Semantics of Programming Languages: A Tool-Oriented Approach*

Jan Heering
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Jan.Heering@cwi.nl

Paul Klint
CWI and University of Amsterdam
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Paul.Klint@cwi.nl

Abstract

By paying more attention to semantics-based tool generation, programming language semantics can significantly increase its impact. Ultimately, this may lead to "Language Design Assistants" incorporating substantial amounts of semantic knowledge.

1 The Role of Programming Language Semantics

Programming language semantics has lost touch with large groups of potential users [39]. Among the reasons for this unfortunate state of affairs, one stands out. Semantic results are rarely incorporated in practical systems that would help language designers to implement and test a language under development, or assist programmers in answering their questions about the meaning of some language feature not properly documented in the language's reference manual. Nevertheless, such systems are potentially more effective in bringing semantics-based formalisms and techniques to the places they are needed than their dissemination in publications, courses, or even exemplary (but little-used) programming languages.

The current situation in which semantics, languages, and tools are drifting steadily further apart is shown in Figure 1. The tool-oriented approach to semantics aims at making semantics definitions more useful and productive by generating as many language-based tools from them as possible. This will, we expect, reverse the current trend as shown in Figure 2. The goal is to produce semantically well-founded languages and tools. Ultimately, we envision the emergence of "Language Design Assistants" incorporating substantial amounts of semantic knowledge.

Table 1 lists the semantics definition methods we are aware of. Examples of their use can be found in [40]. Petri nets, process algebras, and other methods that do not specifically address the semantics of programming languages, are not included. Dating back to the sixties, attribute grammars and denotational semantics are among the oldest methods, while abstract state machines (formerly called evolving algebras), coalgebra semantics, and program algebra are the latest additions to the field. Ironically, while attribute grammars are popular with tool builders, semanticists do not consider them a particularly interesting definition method. Since we will only discuss the various methods in general terms without going into technical details, the reader need not be familiar with them. In any case, the differences between them, while often hard to decipher because the field is highly fragmented and appropriate "dictionaries" are lacking, do not affect our main argument.

Table 2 lists a representative language development system (if any) for the semantics definition methods of Table 1. The last entry, Software Refinery, which has its origins in knowledge-based software

*This research was supported in part by the Telematica Instituut under the *Domain-Specific Languages* project.

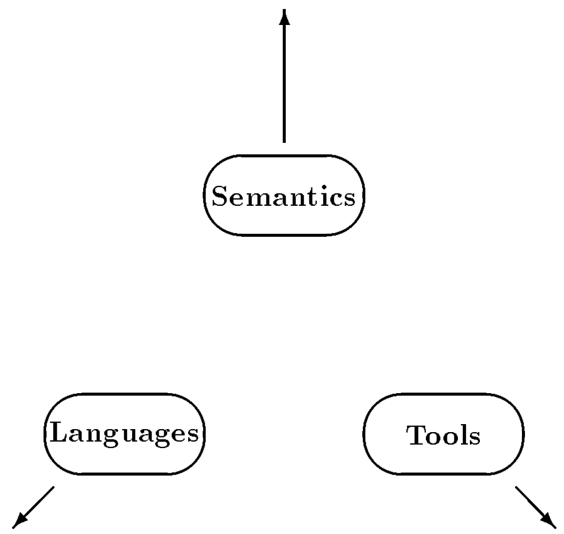

Figure 1: Semantics, languages, and tools are drifting steadily further apart.

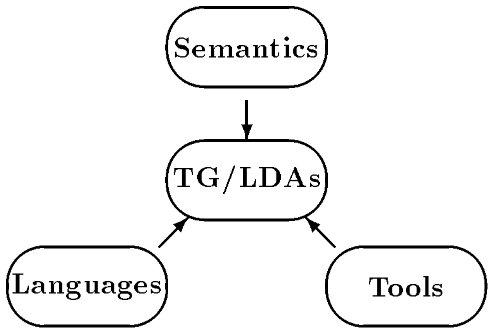

Figure 2: In the tool-oriented approach, semantics, languages, and tools are kept together by Tool Generation (**TG**) and, ultimately, Language Design Assistants (**LDAs**).

Semantics	Definition in terms of
Axiomatic [4]	Pre- and postconditions
Attribute grammars [12]	Attribute propagation rules
Denotational [38]	Lambda-expressions
Algebraic [7]	Equations/rewrite rules
Structural operational [35]/ Natural [23]	Inference rules
Action [31]	Action expressions
Abstract state machines [19]	Transition rules
Coalgebraic [21]	Behavioral specification rules
Program algebra [8]	Equations

Table 1: Current approaches to programming language semantics.

Semantics	System	Developed at
Attribute grammars	Synthesizer Generator [36]	Cornell University
Denotational	PSG [5]	Technical University of Darmstadt
Algebraic	ASF+SDF Meta-Environment [14]	CWI and University of Amsterdam
Structural operational/ Natural	Centaur [9]	INRIA Sophia-Antipolis
Action	ASD [13]	CWI and University of Aarhus
Abstract state machines	Gem-Mex [3]	University of L'Aquila
(Operational)	Software Refinery [29]	Reasoning Systems, Palo Alto

Table 2: Some representative language development systems.

environments research at Kestrel Institute, does not fit any of the current semantics paradigms. The pioneering Semanol system [2] is, to the best of our knowledge, no longer in use and is not included. The systems listed have widely different capabilities and are in widely different stages of development. Before discussing their characteristics and applications in Section 3, we first explain the general ideas underlying the tool-oriented approach to programming language semantics. These were shaped by our experiences with the ASF+SDF Meta-Environment (Table 2) over the past ten years. Finally, we discuss Language Design Assistants in Section 4.

2 A Tool-Oriented Approach to Semantics

The tool-oriented approach to semantics aims at making semantics definitions more useful and productive by generating as many language-based tools from them as possible. This affects many aspects of the way programming language semantics is practiced and upsets some of its dogmas.

Table 3 lists some of the tools that might be generated. In principle, the language definition has to be augmented with suitable tool-specific information for each tool to be generated, and this may require tool-specific language extensions to the core semantics definition formalism. In practice, this is not always necessary since semantics definitions tend to contain a good deal of implicit information that may be extracted and used for tool generation.

Scanner/Parser
Prettyprinter
Syntax-directed editor
Typechecker
(Abstract) interpreter(s)
Dataflow analyzer
Call graph extractor
Partial evaluator
Optimizer
Program slicer
Origin tracker
Debugger
Code generator
Compiler
Profiler
Test case generator
Test coverage analyzer
Regression test tool
Complexity analyzer (metrics)
Documentation generator
Cluster analysis tool
Systematic program modification tool

Table 3: Tools that might be derived from a language definition.

The first entry of Table 3, scanner and parser generation, is standard technology. Lex and Yacc are well-known examples of stand-alone generators for this purpose. Their input formalisms are close to regular expressions and BNF, the *de facto* standard formalisms for regular and context-free grammars, respectively. Unfortunately, for most of the other tools in Table 3 there are no such standard formalisms.

The key features of the tool-oriented approach are:

- Language definitions are primarily tool generator input. They do not have to provide any kind of theoretical "explanation" of the constructs of the language in question nor do they have to become part of a language reference manual.
- An interpreter that can act, among other things, as an "oracle" to programmers needing help will be among the first tools to be generated.
- Writing (large) language definitions loses its esoteric character and becomes similar to any other kind of programming. Semantics formalisms tend to do best on small examples, but lose much of their power as the language definitions being written grow. In the tool-oriented approach, semantics formalisms have to be modular and separate generation (the analogue of separate compilation) has to be supported. Libraries of language constructs become important.
- The tool-oriented approach may require addition of tool-specific features to the core formalism. This leads to an open-ended rather than a "pure" style of semantics description.
- The scope of the tool-oriented approach includes, for instance,

System	Generated tools	Semantic engine
Synthesizer Generator	scanner/parser (LALR), prettyprinter, syntax-directed editor, incremental typechecker, incremental translator, ...	incremental attribute evaluator
PSG	scanner/parser, syntax-directed editor, incremental typechecker (even for incomplete program fragments), interpreter	functional language interpreter
ASF+SDF Meta-Environment	scanner/parser (generalized LR), prettyprinter, syntax-directed editor, typechecker, interpreter, origin tracker, translator, renovation tools, ...	conditional rewrite rule engine
Centaur	scanner/parser (LALR), prettyprinter, syntax-directed editor, typechecker, interpreter, origin tracker, translator, ...	inference rule engine
ASD	scanner/parser, syntax-directed editor, checker, interpreter	conditional rewrite rule engine
Gem-Mex	scanner/parser, typechecker, interpreter, debugger	transition rule engine
Software Refinery	scanner/parser (LALR), prettyprinter, syntax-directed editor, object-oriented parse tree repository (including dataflow relations), Y2K/Euro tools, program slicer, ...	tree manipulation engine

Table 4: Tool generation capabilities of representative language development systems.

- Domain-specific and little languages [15, 37]. Many of the tools in Table 3 are as useful for DSLs as they are for programming languages.
- Software maintenance and renovation tools [16]. Some of these are included at the end of Table 3.
- Compiler toolkits such as CoSy [1], Cocktail [18], OCS [22], SUIF [42], and PIM [6, 17].

3 Existing Language Development Systems

Table 4 summarizes the tool generation capabilities of the representative language development systems listed in Table 2. All of them can generate lexical scanners, parsers, and prettyprinters, many of them can produce syntax-directed editors, typecheckers, and interpreters, and a few can produce various kinds of software renovation tools. To this end, they support one or more specification formalisms, but these differ in generality and application domain.

For instance, the Synthesizer Generator supports attribute grammars with incremental attribute evaluation, which is particularly suitable for typechecking, static analysis and translation, but less suitable for dynamic semantics. The ASF+SDF Meta-Environment supports conditional rewrite rules rather than attribute grammars, and these can be used for defining dynamic semantics as well. Software Refinery comes with a full-blown functional language in which a wide range of computations on programs can be expressed. Other systems provide more specialized specification formalisms. PSG, for instance, uses context relations to describe incremental typechecking (even for incomplete program fragments) and denotational definitions for dynamic semantics. Gem-Mex supports a semi-visual formalism optimized for the definition of programming language semantics and tool generation. It can generate a typechecker, an interpreter, and a debugger.

Table 4 is far from complete. Some other language development systems are SIS [30], PSP [32], GAG [24], SPS [41], MESS [28], Actress [11], Pregmatic [10], LDL [20], and Eli [26]. Many of the tools listed in Table 3 are not generated by any current system. Ample opportunities for tool generation still exist in areas like optimization, dynamic program analysis, testing, and maintenance.

4 Toward Language Design Assistants

The logical next step beyond semantics-based tool generation would lead to a situation similar to that of computer algebra. Large parts of mathematics are being incorporated in computer algebra systems. Conversely, computer algebra itself has become a fruitful mathematical activity, yielding new results of general mathematical interest. In the case of semantics, we see opportunities for "Language Design Assistants" incorporating a substantial amount of both formal and informal semantic knowledge. The latter is found, for instance, in language design rationales and discussion documents produced by standardization bodies. Development of such assistants will not only push semantics even further toward practical application, but also give rise to new theoretical questions.

The Language Design Assistants we have in mind would support the human language designer by providing design choices and performing consistency checks during the design process. Operational knowledge about typical issues like typing rules, scope rules, and execution models should be incorporated in them. Major research questions arise here regarding the acquisition, representation, organization, and abstraction level of the required knowledge. For instance, should it be organized according to any of the currently known paradigms of object-oriented, functional, or logic programming? Or should a higher level of abstraction be found from which these and other, new, paradigms can be derived? How can constraints on the composition of certain features be expressed and checked? Another key question is how to construct a collection of "language feature components" that are sufficiently general to be reusable across a wide range of languages.

Similar considerations apply to tool development. By incorporating knowledge about tool generation in the Language Design Assistant we can envision a Tool Generation Assistant that helps in constructing tools in a more advanced way than the tool generation we had in mind in the previous sections.

To make this perspective somewhat more tangible, consider the relatively simple case of an if-then-else-like conditional construct that has to be modelled as a language feature component. Table 5 gives an impression of the wide range of issues that has to be addressed before such a generic conditional construct can be specialized into a concrete if-then-else-statement or conditional expression in a specific language. It is a research question to design an abstract framework in which these and similar questions can be expressed and answered.

Another major question is how to organize the specialization process from language feature component to concrete language construct. The main alternatives are parameterization and transformation [27]. Using parameterization, specialization of the component in question amounts to instantiating its parameters. Since parameters have to be identified beforehand and instantiation is usually a rather simple mechanism, the adaptability/reusability of a parameterized component is limited. Using transformations, on the other hand, a language feature component is designed without explicit parameters. Specialization is achieved by applying appropriate transformation rules to it to obtain the desired specific case. Clearly, this approach is more flexible since any part of the language feature component can be modified by the transformation rules and can thus effectively act as a parameter. The relation between this approach of meta-level transformation and parameterized modules is largely unexplored.

- ○ What is the type of the expression controlling the selection of one of the two branches.
- ○ How is the controlling expression evaluated (short circuit vs. full evaluation)?
- ○ Is the controlling expression evaluated concurrently with other program parts (with speculative execution of the conditional as a special case)?
- ○ Can the controlling expression have side-effects?
- ○ Can the controlling expression cause exceptions?
- ○ Are jumps from outside into the branches allowed?
- ○ Is the selected branch evaluated concurrently with other program parts?
- ○ Can the evaluation of the selected branch cause side-effects?
- ○ Can the evaluation of the selected branch cause exceptions?
- ○ Does the evaluation of the conditional construct yield a value?

Table 5: Some of the possible parameters of a generic conditional construct.

Although we are not aware of research on Language Design Assistants from the broad perspective sketched here, there is some work pointing in the same general direction:

- The Language Designer's Workbench sketched as future work in [34, 28] has some of the same goals.
- Action semantics [31] also emphasizes libraries of reusable language constructs.
- Plans (no longer pursued) for the Language Development Laboratory [20] included a library of reusable language constructs, a knowledge base containing knowledge of languages and their compilers/interpreters, and a tool for language design.
- The "design and implementation by selection" of languages described in [33, 25] is a case study in high-level interactive composition of predefined language constructs.

Acknowledgements We would like to thank Jan Bergstra, Mark van den Brand, Arie van Deursen, Ralf Lämmel, and Jan Rutten for useful comments on earlier versions.

References

[1] Ace bv. The CoSy compilation system, 1999. www.ace.nl/products/cosy.htm.

[2] E. R. Anderson, F. C. Belz, and E. K. Blum. SEMANOL(73)—A metalanguage for programming the semantics of programming languages. *Acta Informatica*, 6:109–134, 1976.

[3] M. Anlauff, P. W. Kutter, and A. Pierantonio. Formal aspects and development environments for Montages. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF '97)*, Electronic Workshops in Computing. Springer/British Computer Society, 1997.

[4] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 2nd edition, 1997.

[5] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8:547–576, 1986.

[6] J. A. Bergstra, T. B. Dinesh, J. Field, and J. Heering. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems*, 19(5):639–684, 1997.

[7] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[8] J. A. Bergstra and M. E. Loots. Program algebra for component code. Technical Report P9811, Programming Research group, University of Amsterdam, 1998. To appear in *Formal Aspects of Computing.*

[9] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. *ACM SIGPLAN Notices*, 24(2):14–24, 1989. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.*

[10] M. G. J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments.* PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[11] D. F. Brown, H. Moura, and D. A. Watt. Actress: Action semantics directed compiler generator. In U. Kastens and P. Pfahler, editors, *Compiler Construction (CC '92)*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, 1992.

[12] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science.* Springer-Verlag, 1988.

[13] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994. www.cwi.nl/~arie/papers/pschrift.ps.gz.

[14] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

[15] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[16] A. van Deursen, P. Klint, and C. Verhoef. Research issues in software renovation. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE '99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1999.

[17] J. Field, J. Heering, and T. B. Dinesh. Equations as a uniform framework for partial evaluation and abstract interpretation. *ACM Computing Surveys*, 30(3es), 1998. Electronic supplement: 1998 Symposium on Partial Evaluation (SOPE '98).

[18] J. Grosch and H. Emmelmann. A tool box for compiler construction. In D. Hammer, editor, *Compiler Compilers (CC '90)*, volume 477 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[19] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 7–36. Oxford University Press, 1995.

[20] J. Harm, R. Lämmel, and G. Riedewald. The Language Development Laboratory. In M. Haveraaen and O. Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory (NWPT '96)*, Research Report 248, pages 77–86. University of Oslo, Department of Informatics, 1997. www.cwi.nl/~ralf.

[21] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62:222–259, 1997. www.cwi.nl/~janr.

[22] T. P. Justice and T. Budd. OCS: An object-oriented compiler construction toolkit. Technical Report 93-60-10, Oregon State University, Department of Computer Science, 1993.

[23] G. Kahn. Natural Semantics. In F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Symposium on Theoretical Aspects of Computer Science (STACS '87)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[24] U. Kastens. The GAG-System — A tool for compiler construction. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 165–181. Cambridge University Press, 1984.

[25] U. Kastens and P. Pfahler. Compositional design and implementation of domain-specific languages. In R. N. Horspool, editor, *IFIP TC2 WG 2.4 Working Conference on System Implementation 2000: Languages, Methods and Tools*, pages 152–165. Chapman and Hall, 1998.

[26] U. Kastens, P. Pfahler, and M. Jung. The Eli system. In K. Koskimies, editor, *Compiler Construction (CC '98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 294–297. Springer-Verlag, 1998.

[27] R. Lämmel. Declarative aspect-oriented programming. In O. Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, BRICS Notes Series NS-99-1, pages 131–146, 1999.

[28] P. Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[29] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, May 1994.

[30] P. D. Mosses. SIS — Semantics Implementation System: Reference manual and user guide. Technical Report DAIMI MD-30, Computer Science Department, Aarhus University, 1979.

[31] P. D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[32] L. Paulson. A semantics-directed compiler generator. In *Principles of Programming Languages (POPL '82)*, pages 224–233. ACM, 1982.

[33] P. Pfahler and U. Kastens. Language design and implementation by selection. In *Proceedings First ACM SIGPLAN Workshop on Domain-Specific Languages (DSL '97)*, Technical Report, pages 97–108. University of Illinois at Urbana-Champaign, 1997.

[34] U. F. Pleban. Formal semantics and compiler generation. In H. Morgenbrod and W. Sammer, editors, *Programmierumgebungen und Compiler*, pages 145–161. Teubner-Verlag, 1984.

[35] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[36] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, third edition, 1989.

[37] P. H. Salus, editor. *Handbook of Programming Languages Vol. III: Little Languages and Tools*. Macmillan Technical Publishing, 1998.

[38] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[39] D. A. Schmidt. On the need for a popular formal semantics. *ACM Computing Surveys*, 28(4es), 1996. Electronic supplement: Strategic Directions in Computing Research.

[40] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[41] M. Wand. A semantic prototyping system. *SIGPLAN Notices*, 19(6):213–221, 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*.

[42] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.