Copyright and Attribution

Copyright for Code

The code in this book is part of the open-source YAS project: 
http://www.softlang.org/yas
YAS is licensed under the MIT license Details follow.
Copyright 2016–2018 Ralf Lämmel
Permission is hereby granted, free of charge, to any person obtaining a copy of the YAS code including software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Artwork Credits

Cover artwork:
With the assistance of Archina Void and Daniel Dünker.
Licensed under CC BY-SA 4.0
Artwork DMT, acrylic, 2006 by Matt Sheehy is quoted with the artist’s permission.

Credits for per-chapter artwork:
Wojciech Kwasnik.
See the individual chapters for details.
Preface

You must not fall into the trap of rejecting a surgical technique because it is beyond the capabilities of the barber in his shop around the corner.

– Edsger W. Dijkstra, EWD 512

Welcome to the Software Languages Book!

The Notion of a Software Language

A software language is an “artificial language,” the syntax and semantics of which may be realized in software. Software languages are particularly omnipresent in software and systems engineering. While a proper attempt at classification will follow later, here are some illustrative categories of software languages:

- programming languages (e.g., Python, Java, and Haskell);
- modeling languages (e.g., UML, Simulink, and Modelica);
- exchange formats (e.g., JSON and XML);
- markup languages (e.g., HTML);
- domain-specific languages (DSLs) for domains such as the following:
  - parsing;
  - build management;
  - machine control;
  - documentation;
  - configuration.

1 [https://www.cs.utexas.edu/users/EWD/transcriptions/EWD05xx/EWD512.html](https://www.cs.utexas.edu/users/EWD/transcriptions/EWD05xx/EWD512.html)
Software Language Engineering (SLE)

We shall honor fundamental concepts and engineering techniques across different use cases and forms of software languages, with different software technologies used for realization. That is, we endorse and advertise software language engineering (SLE). To quote from the website of (an edition of) the SLE conference, “Software language engineering is the application of systematic, disciplined, and quantifiable approaches to the development (design, implementation, testing, deployment), use, and maintenance (evolution, recovery, and retirement) of these languages.”

A Particular SLE Book

SLE is a relatively young field. (The term “SLE” may be dated back to 2007.) The knowledge of fundamental concepts and engineering techniques is scattered over multiple research communities and technological spaces. Thus, there exist “knowledge silos” with limited understanding of commonalities and specifics in different contexts. The present book is aimed at collecting and organizing scattered knowledge in the form of an accessible textbook. Given the breadth of the SLE field, this book cannot claim full coverage and a balanced presentation of the SLE field. This book is biased as follows:

- Coverage of language processors for source code analysis and manipulation informed by programming language theory and implementation. There is no coverage of runtime forms of metaprogramming such as reflection.
- A focus on application areas such as software analysis (software reverse engineering), software transformation (software re-engineering), software composition (modularity), and domain-specific languages.
- Usage of several programming languages (Java, Python, and Haskell) for illustration with Haskell taking a leading role. There are no illustrations for particular metaprogramming systems, language workbenches, or model-driven engineering technologies, but some of the underlying principles are discussed and pointers to further reading are provided.
- Code blocks (programs and executable specifications) form an integral part of the sequential text flow in this book. Code blocks are placed in non-floating “Illustration” blocks, as opposed to being moved into floating figures. The code is typically explained in subsequent text paragraphs (less so in code comments).

[http://www.sleconf.org/2012/]
Complementary Online Material

There is a website for the book. All software artifacts in the book are part of YAS (Yet Another SLR (Software Language Repository)); The code of YAS is hosted on GitHub. The book’s website provides complementary material, for example, lecture slides and videos.

Structure of the Preface

The rest of this preface provides descriptions as follows: the audience targeted by the book, the background assumed by the book, the characteristics of the book, an outline of the book, trails offered for using the book selectively, and the style of the exercises included throughout the book.

Audience Targeted by this Book

This book is designed as a textbook for self-organized learning and university courses for Bachelor (advanced level) or Master of Computer Science, in the broader context of software engineering.

This book serves those who have an intention of understanding the fundamental concepts and important engineering principles underlying software languages. Readers will acquire much of the operational intelligence needed for dealing with software languages in software development practice. Of course, readers may also need to consult more technology-specific resources when addressing specific problems with the help of specific technologies.

This book is primarily targeted at people in higher education. However, because of the book’s pragmatic (applied) approach, practitioners on a self-learning path may also appreciate the book.

The typical developer may have encountered language design and implementation in practice and may have used technologies, as dictated by the moment. This book raises the developer’s view to a higher level of abstraction and delivers more advanced techniques, for example, small-step semantics, formal type systems, quasi-quotation, term rewriting, and program analysis. The objective is that the book will enable readers to design, implement, assess, integrate, and evolve language-based software. This is an important skill set for software engineers, as languages are permeating software development in an increasing manner.
This book admits several different ‘trails’ (see below), thereby making it useful for different learning objectives and different course designs on the basis of adjusting the level of sophistication and selective inclusion of chapters. Depending on individual background or the local curriculum, some chapters or sections may be skipped or processed by cursory reading or the short videos available may be consulted instead.

Background Assumed by this Book

Required Knowledge

**Moderate programming skills**  One needs to be fluent in an object-oriented and a functional programming language. The illustrative examples in the book are written in many different notations and programming languages. Chapter 2 (“A Story of a Domain-Specific Language”) uses Java and Python in going through many aspects of language implementation in an introductory manner. Beyond Chapter 2, the functional programming language Haskell dominates as the language used for illustration. The book’s Haskell code is straightforward; advanced language features and idioms are avoided.

**Basic software engineering knowledge**  A basic understanding of the software lifecycle (analysis, design, implementation, testing, deployment, maintenance) is required. In particular, the reader needs to have mastered basic aspects of software design and testing. That is, the reader should have previously leveraged and designed domain models (e.g., object models) for different domains. Also, the reader should have some experience with unit testing and in testing the input/output behavior of software components.

Optional Knowledge

**Basic knowledge of theoretical computer science**  The book rehashes the relevant background in a pragmatic manner and hence, such knowledge is optional. This includes notions such as formal languages and computability.

**Basic knowledge of metaprogramming**  Such knowledge is optional because the book develops a particular (limited) view of metaprogramming from the ground up. We focus on source code analysis and manipulation. A few metaprogramming recipes are highlighted in the text. Runtime forms of metaprogramming such as reflection are not discussed.
Characteristics of this Book

SLE concepts and techniques  This book aims to identify, define, and illustrate the fundamental concepts and engineering techniques as relevant to applications of software languages in software development. Examples of these concepts include abstract syntax, compositionality, and type system. Examples of these techniques include parser generation and template processing. Some concepts and techniques will be explained by referring to a lifecycle for software languages or to the architecture of a typical language implementation such as a compiler.

Software engineering perspective  This book presents software languages primarily from a software engineering perspective. That is, the book basically addresses the following question: how to parse, analyze, transform, generate, format, and otherwise process software artifacts in different software languages, as they turn up in software development? This question is of interest in many areas of software engineering, most notably software reverse engineering, software reengineering, model-driven engineering, program comprehension, software analysis, program generation, and mining software repositories.

Diverse languages  This book covers a wide range of software languages—most notably programming languages, domain-specific languages, modeling languages, exchange formats, and specifically also language definition languages (notably grammar and metamodeling notations). Several different technological spaces are exercised, with some emphasis on grammarware and with excursions to modelware, XMLware, and JSONware. Several different programming paradigms are exercised, most notably, functional and object-oriented programming.

Polyglot illustration  Different languages are leveraged to illustrate SLE concepts and techniques. The functional programming language Haskell dominates the book. Additionally, the mainstream programming languages Python and Java are used for illustration. Further, XML, XML Schema, JSON, and JSON Schema are leveraged as mainstream options for exchange formats. ANTLR is used for mainstream parser development. A number of syntax definition formalisms (inspired by Backus-Naur form, Ecore, and algebraic signatures) are developed and systematically used in the book. The standard notion of inference rules for deductive systems is used for representing operational semantics and type systems.

Bits of theory  A deeper understanding of software languages must take into account some fundamental concepts typically studied in the field of programming language theory. In particular, this concerns semantics and type systems. This book presents these topics in a pragmatic manner so that the practical value of semantics definitions and type systems may become clear more easily and the knowledge gained can be applied to software languages other than programming languages; see Chapter 8–11.

Bits of language implementation  The development of interpreters or compilers for programming languages as well as runtime systems, is well understood and covered by existing textbooks. We take a more general view of language implementation, which covers languages other than programming languages and language-based software components other than compilers and interpreters. The
book covers the topic of interpreters relatively well. Compiler construction is covered only in a superficial manner. Runtime systems are not covered.

**Bits of programming paradigms** Just as this book is not a book on compiler construction, it is not a book on programming paradigms either. Nevertheless, the book exercises several paradigms to some extent. That is, languages of different paradigms are defined and implemented with a cursory discussion of the underlying language concepts. Further, languages of different paradigms are used in implementing the examples in the book. A systematic discussion of programming paradigms is beyond the scope of this book.

**Outline of this Book**

Preface
This is the current chapter.

**Chapter 1: The Notion of a Software Language**
The notion of a software language is introduced broadly by means of introducing example languages, classifying languages, discussing the language lifecycle, and identifying the roles of languages in software engineering.

**Chapter 2: A Story of a Domain-Specific Language**
A domain-specific modeling language, FSML, for finite state machines (FSMs) is discussed in terms of language concepts, lifecycle, syntax, operational semantics, and provision of a code generator. Mainstream implementation languages and technologies are leveraged.

**Chapter 3: Foundations of Tree- and Graph-Based Abstract Syntax**
The signature- and metamodel-based definitions of tree- and graph-based syntax and the accompanying notion of conformance are described in a pragmatic manner. The abstract syntaxes of several example languages are defined.

**Chapter 4: Representation of Object Programs in Metaprograms**
The implementation of abstract syntax is discussed, where object models in object-oriented programming or algebraic data types in functional programming serve the purpose of object-program representation in metaprogramming.

**Chapter 5: A Suite of Metaprogramming Scenarios**
Typical scenarios of metaprogramming are discussed and illustrated: interpretation, semantic analysis, transformation, translation. Only the basic idioms of metaprogramming are exercised. Concrete syntax is not considered yet.

**Chapter 6: Foundations of Textual Concrete Syntax**
The grammar-based definition of textual concrete syntax and the accompanying notions of acceptence and parsing are described in a pragmatic manner. The concrete syntaxes of several example languages are defined.

**Chapter 7: Implementation of Textual Concrete Syntax**
Several aspects of the implementation of concrete syntax are discussed: parsing (e.g., by using a parser generator or parser combinators), abstraction (i.e., the
mapping of concrete to abstract syntax), formatting (e.g., by means of template processing), and concrete object syntax.

**Chapter 8: A Primer on Operational Semantics**

The operational approach to semantics definition is described in a pragmatic manner. This approach leverages inference rules (deductive systems) to model the stepwise computation of programs in either big-step or small-step style. The operational semantics of several example languages are defined. The formal definitions can be used, for example, in the implementation of interpreters.

**Chapter 9: A Primer on Type Systems**

The notion of type systems is described in a pragmatic manner. The approach, again, leverages inference rules to assign properties to programs as sound, static predictions of runtime behavior. The type systems of several example languages are defined. The formal definitions can be used, for example, in the implementation of type checkers.

**Chapter 10: An Excursion into the Lambda Calculus**

The lambda calculus is described as a well-known vehicle for studying semantics and type systems of programming language constructs. In fact, a number of specific lambda calculi are discussed, and thereby we encounter polymorphism, structural and nominal typing, and subtyping.

**Chapter 11: An Ode to Compositionality**

The denotational approach to semantics definition is described in a pragmatic manner. This approach leverages functional equations to map program phrases to elements of suitable domains in a compositional (i.e., inductive) style. The denotational semantics of several example languages are defined.

**Chapter 12: A Suite of Metaprogramming Techniques**

Several metaprogramming techniques are described in a pragmatic manner: term rewriting, attribute grammars, multi-stage programming, partial evaluation, and abstract interpretation. The techniques are applied to different metaprogramming scenarios and example languages.

**Postface**

This final chapter summarizes the key concepts covered by the book, identifies omissions in this particular book on software languages, lists complementary textbooks, and mentions relevant academic conferences.

**Trails Offered by this Book**

This book may be walked through several different trails for selective self-learning experiences or course designs. Each trail listed below itemizes chapters to be include at different levels of detail:

- **low** cursory/highly selective coverage;
- **medium** incomplete coverage;
- **high** comprehensive coverage.
All these trails, when understood as course designs, correspond to advanced Bachelor’s or regular Master’s courses. Based on the author’s experience, these trails can be upgraded to research-oriented course designs for Master’s courses. To this end, the literature references provided and specifically, also those from the Postface of the book, and relevant technologies, for example, for metaprogramming, may be studied by students as part of their literature research and project work.

**Trail “An Introduction to Metaprogramming”**

- Chapter 1: *The Notion of a Software Language* [none–low]
- Chapter 2: *A Story of a Domain-Specific Language* [high]
- Chapter 3: *Foundations of Tree- and Graph-Based Abstract Syntax* [medium]
- Chapter 4: *Representation of Object Programs in Metapograms* [medium]
- Chapter 5: *A Suite of Metaprogramming Scenarios* [medium]
- Chapter 6: *Foundations of Textual Concrete Syntax* [medium]
- Chapter 7: *Implementation of Textual Concrete Syntax* [medium]

This is an introductory trail with Chapters 2 and 5 at its heart, complemented by modest coverage of the foundations and implementation of abstract and concrete syntax and, possibly, the notion of a software language in general. The assumption is here that this trail should touch upon different metalanguages (including mainstream options) and a broad variety of relatively simple metaprogramming scenarios and techniques. The complexity of the trail could be tuned by including more or less detail from Chapter 5.

**Trail “A Primer on Programming Language Theory”**

- Chapter 3: *Foundations of Tree- and Graph-Based Abstract Syntax* [low]
- Chapter 4: *Representation of Object Programs in Metapograms* [low]
- Chapter 5: *A Suite of Metaprogramming Scenarios* [low]
- Chapter 8: *A Primer on Operational Semantics* [high]
- Chapter 9: *A Primer on Type Systems* [high]
- Chapter 10: *An Excursion into the Lambda Calculus* [medium–high]
- Chapter 11: *An Ode to Compositionality* [medium–high]

This trail excludes the two introductory chapters, as a broad view of software languages is not required. The trail starts off with a short discussion of abstract syntax. The trail skips over the topic of concrete syntax. The excursion to scenarios of metaprogramming is recommended to introduce the notions of interpretation and type checking without reliance on formal notation. The remaining chapters in the trail deal with formal semantics and type systems in a pragmatic manner. Haskell serves as the implementation language. The complexity of the trail can be tuned by including more or less detail of the lambda calculus and denotational semantics. For instance, abstract interpretation and structural and nominal typing and subtyping may be considered optional.
Preface

Trail “Metaprogramming in Haskell”

- Chapter 1: The Notion of a Software Language [none–low]
- Chapter 3: Foundations of Tree- and Graph-Based Abstract Syntax [low]
- Chapter 4: Representation of Object Programs in Metaprograms [low]
- Chapter 5: A Suite of Metaprogramming Scenarios [high]
- Chapter 6: Foundations of Textual Concrete Syntax [low]
- Chapter 7: Implementation of Textual Concrete Syntax [low]
- Chapter 12: A Suite of Metaprogramming Techniques [high]

Chapters 5 and 12 provide the technical meat for this trail, prepared by modest coverage of the foundations and implementation of abstract and concrete syntax and, possibly, the notion of a software language in general.

Trail “Software Language Engineering”

This is a long trail through all chapters. This trail may be too long for an actual course.

Exercises in the Book

Each exercise is marked with a level of difficulty:

**Basic** These exercises are concerned with using the relevant techniques in a basic manner; the book’s coverage should suffice for solving these exercises.

**Intermediate** These exercises are concerned with aspects of techniques or scenarios of their usage that may go beyond the book’s coverage. Some research (“googling”), teamwork, and advice from an instructor may be needed for solving the exercises.

**Advanced** These exercises are at the level of project assignments that may require weeks of work depending on background and supervision. These exercises could also be used for giving focus to research efforts in a course, for example.

Except for the exercises at the basic level, the formulations given intentionally leave room for creativity. In general, solutions to the exercises will not be explicitly published by the book’s author, although the resources for the book may provide some relevant pieces of information.
I am grateful to my academic peers, who have helped me to learn many of the things that I now want to pass on with this book. I list these peers in somewhat chronological order, Günter Riedewald (my diploma and PhD supervisor to whom I owe so much), Uwe Lämmel (an early mentor who put me on the Prolog and grammar quest), Mark van den Brand (a mentor who helped me during my PhD period in diverse ways), Paul Klint (my key mentor during my PostDoc and senior researcher times at CWI, Amsterdam), Chris Verhoef (my boss at VU, Amsterdam, who introduced me to the art of writing), Simon L. Peyton Jones (with whom I wrote a few great papers and from whom I learned other aspects of the art of writing), Erik Meijer (a very exciting mentor at Microsoft and, more recently, at Facebook), and Jean-Marie Favre (an inspiring colleague and friend of whom you have one of the kind in a lifetime).

There are yet other peers with whom I may have collaborated only more briefly, but they also have helped me to learn things I want to pass on with this book; they are stated here in no well-defined order: Eelco Visser, Dragan Gasevic, Simon Thompson, Krzysztof Czarnecki, Jean Bézivin, James R. Cordy, Oleg Kiselyov, Peter Thiemann, Wolfram Schulte, Wald Taha, Tijs van der Storm, Frédéric Jouault, Robert Hirschfeld, William R. Cook, Alfonso Pierantonio, Marjan Mernik, Dietrich Paulus, Steffen Staab, and Jan Maluszynski.

I kindly acknowledge collaboration with Anya Helene Bagge on initial attempts at a textbook design and continued efforts regarding education in software language engineering. I envy people who can write a textbook in a team effort; I cannot.

Further, I am very grateful to those who provided input for or feedback on (parts of) the book. I want to mention specifically Mahdi Derakhshmanesh, Torsten Grust, Bradford Larsen, Nicolas Laurent, Eliot Miranda, Friedrich Steimann, and Vadim Zaytsev.

I kindly acknowledge collaboration on relevant subjects with current or former graduate or undergraduate students at the University of Rostock; Universiteit van Amsterdam; the Dutch Centre of Mathematics and Computer Science, Amsterdam (CWI); Vrije Universiteit, Amsterdam; Universität Koblenz-Landau; and other places. I want to mention specifically Jan Kort, Joost Visser, Wolfgang Lohmann,
Acknowledgments

Thiago T. Bartolomei, Markus Kaiser, Ekaterina Pek, Andrei Varanovich, Vadim Zaytsev, Marcel Heinz, Lukas Härtel, Johannes Härtel, Kevin Klein, and Simon Schauss.

Several editions of my courses on software language engineering and programming language theory have helped in collecting and maturing the content; thanks to all the students on these courses for bearing with me.

I kindly acknowledge the artistic work of Wojciech Kwasnik and his collaborators on the book’s cover and the per-chapter artwork. I very much enjoyed the endeavor—a “deep art” approach. The artwork on the book’s cover shows a tower (inspired by the Tower of Babel), suggesting a notion of a “Tower of Software Languages”. The tower is the output of a neural algorithm applied to a simpler (computed) tower and used a style image by Matt Sheehy for “morphing”. “Tron design” was applied at the border of the tower’s shape. The images for the per-chapter artwork were derived based on the following pattern: the image of a computer scientist to be honored was composed with artwork by Vincent van Gogh for the person’s background; artwork by Matt Sheehy (the same as for the book’s cover) was used to morph the person’s clothes and “Tron design” was applied at the border of the person’s shape. Thus, there is a constructive similarity between the “Tower of Software Languages” and the morphed images of the persons. See [http://www.softlang.org/book-art](http://www.softlang.org/book-art) for information on the art and the way in which computer scientists are honored in this book.

Much of the book was written in Koblenz in proximity to the Mosel and Rhine river, perhaps, in the morning, while eating scrambled eggs and drinking coffee at Baeckerei Hoefer (Ferdinand-Sauerbruch-Straße); in the Sunday afternoons, while drinking Darjeeling or, possibly, Riesling at Kaffeewirtschaft (Münzplatz); or on Saturday or Sunday evenings on Holger’s rocking chair.

Dear Olya, thank you for being there and sharing your life with me.

Ralf Lämmel (Software Language Engineer)
March 1st, 2018
Contents

List of Recipes ................................................................. xxvii

Acronyms ................................................................. xxix
Fabricated languages ............................................................ xxix
Other acronyms ............................................................... xxx

I The Notion of a Software Language ........................................ 1
1.1 Examples of Software Languages ......................................... 2
1.1.1 Real-World Software Languages ..................................... 2
1.1.2 Fabricated Software Languages ....................................... 3
1.1.2.1 BNL: A Language of Binary Numbers .......................... 5
1.1.2.2 BTL: An Expression Language ................................ 5
1.1.2.3 BL: A Language for Buddy Relationships ..................... 6
1.1.2.4 BFPL: A Functional Programming Language ................. 6
1.1.2.5 BIPL: An Imperative Programming Language .................. 7
1.1.2.6 FSML: A Language for Finite State Machines ............... 7
1.1.2.7 BGL: A Language for Context-Free Grammars ............. 9
1.2 Classification of Software Languages ...................................... 9
1.2.1 Classification by Paradigm ........................................... 10
1.2.2 Classification by Type System ...................................... 11
1.2.3 Classification by Purpose ........................................... 12
1.2.4 Classification by Generality or Specificity ......................... 13
1.2.5 Classification by Representation .................................. 14
1.2.6 Classification by Notation ......................................... 15
1.2.7 Classification by Degree of Declarativeness ..................... 15
1.3 The Lifecycle of Software Languages .................................. 17
1.3.1 Language Definition ............................................... 18
1.3.2 Language Implementation ........................................ 20
1.3.2.1 Compilation versus Interpretation .............................. 20
1.3.2.2 Architecture of a Compiler ................................... 21
1.3.2.3 Classification of Language Processors ......................... 22
## Contents

1.3.2.4 Metaprogramming Systems ........................................ 25
1.3.2.5 Language Workbenches ........................................... 26
1.3.3 Language Evolution .................................................. 27

1.4 Software Languages in Software Engineering ......................... 28
1.4.1 Software Re-Engineering ......................................... 28
1.4.2 Software Reverse Engineering .................................. 30
1.4.3 Software Analysis ................................................ 31
1.4.4 Technological Spaces .............................................. 33
1.4.5 Model-Driven Engineering ....................................... 36

Summary and outline ....................................................... 38
References ............................................................... 38

2 A Story of a Domain-Specific Language .................................. 51
2.1 Language Concepts .................................................... 52
2.2 Internal DSL .......................................................... 54
2.2.1 Baseline Object Model ........................................... 54
2.2.2 Fluent API ....................................................... 57
2.2.3 Interpretation ..................................................... 61
2.2.4 Well-Formedness ................................................ 63

2.3 External DSL .......................................................... 66
2.3.1 Syntax Definition ................................................ 67
2.3.2 Syntax Checking .................................................. 68
2.3.3 Parsing ............................................................ 71
2.4 DSL Services .......................................................... 74
2.4.1 Interchange Format .............................................. 74
2.4.2 Code Generation ................................................ 76
2.4.3 Visualization ...................................................... 82

Summary and outline ....................................................... 84
References ............................................................... 85

3 Foundations of Tree- and Graph-Based Abstract Syntax .............. 87
3.1 Tree-Based Abstract Syntax .......................................... 88
3.1.1 Trees versus Terms .............................................. 88
3.1.2 A Basic Signature Notation .................................... 89
3.1.3 Abstract Syntax Trees .......................................... 90
3.1.4 An Extended Signature Notation ................................ 91
3.1.5 Illustrative Examples of Signatures ......................... 92
3.1.5.1 Syntax of Simple Expressions .......................... 92
3.1.5.2 Syntax of Simple Imperative Programs ............... 92
3.1.5.3 Syntax of Simple Functional Programs ............... 93
3.1.5.4 Syntax of Finite State Machines ....................... 94
3.1.6 Languages as Sets of Terms .................................. 95
3.1.7 Conformance to a Signature ................................ 96

3.2 Graph-Based Abstract Syntax ....................................... 96
3.2.1 Trees versus Graphs ............................................ 97
## Contents

3.2.2 Languages as Sets of Graphs ........................................... 98
3.2.3 A Metamodeling Notation ............................................. 99
3.2.4 Conformance to a Metamodel ......................................... 100
3.2.5 Illustrative Examples of Metamodels ................................. 101
3.2.5.1 Syntax of Finite State Machines ................................ 101
3.2.5.2 Syntax of Simple Functional Programs ......................... 102
3.3 Context Conditions ....................................................... 102
3.4 The Metametalevel ....................................................... 103
3.4.1 The Signature of Signatures .......................................... 104
3.4.2 The Signature of Metamodels ........................................ 105
3.4.3 The Metamodel of Metamodels ...................................... 106
Summary and outline ....................................................... 107
References ................................................................. 108

4 Representation of Object Programs in Metaprograms ............... 109
4.1 Representation Options .................................................... 110
4.1.1 Untyped Representation ............................................ 110
4.1.2 Universal Representation ............................................ 111
4.1.3 Typeful Representation ............................................ 112
4.1.3.1 Algebraic Data Type-Based Representation .................. 112
4.1.3.2 Object-Based Representation .................................. 114
4.1.3.3 Reference Relationships ....................................... 115
4.1.3.4 Smart Constructors ............................................. 118
4.1.4 Interchange Formats .................................................... 120
4.1.4.1 JSON Representation ........................................... 120
4.1.4.2 XML Representation ........................................... 121
4.2 Conformance Checking .................................................... 122
4.2.1 Language-Specific Conformance Checking ......................... 122
4.2.2 Generic Conformance Checking ..................................... 123
4.2.3 Schema-Based Conformance Checking ............................... 125
4.3 Serialization ............................................................. 128
4.4 AST-to-ASG Mapping ...................................................... 129
Summary and outline ....................................................... 133
References ................................................................. 133

5 A Suite of Metaprogramming Scenarios ............................... 135
5.1 Interpretation ............................................................. 136
5.1.1 Basics of Interpretation ............................................. 136
5.1.2 Interpretation with Stores .......................................... 138
5.1.3 Interpretation with Environments .................................. 141
5.1.4 Stepwise Interpretation ............................................. 143
5.2 Compilation ............................................................. 146
5.2.1 Architecture of a Compiler ........................................ 147
5.2.2 Translation to Assembly Code ..................................... 148
5.2.3 Translation to Machine Code ...................................... 151
9 A Primer on Type Systems ................................................. 271
  9.1 Types ................................................................. 272
  9.2 Typing Judgments .................................................... 272
  9.3 Typing Rules ......................................................... 273
  9.4 Typing Derivations ................................................... 274
  9.5 Type Safety ........................................................... 274
  9.6 Type Checking ......................................................... 277
  9.7 More Examples of Type Systems ..................................... 278
      9.7.1 Well-Typedness of Simple Imperative Programs ........... 278
      9.7.2 Well-Typedness of Simple Functional Programs ......... 284
      9.7.3 Well-Formedness of Finite State Machines ............... 287
  Summary and outline .................................................... 287
  References ............................................................... 288

10 An Excursion into the Lambda Calculus ............................. 289
  10.1 The Untyped Lambda Calculus ..................................... 290
      10.1.1 Syntax ......................................................... 290
      10.1.2 Semantics ...................................................... 291
      10.1.3 Substitution .................................................... 292
      10.1.4 Predefined Values and Operations ......................... 294
      10.1.5 Fixed-Point Computation .................................... 295
      10.1.6 Interpretation .................................................. 296
      10.1.7 Turing Completeness .......................................... 298
  10.2 The Simply Typed Lambda Calculus ............................... 299
      10.2.1 Syntax ......................................................... 299
      10.2.2 Semantics ...................................................... 300
      10.2.3 Type System .................................................... 300
      10.2.4 Type Checking .................................................. 301
      10.2.5 Type Erasure .................................................... 302
  10.3 System $F$ .......................................................... 303
      10.3.1 Syntax ......................................................... 304
      10.3.2 Semantics ...................................................... 305
      10.3.3 Type System .................................................... 306
      10.3.4 Type Erasure .................................................... 307
  10.4 Type-System Extensions ............................................ 309
      10.4.1 Records and Variants ......................................... 309
      10.4.2 Structural Type Equivalence ................................ 312
      10.4.3 Structural Subtyping ......................................... 312
      10.4.4 Nominal Typing ................................................ 315
  Summary and outline .................................................... 318
  References ............................................................... 318
## An Ode to Compositionality

11.1 Compositionality ................................................................. 320
11.2 Direct Style ................................................................. 320
   11.2.1 Semantic Domains ...................................................... 321
   11.2.2 Semantic Functions ................................................... 321
   11.2.3 Semantic Combinators ............................................... 322
   11.2.4 Fixed-Point Semantics .............................................. 323
   11.2.5 Direct-Style Interpreters ......................................... 325
11.3 Continuation Style ......................................................... 328
   11.3.1 Continuations .......................................................... 328
   11.3.2 Continuation-Style Interpreters .................................. 329
   11.3.3 Semantics of Gotos .................................................... 330
Summary and outline .............................................................. 333
References ............................................................................ 334

## A Suite of Metaprogramming Techniques

12.1 Term Rewriting ............................................................... 336
   12.1.1 Rewrite Rules ............................................................ 336
   12.1.2 Encoding Rewrite Rules .............................................. 338
   12.1.3 Normalization ............................................................. 340
   12.1.4 Strategic Programming ................................................ 341
   12.1.5 Rewriting-Related concerns ........................................ 345
      12.1.5.1 Other Traversal Idioms ........................................ 345
      12.1.5.2 Concrete Object Syntax ........................................ 346
      12.1.5.3 Graph Rewriting and Model Transformation ............... 346
      12.1.5.4 Origin Tracking .................................................. 346
      12.1.5.5 Layout Preservation ............................................ 346
   12.2 Attribute Grammars ...................................................... 347
      12.2.1 The Basic Attribute Grammar Formalism ....................... 347
      12.2.2 Attribute Evaluation ............................................... 350
      12.2.3 Attribute Grammars as Functional Programs .................. 354
      12.2.4 Attribute Grammars with Conditions .......................... 356
      12.2.5 Semantic Actions with Attributes .............................. 358
   12.3 Multi-Stage Programming ............................................... 363
      12.3.1 Inlining as an Optimization Scenario .......................... 364
      12.3.2 Quasi-Quotation and Splicing .................................... 364
      12.3.3 More Typeful Staging .............................................. 366
   12.4 Partial Evaluation ........................................................ 368
      12.4.1 The Notion of a Residual Program ............................... 368
      12.4.2 Interpretation with Inlining ..................................... 370
      12.4.3 Interpreter with Memoization ................................... 375
   12.5 Abstract Interpretation ................................................ 380
      12.5.1 Sign Detection as an Optimization Scenario .................. 380
      12.5.2 Semantic Algebras .................................................. 381
      12.5.3 Concrete Domains ................................................... 382
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5.4 Abstract Domains</td>
<td>383</td>
</tr>
<tr>
<td>12.5.5 Examples of Abstract Interpreters</td>
<td>386</td>
</tr>
<tr>
<td>12.5.5.1 A Type-Checking Interpreter</td>
<td>386</td>
</tr>
<tr>
<td>12.5.5.2 A Sign-Detection Interpreter</td>
<td>388</td>
</tr>
<tr>
<td>Summary and outline</td>
<td>393</td>
</tr>
<tr>
<td>References</td>
<td>394</td>
</tr>
</tbody>
</table>

**Postface**

| The importance of Software Language Engineering | 399 |
| Software Languages: Key Concepts | 400 |
| Omissions in This Book | 401 |
| Complementary Textbooks | 403 |
| Software Languages in Academia | 405 |
| Feedback Appreciated | 408 |
| References | 408 |

**Index**

|  | 415 |
List of Recipes

2.1 Recipe (Development of a fluent API) ................................................. 61
2.2 Recipe (Development of an interpreter) ............................................. 63
2.3 Recipe (Development of a constraint checker) ..................................... 66
2.4 Recipe (Authoring a grammar) ............................................................ 67
2.5 Recipe (Development of a syntax checker) .......................................... 69
2.6 Recipe (Development of a parser) ....................................................... 73
2.7 Recipe (Development of a code generator) .......................................... 82

5.1 Recipe (Authoring an abstract syntax definition) ................................. 107

5.1 Recipe (Implementation of a conformance checker) .............................. 133

5.1 Recipe (Development of an interpreter (continued)) ........................... 138
5.2 Recipe (Development of a software transformation) ............................. 167

8.1 Recipe (Implementation of inference rules) .......................................... 247

11.1 Recipe (Compositional interpretation) ................................................ 327

12.1 Recipe (Design of a strategic program) ............................................. 345
12.2 Recipe (Design of an attribute grammar) ......................................... 362
12.3 Recipe (Design of a multi-stage program) ...................................... 368
12.4 Recipe (Design of a partial evaluator) ............................................ 379
12.5 Recipe (Design of an abstract interpreter) ........................................ 393
Acronyms

Fabricated Languages

In this book, several software languages have been “fabricated” to capture core design aspects of diverse real-world software languages. See Section 1.1.2 for a detailed discussion. Here is a summary:

- BAL  Basic Assembly Language
- BFPL  Basic Functional Programming Language
- BGL  Basic Grammar Language
- BIPL  Basic Imperative Programming Language
- BL  Buddy Language
- BML  Basic Machine Language
- BNL  Binary Number Language
- BSL  Basic Signature Language
- BTL  Basic TAPL Language
- EFPL  Extended Functional Programming Language
- EGL  Extended Grammar Language
- EIPL  Extended Imperative Programming Language
- EL  Expression Language
- ESL  Extended Signature Language
- FSML  Finite State Machine Language
- MML  MetaModeling Language
- TLL  Typed Lambda Language
- ULL  Untyped Lambda Language
### Other Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>abstract data type</td>
</tr>
<tr>
<td>AG</td>
<td>attribute grammar</td>
</tr>
<tr>
<td>AOP</td>
<td>aspect-oriented programming</td>
</tr>
<tr>
<td>ASG</td>
<td>abstract syntax graph</td>
</tr>
<tr>
<td>AST</td>
<td>abstract syntax tree</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus Naur form</td>
</tr>
<tr>
<td>ccpo</td>
<td>chain complete partial order</td>
</tr>
<tr>
<td>CFG</td>
<td>context-free grammar</td>
</tr>
<tr>
<td>COP</td>
<td>context-oriented programming</td>
</tr>
<tr>
<td>CPS</td>
<td>continuation-passing style</td>
</tr>
<tr>
<td>CST</td>
<td>concrete syntax tree</td>
</tr>
<tr>
<td>DSL</td>
<td>domain-specific language</td>
</tr>
<tr>
<td>DSML</td>
<td>domain-specific modeling language</td>
</tr>
<tr>
<td>EBNF</td>
<td>extended Backus Naur form</td>
</tr>
<tr>
<td>FSM</td>
<td>finite state machine</td>
</tr>
<tr>
<td>IDE</td>
<td>integrated development environment</td>
</tr>
<tr>
<td>IR</td>
<td>intermediate representation</td>
</tr>
<tr>
<td>JIT</td>
<td>just in time</td>
</tr>
<tr>
<td>LMS</td>
<td>lightweight modular staging</td>
</tr>
<tr>
<td>MDE</td>
<td>model-driven engineering</td>
</tr>
<tr>
<td>OO</td>
<td>object oriented/orientation</td>
</tr>
<tr>
<td>OOP</td>
<td>object-oriented programming</td>
</tr>
<tr>
<td>PEG</td>
<td>parsing expression grammar</td>
</tr>
<tr>
<td>RDF</td>
<td>resource description framework</td>
</tr>
<tr>
<td>SLR</td>
<td>software language repository</td>
</tr>
<tr>
<td>UML</td>
<td>unified modeling language</td>
</tr>
</tbody>
</table>