Book Reviews

Computational Semantics with Functional Programming

Jan van Eijck* **and Christina Unger**‡
(*CWI, Amsterdam and Utrecht University; ‡University of Bielefeld)

Cambridge: Cambridge University Press, 2010, xv+405 pp; hardbound,
ISBN 978-0-521-76030-0, $99.00; paperbound, ISBN 978-0-521-75760-7, $40.00

Reviewed by
Robin Cooper
University of Gothenburg

There has been a recent intensification of interest in "semantics" in computational linguistics. I write the word in scare quotes because there are very different views of what computational semantics is. Broadly, it divides into the view that word meaning can be modeled in distributional terms and the view that meaning is to be viewed in terms of model theory of the kind employed in formal semantics deriving from the seminal work of Richard Montague (1974). This book is firmly placed in the latter logic-based semantics camp.

If you want to learn about the logical approach to computational semantics there are three basic texts you can go to: Blackburn and Bos (2005), which uses Prolog; Bird, Klein, and Loper (2009, chapter 10), which covers the essential aspects of Blackburn and Bos using Python within the popular Natural Language Toolkit (NLTK); and the present book, which uses the functional programming language Haskell. All three of these references will teach you both semantics and the required programming skills at the same time. So how do you choose between them?

One relevant issue is the choice of programming language. Prolog, a logic programming language, seems like a natural choice for logic-based semantics and, indeed, as Blackburn and Bos show, it provides support for writing concise and transparent code which is close to the kind of formalism used in theoretical logic-based semantics. It is not without its problems, however. Prolog variables are associated with unification-based binding, which is not the same as the kind of binding by quantifiers and operators that is used in logic. A second problem is that Prolog is a relation-based language and does not have a direct implementation of functions that return a value. Formal semantics building on Montague's work makes heavy use of the λ-calculus with λ-expressions denoting functions, and semantic composition is largely defined in terms of function-argument application. Blackburn and Bos implement a version of the λ-calculus in Prolog but this leads to a third problem: Formal semantics uses a typed version of the λ-calculus, yet standard Prolog is not a typed language (apart from making basic distinctions between integers, lists, etc.). Python, being object-oriented, allows a partial solution to the typing problem and it also has functions. It is a very flexible language and allows transparent coding of semantic formalisms by its powerful string processing. But coding semantic formalisms in terms of strings, although providing a great deal of flexibility, does not give you the feeling that the language is providing support for or insight into the logical operations that are being performed.

Van Eijck and Unger's book enters into this discussion with Haskell, a language based on the λ-calculus, with a strict typing system that requires the programmer to declare the types of all objects that are to be used in a program. As a functional

programming language (in the sense of being based on functions that take arguments and return values) it comes with a proper logical notion of variable binding rather than the unification variety. Furthermore, it has static typing, which means that it checks your program for type errors at compile time. This means that if you have made an error you do not have to wait until it shows up in some particular example at run time (perhaps after you have delivered your system to your client). The core of this language then provides exactly the tools that the logical semanticist needs. Here the semanticist is getting real support from the language, not just the flexibility to implement what she needs. A few words of caution are appropriate, however. If the type system that you want to implement does not exactly match the type system of the programming language, you may well be better off doing your implementation in a non-typed language so that the two type systems do not interfere with each other, though how you feel about this issue depends very much on your programming background and preferences. Van Eijck and Unger show, in my view magnificently, that Haskell's type system *does* match the type system used in the classical approach to formal semantics. Some might feel that there is a slight cost, in that direct implementation in Haskell involves using Haskell's syntax, which is a bit different from the kind of notation used in classical formal semantics. Haskell's syntax is extremely slick and concise. For example, it seems to follow the philosophy: "Don't put any kind of brackets around expressions, when you could simply write them next to each other and the context will determine what that means." This can be initially unnerving to linguists used to more florid notations with an element of redundancy.

One of the great contributions that Blackburn and Bos made was showing how logic-based computational semantics can be connected to off-the-shelf first-order theorem provers and model builders such as Prover9 and Mace and showing how this connection can be exploited in semantics and discourse processing. This work has also been incorporated into the NLTK version of semantics. Although you will find discussion of inference in van Eijck and Unger's book, you will not find accounts of how to connect to external theorem provers. This is probably a principled decision. If you learn semantics from the Blackburn and Bos or NLTK texts you might be forgiven for coming to the conclusion that the main aim of using the λ-calculus in linguistic semantics is to get a compositional treatment that will always allow you to reduce a first-order formula that can be sent off to a fast first-order theorem prover or model builder and you may just miss their occasional references to the fact that not all natural language sentences correspond to first-order formulae. This means that you will not be able to adequately treat sentences containing certain generalized quantifiers such as *most* in *most customers prefer this product* or intensional verbs such as *want* in *I want to go to Chicago*. Van Eijck and Unger, on the other hand, have a fairly detailed discussion of generalized quantifiers in Chapter 7 and devote the whole of Chapter 8 to intensionality. Although it would, of course, be a challenge to cover all the major results of formal semantics within a single introductory textbook on computational semantics, at the end of this book one has the impression that we have the tools we need to go further, whereas with the other introductions we have the impression that computational semantics deals with expressions of natural language whose interpretations can be represented in first-order logic.

In general, van Eijck and Unger place semantics in a broader philosophical and logical setting. Chapter 1 deals with the formal, logical approach to natural language, and Chapter 2 with the tools used in functional programming. Chapter 3 provides a practical introduction to the parts of the programming language Haskell that you need to know in order to be able to follow the book. Chapters 4 and 5 deal, respectively, with

syntax and semantics for fragments. The intention here was, I imagine, to introduce things gently by starting with the syntax of games rather than of languages. I was not sure that this was the most useful way to introduce things for linguists. It was not entirely clear to me what the syntax of a game was meant to represent (the game itself?, the language you use when playing the game or when describing the game?) and this was not helped by having the semantics coming in the next chapter after discussing the syntax of a fragment of English, propositional logic, predicate logic, and an extension of predicate logic with function symbols. Chapter 6 deals with model checking with predicate logic, and finally with Chapter 7 we get down to some serious compositional semantics for natural language. The remaining chapters build on this and deal with more advanced topics: intensionality in Chapter 8, parsing in Chapter 9, quantifier scope in Chapter 10, continuation semantics in Chapter 11, discourse representation in Chapter 12, and communication and information in Chapter 13. If you make it to the end, the Afterword tells you to treat yourself to a beer as a reward.

As stated in the Preface, the book is directed to linguists, logicians, and functional programmers and provides basically all you need to know if you are coming from one of these areas and do not know about the other two. This is a major achievement and practically everything you need is there, clearly and concisely expressed. The learning curve is steep, however, and there is a great deal of complex material to digest. If you are on your own without a sympathetic teacher, you might feel that you have earned something stronger than a beer by the time you get to the end.

This book was developed over a period of about ten years and, besides being an introductory textbook to computational semantics that every serious student of the field should study, it represents a mature major research contribution demonstrating the close relationship between classical formal semantics and modern functional programming.

References

Bird, Steven, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, Sebastopol, CA.

Blackburn, Patrick and Johan Bos. 2005. *Representation and Inference for Natural Language: A First Course in Computational Semantics*. CSLI Studies in Computational Linguistics. CSLI Publications, Stanford, CA.

Montague, Richard. 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Edited and with an introduction by Richmond H. Thomason. Yale University Press, New Haven, CT.

Robin Cooper is Professor of Computational Linguistics at the University of Gothenburg and director of the Swedish National Graduate School of Language Technology. His address is Department of Philosophy, Linguistics and Theory of Science, University of Gothenburg, Box 200, S-405 30 Göteborg, Sweden; e-mail: cooper@ling.gu.se.