

Computer Analogy Reconsidered

from a Perspective of Cognitive Linguistics and Object-Oriented Programming

Yoichiro Hasebe
Tokushima Bunri University
yhasebe@tokushima.bunri-u.ac.jp

1. Introduction

In recent studies in cognitive science, considerable criticism has been directed at the once-popular idea of likening our cognitive abilities to software in a digital computer. Until recently, this analogy—often called “computer analogy (CA)” —had been considered to be apt for analyzing how the human mind and language work as well as for developing artificial intelligence systems. However, many attempts based on CA ended in failure, and as a result, cognitive theories embracing CA lost much of their appeal.

Nevertheless, the analogy between the human mind and a computer is not a concept that should be discredited quickly; it can still be meaningful in an interesting way. Computer software, though itself being a purely formalistic kind of thing, is a product of many stages of development that definitely need “cognitive” work on the part of human beings, i.e., “programmers.” Moreover, software engineering has evolved rapidly in the last few decades after the most primitive types of programming languages were first developed. Due to this rapid evolution and the increasing recognition and acceptance of a new paradigm called “object-oriented programming,” significant attention is currently being paid to the cognitive aspect of software development. Therefore, it seems reasonable to expect that studies in cognitive science can benefit from advancements in software engineering, which will enable the development of a new type of CA that is truer to the structure and working of human cognition and language.

In this paper, I will elaborate this idea with a specific reference to one of the relatively new fields in cognitive science, namely, cognitive linguistics. The basic tenets in cognitive linguistics were developed in an attempt to critically reconsider the generative theory of language, which was put forth in the 1950s and has had enormous influence since then not only on linguistics but also on other disciplines including computer science. Interestingly, many theoretical notions and concepts in cognitive linguistics have remarkable similarities to those in object-oriented programming (OOP), a new paradigm of computer programming that attempts to overcome the defects of more formalistic methodologies that were used in the past.¹ Based on these observations, I will argue that the redefinition of CA in a more contemporary context can offer a new perspective on the study of human cognition and language.

2. Computer Programming as a Cognitive Process

In this section, I will first examine the reasons for which the traditional CA has been criticized

as being inappropriate in many studies in cognitive science; then, I will point out an issue that has not been considered in many such arguments against CA—the fact that a computer system is an artifact that can only be developed by human beings with their basic cognitive abilities.

At least three arguments exist against CA. The first is based on the fact that it seems improbable that any type of symbolic system is inherently implemented in the human brain. The physical architecture of the brain comprised of a great number of neurons does numerous types of information processing; a digital computer is also supposed to do similar processing with a different architecture. However, the human brain is required to deal with not only the formal aspect of information but also the semantic one and the relation between the two. A digital computer, on the other hand, only handles the former. Whenever the computer seems to be processing what are generally called “symbols,” it is merely manipulating the symbolic forms using some type of syntax that is already provided; the interpretation of the results of the computing process is always the responsibility of the person who uses the system.²

The second argument against CA arises from difficulties in assuming that high-level cognitive activities such as language use are made possible by a cooperative functioning of separate modules with specialized purposes. Generative grammar, for example, utilizes this type of “divide and conquer” method for explaining language; it makes a rigid distinction between language use and language competence—the latter is an independent system in the brain and is further divided into modules for specialized purposes such as syntax and semantics. However, studies in cognitive linguistics and some other related areas have revealed that the above theory is unlikely, especially when considering the significant roles that more fundamental cognitive abilities have in our language ability. Thus, understanding the place of language in the human brain in an analogous manner to that of program modules in a digital computer appears to be an incorrect approach.³

The third argument against CA is concerned with the nature of hardware, not software, of a digital computer. It was shown that systems with a rule-based, sequential processing architecture are not effective enough to deal with many highly complex tasks such as visual and aural recognition, but those with a parallel distributed processing architecture are often quite capable of performing such tasks.⁴ This implies that human cognitive abilities cannot be fully simulated using a conventional digital computer. Therefore, a computer with a more brain-like architecture for computation should be developed for accurately simulating the tasks handled by the human brain.⁵

The above arguments against CA are basically valid. However, there is a caveat. Many such arguments are based on the presupposition that a computer system with its implemented programs is a self-contained, autonomous object. It is true that a computer system appears to operate rather automatically—once it is configured properly, it accepts incoming data, processes them according to a given set of rules, and changes its own state for later processing or outputs the result; all these stages may occur without much human intervention. However, the implemented set of rules that determines the function of the system as a whole is nothing other than the result of the work by human programmers, which is intrinsically subjective and intentional. Hence, the design of a computer program inevitably reflects more or less subjective views toward the problem domain.

In fact, it does not matter how the program code was originally written (the high-level language used) or how it was designed as long as the code clearly instructs the computer to

output the desired results. To be executed, a program must be translated to machine language that consists of only a sequence of 0s and 1s. For programmers, however, the manner in which program code is designed and written is of great importance because the efficiency of development and ease of maintenance largely depend on that. Therefore, researchers in the fields of computer science and software engineering have incessantly tried to improve the methodology of programming, and they have developed languages that are able to more closely reflect the programmers' views toward the problem domain and the relationships among elements observed in it. Thus, it is possible, and even necessary, to redefine CA in terms of the cognitive aspects by not treating "computer" as merely a general term for artificial devices with a certain type of physical architecture.⁶

3. Levels of Programming Languages

In this section, different levels of programming languages are briefly described. It is shown that the development of languages is a gradual refinement to the framework with which programmers can express their conception and computational implementation of the problem domain.

As mentioned already, a digital computer can only execute code sequences written in a format called "machine language." Machine language is essentially a representation of the instructions at the lowest level that can be directly fed to the central processing unit. Naturally, the structure of machine language is based on a purely architecture-oriented logic, and in fact, it is composed of rather cryptic sequences of 0s and 1s (or ons and offs). In many cases, machine language expresses an instruction with four-bit sequences of binary data; each instruction is so primitive that it only instructs the processor to perform tasks such as loading a data item from a certain memory address and storing it in another address. Therefore, it is extremely difficult, or at least painstaking, for programmers to deal directly with this type of program code.

| instruction | machine language | assembly language |
|---|---------------------|-------------------|
| Load an item of data from memory address 366. | 0000 0001 0110 1110 | LOAD x |
| Add two numbers held in registers 1 and 2. | 0100 0000 0001 0010 | ADD R1 R2 |
| Jump to instruction 13 if the result of the previous operation is zero. | 1100 0000 0000 1101 | JUMPZ h |

Table 1

Thus, for the purpose of simplifying the reading and writing of such obscure sequences of machine language, a mnemonic system was invented. This alternative—known as "assembly language"—consists of words that directly correspond to instructions in binary code, and it is translated, or "assembled," to binary code when executed. Table 1 shows examples of representations in machine language and assembly language, both of which instruct the processor to perform the same tasks.

However, programming in assembly language is not essentially different from that in machine language. This is because it is still not possible to directly express concepts in the problem domain using assembly language—the programmer has to deal with a very low level of abstraction wherein most instructions are expressed in terms of memory addresses. Accordingly, even to implement a simple task such as that represented below in (1), i.e., finding the area of a triangle with sides a, b, and c, one needs to write a significantly lengthy series of instructions in assembly language, as shown in (2).⁷

$$(1) \sqrt{(s \times (s - a) \times (s - b) \times (s - c))}, \text{ where } s = (a + b + c) / 2$$

```
(2) LOAD R1 a; ADD R1 b; ADD R1 c; DIV R1 #2; LOAD R2 R1;
    LOAD R3 R1; SUB R3 a; MULT R2 R3;
    LOAD R3 R1; SUB R3 b; MULT R2 R3;
    LOAD R3 R1; SUB R3 c; MULT R2 R3; LOAD R0 R2; CALL sqrt
```

Consequently, many so-called high-level languages that enable the programmer to think at a higher level of abstraction were developed; this is because the human mind is not well suited for reading and writing lower-level languages such as machine and assembly languages. In fact, what can be expressed in sophisticated languages can also be written in assembly language or even machine language. However, high-level languages such as COBOL, LISP, BASIC, and C are equipped with many useful features that do not exist in the lower-level languages; by making use of these features, the programmer can design and write programs more easily and effectively. Among such features found in most high-level languages at present are “expressions,” “data types,” “control structures,” and “functions.” The algebraic statement in (1) can be represented in a concise manner using a high-level language, as shown in (3).⁸

```
(3) let s = (a + b + c) / 2
    in sqrt (s * (s - a) * (s - b) * (s - c))
```

However, programming in a high-level language does not necessarily allow every concept in the problem domain to be represented directly in the source code. It is inevitable for the programmer to go through a thought process of extracting essential elements and relations from the problem domain and translating them to appropriate step-by-step sequential procedures that can yield desirable results. This method of developing a computational system can actually produce a good source code that is compact and highly efficient in terms of execution time. However, such codes tend to be extremely complex and hard to understand; they can become so convoluted that it becomes virtually impossible for anyone who was not involved in the development process to debug or try to add extra functions; even the very people who wrote the code may find them extremely difficult.

To overcome these problems, OOP was invented. The basic concepts of OOP were developed in the 1960s and 70s. It became widely recognized during the 1980s and 90s as OOP languages such as Smalltalk, C++, and Java gained increasing popularity. OOP emphasizes the significance of programmers’ viewpoints for solving the problem in question.

The basic programming procedure in OOP can be described as follows. First, certain aspects of the problem domain are focused on and elements participating in the domain and relationships among them are identified. Then, these elements are computationally implemented as “objects” that possess not only their own static properties but also dynamic behaviors that determine how they are related to other objects. The resulting source codes that are written by following this object-oriented procedure tend to be quite readable; thus, it is easy to both maintain and extend them.

Accordingly, programming languages and software design methods have changed and increased in sophistication in the last few decades. It can be said that they have gradually evolved to place greater importance on logic on the part of the programmers while distancing them from the physical mechanism of the computer. The arguments against CA in cognitive science are valid as long as they are concerned with the architectural difference between the human cognitive system and the digital computer. Nevertheless, considering the fact that a computer system is nothing more than an artifact that is created and used by humans, it seems quite reasonable and necessary to extend the general definition of the concept of the computer to include its design aspects. Such an extended concept of the computer can offer a strikingly close resemblance to the mechanism of human cognitive systems. This becomes particularly apparent when a comparison is made between concepts in OOP and those in cognitive linguistics, as presented in the next section.⁹

4. Theoretical Concepts of OOP and Cognitive Linguistics

OOP aims to simulate a problem construed from a certain point of view that can subsume the elements taking part in the problem domain. Apart from possessing basic features of non-OOP languages, OOP languages are equipped with many advanced features that most non-OOP languages do not possess. These advanced features make it possible for programmers to think about the problem domain in a realistic manner and implement the concepts in it without putting in effort to reorganize them according to the logic of the digital computer. The tedious procedure of reorganization can be left to the compiler, which transforms the source code into native code written in machine language. In fact, the theoretical concepts of OOP and those of cognitive linguistics bear a remarkable resemblance despite the apparent difference between these two fields. In this section, I will introduce some of those concepts and provide a brief description of each of them.¹⁰

4.1 Objects, Classes, and Instances

The concepts of “object” in OOP and “element” in Ronald Langacker’s cognitive grammar seem to be compatible with each other. An object may be an instance of some type of class, which is a higher-order structure that works as a prototype of all the possible objects sharing the same types of properties and behaviors. In cognitive grammar, elements are defined in a similar fashion. Among the properties of an element, the distinction between “thing” and “relation” is the most important. Furthermore, the set of possible behaviors exhibited by an element can be defined in cognitive grammar using various theoretical notions such as “action chains,” “active zones,” and “trajector/landmark” (Langacker 1991, 1999).

Moreover, just as objects in OOP can form increasingly larger structures by relating

themselves with other objects through “methods” (functional implementation of behaviors) of their own or those of the other objects and/or by including other objects as their internal components, elements in cognitive grammar can act upon each other and/or merge themselves into a composite structure. A composite structure itself takes on the status of an element and can, in turn, serve as a component structure for further composition with other elements; this scenario is similar to the object composition in OOP.

4.2 Inheritance, Specialization, and Generalization

In OOP, one can easily make new objects with features similar to those of existing objects by allowing the new objects to inherit properties and behaviors from the existing ones. While doing so, it is either possible to override and alter some of the inherited properties and behaviors or even implement additional features and functionalities to make the new objects more specialized in certain respects. Conversely, one can also increase the abstraction or generalize details of rather specialized objects and make so-called super classes. In this manner, many new objects with the desired properties can be constructed easily.

It is quite obvious that the philosophy behind these characteristics of objects in OOP have much in common with ideas in cognitive linguistics with regard to relationships among meanings and functions of words and expressions. For instance, Langacker’s network model of linguistic categories has the concepts of elaboration, extension, and schematization, which may be considered to be almost identical counterparts of the OOP concepts of inheritance, specialization, and generalization. Moreover, the radial network model of categories, which has a great importance in many cognitive linguistic theories, is considered to be fairly similar to Langacker’s network model, and accordingly has also much in common with the object model in OOP.

4.3 Polymorphism

The mechanism of inheritance between objects endows OOP with another powerful feature called polymorphism. In many OOP languages, the programmer can extend an object by implementing more than one method with the same name; these methods have more or less the same purposes but take different types of arguments. A user can call the appropriate method merely by passing any type of argument to the object as long as the object’s application program interface (API) lists that as a possible argument type for methods of the specified name. For instance, an object for calculating the area of a shape can be configured so as to deal with many different shapes without requiring the users to think about whether the shape in question is a circle, triangle, or square.

Linguistic polysemy, one of the most actively debated research areas in linguistics, can be considered to be a naturally occurring phenomenon of what is simulated as polymorphism in OOP. In linguistic literature, the exact mechanism that gives rise to polysemic expressions does not seem to be completely clear as yet. Studies in cognitive linguistics, however, have suggested some promising research ideas that are even capable of accounting for problems regarding metaphoric meanings and literal ones, which are the most extreme cases of polysemy. Among those ideas in linguistic polysemy, Gilles Fauconnier’s theory of mental spaces might be the most notable from the perspective of the present study. This is because the theory

acknowledges the importance of different domains as well as that of different roles that elements with the same name can have according to the domain they are involved in (Fauconnier 1985). These are, in fact, two of the most important concepts in OOP as well.

4.4 Data Hiding and Encapsulation

Since OOP emphasizes the importance of data integrity, many OOP languages have features to enforce it. Properties and methods of an object that should not be altered or not even be accessible to other objects can be made “private.” These private properties and methods are encapsulated in the object and become virtually invisible to other objects. When it is necessary for an object to obtain a value or execute a method that is privately held in another object, the client object has to access that through the other object’s API and request it to execute the desired procedures. Thus, at least ideally, every object is supposed to be more or less autonomous in the sense that it provides its services only to objects that request it in a designated manner. The details of the internal mechanism adopted to enable the services do not matter as long as the result fully satisfies what the object advertises in its API.

A similar mechanism is often observed in the case of natural languages. Words and phrases do not necessarily convey their meanings and purposes by themselves; they do so only when used in appropriate contextual and grammatical environments (Kay 1997). For example, the meaning of the noun *click* as the pressing of a button of a pointing device is conveyed only when it is used in the right contextual and grammatical environment. Otherwise, *click* may either be interpreted as a sharp sound or as a verb instead of a noun.

The linguistic phenomena of grammaticalization and subjectification are good examples to make this point (Hopper and Traugott 1993, Sweetser 1990). A novel linguistic expression is likely to be fully analyzable when it is first devised; basically, it conveys only the meaning that its component structures express collectively. Then, as time goes by, the same expression may gradually take on the status of a set phrase, an idiom, or one of the limited instances of a certain grammatical unit (as in the case of *be going to*, see Langacker (1990)). At this stage, the expression tends to have lost much of its original analyzability (in other words, public interface to the internals); it may be left with highly specific meanings and functions that are conveyed only within appropriate contextual and grammatical environments. These phenomena offer a strong analogy to the fact that in OOP, objects with complexly structured internal mechanisms tend to be used only for highly specific purposes; they are not very extendable, and they are just required to perform the relevant tasks. The users of such objects, however, do not care much about their mechanisms or the manner in which they were constructed, just like a speaker of English does not have to know much about the morphology or etymology of words to just use them.

5. Goals of OOP and Cognitive Linguistics

It has been established that the methodologies of cognitive linguistics and OOP have much in common. Now, the question is what makes these two fields with seemingly different purposes—elucidating the mechanisms behind human cognition and language (cognitive linguistics) and improving the efficiency of software development and the robustness and maintainability of computer programs (OOP)—share so many features and conceptual aspects?

To answer this question, it would be helpful to summarize the basic approaches used in OOP and cognitive linguistics

OOP attempts to enable programmers to find a natural, or at least coherent, description of the elements and relations observed in the problem domain; the premise is that OOP is a secure method to easily develop robust software of high quality, although there “could” be other ways to write programs having the same functions more concisely and even elegantly. Priority is not given to the efficiency of the source code that is to be compiled into machine language, but to the efficiency of the process by which the human mind tries to analyze and describe real-world events and problems in a machine-readable format. Therefore, it can be said that OOP is a type of “cognitive engineering” since it is a framework for producing something that is in accordance with the manner in which human cognition works naturally.

Cognitive linguistics, on the other hand, tries to study human cognition by taking an approach that considers language as a reflection of our fundamental view toward the world. The ability to generate and interpret sentences is not preprogrammed and modularized somewhere in the brain as a series of computational procedures; this ability arises from the interaction between basic cognitive abilities and the experience gained by numerous encounters with events and entities in the world. This approach, therefore, may be referred to as “cognitive reverse-engineering” because it attempts to find out the basic structures and patterns of cognition by examining an already existing construct called language.

Thus, I argue that the question raised above can be answered by stating that both OOP and cognitive linguistics have a common objective, namely, determining the most fundamental and natural ways in which the human mind works when it deals with various problems in the world. Therefore, the difference between the two disciplines, aside from numerous superficial ones, may only be the directions taken to achieve the objective, i.e., either constructing artifacts (computer programs) or analyzing existing ones (natural languages).¹¹

6. Conclusion

The idea presented in this paper may not necessarily be new. Similar observations have been made both in cognitive science and computer science. For example, Deacon (1997) pointed out the analogical relationship between the impact of the invention of the graphical user interface (GUI), which is object oriented by nature, on computer operation in general and the meaning of the emergence of a human language. Furthermore, there have also been some attempts to implement a cognitive linguistic framework into a computational system (Barnden 1998, Holmqvist 1998).

However, it seems worth pointing out that the analogy of cognitive linguistics and OOP should be pursued in greater detail because it has much more to offer than just providing another novel way of describing these relatively new theoretical frameworks. If cognitive linguistics and OOP truly have the same objective with different approaches, as argued in the previous section, both will greatly benefit from further research based on this study. It is time to get over the preconceived ideas against the traditional (and undoubtedly wrong) CA and appreciate the significant value of the analogy between language and cognition on one hand and computer and software development on the other.

Notes

1. Introductory information on OOP is available in many books on computing and programming, which are not necessarily academic or highly technical. Barker (2000) and Weisfeld (2000) are among those that are fairly accessible.
2. The reasoning behind this argument was vigorously expounded by the philosopher John Searle. As Searle (2002) describes, the meaning of “computer” has changed drastically since Alan Turing used this word in his monumental 1950 article on the nature of computing and intelligence.
3. Lakoff and Johnson (1999) attack the popular view of the human mind as a computer program composed of many subprograms with specialized functions, and they argue that its popularity is due to the ironical fact that the “mind as computer” metaphor seems “intuitive” to many people.
4. See MacWhinney (2000) and Elman (2001) for implications of the idea of parallel distributed processing (also known as “connectionism”) on the study of language and language learning.
5. At present, many experiments in parallel distributed processing are conducted using computers that can only process sequentially (von Neumann computers); they are configured so as to simulate parallel processing computers.
6. In fact, practicing systems developers have been noticing the psychological aspect of programming and its effects on the quality of the development process and the final product (Weinberg 1971). Unfortunately, this awareness has not been adequately conveyed to researchers studying cognition in more general terms.
7. The examples of program codes in this section are borrowed from Watt and Brown (2000).
8. The language used in this code is not one of the major high-level languages; it is a pseudo-language used in Watt and Brown (2000) for instructional purposes. Nevertheless, this does not affect the point made here.
9. Some researchers have noticed the analogical relationship between the levels of programming languages and levels of structures of human cognition. Among them are Rumelhart and McClelland (1986) and Jackendoff (1992). However, their arguments are only aimed at describing higher-level functions of the brain as abstract computational processes. Moreover, they do not explore the nature of OOP languages, which is radically different from that of older high-level languages that they refer to.
10. There are many other points where OOP and cognitive linguistics show resemblance, which are not taken up in this section. Two of such points worth mentioning here are that (1) OOP has a meticulous diagramming system called “Unified Modeling Language (UML)” to schematically represent objects and their interrelations, and (2) in OOP, there is a general recognition of the importance of accumulated knowledge on constructions used frequently in practice, which are often called “design patterns.” For further information and examples, see Scott (2001) for an accessible introduction to UML and Gamma et al. (1995) for the first attempt in compiling a useful collection of design patterns.
11. The fact that there are two approaches for finding out how human cognition works—one synthetic and the other analytic—is also explicated in Franklin (1997).

References

- Barker, Jacquie. 2000. *Beginning Java Objects: From Concepts to Code*. Birmingham: Wrox Press.
- Barnden, John A. 1998. An AI System for Metaphorical Reasoning about Mental States in Discourse. In Jean-Pierre Koenig, *Discourse and Cognition: Bridging the Gap*. Stanford: CSLI, 167–88.
- Deacon, Terrence W. 1997. *The Symbolic Species: The Co-evolution of Language and the Brain*. New York: W. W. Norton.
- Elman, Jeffrey L. 2001. Connectionism and Language Acquisition. In Michael Tomasello and Elizabeth Bates, *Language Development: The Essential Readings*. Oxford: Blackwell, 295–306.
- Fauconnier, Gilles. 1985. *Mental Spaces: Aspects of Meaning Construction in Natural Language*. Cambridge, MA: MIT Press.
- Franklin, Stan. 1997. *Artificial Minds*. Cambridge, MA: MIT Press.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Holmqvist, Kenneth. 1998. Conceptual Engineering: Implementing Cognitive Semantics. In Jens Allwood and Peter Gärdenfors, *Cognitive Semantics: Meaning and Cognition*. Amsterdam: John Benjamins, 153–71.
- Hopper, Paul J. and Elizabeth C. Traugott. 1993. *Grammaticalization*. Cambridge: Cambridge University Press.
- Jackendoff, Ray. 1992. *Consciousness and the Computational Mind*. Cambridge, MA: MIT Press.
- Kay, Paul. 1997. *Words and the Grammar of Context*. Stanford: CSLI.
- Lakoff, George. 1987. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. Chicago: The University of Chicago Press.
- Lakoff, George and Mark Johnson. 1999. *Philosophy in the Flesh: The Embodied Mind and its Challenge to Western Thought*. New York: Basic Books.
- Langacker, Ronald W. 1990. Subjectification. *Cognitive Linguistics* 1, 5–38.
- Langacker, Ronald W. 1991. *Foundations of Cognitive Grammar, Vol. 2: Descriptive Application*. Stanford: Stanford University Press.
- MacWhinney, Brian. 2000. Connectionism and Language Learning. In Michael Barlow and Suzanne Kemmer, *Usage-Based Models of Language*. Stanford: CSLI, 121–49.
- Rumelhart, David E. and James L. McClelland. 1986. PDP Models and General Issues in Cognitive Science. In David E. Rumelhart, James L. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA: MIT Press, 110–46.
- Scott, Kendall. 2001. *UML Explained*. Reading, MA: Addison-Wesley.
- Searle, John R. 2002. *Consciousness and Language*. Cambridge: Cambridge University Press.
- Sweetser, Eve E. 1990. *From Etymology to Pragmatics: Metaphorical and Cultural Aspects of Semantic Structure*. Cambridge: Cambridge University Press.
- Turing, Alan M. 1950. Computing Machinery and Intelligence. *Mind* 59, 433–60.
- Watt, David A. and Deryck F. Brown. 2000. *Programming Language Processors in Java: Compilers and Interpreters*. New York: Prentice Hall.
- Weinberg, Gerald M. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.
- Weisfeld, Matt. 2000. *The Object-Oriented Thought Process*. Indianapolis: Sams.

<要旨>

コンピュータ・アナロジー再考 認知言語学とオブジェクト指向の観点から

長谷部 陽一郎
徳島文理大学

近年の認知科学では、かつては有力な見方であった、ヒトの認知能力や言語能力を脳というハードウェアに対するソフトウェアとみなすいわゆるコンピュータ・アナロジー（CA）に対する批判が数多くなされてきた。本稿では、CAの本質をいま一度問い直し、現代のコンピュータ科学およびプログラミング理論に照らし合わせて考えるならば、それが必ずしも完全に否定されるべきものでないことを主張する。そして、近年のソフトウェア開発において特に重要な意味を担ってきたオブジェクト指向プログラミング（OOP）の方法論と、Langacker、Lakoffらの理論に代表される認知言語学の方法論との共通性に注目し、新たな視点からCAを見直すならば、むしろ認知科学に新たな展望がもたらされる可能性があることを論じる。

CAに対する批判の論点としては様々なものがあるが、主要なものは次の3つである。第1に、記号情報処理のあり方に関する脳とコンピュータにおける本質的な相違、第2に、ある種の機能に特化した複数のモジュール的構造を仮定することの問題点、そして第3に、線条的処理を想定したモデルによってヒトの認知構造を捉えることの不適切さである。これらの批判は基本的に正しい。しかし問題はその多くがコンピュータ・プログラムを、人間の視点から完全に切り離された存在物であるかのように見なしていることである。実際には、あらゆるプログラムの設計にはプログラマという主体が問題領域を主観的に切り取る視点が必然的に反映される。このことを考えると、コンピュータ・プログラムは一種の自律的な存在物としてではなく、むしろその開発工程を含めた一連の認知プロセスの結果物として捉えられるべきであると言える。

ソフトウェア開発の効率を向上させ、より頑強で保守・拡張性の高いプログラムの作成を可能にするために、プログラミング言語はこの数十年の間に、コンピュータの側の形式的な論理に基づく低レベル言語から、ヒトの知覚や認識の性質を取り入れた、プログラマの側の非形式的な論理に基づく高レベル言語へと進化していった。その一つの結果としてのOOPの方法論は、プログラミングにおける「世界を見る視点の重要性」を踏まえ、従来の手続き型プログラミングの方法論には含まれなかった、より高次の理論的概念を機能として実現している。そして興味深いことに、その多くが、言語学という分野の発展の一つの成果である認知言語学において広く論じられている理論的概念との間に対応関係を示している。例を挙げると、OOPにおける「オブジェクト」の概念は認知言語学でいうところの「要素」に、「継承・汎化・特殊化」の概念は「カテゴリーのネットワークモデル」に、「多態性」の概念は「多義性」にそれぞれ対応している。さらに「データの隠蔽・カプセル化」の概念は、自然言語における「文法化・主体化」との間に対応関係を示している。

OOPと認知言語学は、ある共通の対象に異なる方向からアプローチする2つの分野であると言える。前者が、ソフトウェアの開発という構成論的な作業の効率化のために人間の認知システムの基本的な構造を探る「認知的エンジニアリング」であるのに対し、後者は、自然言語の本質の追及という多分に分析的な作業における理論的基盤として人間の認知システムの構造を解明しようとする「認知的リバース・エンジニアリング」であると言える。本稿は、これらの事実に基づき、現在、認知科学の様々な領域で一種の共通認識となっている「誤謬としてのCA」とらわれることなく、より現代的な視点からそれを再定義しなおすことを提案するものである。