Postface

Abstract We wrap up this book by (i) restating the importance of competence software languages in the broader context of computer science and IT; (ii) summarizing the key concepts of software languages as captured in this book; (iii) identifying the omissions in this book, to the extent they are clear to the author; (iv) listing complementary textbooks; and (v) listing relevant academic conferences. In this manner, we also provide pointers to further reading of different kinds.

The importance of Software Language Engineering

Software language engineering (SLE) basically proxies for software engineering with awareness of the languages involved such that software artifacts are treated in a syntax- and semantics-aware manner and particular attention is paid to software language definition and implementation, as well as other phases of the software language lifecycle. A "philosophical" discussion of the term "software languages" and the engineering thereof can be found elsewhere [39].

SLE is becoming a fundamental competence in computer science, best comparable to competences in algorithms, data structures, networking, security, software architecture, design, testing, formal methods, data mining, and artificial intelligence. The importance of SLE competence has increased in recent years and may continue to increase because of general developments in computer science. That is, while the classic language implementation tasks, i.e., the design and implementation of interpreters and compilers for mostly general-purpose programming languages, affect only a few software engineers directly, software engineers and other IT professionals and scientists are increasingly ending up in more diverse contexts with relevance to SLE such as the following:

- Design, implementation, and usage of internal and external DSLs that support problem or technical domains, for example, user interfaces, web services, configuration, testing, data exchange, interoperability, deployment, and distribution.

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7

- Software reverse engineering and re-engineering in many forms, for example, analysis of projects regarding their dependence on open-source software, integration of systems, and migration of systems constrained by legislation or technology.
- Data extraction in the context of data mining, information retrieval, machine learning, mining software repositories, big data analytics, data science, computational social science, digital forensics, and artificial intelligence, with diverse input artifacts to be parsed and interchange formats to conform to.

Software Languages: Key Concepts

What are the key concepts regarding software languages? In this particular book, we have identified, explained, illustrated, and connected some concepts, as summarized below.

The basic starting point is (object program) *representation* of artifacts that are software language elements so that they are amenable to programmatic processing, i.e., *metaprogramming*. Thus, a *metaprogram* is a program that processes other programs or software artifacts that are elements of software languages. Ordinary programming languages may serve for metaprogramming, but we may also use more specialized *metaprogramming languages* or *systems* with dedicated metaprogramming expressiveness (e.g., for *term rewriting* or *concrete object syntax*) and infrastructure (e.g., for *template processing* or *parser generation*).

The *syntax* of a software language defines the language as a set of strings, trees, or graphs; it also defines the structure of software language elements, thereby facilitating their representation in metaprograms. Different formalisms may be used for syntax definition, for example, *grammars*, *signatures*, or *metamodels*. We may check for *conformance*, i.e., given a string, a tree, or a graph, we may check whether it complies with a given syntax definition. We may also engage in *parsing*, such that we analyze the structure of some input and possibly map language elements from *concrete syntax* to *abstract syntax*. We may further engage in *formatting* such that we render language elements according to some concrete syntax.

The syntax (and semantics) of syntax definitions gives rise to a *metametalevel*. It is important to master representation and syntax across *technological spaces* [76] because one is likely to encounter different spaces in practice.

The *semantics* of a software language defines the meaning of language elements. A software language engineer may not be interested in the formal study of semantics in itself, but may be very well interested in the applications of semantics for the purpose of (metaprograms for) *interpretation* (i.e., actual or *abstract interpretation*), *semantic analysis* (e.g., *well-formedness* or *type checking*), *transformation* (e.g., *optimization* or *refactoring*), and *translation* (e.g., *compilation* or *code generation* in a DSL implementation). We contend that there is simply no reasonable way to author metaprograms without some degree of understanding of the *operational* or *denotational* semantics of the object and metalanguages involved. All these metaprograms

are syntax-driven and essentially rule-based in how they interpret, analyze, transform, or translate object programs. In more disciplined cases, these metaprograms may obey the principle of *compositionality* (i.e., some kind of structural recursion).

Omissions in This Book

This book describes the author's particular view of the software language world and it is limited by what can be reasonably fitted into a book, also taking into account a certain target audience and making certain assumptions about the background of readers, as discussed in the Preface. Thus, omissions are to be expected, and we itemize them here. We do this also to support further reading, maybe also in the context of research-oriented course designs. We group these omissions by major software language topics.

Meta-programming In this book, we favor a grammarware- and metaprogramming-based view of software languages; we have complemented this view with occasional excursions into the technological space of *model-driven engineering* (*metamodeling*, *model transformation*) [10, 125]. We implemented transformations in the style of rewriting; we did not properly describe or exercise *model transformation languages* [92, 26, 58, 2, 60]. The coverage of metaprogramming is limited in this book. We focused mainly on source-code analysis and manipulation, where source code should again be understood in a broad sense, not limited to programming languages. We managed an excursion to program generation. We did not cover *reflection* or limited (disciplined) concepts derived from reflection [67, 36, 30], *bytecode analysis and manipulation* [27, 21], higher-level programming constructs for modularity with lower-level semantics involving reflection, for example, *context-oriented programming* (COP) [50], *aspect-oriented programming* (AOP) [65, 78, 149, 64, 66], or *morphing* [54].

Domain-specific languages This book provides a backbone regarding representation, syntax, parsing, formatting, basic language concepts, interpretation, and typing for DSLs or software languages generally. Coverage of domains was very sparse; some additional problem domains for DSLs are, for example, telecom services, insurance products, industrial automation, medical device configuration, ERP configuration, and phone UI applications [139]. The book also left out the interesting topic of *domain-specific optimization* [75, 12, 68]. We adopted a basic metaprogramming approach without explicit dependence on or systematic discussion of more advanced features of *metaprogramming languages* and *systems* [52, 151, 152, 93, 118, 19, 35, 32, 29, 143, 124], and also without coverage of DSL development with *language workbenches* [33, 34, 62, 61, 147, 145, 148, 141]. We focused on textual concrete syntax; we did not cover *visual syntax* [88, 96, 73], in particular, we did not cover it in terms of editing. We also omitted coverage of *syntax-directed editing* [11, 119] and *projectional editing* [142, 146, 144]. Overall, we did not cover the lifecycle of DSLs too well; we

provided some implementation options, but we were very sparse on design. Some recommended further reading is [52, 151, 152, 93, 118, 19, 35, 32, 29, 143, 124]. There were also specific topics on the *software language lifecycle* that fell off the desk, unfortunately, for example, *test-data generation* [114, 89, 13, 48, 17, 134, 136, 47, 77, 59, 82] and *language-usage analysis* [72, 43, 80, 7, 22, 38, 81].

Compilation As *compiler construction* is a well-established subject with excellent textbook coverage (see the next section), the present book makes no attempt at covering compiler construction. It so happens that simple aspects of frontends and translation were covered, but the more interesting aspects of the vital use of *intermediate representations* and *high-* and *low-level optimizations* were not. These days, *compiler frameworks*, as a form of language infrastructure possibly including runtime system components, support compiler construction, for example, LLVM [83], which is widely used, or more specialized frameworks [28, 115, 37, 20, 86] that target different architectures or optimization philosophies. Another interesting framework is Graal/Truffle, which combines aspects of cross-compilation and interpreter specialization [153, 31, 120].

Grammars While we covered tree-based abstract syntax definition by means of algebraic signatures, we did not discuss the related notion of *tree grammars* [23]. While we covered graph-based abstract syntax definition (i.e., metamodeling, including reference relationships), we did not discuss the related field of *graph grammars* [122]. As for string languages, we focused completely focused on context-free grammars, and did not cover other grammar formalisms such as *regular grammars* [53] and *parsing expression grammars* (PEGs) [40]. Regular grammars are useful, for example, for efficient scanning [1] and lexical fact extraction [101, 70]. PEGs provide an alternative formalism for parsing.

Parsing We did not explore *grammar classes* and we dealt superficially with *parsing algorithms* [1, 45], thereby leaving out a discussion of the space of options with regard to precise recognition, efficiency, restrictions imposed on the grammar author, and guarantees regarding grammar ambiguities. Increasingly, "generalized" parsing approaches are being used: *generalized LR parsing* [140, 117], *generalized LL parsing* [57, 126], LL(∗) [109], and PEGs [40]. We certainly also missed various forms of parsing used in software engineering. For instance, there are specific forms of grammars (or parsing) such as *island* or *tolerant grammars*, aiming at robustness in the in response to the complexity of language syntax, diversity of dialects, or handling parser errors in an interactive context [97, 98, 71, 104].

Semantics and types We covered the basics of semantics and types, aiming at the pragmatic realization of interpreters and type checkers. While we developed the basics of operational and denotational semantics, we did not cover some of the refinements that improve usability, for example, modular operational semantics [100], action semantics [99], and monad transformers [85]. We also did not cover approaches to semantics that are less directly usable for interpretation – in particular, all forms of axiomatic semantics [51, 138]. We skipped over the foundations of formal semantics and type systems; we refer to the next section for textbooks on programming language theory. We focused on "classical" static

typing; we omitted several advanced concepts, for example, effect systems [102], liquid types [121], soft typing, gradual typing [18, 130, 131, 132], dependently typed programming [90, 106, 107, 15], and the scale between *static* and *dynamic typing* [91]. We touched upon *type safety*, but omitted a proper discussion of the more general notion of (mechanized) *metatheory* [113, 4], which might possibly also depend on *theorem provers* or *proof assistants*, for example, Twelf [112, 84], Coq [8], and Isabelle/HOL [105], or *dependently typed* programming languages, for example, Agda [14, 107].

Language concepts We exercised a range of languages for illustration. We did not intend to cover programming paradigms systematically; see the next section for textbook recommendations. Various language concepts were not covered, for example, object-oriented programming (OOP) [55, 16], type dispatch or type case [44, 24, 25], polytypic or generic functional programming [56, 49, 79], and information hiding or data abstraction [95, 133, 150]. We covered some parts of the lambda cube [6]; we did not cover complementary calculi, for example, process calculi or process algebras (CCS, CSP, π-calculus, etc.) [5, 94, 123] for concurrent programming.

Complementary Textbooks

This book can be usefully compared and complemented with textbooks in neighboring areas. In this manner, we may also provide pointers to further reading. We have classified textbooks into the following categories:

Programming language theory Textbooks in this category cover topics such as lambda calculi, formal semantics, formal type systems, metatheory, and program analysis. Examples include Pierce's "Types and Programming Languages" [113], Friedman and Wand's "Essentials of Programming Languages" [42], and Gunter's "Semantics of Programming Languages: Structures and Techniques" [46]. We also refer to Nielson and Nielson's "Semantics with Applications: An Appetizer" [103] as an introductory text and Slonneger and Kurtz' "Formal Syntax and Semantics of Programming Languages" [135] for a less formal, more practical approach to programming language theory.

The present book covers only some basic concepts of formal semantics and type systems, such as interpretation based on big-step and small-step semantics, the scheme of compositional (denotational) semantics, and type checking – without covering language concepts deeply, and without properly covering fundamental aspects such as metatheory (e.g., the soundness property on pairs of semantics and type system). This book goes beyond textbooks on programming language theory by covering the lifecycle, syntax implementation (parsing and formatting), and metaprogramming with applications to, for example, software reverse engineering and re-engineering.

Programming paradigms Textbooks in this category cover various paradigms (such as the imperative, functional, logical, and OO paradigms). The organiza-

tion may be more or less aligned with an assumed ontology of language concepts. Typically, an interpreter-based approach is used for illustration. Examples include Sebesta's "Concepts of Programming Languages" [128], Sethi's "Programming Languages: Concepts and Constructs" [129] and Scott's "Programming Language Pragmatics" [127]. These books also cover, to some extent, programming language theory and compiler construction.

The present book is not concerned with a systematic discussion of programming paradigms and programming language concepts. Nevertheless, the book exercises (in fact, "defines") languages of different paradigms and discusses various language concepts in a cursory manner. This book goes beyond textbooks on programming paradigms by covering metaprogramming broadly, which is not a central concern in textbooks on paradigms.

Compiler construction This is the classical subject in computer science that, arguably, comes closest to the subject of software languages. Examples of textbooks on compiler construction and overall programming language implementation include Aho, Lam, Sethi, and Ullman's seminal "Compilers: Principles, Techniques, and Tools" [1], Louden's "Compiler Construction: Principles and Practice" [87], and Appel's product line of textbooks such as Appel and Palsberg's " Modern Compiler Implementation in Java" [3].

The present book briefly discusses compilation (translation), but it otherwise covers compiler construction at best superficially. For instance, lower-level code optimization and code generation are not covered. This book covers language implementation more broadly than textbooks on compiler construction, with regard to both the kinds of software languages and the kinds of language-based software components. Most notably, this book covers metaprogramming scenarios other than compilation, and metaprogramming techniques other than those used in a typical compiler.

Hybrids There are a number of books that touch upon several of the aforementioned topics in a significant manner. There is Krishnamurthi's "Programming Languages: Application and Interpretation" [74], which combines programming language theory and programming paradigms in a powerful manner. There is Ranta's "Implementing Programming Languages: An Introduction to Compilers and Interpreters" [116] with coverage of programming paradigms and compiler construction. There is also Stuart's "Understanding Computation: From Simple Machines to Impossible Programs" [137], which is exceptionally broad in scope: it covers various fundamental topics in computer science, including parsing and interpretation; it explains all notions covered to the working Ruby programmer in a pragmatic manner.

The present book aims at a deeper discussion of the implementation and lifecycle of software languages in the broader context of software engineering, with the central topic being metaprogramming in the sense of source-code analysis and manipulation.

Domain-specific languages There are some more or less recent textbooks on DSLs. Fowler's "Domain-Specific Languages" [41] discusses relatively basic or mainstream OO techniques and corresponding patterns for language implemen-

tation and embedding specifically. Kleppe's "Software Language Engineering: Creating Domain-Specific Languages Using Metamodels" [69] and Kelly and Tolvanen's "Domain-Specific Modeling: Enabling Full Code Generation" [63] exercise the modeling- and metamodeling-based view of language design and implementation, as opposed to the use of standard programming languages and language implementation technology. Voelter et al.'s "DSL Engineering: Designing, Implementing and Using Domain-Specific Languages" [143] focuses on specific technologies such as MPS, xText, and Stratego/Spoofax. Parr's "Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages" [108] is a practical guide to using the ANTLR technology for language implementation. Bettini's "Implementing Domain-Specific Languages with Xtext and Xtend" [9] focuses on practitioners specifically interested in the Xtext stack.

The present book is not limited to domain-specific languages; it discusses software languages in a broad sense. Programming languages and semantics-based techniques such as partial evaluation and abstract interpretation are also covered to some extent. The book discusses software language engineering without commitment to a specific metaprogramming system.

Software Languages in Academia

Let us now connect the broad area of software languages to some established academic conference series. In this manner, we will also hint at resources for carrying out research on software languages. The conferences listed below are loosely ordered by decreasing coverage of the software language area. Thus, we start off with the conferences on Software Language Engineering and close the list with more specialized conferences focusing on specific aspects of software languages or specific categories such as programming languages. It goes without saying that this selection and its ordered presentation, just as much as the characterization of the individual conferences series, are subjective. We link each conference acronym to its manifestation in the DBLP database.

- *SLE*[3]: **Software Language Engineering**. This conference series covers the full range of software language topics in a balanced manner. The conference series was specifically created to unite the different perspectives on software languages such as those in the communities of grammarware and modelware.
- *SCAM*[4]: **Source Code Analysis and Manipulation**. This conference series takes a broad view of source code and covers a wide range of forms and purposes of software analysis and transformation such as parsing, smells, metrics, slicing, and clone detection.

[3] SLE: http://dblp.uni-trier.de/db/conf/sle/ (at DBLP)

[4] SCAM: http://dblp.uni-trier.de/db/conf/scam/ (at DBLP)

- *MODELS*[5]: **Model Driven Engineering Languages and Systems**. This conference series covers the field of software languages in terms of modeling, metamodeling, and model transformation while assuming an increasingly broad interpretation of modeling etc.
- *ECMFA*[6]: **European Conference on Model Driven Architecture - Foundations and Applications**. This conference series is similar to MODELS.
- *MODELSWARD*[7]: **Model-Driven Engineering and Software Development**. This conference series is similar to MODELS.
- *ICMT*[8]: **International Conference on Model Transformation**. This conference series covers the field of software languages in terms of model transformation while assuming an increasingly broad interpretation of model transformation by being inclusive in terms of technological spaces.
- *MSR*[9]: **Mining Software Repositories**. This conference series covers analysis of all kinds of artifacts in the broad sense of software repositories – not just source code, but also commit messages and bug reports. The conference series goes beyond the field of software languages by being also inclusive of methods from the fields of text analysis, natural language processing, information retrieval, machine learning, and data mining.
- *ICPC*[10]: **International Conference on Program Comprehension**. This conference series focuses on methods and tools for program comprehension, which includes a wide range of types of software analysis, visualization, cognitive theories, and other things. Software languages play a key role in terms of the artifacts to be analyzed.
- *SANER*[11]: **Software Analysis, Evolution, and Reengineering** (formerly WCRE (Working Conference on Reverse Engineering) and Conference on Software Maintenance and Reengineering (CSMR)). This conference series covers the broad areas of software reverse engineering, software re-engineering, and – even more broadly – software maintenance, software evolution, and software analysis. Software languages play a key role in terms of the artifacts to be analyzed or transformed.
- *ICSME*[12]: **International Conference on Software Maintenance and Evolution** (formerly ICSM (International Conference on Software Maintenance)). This conference series is similar to SANER.
- *CC*[13]: **Compiler Construction**. This conference series focus on language implementation, specifically compiler construction, which is a classic core component

[5] MODELS: http://dblp.uni-trier.de/db/conf/models/ (at DBLP)

[6] ECMFA: http://dblp.uni-trier.de/db/conf/ecmdafa/ (at DBLP)

[7] MODELSWARD: http://dblp.uni-trier.de/db/conf/modelsward/ (at DBLP)

[8] ICMT: http://dblp.uni-trier.de/db/conf/icmt/ (at DBLP)

[9] MSR: http://dblp.uni-trier.de/db/conf/msr/ (at DBLP)

[10] ICPC: http://dblp.uni-trier.de/db/conf/iwpc/ (at DBLP)

[11] SANER: http://dblp.uni-trier.de/db/conf/wcre/ (at DBLP)

[12] ICSME: http://dblp.uni-trier.de/db/conf/icsm/ (at DBLP)

[13] CC: http://dblp.uni-trier.de/db/conf/cc/ (at DBLP)

of the software language field. Language implementation aspects other than those directly relevant to compilation are not systematically covered.

- *PEPM*[14]: **Partial Evaluation and Semantic-Based Program Manipulation**. This conference series is concerned with program manipulation, partial evaluation, and program generation. The focus is on semantics-based methods and programming languages (including domain-specific languages) as opposed to engineering and software languages generally.
- *ICSE*[15]: **International Conference on Software Engineering**. This conference series covers software engineering broadly. A significant percentage of ICSE papers involve language-centric tools and methods, for example, in the sense of refactorings, IDEs, and automated testing. Combinations of software language engineering and empirical software engineering are common – just as in the case of the MSR conferences.
- *ASE*[16]: **Automated Software Engineering**. This conference series is similar to ICSE, except that it entirely focuses on automated aspects of software engineering.
- *PLDI*[17]: **Programming Language Design and Implementation**. This conference series covers all areas of programming language research, including the design, implementation, theory, and efficient use of languages. There is a clear focus on implementation, though, for example, innovative and creative approaches to compile-time and runtime technology and results from implementations.
- *POPL*[18]: **Principles of Programming Languages**. This conference series covers all aspects of programming languages and programming systems. Historically, POPL papers have been theoretical – they develop formal frameworks; more recently, experimental papers and experience reports have been encouraged.
- *ECOOP*[19]: **European Conference on Object-Oriented Programming**. This conference series covers all areas of object technology and related software development technologies with a focus on foundations (semantics, types, semantics-based tools, language implementation).

In addition to the established conferences listed above, let us also point out an emerging conference series, http://programming-conference.org/, which is associated with a dedicated journal, http://programming-conference.org/journal/. This conference series promises, as it develops further, to be very relevant in terms of software language topics.

It would also be useful to itemize journals in a similar manner, but we leave this as an "advanced exercise" to the reader. One could, for example, set up and execute a suitable methodology to review computer-science journals in terms of their coverage

[14] PEPM: http://dblp.uni-trier.de/db/conf/pepm/ (at DBLP)

[15] ICSE: http://dblp.uni-trier.de/db/conf/icse/ (at DBLP)

[16] ASE: http://dblp.uni-trier.de/db/conf/kbse/ (at DBLP)

[17] PLDI: http://dblp.uni-trier.de/db/conf/pldi/ (at DBLP)

[18] POPL: http://dblp.uni-trier.de/db/conf/popl/ (at DBLP)

[19] ECOOP: http://dblp.uni-trier.de/db/conf/ecoop/ (at DBLP)

of the software language area. To this end, one could follow common guidelines for a systematic mapping study [110, 111].

Feedback Appreciated

Readers are strongly encouraged to get in touch with the book's author, who is looking forward to incorporating any feedback received into a future revision of this book and to advertise contributed resources. Please see the book's website[20] for contact information.

References

1. Aho, A., Monica S., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison Wesley (2006). 2nd edition
2. Amrani, M., Combemale, B., Lucio, L., Selim, G.M.K., Dingel, J., Traon, Y.L., Vangheluwe, H., Cordy, J.R.: Formal verification techniques for model transformations: A tridimensional classification. J. Object Technol. **14**(3), 1–43 (2015)
3. Appel, A., Palsberg, J.: Modern Compiler Implementation in Java. Cambridge University Press (2002). 2nd edition
4. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The PoplMark challenge. In: Proc. TPHOLs, *LNCS*, vol. 3603, pp. 50–65. Springer (2005)
5. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press (1990)
6. Barendregt, H.: Introduction to generalized type systems. J. Funct. Program. **1**(2), 125–154 (1991)
7. Baxter, G., Frean, M.R., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.D.: Understanding the shape of Java software. In: Proc. OOPSLA, pp. 397–412. ACM (2006)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
9. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2013)
10. Bézivin, J.: Model driven engineering: An emerging technical space. In: GTTSE 2005, Revised Papers, *LNCS*, vol. 4143, pp. 36–64. Springer (2006)
11. Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: CENTAUR: the system. In: Proc. SDE 1988, pp. 14–24. ACM (1989)
12. van den Bos, J., van der Storm, T.: Domain-specific optimization in digital forensics. In: Proc. ICMT, *LNCS*, vol. 7307, pp. 121–136. Springer (2012)
13. Boujarwah, A., Saleh, K.: Compiler test suite: Evaluation and use in an automated test environment. Inf. Softw. Technol. **36**(10), 607–614 (1994)
14. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda – A functional language with dependent types. In: Proc. TPHOLs, *LNCS*, vol. 5674, pp. 73–78. Springer (2009)
15. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**(5), 552–593 (2013)

[20] http://www.softlang.org/book

16. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: PolyTOIL: A type-safe polymorphic object-oriented language. ACM Trans. Program. Lang. Syst. **25**(2), 225–290 (2003)
17. Burgess, C.J., Saidi, M.: The automatic generation of test cases for optimizing Fortran compilers. Inf. Softw. Technol. **38**(2), 111–119 (1996)
18. Cartwright, R., Fagan, M.: Soft typing. In: Proc. PLDI, pp. 278–292. ACM (1991)
19. Ceh, I., Crepinsek, M., Kosar, T., Mernik, M.: Ontology driven development of domain-specific languages. Comput. Sci. Inf. Syst. **8**(2), 317–342 (2011)
20. Chandramohan, K., O'Boyle, M.F.P.: A compiler framework for automatically mapping data parallel programs to heterogeneous MPSoCs. In: Proc. CASES, pp. 1–10. ACM (2014)
21. Chiba, S.: Load-time structural reflection in Java. In: Proc. ECOOP, *LNCS*, vol. 1850, pp. 313–336. Springer (2000)
22. Collberg, C.S., Myles, G., Stepp, M.: An empirical study of Java bytecode programs. Softw., Pract. Exper. **37**(6), 581–641 (2007)
23. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata (2007)
24. Crary, K., Weirich, S., Morrisett, J.G.: Intensional polymorphism in type-erasure semantics. In: Proc. ICFP, pp. 301–312. ACM (1998)
25. Crary, K., Weirich, S., Morrisett, J.G.: Intensional polymorphism in type-erasure semantics. J. Funct. Program. **12**(6), 567–600 (2002)
26. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–646 (2006)
27. Dahm, M.: Byte code engineering. In: Java-Informations-Tage, pp. 267–277 (1999)
28. Dai, X., Zhai, A., Hsu, W., Yew, P.: A general compiler framework for speculative optimizations using data speculative code motion. In: Proc. CGO, pp. 280–290. IEEE (2005)
29. Degueule, T.: Composition and interoperability for external domain-specific language engineering. Ph.D. thesis, Université de Rennes 1 (2016)
30. Denker, M.: Sub-method structural and behavioral reflection. Ph.D. thesis, University of Bern (2008)
31. Duboscq, G., Würthinger, T., Mössenböck, H.: Speculation without regret: Reducing deoptimization meta-data in the Graal compiler. In: Proc. PPPJ, pp. 187–193. ACM (2014)
32. Erdweg, S.: Extensible languages for flexible and principled domain abstraction. Ph.D. thesis, Philipps-Universität Marburg (2013)
33. Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: The state of the art in language workbenches – conclusions from the language workbench challenge. In: Proc. SLE, *LNCS*, vol. 8225, pp. 197–217. Springer (2013)
34. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
35. Erwig, M., Walkingshaw, E.: Semantics first! – rethinking the language design process. In: Proc. SLE 2011, *LNCS*, vol. 6940, pp. 243–262. Springer (2012)
36. Fähndrich, M., Carbin, M., Larus, J.R.: Reflective program generation with patterns. In: Proc. GPCE, pp. 275–284. ACM (2006)
37. Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems **46**(2), 251–300 (2010)
38. Favre, J., Gasevic, D., Lämmel, R., Pek, E.: Empirical language analysis in software linguistics. In: Proc. SLE 2010, *LNCS*, vol. 6563, pp. 316–326. Springer (2011)
39. Favre, J.M., Gasevic, D., Lämmel, R., Winter, A.: Guest editors' introduction to the special section on software language engineering. IEEE Trans. Softw. Eng. **35**(6), 737–741 (2009)

40. Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation. In: Proc. POPL, pp. 111–122. ACM (2004)
41. Fowler, M.: Domain-Specific Languages. Addison-Wesley (2010)
42. Friedman, D., Wand, M.: Essentials of Programming Languages. MIT Press (2008). 3rd edition
43. Gil, J., Maman, I.: Micro patterns in Java code. In: Proc. OOPSLA, pp. 97–116. ACM (2005)
44. Glew, N.: Type dispatch for named hierarchical types. In: Proc. ICFP, pp. 172–182. ACM (1999)
45. Grune, D., Jacobs, C.: Parsing Techniques: A Practical Guide. Monographs in Computer Science. Springer (2007). 2nd edition
46. Gunter, C.: Semantics of Programming Languages: Structures and Techniques. MIT Press (1992)
47. Harm, J., Lämmel, R.: Two-dimensional approximation coverage. Informatica (Slovenia) **24**(3) (2000)
48. Harm, J., Lämmel, R., Riedewald, G.: The Language Development Laboratory — LDL. In: Proc. NWPT, pp. 77–86 (1997). Research Report 248. University of Oslo
49. Hinze, R.: A new approach to generic functional programming. In: Proc. POPL, pp. 119–132. ACM (2000)
50. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)
51. Hoare, C.: An axiomatic basis for computer programming (reprint). Commun. ACM **26**(1), 53–56 (1983)
52. Hoare, C.A.R.: Hints on programming language design. Tech. rep., Stanford University (1973)
53. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Pearson (2013). 3rd edition
54. Huang, S.S., Smaragdakis, Y.: Morphing: Structurally shaping a class by reflecting on others. ACM Trans. Program. Lang. Syst. **33**(2), 6 (2011)
55. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
56. Jansson, P., Jeuring, J.: Polyp – A polytypic programming language. In: Proc. POPL, pp. 470–482. ACM (1997)
57. Johnstone, A., Scott, E.: Modelling GLL parser implementations. In: Proc. SLE 2010, *LNCS*, vol. 6563, pp. 42–61. Springer (2011)
58. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008)
59. Kalinov, A., Kossatchev, A., Petrenko, A., Posypkin, M., Shishkov, V.: Coverage-driven automated compiler test suite generation. ENTCS **82**(3) (2003)
60. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: A survey of the first wave. In: Conceptual Modelling and Its Theoretical Foundations – Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday, *LNCS*, vol. 7260, pp. 197–215. Springer (2012)
61. Kats, L.C.L., Visser, E.: The Spoofax language workbench. In: Companion SPLASH/OOPSLA, pp. 237–238. ACM (2010)
62. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proc. OOPSLA, pp. 444–463. ACM (2010)
63. Kelly, S., Tolvanen, J.: Domain-Specific Modeling. IEEE & Wiley (2008)
64. Kiczales, G.: Aspect-oriented programming. In: Proc. ICSE, p. 730. ACM (2005)
65. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proc. ECOOP, *LNCS*, vol. 1241, pp. 220–242. Springer (1997)
66. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proc. ICSE, pp. 49–58. ACM (2005)
67. Kiczales, G., des Rivieres, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)

68. Kim, Y., Kiemb, M., Park, C., Jung, J., Choi, K.: Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: Proc. DATE, pp. 12–17. IEEE (2005)
69. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley (2008)
70. Klusener, A.S., Lämmel, R., Verhoef, C.: Architectural modifications to deployed software. Sci. Comput. Program. **54**(2-3), 143–211 (2005)
71. Klusener, S., Lämmel, R.: Deriving tolerant grammars from a base-line grammar. In: Proc. ICSM, pp. 179–188. IEEE (2003)
72. Knuth, D.E.: An empirical study of FORTRAN programs. Softw., Pract. Exper. **1**(2), 105–133 (1971)
73. Kolovos, D.S., Rose, L.M., bin Abid, S., Paige, R.F., Polack, F.A.C., Botterweck, G.: Taming EMF and GMF using model transformation. In: Proc. MODELS, *LNCS*, vol. 6394, pp. 211–225. Springer (2010)
74. Krishnamurthi, S.: Programming Languages: Application and Interpretation. Brown University (2007). https://cs.brown.edu/~sk/Publications/Books/ProgLangs/
75. Kronawitter, S., Stengel, H., Hager, G., Lengauer, C.: Domain-specific optimization of two Jacobi smoother kernels and their evaluation in the ECM performance model. Parallel Processing Letters **24**(3) (2014)
76. Kurtev, I., Bézivin, J., Akşit, M.: Technological spaces: An initial appraisal. In: Proc. CoopIS, DOA 2002, Industrial track (2002)
77. Lämmel, R.: Grammar testing. In: Proc. FASE, *LNCS*, vol. 2029, pp. 201–216. Springer (2001)
78. Lämmel, R.: A semantical approach to method-call interception. In: Proc. AOSD, pp. 41–55. ACM (2002)
79. Lämmel, R., Jones, S.L.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Proc. TLDI, pp. 26–37. ACM (2003)
80. Lämmel, R., Kitsis, S., Remy, D.: Analysis of XML schema usage. In: Proc. XML (2005)
81. Lämmel, R., Pek, E.: Understanding privacy policies – A study in empirical analysis of language usage. Empir. Softw. Eng. **18**(2), 310–374 (2013)
82. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: Proc. TestCom, *LNCS*, vol. 3964, pp. 19–38. Springer (2006)
83. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO, pp. 75–88. IEEE (2004)
84. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of standard ML. In: Proc. POPL, pp. 173–184. ACM (2007)
85. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: Proc. POPL, pp. 333–343. ACM (1995)
86. Liang, Y., Xie, X., Sun, G., Chen, D.: An efficient compiler framework for cache bypassing on GPUs. IEEE Trans. CAD Integr. Circ. Syst. **34**(10), 1677–1690 (2015)
87. Louden, K.: Compiler Construction: Principles and Practice. Cengage Learning (1997)
88. Marriott, K., Meyer, B. (eds.): Visual Language Theory. Springer (1998)
89. Maurer, P.: Generating test data with enhanced context-free grammars. IEEE Softw. **7**(4), 50–56 (1990)
90. McBride, C.: Epigram: Practical programming with dependent types. In: Proc. AFP, *LNCS*, vol. 3622, pp. 130–170. Springer (2004)
91. Meijer, E., Drayton, P.: Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages (2005). Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.69.5966
92. Mens, T.: Model Transformation: A Survey of the State of the Art, pp. 1–19. John Wiley & Sons, Inc. (2013)
93. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)

94. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press (1999)
95. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Trans. Program. Lang. Syst. **10**(3), 470–502 (1988)
96. Moody, D.L.: The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng. **35**(6), 756–779 (2009)
97. Moonen, L.: Generating robust parsers using island grammars. In: Proc. WCRE, pp. 13–22. IEEE (2001)
98. Moonen, L.: Lightweight impact analysis using island grammars. In: Proc. IWPC, pp. 219–228. IEEE (2002)
99. Mosses, P.D.: Theory and practice of action semantics. In: Proc. MFCS, *LNCS*, vol. 1113, pp. 37–61. Springer (1996)
100. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. **60-61**, 195–228 (2004)
101. Murphy, G.C., Notkin, D.: Lightweight lexical source model extraction. ACM Trans. Softw. Eng. Methodol. **5**(3), 262–292 (1996)
102. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design, Recent Insight and Advances, *LNCS*, vol. 1710, pp. 114–136. Springer (1999)
103. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer (2007)
104. Nilsson-Nyman, E., Ekman, T., Hedin, G.: Practical scope recovery using bridge parsing. In: Proc. SLE 2008, *LNCS*, vol. 5452, pp. 95–113. Springer (2009)
105. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
106. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology (2007)
107. Norell, U.: Dependently typed programming in Agda. In: AFP 2008, Revised Lectures, *LNCS*, vol. 5832, pp. 230–266. Springer (2009)
108. Parr, T.: Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages. Pragmatic Bookshelf (2010)
109. Parr, T., Fisher, K.: LL(*): The foundation of the ANTLR parser generator. In: Proc. PLDI, pp. 425–436. ACM (2011)
110. Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M.: Systematic mapping studies in software engineering. In: Proc. EASE, Workshops in Computing. BCS (2008)
111. Petersen, K., Vakkalanka, S., Kuzniarz, L.: Guidelines for conducting systematic mapping studies in software engineering: An update. Inf. Softw. Technol. **64**, 1–18 (2015)
112. Pfenning, F., Schürmann, C.: System description: Twelf – A meta-logical framework for deductive systems. In: Proc. CADE-16, *LNCS*, vol. 1632, pp. 202–206. Springer (1999)
113. Pierce, B.: Types and Programming Languages. MIT Press (2002)
114. Purdom, P.: A sentence generator for testing parsers. BIT **12**(3), 366–375 (1972)
115. Raghavan, P., Lambrechts, A., Absar, J., Jayapala, M., Catthoor, F., Verkest, D.: Coffee: COmpiler Framework for Energy-aware Exploration. In: Proc. HiPEAC, *LNCS*, vol. 4917, pp. 193–208. Springer (2008)
116. Ranta, A.: Implementing Programming Languages: An Introduction to Compilers and Interpreters. College Publications (2012)
117. Rekers, J.: Parser generation for interactive environments. Ph.D. thesis, University of Amsterdam (1992)
118. Renggli, L.: Dynamic language embedding with homogeneous tool support. Ph.D. thesis, Universität Bern (2010)
119. Reps, T.W., Teitelbaum, T.: The Synthesizer Generator – A System for Constructing Language-Based Editors. Texts and Monographs in Computer Science. Springer (1989)
120. Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T., Mössenböck, H.: Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle. In: Proc. VMILSPLASH, pp. 6–15. ACM (2016)

121. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proc. PLDI, pp. 159–169. ACM (2008)
122. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Publishing Company (1997). Volume 1: Foundations
123. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
124. Schauss, S., Lämmel, R., Härtel, J., Heinz, M., Klein, K., Härtel, L., Berger, T.: A chrestomathy of DSL implementations. In: Proc. SLE. ACM (2017). 12 pages
125. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)
126. Scott, E., Johnstone, A.: Structuring the GLL parsing algorithm for performance. Sci. Comput. Program. **125**, 1–22 (2016)
127. Scott, M.: Programming Language Pragmatics. Morgan Kaufmann (1996). 3rd edition
128. Sebesta, R.W.: Concepts of Programming Languages. Addison-Wesley (2012). 10th edition
129. Sethi, R.: Programming Languages: Concepts and Constructs. Addison Wesley (1996). 2nd edition
130. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Proc. Workshop on Scheme and Functional Programming, pp. 81–92. University of Chicago (2006)
131. Siek, J.G., Taha, W.: Gradual typing for objects. In: Proc. ECOOP, *LNCS*, vol. 4609, pp. 2–27. Springer (2007)
132. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: Proc. SNAPL, *LIPIcs*, vol. 32, pp. 274–293. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2015)
133. Simonet, V.: An extension of HM(X) with bounded existential and universal data-types. In: Proc. ICFP, pp. 39–50. ACM (2003)
134. Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: Proc. DSL, pp. 1–13. USENIX (1999)
135. Slonneger, K., Kurtz, B.: Formal Syntax and Semantics of Programming Languages. Addison Wesley (1995)
136. Slutz, D.: Massive stochastic testing for SQL. Tech. Rep. MSR-TR-98-21, Microsoft Research (1998). A shorter form of the paper appeared in the Proc. VLDB 1998
137. Stuart, T.: Understanding Computation: From Simple Machines to Impossible Programs. O'Reilly (2013)
138. Tennent, R.: Specifying Software. Cambridge University Press (2002)
139. Tolvanen, J., Kelly, S.: Defining domain-specific modeling languages to automate product derivation: Collected experiences. In: Proc. SPLC, *LNCS*, vol. 3714, pp. 198–209. Springer (2005)
140. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proc. IJCAI, pp. 756–764. Morgan Kaufmann (1985)
141. Visser, E., Wachsmuth, G., Tolmach, A.P., Neron, P., Vergu, V.A., Passalaqua, A., Konat, G.: A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In: Proc. SPLASH, Onward!, pp. 95–111. ACM (2014)
142. Voelter, M.: Embedded software development with projectional language workbenches. In: Proc. MODELS, *LNCS*, vol. 6395, pp. 32–46. Springer (2010)
143. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering – Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
144. Voelter, M., Lisson, S.: Supporting diverse notations in MPS' projectional editor. In: Proc. GEMOC@Models 2014, *CEUR Workshop Proceedings*, vol. 1236, pp. 7–16. CEUR-WS.org (2014)
145. Voelter, M., Ratiu, D., Kolb, B., Schätz, B.: mbeddr: instantiating a language workbench in the embedded software domain. Autom. Softw. Eng. **20**(3), 339–390 (2013)
146. Völter, M., Siegmund, J., Berger, T., Kolb, B.: Towards user-friendly projectional editors. In: Proc. SLE, *LNCS*, vol. 8706, pp. 41–61. Springer (2014)

147. Völter, M., Visser, E.: Language extension and composition with language workbenches. In: Companion SPLASH/OOPSLA, pp. 301–304. ACM (2010)
148. Wachsmuth, G., Konat, G.D.P., Visser, E.: Language design with the Spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)
149. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. ACM Trans. Program. Lang. Syst. **26**(5), 890–910 (2004)
150. Wehr, S., Lämmel, R., Thiemann, P.: JavaGI: Generalized interfaces for Java. In: Proc. ECOOP, *LNCS*, vol. 4609, pp. 347–372. Springer (2007)
151. Wile, D.S.: Lessons learned from real DSL experiments. In: Proc. HICSS-36, p. 325. IEEE (2003)
152. Wile, D.S.: Lessons learned from real DSL experiments. Sci. Comput. Program. **51**(3), 265–290 (2004)
153. Würthinger, T.: Graal and Truffle: Modularity and separation of concerns as cornerstones for building a multipurpose runtime. In: Proc. Modularity, pp. 3–4. ACM (2014)

Index

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7

P

Q

R

S