Fads and Fallacies about Logic

John F. Sowa, *VivoMind Intelligence, Inc.*

Throughout the history of AI, logic has been praised by its admirers, maligned by its detractors, and discussed in confusing and misleading terms by almost everybody. Among the pioneers, John McCarthy has always been a strong promoter of logic, but Marvin Minsky has been a skeptic who experimented with a wide range of alternatives. Roger Schank had no doubts about logic, which he denounced at every opportunity. He introduced the distinction between the *neats*, who used logic for everything, versus the *scruffies* like himself, who developed notations that were specifically designed for the problem at hand.

Even advocates of logic have disagreed among themselves about its role, the subset appropriate to any particular problem, and the trade-offs among ease of use, expressive power, and computational complexity. The debates introduced many valuable ideas, but the hype and polemics have often confused the issues. This article reviews the controversies and suggests design options that can take advantage of the strengths of logic while avoiding the fads and fallacies.

Language and logic

No discussions about logic have been more confused and confusing than the debates about how logic is related to natural languages. Historically, logic evolved from language. Its name comes from the Greek *logos*, which means word or reason and includes any language or method of reasoning used in any of the -ology fields of science and engineering. Aristotle developed formal logic as a systematized method for reasoning about the meanings expressed in ordinary language. For the next two millennia, formal logic was expressed in a stylized or *controlled* subset of a natural language: originally Greek, then Latin, and later modern languages.

In the 19th and 20th centuries, mathematicians took over the development of logic in notations that diverged far from its roots. Yet every operator in logic is a specialization of some word or phrase in natural language: $\exists$ for *there exists*, $\forall$ for *every*, $\wedge$ for *and*, $\vee$ for *or*, $\supset$ for *if-then*, $\sim$ for *not*, $\Diamond$ for *possibly*, and $\Box$ for *necessarily*. The metalevel words for talking about logic and deduction are the same words used for the corresponding concepts in natural languages: *truth*, *falsity*, *reasoning*, *assumption*, *conclusion*, and *proof*. Although mathematical logic might look very different from ordinary language, every formula in logic expresses the same meaning as some natural language sentence. Furthermore, every step of every proof corresponds to an argument in ordinary language that's just as correct and cogent as the formal version.

What makes formal logic hard to use is its rigidity and its limited set of operators. Natural languages are richer, more expressive, and much more flexible. That flexibility permits vagueness, which some logicians consider a serious flaw, but a precise statement on any topic is impossible until all the details are determined. Formal logic can only express the final result of a lengthy process of analysis and design. Natural language, however, can express every step from the earliest hunch or tentative suggestion to the finished specification.

In short, there are two equal and opposite fallacies about logic and language: at one extreme, logic is unnatural and irrelevant; at the other extreme, natural language is incurably vague. A more balanced view should recognize the virtues of both: logic is the basis for precise reasoning in every natural language; but without vagueness in the early stages of a project, it would be impossible to explore all the design options.

What is logic?

Unreadability is a common complaint about logic, but that's only true of 20th century mathematical logic. In the middle ages, the usual notation was a controlled natural language, often supplemented with diagrams. For example, the *Tree of Porphyry* (figure 1) displayed Aristotle's categories and method of definition by genus and differentiae. Ramon Lull invented a method of defining categories by rotating and aligning disks with inscribed attributes. Inspired by Lull's system, Gottfried Leibniz used algebra to define categories by conjunctions of attributes, which he encoded as prime numbers. In the 19th century, George Boole used algebra to represent propositions, and Gottlob

1541-1672/07/$25.00 © 2007 IEEE
Published by the IEEE Computer Society

Frege invented a tree notation to represent the quantifiers and operators of first-order logic. Charles Peirce represented FOL by adding quantifiers to Boolean algebra, but he later invented *existential graphs* as a logically equivalent notation.

Many logicians limit logic to a narrow range of mathematical notations, and most nonlogicians don't think of their notations as a kind of logic. Yet almost any declarative notation—graphic or linear—could be treated as a version of logic. A *logic* is any precise notation for expressing statements that can be judged true or false. A *rule of inference* is a truth-preserving transformation: when applied to a true statement, the result is guaranteed to be true. To clarify the notion of "judging," Alfred Tarski defined a *model* as a set of entities and a set of relationships among those entities. *Model theory* is a systematic method for evaluating a statement's truth in terms of a model. According to these definitions, a database, in relational or network format, is a model, and the method of evaluating the WHERE clause of an SQL statement is equivalent to Tarski's evaluation function. With a precise specification and an evaluation function, many other notations, including controlled natural languages, could qualify as a version of logic.

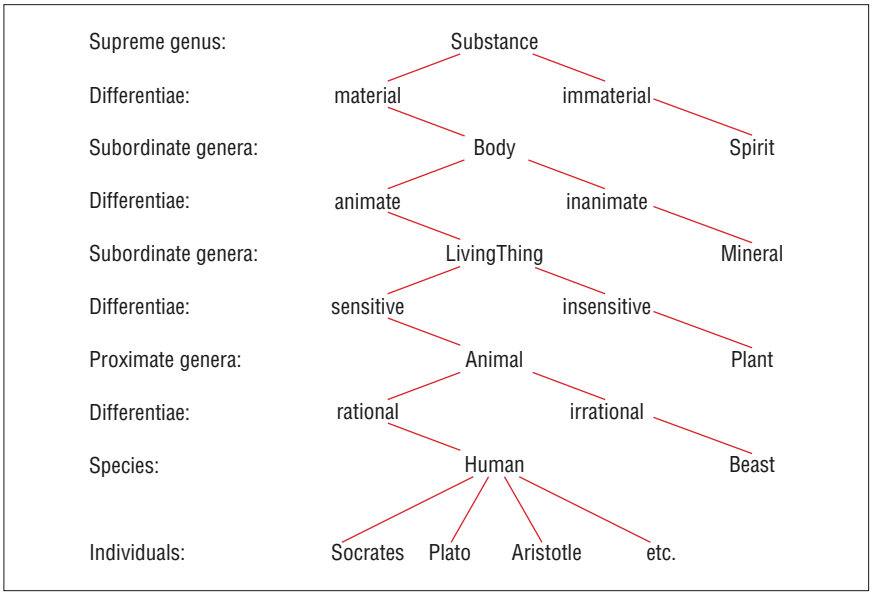

Figure 1. The *Tree of Porphyry* displays Aristotle's method of defining categories.

Deriving procedures from declarations

A procedure can only be used in one way, but a declarative specification has many possible uses. Directions from a highway to a hotel, for example, specify a procedure for following a single path, but a map is a declarative specification that determines all possible paths. With a map, a person or computer can derive a procedure by tracing a path between any two points. In computer science, *grammars* are logic-based declarations with many possible uses: analyzing a sentence, generating a sentence, detecting errors in a sentence, or suggesting corrections to a sentence.

For over 40 years, AI researchers have periodically revived debates about procedural or declarative languages. In database systems, tables and graphs are equivalent ways of expressing data, but queries about the data can be represented in two ways: a path-based procedure for navigating the data or a declarative statement in a logic-based language such as SQL. In the 1970s, the debate was settled in favor of SQL, but object-oriented databases have revived the argument for a procedural approach:

- Data represented in a tree or graph can be viewed as a road map.
- Procedural directions are the most efficient way to tell a computer how to get from one point to another.
- If the programmer specifies the access path, the computer doesn't need complex algorithms to derive it.

For some systems, these arguments are valid. But the following arguments from the 1970s are just as valid today:

- Tables and graphs specify logically equivalent access paths.
- Optimizing algorithms can derive more efficient paths than most programmers.
- A query that takes a few lines to declare in SQL can expand to several pages of highly error-prone procedural directions.

For these reasons, the major vendors of object-oriented databases support SQL as an alternative to path-based procedures. When given the choice, most users prefer SQL, even for databases whose native organization is a network.

In summary, the choice of procedural or declarative methods depends on the available tools. Because current computers execute procedures, a compiler must translate declarations to procedures. Although declarations allow different procedures to be generated for different purposes, good performance requires good optimization algorithms. If no optimizer is available, programmers may have to use a procedural language that maps directly to machine language.

Object language and metalanguage

Language about language, or metalanguage, is ubiquitous. As an example, the psycholinguist John Limber[1] recorded the following sentence by Laura, a 34-month-old child:

> When I was a little girl, I could go, "Geek geek," like that; but now I can go, "This is a chair."

This sentence involves metalanguage about the quotations. But logically, the modal auxiliaries "can" and "could" are metalevel comments about the containing clauses. In effect, Laura's sentence includes two levels of metalanguage about the quotations.

In adult speech, metalanguage is so common that it's often unrecognized. The modal verbs are a special case of metalevel expressions that include adverbs such as "possibly" and clauses such as "I doubt that" or "The odds are against it." Like modality, theories of probability, certainty, or fuzziness can be represented by first-order statements about first-order statements.

Some people argue that metalevel representations are complex or inefficient. But for

many applications, metalanguage can significantly reduce the complexity, as in the following controlled English sentences and their translations to an algebraic notation:

"Every dog is an animal"
$\Rightarrow (\forall x)(\text{dog}(x) \supset \text{animal}(x))$

"Dog is a subtype of Animal"
$\Rightarrow \text{Dog} < \text{Animal}$

The first sentence is translated to predicate logic, but the second is a metalevel statement about types. Statements with the relation < between two constants define a type hierarchy, which could be encoded in bit strings or products of primes. Subsequent reasoning with a *typed* or *sorted* logic can replace a chain of inferences with a divide instruction or a bit-string comparison.

These examples illustrate an important use of metalanguage: stratifying the knowledge representation in levels that can be processed independently by simpler algorithms. Since antiquity, logicians have considered the type hierarchy a privileged level with greater *entrenchment* than ordinary assertions. They invented special techniques for reasoning about it, such as Aristotle's syllogisms, Porphyry's tree, or modern *description logics*. The greater entrenchment gives type statements a modal effect of being *necessarily true*.

As another example of entrenchment, database constraints are obligatory with respect to ordinary assertions called *updates*. Three levels—a type hierarchy, database constraints, and updates—can support multimodal reasoning with a mixture of modes that are necessary or obligatory for different reasons.[2]

Expressive power and computational complexity

Computational complexity is a property of an algorithm, and the complexity of a program depends on the complexity of the algorithm it embodies. Statements in logic have no inherent complexity apart from the algorithms that process them for various purposes. Algorithms for proving statements in first-order logic, for example, might take an exponential amount of time or even loop forever. Yet the worst cases rarely occur, and theorem provers can be efficient on the FOL statements people actually use. In 1910, when Whitehead and Russell wrote the *Principia Mathematica*, theories of computational complexity were unknown. Yet in 1960, a program by Hao Wang proved all 378 of their theorems in propositional and first-order logic in just 7 minutes.[3] That was an average of 1.1 seconds per theorem on an IBM 704, a vacuum-tube machine with an 83-kilohertz CPU and 144 Kbytes of storage.

For database queries and constraints, SQL supports full FOL, but a database system can evaluate most queries in linear or logarithmic time, and even the worst-case examples take no more than polynomial time. That performance enables SQL algorithms to process terabytes or petabytes of data and makes FOL, as expressed in SQL, the most widely used version of logic in the world. For example, consider the query:

Find John Doe's department, manager, and salary.

The language used to state a problem has no effect on complexity. Reducing the expressive power of a language only makes some problems impossible to state.

With an index on the employee field, a relational database could find the answer in time proportional to (log N), where N is the number of employees. The following query would take (N log N) time:

Find all employees who earn more than their managers.

This query would take N steps to find the manager and salary of each employee. Finding the salary of each employee's manager introduces the (log N) factor.

Even complex queries can be evaluated efficiently if they process a subset of the database. Suppose that the complex condition in the following query takes time proportional to the cube of the number of entries:

For all employees in department C99, find [*complex condition*].

If a company has 10,000 employees, N^3 would be a trillion. But if department C99 has only 20 employees, then 20^3 is only 8,000.

Although computational complexity is important, complexity is a property of algorithms and only indirectly a property of problems. The language used to state a problem has no effect on complexity. Reducing the expressive power of the language cannot solve any problems faster; it only makes some problems impossible to state.

Using logic in practical systems

The hardest knowledge representation task is to analyze knowledge about a domain and state it precisely in any language. Since the 1970s, knowledge engineers and systems analysts have been eliciting knowledge from domain experts and encoding it in computable forms. Unfortunately, database-design tools have been disjoint from expert-system tools; they use different notations that require different skills and often different specialists. If all the tools were based on a common, readable notation for logic, the number of specialists required and the amount of training they need could be reduced. Furthermore, the domain experts would be able to read the knowledge representation, detect errors, and even correct them.

Before the 20th century, people used readable notations for logic: controlled natural languages supplemented with type hierarchies and related diagrams. Although full natural language with all its richness, flexibility, and vagueness is still a major research area, the technology for supporting controlled NLs has been available since the 1970s. Two major obstacles have prevented such languages from becoming commercially successful:

- The supporting tools have been isolated from the mainstream of commercial software development.
- Most software developers aren't linguists or logicians.

The second challenge is easier to address. The ontologies being developed today are usually aligned to freely available resources, such as WordNet, which contains sufficient linguistic information for processing controlled NLs. But the challenge of integrating all the tools used in software design and development isn't a technical problem. It's an even more daunting problem of fads, trends, politics, and standards.

Although controlled NLs are easy to read, the people who write them need training, good help facilities, and tools for mapping the languages to current software. An example of such a tool is a knowledge compiler that extracted a subset of axioms from the Cyc system to drive a deductive database.[4] It translated Cyc axioms, stated in a superset of FOL, to constraints for an SQL database and to Horn-clause rules for an inference engine. Although the knowledge engineers had used a very expressive dialect of logic, the compiler translated 84 percent of the axioms they wrote directly to Horn-clause rules (4,667 of the 5,532 axioms extracted from Cyc). It translated the remaining 865 axioms to SQL constraints that ensured the consistency of all database updates with the axioms.

A solid foundation in logic can make commercial software easier to use, and it does not require the people who use it to have formal training in logic. The fads and fallacies that block such use are logicians' disdain for readable notations, nonlogicians' fear of formal logic, and the lack of any coherent policy for integrating the development tools.[5,6] The logic-based languages of the Semantic Web are useful, but they're not integrated with the SQL language of relational databases, the design and development tools based on the Unified Modeling Language, or the legacy systems that won't disappear for many decades to come. A better integration is possible with tools based on logic at the core, diagrams and controlled NLs at the human interfaces, and compiler technology for mapping logic to both new and legacy software. ■

References

1. J. Limber, "The Genesis of Complex Sentences," *Cognitive Development and the Acquisition of Language*, T. Moore, ed., Academic Press, 1973, pp. 69–186; http://pubpages.unh.edu/~jel/JLimber/Genesis_complex_sentences.pdf.

2. J.F. Sowa, "Laws, Facts, and Contexts: Foundations for Multimodal Reasoning," *Knowledge Contributors*, V.F. Hendricks, K.F. Jørgensen, and S.A. Pedersen, eds., Kluwer Academic Publishers, 2003, pp. 145–184.

3. H. Wang, "Toward Mechanical Mathematics," *IBM J. Research and Development*, vol. 4, Jan. 1960, pp. 2–22; www.research.ibm.com/journal/rd/041/ibmrd0401B.pdf.

4. B.J. Peterson, W.A. Andersen, and J. Engel, "Knowledge Bus: Generating Application-Focused Databases from Large Ontologies," *Proc. 5th Int'l Workshop Knowledge Representation Meets Databases*, CEUR-WS.org, 1998; http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10.

5. J.F. Sowa, "Concept Mapping," slide presentation at *Am. Educational Research Assoc. 2006 Ann. Meeting*, 2006; www.jfsowa.com/talks/cmapping.pdf.

6. J.F. Sowa, "Extending Semantic Interoperability to Legacy Systems and an Unpredictable Future," slide presentation at the *Collaborative Expedition Workshop*, 2006; www.jfsowa.com/talks/extend.pdf.

John F. Sowa is a cofounder of VivoMind Intelligence, Inc., an AI company that's developing software based on some of the ideas discussed in this article. He retired from IBM after working for 30 years on R&D projects in AI and related areas. He has a PhD in computer science from the Vrije Universiteit in Brussels. He is a fellow of the AAAI. Contact him at sowa@bestweb.net.