him in **C+** represented. The machine language statements passed out of the room will include the representation of the rule (R), IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN, since there was a statement representing that rule in the high-level program handed to the occupant. And this is because, by hypothesis (given us by Searle) there is a nomic relation ensuring that the statement handed to the occupant “if n ≤ 10 {q1 = x};” will be translated to its machine-language instruction “0110010001110001”. In a computer, this translation will occur just as dependably, and not just by hypothesis, since (de-bugged) compilers tend to be very reliable. In the Chinese Room, Searle ensures us by his hypothesis of the very same degree of reliability -- it is just delivered a bit more slowly, since the translation is delivered by ‘meatware’ (a person) rather than ‘hardware’ (a CPU). Thus the machine language statement will also represent the rule (R), by virtue of a nomic relation realized by the compiler, whether that compiler be instantiated in meatware or hardware.

[8]As correctly pointed out in (Dretske, 1981), our knowledge of these processes of transmission and translation is irrelevant to the accurate delivery of the information.
[9]Due to the Xerox principle of information (If A carries the information that B, and B carries the information that C, then C carries the information that C), the information content will be conserved at the various stages discussed in the example. See (Dretske, 1981).

5. Conclusion

We have seen that there are three possibilities for Cummins as to what the *device* CHESS is. We have also seen that none of these three meet his criteria for what this device should do: namely, execute a rule (R) without representing it. Thus the *device* CHESS does not exist as Cummins imagines it. We have also seen that analogies with the Chinese Room argument do not save the notion of inexplicit information. And this means that the strategy of attributing inexplicit information to a chess-playing computer by virtue of the computer *executing* a rule (R), where (R) is an instruction in a program, fails.

This conclusion means that if Dennett's chess playing computer carries inexplicit information, it does so for different reasons than those examined here. Nevertheless, Cummins gives us something worth thinking about when he says we can rightly intentionally characterize a system, even though the propositional contents of our characterizations are not 'tokened' (explicitly represented) in the system. Moreover, this inexplicit content, being somehow "lodged in . . . physical facts" is a lot closer to doing some real causal work than mere 'emergent' contents are. So now at least we have seen an alternative to Dennett's view, though I believe it is an alternative that fails.

References

Aho, A. and Ullman, J. (1979), *Principles of Compiler Design*, Reading, MA: Addison Wesley.

Cummins, R. (1986), 'Inexplicit Information', in M. Brand and R. Harnish, eds., *The Representation of Knowledge and Belief* , Tucson, AZ: University of Arizona Press, pp. 116-126.

Dennett, D. (1978), *Brainstorms*, Cambridge, MA: MIT Press.

Dennett, D. (1987), *The Intentional Stance*, Cambridge, MA: MIT Press.

Dretske, F., (1981), *Knowledge and the Flow of Information*, Cambridge, MA: MIT Press.

Dretske, F., (1988), *Explaining Behavior*, Cambridge, MA: MIT Press.

Searle, J., (1980), 'Minds, Brains and Programs', *The Behavioral and Brain Sciences* **3**, pp. 417-424.