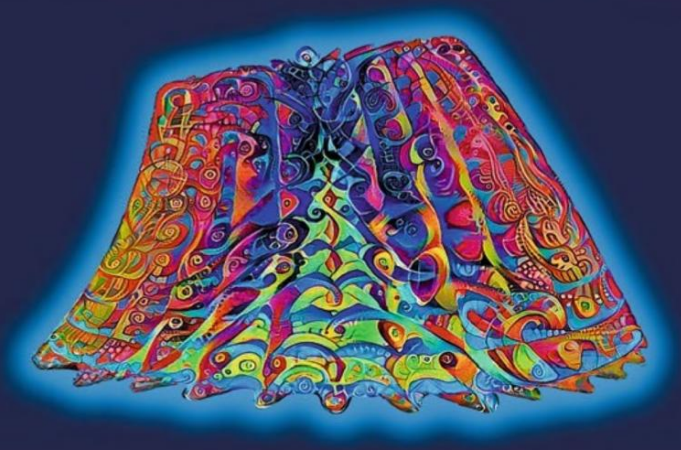

Ralf Lämmel

Software Languages

Syntax, Semantics, and Metaprogramming

Springer

Ralf Lämmel

Software Languages

Syntax, Semantics, and Metaprogramming

Ralf Lämmel
Computer Science Department
Universität Koblenz-Landau
Koblenz, Germany

ISBN 978-3-319-90798-7 ISBN 978-3-319-90800-7 (eBook)
https://doi.org/10.1007/978-3-319-90800-7

Library of Congress Control Number: 2018942228

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG part of Springer Nature.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

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.

Artwork Credits

Cover artwork:
Wojciech Kwasnik, *The Tower of Software Languages*, 2017.
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.

Contents

List of Recipes

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

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

Chapter 1
The Notion of a Software Language

JEAN-MARIE FAVRE.[1]

Abstract In this chapter, we characterize the notion of "software language" in a broad sense. We begin by setting out diverse examples of programming, modeling, and specification languages to cover a wide range of use cases of software languages in software engineering. Then, we classify software languages along multiple dimensions and describe the lifecycle of software languages, with phases such as language definition and implementation. Finally, we identify areas in software engineering that involve software languages in different ways, for example, software reverse engineering and software re-engineering.

[1] When the "Software Languages" community was formed around 2005–2007, Jean-Marie Favre was perhaps the key pillar and visionary and community engineer. His views and interests are captured very well in publications like these: [105, 104, 106, 100, 103].

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_1

1.1 Examples of Software Languages

In this book, we discuss diverse software languages; we may use them for illustrative purposes, and we may even define or implement them or some subsets thereof. For clarity, we would like to enumerate all these languages here in one place so that the reader will get an impression of the "language-related profile" of this book.

1.1.1 Real-World Software Languages

By "real-world language", we mean a language that exists independently of this book and is more or less well known. We begin with *programming languages* that will be used for illustrative code in this book. We order these languages loosely in terms of their significance in this book.

- *Haskell*[2]: The functional programming language Haskell
- *Java*[3]: The Java programming language
- *Python*[4]: The dynamic programming language Python

We will use some additional software languages in this book; these languages serve the purpose of specification, modeling, or data exchange rather than programming; we order these languages alphabetically.

- *ANTLR*[5]: The grammar notation of the ANTLR technology
- *JSON*[6]: The JavaScript Object Notation
- *JSON Schema*[7]: The JSON Schema language
- *XML*[8]: Extensible Markup Language
- *XSD*[9]: XML Schema Definition

Furthermore, we will refer to diverse software languages in different contexts, for example, for the purpose of language classification in Section 1.2; we order these languages alphabetically.

- *Alloy*[10]: The Alloy specification language
- *CIL*[11]: Bytecode of .NET's CLR

---

[2] Haskell language: https://www.haskell.org/

[3] Java language: https://en.wikipedia.org/wiki/Java_(programming_language)

[4] Python language: https://www.python.org/

[5] ANTLR language: http://www.antlr.org/

[6] JSON language: https://en.wikipedia.org/wiki/JSON

[7] JSON Schema language: http://json-schema.org/

[8] XML language: https://en.wikipedia.org/wiki/XML

[9] XSD language: https://en.wikipedia.org/wiki/XML_Schema_(W3C)

[10] Alloy language: http://alloy.mit.edu/alloy/

[11] CIL language: https://en.wikipedia.org/wiki/Common_Intermediate_Language

- *Common Log Format*[12]: The NCSA Common log format
  DocBook[13]: The DocBook semantic markup language for documentation
- *FOAF*[14]: The friend of a friend ontology
- *INI file*[15]: The INI file format
- *Java bytecode*[16]: Bytecode of the JVM
- *make*[17]: The make tool and its language
- *OWL*[18]: Web Ontology Language
- *Prolog*[19]: The logic programming language Prolog
- *QTFF*[20]: QuickTime File Format
- *RDF*[21]: Resource Description Framework
- *RDFS*[22]: RDF Schema
- *Scala*[23]: The functional OO programming language Scala
- *Smalltalk*[24]: The OO reflective programming language Smalltalk
- *SPARQL*[25]: SPARQL Protocol and RDF Query Language
- *UML*[26]: Unified Modeling Language
- *XPath*[27]: The XML path language for querying
- *XSLT*[28]: Extensible Stylesheet Language Transformations

1.1.2 Fabricated Software Languages

In this book, we "fabricated" a few software languages: these are small, idealized languages that have been specifically designed and implemented for the purposes of the book, although in fact these languages are actual or de facto subsets of real-world software languages. The language names are typically acronyms with expansions hinting at the nature of the languages. Language definitions of language-based

[12] Common Log Format language: https://en.wikipedia.org/wiki/Common_Log_Format
[13] DocBook language: https://en.wikipedia.org/wiki/DocBook
[14] FOAF language: http://semanticweb.org/wiki/FOAF.html
[15] INI file language: https://en.wikipedia.org/wiki/INI_file
[16] Java bytecode language: https://en.wikipedia.org/wiki/Java_bytecode
[17] make language: https://en.wikipedia.org/wiki/Make_(software)
[18] OWL language: https://en.wikipedia.org/wiki/Web_Ontology_Language
[19] Prolog language: https://en.wikipedia.org/wiki/Prolog
[20] QTFF language: https://en.wikipedia.org/wiki/QuickTime_File_Format
[21] RDF language: https://www.w3.org/RDF/
[22] RDFS language: https://www.w3.org/TR/rdf-schema/
[23] Scala language: https://en.wikipedia.org/wiki/Scala_(programming_language)
[24] Smalltalk language: https://en.wikipedia.org/wiki/Smalltalk
[25] SPARQL language: https://en.wikipedia.org/wiki/SPARQL
[26] UML language: https://en.wikipedia.org/wiki/Unified_Modeling_Language
[27] XPath language: https://en.wikipedia.org/wiki/XPath
[28] XSLT language: https://en.wikipedia.org/wiki/XSLT

software components are available for these languages from the book's repository.[29] The footnotes in the following list link to the repository locations for the languages.

- *BAL*: *B*asic *A*ssembly *L*anguage
- *BFPL*: *B*asic *F*unctional *P*rogramming *L*anguage
- *BGL*: *B*asic *G*rammar *L*anguage
- *BIPL*: *B*asic *I*mperative *P*rogramming *L*anguage
- *BML*: *B*inary *M*achine *L*anguage
- *BNL*: *B*inary *N*umber *L*anguage
- *BSL*: *B*asic *S*ignature *L*anguage
- *BTL*: *B*asic *T*APL *L*anguage
- *BL*: *B*uddy *L*anguage
- *EFPL*: *E*xtended *F*unctional *P*rogramming *L*anguage
- *EGL*: *E*xtended *G*rammar *L*anguage
- *EIPL*: *E*xtended *I*mperative *P*rogramming *L*anguage
- *EL*: *E*xpression *L*anguage
- *ESL*: *E*xtended *S*ignature *L*anguage
- *FSML*: *F*inite *S*tate *M*achine *L*anguage
- *MML*: *M*eta *M*odeling *L*anguage
- *TLL*: *T*yped *L*ambda *L*anguage
- *Text*: The "language" of text (such as Unicode 8.0 strings)
- *ULL*: *U*ntyped *L*ambda *L*anguage

In the rest of this section, we quickly introduce some of these languages, thereby providing a first indication of the diversity of language aspects covered by the book.

***B*inary *N*umber *L*anguage (BNL)** A trivial language of binary numbers with an intended semantics that maps binary to decimal values.

***B*asic *T*APL *L*anguage (BTL)** A trivial expression language in reference to the TAPL textbook (Types and programming languages [210]).

***B*uddy *L*anguage (BL)** A trivial language for modeling persons in terms of their names and buddy relationships.

***B*asic *F*unctional *P*rogramming *L*anguage (BFPL)** A really simple functional programming language which is an actual syntactic subset of the established programming language *Haskell*.

***B*asic *I*mperative *P*rogramming *L*anguage (BIPL)** A really simple imperative programming language which is a de-facto subset of the established programming language C.

***F*inite *S*tate *M*achine *L*anguage (FSML)** A really simple language for behavioral modeling which is variation on statecharts of the established modeling language UML.

***B*asic *G*rammar *L*anguage (BGL)** A specification language for concrete syntax, which can also be executed for the purpose of parsing; it is a variation on the established Backus-Naur form (BNF).

[29] http://github.com/softlang/yas

1.1.2.1 BNL: A Language of Binary Numbers

We introduce *BNL* (*B*inary *N*umber *L*anguage). This is a trivial language whose elements are essentially the binary numbers. Here are some binary numbers and their associated "interpretations" as decimal numbers:

- **0**: 0 as a decimal number;
- **1**: 1 as a decimal number;
- **10**: 2 as a decimal number;
- **11**: 3 as a decimal number;
- **100**: 4 as a decimal number;
- **101**: 5 as a decimal number;
- **101.01**: 5.25 as a decimal number.

Thus, the language contains integer and rational numbers – only positive ones, as it happens. BNL is a trivial language that is nevertheless sufficient to discuss the most basic aspects of software languages such as *syntax* and *semantics*. A syntax definition of BNL should define valid sequences of digits, possibly containing a period. A semantics definition of BNL could map binary to decimal numbers. We will discuss BNL's abstract syntax in Chapter 3 and the concrete syntax in Chapter 6.

1.1.2.2 BTL: An Expression Language

We introduce *BTL* (*B*asic *T*APL *L*anguage). This is a trivial language whose elements are essentially expressions over natural numbers and Boolean values. Here is a simple expression:

```
pred if iszero zero then succ succ zero else zero
```

The meaning of such expressions should be defined by expression evaluation. For instance, the expression form `iszero` *e* corresponds to a test of whether *e* evaluates to the natural number `zero`; evaluation of the form is thus assumed to return a Boolean value. The expression shown above evaluates to `zero` because `iszero zero` should compute to `true`, making the `if`-expression select the then-branch `succ succ zero`, the predecessor of which is `succ zero`.

An interpreter of BTL expressions should recursively evaluate BTL expression forms. BTL is a trivial language that is nevertheless sufficient to discuss basic aspects of interpretation (Chapter 5), semantics (Chapter 8), and type systems (Chapter 9).

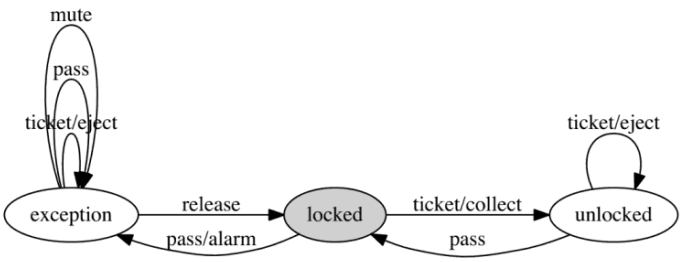

Fig. 1.2 A finite state machine for a turnstile.

The FSM also identifies possible transitions between states triggered by "events," possibly causing "actions"; see the edges in the visual notation.

These are these states in the turnstile FSM:

- *locked*: The turnstile is locked. No passenger is allowed to pass.
- *unlocked*: The turnstile is unlocked. A passenger may pass.
- *exception*: A problem has occurred and metro personnel need to intervene.

There are input symbols which correspond to the events that a user or the environment may trigger. There are output symbols which correspond to the actions that the state machine should perform upon a transition. These are some of the events and actions of the turnstile FSM:

- Event *ticket*: A passenger enters a ticket into the card reader.
- Event *pass*: A passenger passes through the turnstile, as noticed by a sensor.
- Action *collect*: The ticket is collected by the card reader.
- Action *alarm*: An alarm is turned on, thereby requesting metro personnel.

The meanings of the various transitions should be clear. Consider, for example, the transition from the source state "locked" to the target state "unlocked", which is annotated by "ticket/collect" to mean that the transition is triggered by entering a ticket and the transition causes ticket collection to happen.

FSML is a domain-specific modeling language (DSML). FSML supports *state-based modeling* of systems. The specification can be executed to simulate possible behaviors of a turnstile. The specification could also be used to generate a code skeleton for controlling an actual turnstile, as part of an actual metro system. FSML is a trivial language that can be used to discuss basic aspects of domain-specific language definition and implementation. For what it matters, languages for state-based behavior are widely established in software and systems engineering. For instance, the established modeling language UML consists, in fact, of several modeling languages; UML's state machine diagrams are more general than FSML. We will discuss FSML in detail in Chapter 2.

1.1.2.7 BGL: A Language for Context-Free Grammars

We introduce *BGL* (*B*asic *G*rammar *L*anguage). This language can be used to define the *concrete textual syntax* of other software languages. Thus, BGL gets us to the metalevel. Here is an illustration of BGL – a definition of the syntax of BNL – the language of binary numbers, as introduced earlier:

```
[number] number : bits rest ; // A binary number
[single] bits : bit ; // A single bit
[many] bits : bit bits ; // More than one bit
[zero] bit : '0' ; // The zero bit
[one] bit : '1' ; // The nonzero bit
[integer] rest : ; // An integer number
[rational] rest : '.' bits ; // A rational number
```

Each line is a grammar production (a rule) with the syntactic category (or the so-called nonterminal) to the left of ":" and its definition to the right of ":". For instance, the first production defines that a binary number consists of a bit sequence bits for the integer part followed by rest for the optional rational part. The right-hand phrases compose so-called terminals ("0", "1", and ".") and nonterminals (bit, bits, rest, and number) by juxtaposition. The rules are labeled, thereby giving a name to each construct.

BGL is a domain-specific modeling language in that it supports modeling (or specifying or defining) concrete textual syntax. One may "execute" BGL in different ways. Most obviously, one may execute a BGL grammar for the purpose of accepting or *parsing* input according to the syntax defined. BGL, like many other notations for syntax definition, is grounded in the fundamental formalism of *context-free grammars* (CFGs). BGL is a variation on BNF [21]. There exist many real-world notations for syntax definition [277]; they are usually more complex than BGL and may be tied to specific technology, for example, for parsing. We will develop BGL in detail in Chapter 6.

1.2 Classification of Software Languages

There are hundreds or even thousands of established software languages, depending on how we count them. It may be useful to group languages in an ontological manner. In particular, a *classification* of software languages (i.e., a language taxonomy) is a useful (if not necessary) pillar of a definition of "software language".

Wikipedia, which actually uses the term "computer language" at the root of the classification, identifies the following top-level classifiers:[30]

- data-modeling languages;
- markup languages;
- programming languages;
- specification languages;
- stylesheet languages;
- transformation languages.

Any such branch can be classified further in terms of constructs and concepts. For instance, in the case of programming languages, there exist textbooks on programming languages, programming paradigms, and programming language theory such as [199, 232], which identify constructs and concepts. There is also scholarly work on the classification of programming languages [20, 90] and the identification of language concepts and corresponding paradigms [258].

Several classes of software languages (other than programming languages) have been identified, for example, *model transformation languages* [75], *business rule modeling languages* [239], *visual languages* [46, 49, 190], and *architecture description languages* [192]. There is more recent work aimed at the classification of software languages (or computer languages) more broadly [13, 237, 3, 171]. The *101companies* project[31] [102, 101, 173, 166] is also aimed at a taxonomy of software languages, but the results are of limited use, at the time of writing.

In the remainder of this section, we classify software languages along different dimensions. A key insight here is that a single classification tree is insufficient. Multiple inheritance may be needed, or orthogonal dimensions may need to be considered separately.

1.2.1 Classification by Paradigm

When focusing on programming languages as a category of software languages, classification may be based on the *programming paradigm*. A paradigm is characterized by a central notion of programming or computing. Here is an incomplete, classical list of paradigms:

Imperative programming Assignable (updatable) variables and updatable (in-place) data structures and sequential execution of statements of operations on variables. Typically, procedural abstractions capture statements that describe control flow with basic statements for updates. We exercise imperative programming with the fabricated language BIPL (Section 1.1.2.5) in this book.

[30] We show Wikipedia categories based on a particular data-cleaning effort [171]. This is just a snapshot, as Wikipedia is obviously evolving continuously.

[31] http://101companies.org

Functional programming The application of functions models computation with compound expressions to be reduced to values. Functions are first-class citizens in that functions may receive and return functions; these are higher-order functions. We exercise functional programming with the fabricated language BFPL (Section 1.1.2.4) in this book – however, though higher-order functions are not supported.

Object-oriented (OO) programming An object is a capsule of state and behavior. Objects can communicate with each other by sending messages, the same message being implementable differently by different kinds of objects. Objects also engage in structural relationships, i.e., they can participate in whole–part and reference relationships. Objects may be constructed by instantiation of a given template (e.g., a class). *Java* and C# are well-known OO programming languages.

Logic programming A program is represented as a collection of logic formulae. Program execution corresponds to some kind of proof derivation. For instance, *Prolog* is a well-known logic programming language; computation is based on depth-first, left-to-right proof search through the application of definite clauses.

There exist yet other programming or computing notions that may characterize a paradigm, for example, message passing and concurrency. Many programming languages are, in fact, *multi-paradigm* languages in that they support several paradigms. For instance, *JavaScript* is typically said to be both a functional and an imperative OO programming language and a scripting language. Programming languages may be able to support programming according to a paradigm on the basis of some encoding scheme without being considered a member of that paradigm. For instance, in Java prior to version 8, it was possible to encode functional programs in Java, while proper support was added only in version 8.

Van Roy offers a rich discussion of programming paradigms [258]. Programming concepts are the basic primitive elements used to construct programming paradigms. Often, two paradigms that seem quite different (for example, functional programming and object-oriented programming) differ by just one concept. The following are the concepts discussed by Van Roy: record, procedure, closure, continuation, thread, single assignment, (different forms of) cell (state), name (unforgeable constant), unification, search, solver, log, nondeterministic choice, (different forms of) synchronization, port (channel), clocked computation. Van Roy identifies 27 paradigms, which are characterized as sets of programming concepts. These paradigms can be clearly related in terms of the concepts that have to be added to go from one paradigm to another.

1.2.2 Classification by Type System

Furthermore, languages may also be classified in terms of their typing discipline or type system [210] (or the lack thereof). Here are some important options for programming languages in particular:

Static typing The types of variables and other abstractions (e.g., the argument and result types of methods or functions) are statically known, i.e., without executing the program – this is at compile time for compiled languages. For instance, Haskell and Java are statically typed languages.

Dynamic typing The types of variables and other abstractions are determined at runtime. A variable's type is the type of the value that is stored in that variable. A method or function's type is the one that is implied by a particular method invocation or function application. For instance, Python is a dynamically typed language.

Duck typing The suitability of a variable (e.g., an object variable in object-oriented programming) is determined at runtime on the basis of checking for the presence of certain methods or properties. Python uses duck typing.

Structural typing The equivalence or subtyping relationship between types in a static typing setting is determined on the basis of type structure, such as the components of record types. Scala supports some form of structural typing.

Nominal typing The equivalence or subtyping relationship between types in a static typing setting is determined on the basis of explicit type names and declared relationships between them. Java's reference types (classes and interfaces including "extends" and "implements" relationships) commit to nominal typing.

1.2.3 Classification by Purpose

Languages may be classified on the basis of the *purpose* of the language (its usage) or its elements. Admittedly, the term "purpose" may be somewhat vague, but the illustrative classifiers in Table 1.1 may convey our intuition. We offer two views: the purpose of the language versus that of its elements; these two views are very similar.

Table 1.1 Classification by the purpose of language elements

Purpose (language)	**Purpose** (element)	**Classifier**	**Example**
Programming	Program	Programming language	Java
Querying	Query	Query language	XPath
Transformation	Transformation	Transformation language	XSLT
Modeling	Model	Modeling language	UML
Specification	Specification	Specification language	Alloy
Data representation	Data	Data format	QTFF (QuickTime file format)
Documentation	Documentation	Documentation language	DocBook
Configuration	Configuration	Configuration language	INI file
Logging	Log	Log format	Common Log Format
...	...	...	...

tion of Section 1.1.2.7. Several of the illustrative languages of Section 1.1 were introduced as string languages.

Tree language (See also "markup language" below.) Language elements are represented, viewed, and edited as trees, for example, as XML trees or JSON dictionaries. A tree language is defined in terms of a suitable grammar or data modeling notation, for example, XSD in the case of XML. As it happens, we did not present any tree languages in Section 1.1.2.7, but we will discuss tree-based abstract syntax definitions later for some of the string languages that we have already seen. Tree languages play an important role in language implementation.

Graph language Language elements are represented, viewed, and edited as graphs, i.e., more or less constrained collections of nodes and edges. Appropriate grammar and data modeling notations exist for this case as well. The language BL for buddy relationships (Section 1.1.2.3) was introduced as a graph language and we hinted at a visual concrete syntax. A graph language may be coupled with a string or tree language in the sense of alternative representations of the same "conceptual" language. For instance, BL may be represented in a string-, tree-, or graph-based manner.

1.2.6 Classification by Notation

One may also distinguish languages in terms of notation; this classification is very similar to the classification by representation:

Textual (text) language This is essentially a synonym for "string language".

Markup language Markup, as in XML, is used as the main principle for expressing language elements. The use of markup is one popular notation for tree languages. With an appropriate semantics of identities, markup can also be used as a notation for graphs. Not every tree language relies on markup for the notation. For instance, JSON provides another, more dictionary-oriented notation for tree languages.

Visual (graphical) language A visual notation is used. The languages BL for buddy relationships (Section 1.1.2.3) and FSML for state-based modeling (Section 1.1.2.6) were introduced in terms of a visual notation.

1.2.7 Classification by Degree of Declarativeness

An (executable) language may be said to be (more or less) *declarative*. It turns out to be hard to identify a consensual definition of declarativeness, but this style of classification is nevertheless common. For instance, one may say that programs (or models) of a declarative language describe more the "what" than the "how". That is, a declarative program's semantics is not strongly tied to execution order.

Let us review the languages of Section 1.1:

Binary Number Language (BNL) A trivial language.

Buddy Language (BL) A trivial language.

Basic Functional Programming Language (BFPL) This language is "pure", i.e., free of side effects. Regardless of the evaluation order of subexpressions, complete evaluation of a main expression should lead to the same result – modulo some constraints to preserve termination. For instance, argument expressions of a function application could be evaluated in different orders without affecting the result. Thus, BFPL is a declarative programming language.

Basic Imperative Programming Language (BIPL) This language features imperative variables such that the execution order of statements affects the result of computation. Thus, BIPL is not a declarative programming language.

Finite State Machine Language (FSML) This language models finite states and event- and action-labeled transitions between states. Its actual semantics or execution order is driven by an event sequence. FSML would usually be regarded as a declarative (modeling) language.

Basic Grammar Language (BGL) This grammar notation defines sets of strings in a rule-based manner. Thus, BGL's most fundamental semantics is "declarative" in the sense that it is purely mathematical, without reference to any operational details. Eventually, we may attach a more or less constrained operational interpretation to BGL so that we can use it for efficient, deterministic parsing. Until that point, BGL would be usually regarded as a declarative (specification) language.

We may also consider subcategories of declarative languages such that it is emphasized how declarativeness is achieved. Two examples may suffice:

Rule-based language Programs are composed from rules, where a rule typically combines a condition and an action part. The condition part states when the rule is applicable; the action part states the implications of rule application. Some logic programming languages, for example, Prolog, can be very well considered to be rule-based languages. Some schemes for using functional programming, for example, in interpretation or program transformation, also adopt the rule-based approach. Event-driven approaches may also use rules with an additional "event" component, for example, the "event condition action" (ECA) paradigm, as used in active databases [88].

Constraint-based language Programs involve constraints as means of selecting or combining computations. These constraints are aggregated during program execution and constraint resolution is leveraged to establish whether and how given constraints can be solved. For instance, there exist various constraint-logic programming languages which enrich basic logic programming with constraints on sets of algebras for numbers [117].

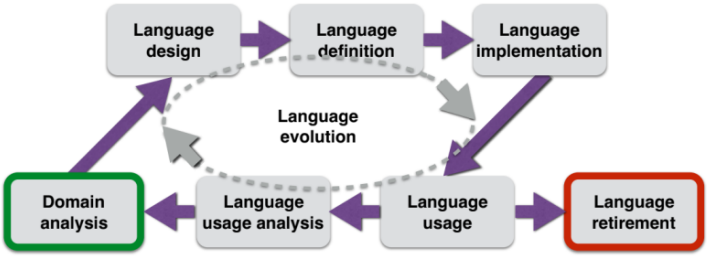

Fig. 1.3 The lifecycle of a software language. The nodes denote phases of the lifecycle. The edges denote transitions between these phases. The lifecycle starts with *domain analysis*. The lifecycle ends (theoretically) with *language retirement*. We may enter cycles owing to language evolution.

Exercise 1.2 (Classification of *make*) [Basic level]
Study the language used by the well-known make *utility and argue that the language is declarative and identify what subcategory of declarativeness applies.*

1.3 The Lifecycle of Software Languages

The notion of a software lifecycle can be usefully adopted for languages. That is, a language goes through a lifecycle, possibly iterating or skipping some phases; see Fig. 1.3. These phases are described in some detail as follows:

Domain analysis A domain analysis is required to discover the domain that is to be addressed by a new language. A domain analysis answers these questions: What are the central concepts in the domain? For instance, the central concepts are states and transitions between states for FSML, differential equations and their solution in a language for weather forecasts, or layouts and their rendering in a language for HTML/XML stylesheets. These concepts form the foundation of language design and for everything onwards. Arguably, no domain analysis is performed for general-purpose programming languages.

Language design The domain concepts are mapped, at the design level of abstraction, to language constructs and language concepts. The emerging language should be classified in terms of paradigm, degree of declarativeness, and other characteristics. A language may be presented as a composition of very specific language constructs as well as reusable language constructs, for example, for basic expressions or modules. The first samples are written so that a syntax emerges, and the transition to the phase of language definition has then begun.

Language definition The language design is refined into a language definition. Most notably, the syntax and semantics of the language are defined. Assuming

an executable language definition, a first language implementation (a proof of concept) is made available for experiments, so that the transition to the phases of language implementation and usage has begun.

Language implementation The language is properly implemented. Initially, a usable and efficient compiler or interpreter needs to provided. Eventually, additional *language processors* and tool support may be provided, for example, documentation tools, formatters, style checkers, and refactorings. Furthermore, support for an integrated development environment (IDE) may be implemented.

Language usage The language is used in actual software development. That is, language artifacts are "routinely" authored and a body of software artifacts acquire dependencies on the language. This is not explicitly modeled in Fig. 1.3, but the assumption is, of course, that the language implementation is continuously improved and new language processors are made available.

Language evolution Language definitions may revised to incorporate new language features or respond to experience with language usage. Obviously, changes to language definitions imply work on language implementations. Language changes may even break backward compatibility, in which cases these changes will necessitate migration of existing code in those languages.

Language usage analysis Language evolution and the systematic improvement of domain analysis as well as language design, definition, and implementation, may benefit from language usage analysis [155, 100, 172], as an empirical element of the lifecycle. By going through the lifecycle in cycles, the language may evolve in different ways. For instance, the language may be extended so that a new version becomes available, which again needs to be implemented and put to use.

Language retirement In practice, languages, once adopted, are rarely retired completely, because the costs and risks of retirement are severe impediments. Retirement may still happen in the narrow scope of projects or organizations. In theory, a language may become obsolete, i.e., there are no software artifacts left that depend on that language. Otherwise, *language migration* may be considered. That is, software artifacts that depend on a language are migrated (i.e., transformed manually or automatically) to another language.

Many aspects of these phases, with some particular emphasis on the lifecycle of DSLs are discussed in [133, 272, 273, 197, 214, 56, 98, 94, 78, 265, 229]. In the present book, the focus is on language definition and implementation; we are concerned only superficially with domain analysis, language design, evolution, and retirement.

1.3.1 Language Definition

Let us have a deeper look at the lifecycle phase of language definition. A language is defined to facilitate implementation and use of the language. There are these aspects of language definition:

Syntax The definition of the syntax consists of rules that describe the valid language elements which may be drawn from different "universes": the set of all strings (say, text), the set of all trees (of some form, e.g., XML-like trees), or the set of all graphs (of some form). Different kinds of formalisms may be used to specify the rules defining the syntax. We may distinguish *concrete* and *abstract syntax* – the former is tailored towards users who need to read and write language elements, and the latter is tailored towards language implementation. Abstract syntax is discussed in Chapters 3and 4. Concrete syntax is discussed in Chapters 6and 7.

Semantics The definition of semantics provides a mapping from the syntactic categories of a language (such as statements and expressions) to suitable domains of meanings. The actual mapping can be defined in different ways. For instance, the mapping can be defined as a set of syntax-driven inference rules which model the stepwise execution or reduction of a program; this is known as small-step operational semantics (Chapter 8). The mapping can also be applied by a translation, for example, by a model-to-model transformation in model-driven engineering (MDE).

Pragmatics The definition of the pragmatics explains the purpose of language concepts and provides recommendations for their usage. Language pragmatics is often defined only informally through text and samples. For instance, the pragmatics definition for a C-like language with arrays may state that arrays should be used for efficient (constant-time) access to indices in ordered collections of values of the same type. Also, arrays should be favored over (random-access) files or databases for as long as in-memory representation of the entire data structure is reasonable. In modeling languages for finite state machine (e.g., FSML), events proxy for sensors and actions proxy for actors in an embedded system.

Types Some languages also feature a *type system* as a part of the language definition. A type system provides a set of rules for assigning or verifying *types*, i.e., properties of language phrases, for example, different expression types such as "int" or "string" in a program with expressions. We speak of *type checking* if the type system is used to check explicitly declared types. We speak of *type inference* if the type system is used additionally to infer missing type declarations. A type system needs to be able to bind names in the sense that any use of an abstraction such as a variable, a method, or a function is linked to the corresponding declaration. Such *name binding* may defined as part of the type system or they may be defined somewhat separately. We discuss types in detail in Chapter 9. Even when a language does not have an interesting type system, i.e., different types and rules about their use in abstractions, the language may still feature other constraints regarding, for example, the correct use of names. Thus, we may also speak of *well-formedness* more generally, as opposed to *well-typedness* more specifically. For instance, in FSML, the events handled by a given source state must be distinct for the sake of determinism.

When definitions of syntax, types, and semantics are considered formal artifacts such that these artifacts are treated in a formal (mathematical) manner, then we operate within the context of *programming language theory*. A formal approach

pass compilation") or may be integrated into one phase ("single-pass compilation"). The components are explained more in detail as follows:

Parser A parser verifies the conformance of given input (i.e., text) to the syntax rules of a language and represents the input in terms of the structure defined by the rules. A parser performs parsing. Compilers and interpreters begin by parsing. Many other language processors, as discussed below, also involve parsing.

Semantic analysis A syntax tree only represents the structure of the source code. For any sort of nontrivial treatment such as code generation, the syntax tree needs to be enriched with attributes and links related to typing and name binding. Names with their bindings and other attributes may be aggregated in a data structure which is referred to as a symbol table or environment.

Code generator The enriched syntax tree is translated, more or less directly, into machine code, i.e., code of some actual or virtual machine. In particular, code generation involves resource and storage decisions such as register allocation, i.e., assigning program variables to processor registers of the target machine. In this book, few technicalities of code generation are discussed; this topic is covered perfectly by the literature on compiler construction.

Ideally, the components are described by specifications such as grammars, type systems, name-binding rules, and rewrite systems, as indicated in Fig. 1.4. In practice, the components are often implemented in a more ad hoc fashion.

This is a simplified data flow, because actual compilers may involve additional phases. That is, parsing may consist of several phases in itself: preprocessing; lexical analysis (scanning, lexing, or tokenization); syntax analysis including parse-tree construction and syntax desugaring. Also, there may be extra steps preceding code generation: translation to a (simpler) *intermediate representation* (IR) and IR-level optimization. Further, code generation may also involve *optimization* at the level of the target language and a separation between translation to assembly code, mapping to machine code, and some elements of linking. Finally, code generation may actually rely on translation such that the given input language is translated into a well-defined subset of an existing (programming) language so that an available compiler can be used afterwards.

Exercise 1.3 (An exercise on language implementation) [Basic level]
Research the current version of the JDK (Java Development Kit) and identify and characterize at least two language implementations that are part of it.

1.3.2.3 Classification of Language Processors

Languages are implemented in many ways other than just regular compilers and interpreters. We use the term "language processor" to refer to any sort of functionality for automated processing of software artifacts in a language-aware manner,

i.e., with more or less awareness of the syntax, types, and semantics of the artifacts. Examples of language processors include documentation generators, refactoring tools, bug checkers, and metrics calculation tools. Language processors often consist of several components and perform processing in phases, as we discussed above for compilers. Rather than classifying language processors directly, let us classify language-based software components. We do not make any claim of completeness for this classification. Several of the classifiers below will reappear in the discussion of the role of software languages across different software engineering areas (Section 1.4):

Parser or text-to-model transformation The term "parser" has already been introduced in the context of compilation and interpretation. The term "text-to-model transformation" is specifically used in the MDE community when one wants to emphasize that the result of parsing is not a parse *tree*, but rather a model in the sense of metamodeling, thus potentially involving, for example, references after completing name binding.

Unparser, formatter, pretty printer, or model-to-text transformation An artifact is formatted as text, possibly also subject to formatting conventions for the use of spaces and line breaks. Formatting may start from source code (i.e., text), concrete syntax trees (i.e., parse trees), or abstract syntax trees. Formatting is typically provided as a service in an IDE.

Preprocessor As part of parsing, code may be subject to macro expansion and conditional compilation. Such preprocessing may serve the purpose of, for example, configuration management in the sense of software variability and desugaring in the sense of language extension by macros. Interestingly, preprocessing gives rise to a language of its own for the preprocessing syntax such that the preprocessor can be seen as an interpreter of that language; the result type of this sort of interpretation is, of course, text [99]. One may also assume that a base language is extended by preprocessing constructs so that preprocessing can be modeled as a translation from the extended to the base language. In fact, some macro system work in that manner. In practice, preprocessing is often used in an undisciplined (i.e., not completely syntax-aware) manner [29, 18, 184].

Software transformation or model-to-model transformation A software transformation is a mapping between software languages. The term "model-to-model transformation" is used in the model transformation and MDE community. We may classify transformations in terms of whether the source and target languages are the same and whether the source and target reside at the same level of abstraction [195]. Thus:

- **Exogenous transformation** The source and target languages are different, as in the case of code generation (translation) or language migration.
- **Endogenous transformation** The source and target languages are the same, as in the case of program refactoring or compiler optimization [116, 196]. We can further distinguish in-place and out-place transformations [195, 35] in terms of whether the source model is "reused" to produce the target model. (Exogenous transformations are necessarily out-place transformations.)

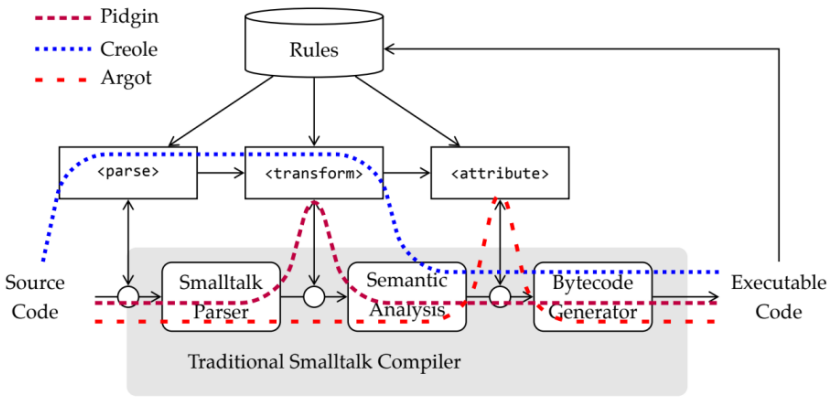

Fig. 1.5 The code compilation pipeline of Helvetia, showing multiple interception paths; there are hooks to intercept parsing <parse>, AST transformation <transform>, and semantic analysis <attribute>. Source: [215]. Additional capabilities of Helvetia support editing (coloring), debugging, etc. © 2010 Springer.

and PLT Redex [107]), *compiler frameworks* (e.g., LLVM [180]), and *modeling frameworks* (e.g., AM3 [24]).

Metaprogramming and software language engineering efforts may be "advertised" through *software language repositories* (SLRs) [165], i.e., repositories with components for language processing (interpreters, translators, analyzers, transformers, pretty printers, etc.). Further examples of SLRs include the repositories for Krishnamurthi's textbook on programming languages [160], Batory's Prolog-based work on teaching MDE [28], Zaytsev et al.'s software language processing suite (SLPS) [278], and Basciani et al.'s extensible web-based modeling platform MDEForge [26].

1.3.2.5 Language Workbenches

Metaprogrammers may also be supported in an interactive and integrated fashion. Accordingly, the notion of *language workbenches* [96, 97, 144, 143, 267, 266, 269, 263] encompasses enhanced metaprogramming systems that are, in fact, IDEs for language implementation. A language workbench assumes specialized language definitions that cater for IDE services such as syntax-directed, structural, or projectional editing, coloring, synthesis of warnings and errors, package exploration, quick fixes, and refactorings.

Figure 1.5 illustrates the compilation pipeline of the metaprogramming system Helvetia [214, 215]. In fact, Helvetia is an extensible development environment for embedding DSLs into a host language (Smalltalk) and its tools such as the editor and debugger. Thus, Helvetia is a language workbench.

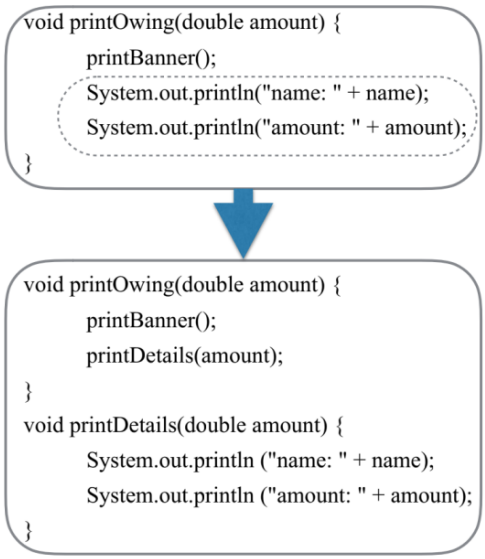

Fig. 1.6 Illustration of the "extract method" refactoring.

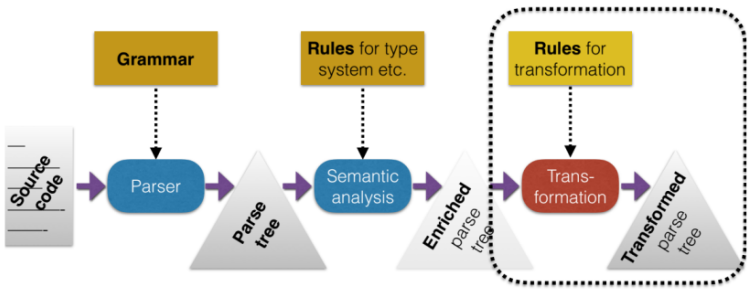

Fig. 1.7 Overall data flow for a re-engineering transformation. We have marked the phase which replaces code generation in the standard data flow for compilation.

228]. Even in the simple example at hand, some constraints have to be met; for example, the extracted statements must not return.

Figure 1.7 shows the overall data flow for a re-engineering transformation as needed, for example, for refactoring or restructuring. This data flow should be compared with the data flow for compilation; see Fig. 1.4. The two data flows share the phases of parsing and semantic analysis. The actual transformation is described (ideally) by declarative rules of a transformation language. Not every re-engineering use case requires a full-blown semantic analysis, which is why we have grayed out slightly the corresponding phase in Fig. 1.7. In fact, not even a proper syntax-aware transformation is needed in all cases, but instead a lexical approach may be applicable [152].

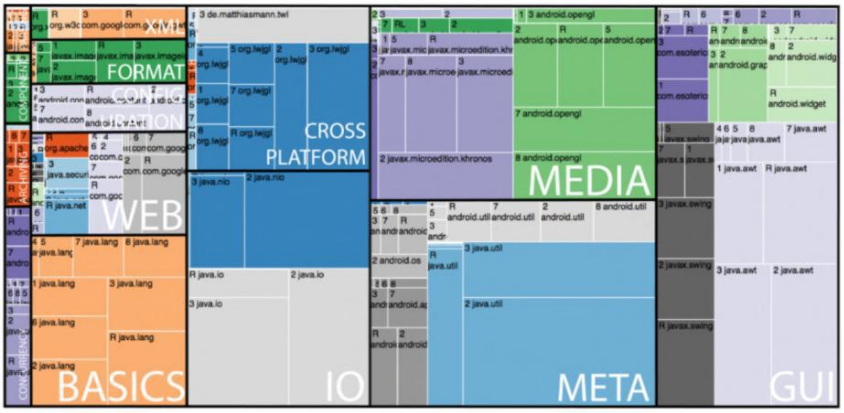

Fig. 1.8 An API-usage map for an open-source Java project. The complete rectangle (in terms of its size) models the references to all APIs made by all developers. The nested rectangles partition references by domain (e.g., GUI rather than Swing or AWT). The rectangles nested further partition references by API; one color is used per API. Within each such rectangle, the contributions of distinct developers (1, . . . , 8 for the top-eight committers and "R" for the rest) are shown. Source: [4].

1.4.2 Software Reverse Engineering

We quote: "reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [59]. For instance, we may extract a call graph from a system, thereby identifying call sites (such as packages, files, classes, methods, or functions) and actual calls (such as method or function calls). Reverse engineering may also be concerned with *architecture recovery* [126, 128, 158, 33], for example, the identification of components in a legacy system. Overall, reverse engineering is usually meant to help with program comprehension and to prepare for software re-engineering or to otherwise facilitate software development.

Figure 1.8 shows the visual result of a concrete reverse engineering effort aimed at understanding API usage in Java projects [4]. The tree map groups API references (i.e., source code-level references to API methods) so that we can assess the contributions of different APIs and of individual developers for each API to the project.

Figure 1.9 shows the overall data flow for a reverse engineering component that is based on the paradigm of fact extraction [109, 201, 185, 27]. Just as in the cases of compilation or transformation for re-engineering, we begin with parsing and (possibly customized) semantic analysis. The data flow differs in terms of last phase for fact extraction. The extracted facts can be thought of as sets of tuples, for example, pairs of caller/callee sites to be visualized eventually as a call graph.

Reverse engineering often starts from some sort of fact extraction. Reverse engineering may also involve data analysis based, for example, on relational alge-

Figure 1.10 gives an example of how metrics and simple visualization can be combined to analyze a software process – in this case, a process for the improvement of a grammar [7]. The changes of the values of the metrics can be explained as consequences of the specific grammar revisions applied at the corresponding commit points.

1.4.4 Technological Spaces

We quote: "A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference regular meetings" [161].

For instance, there are the following technological spaces, which we characterize in a keyword style by pointing out associated languages, technologies, and concepts:

Grammarware string, grammar, parsing, CST, AST, term, rewriting, ...
XMLware XML, XML infoset, DOM, DTD, XML Schema, XPath, XQuery, XSLT, ...
JSONware JSON, JSON Schema, ...
Modelware UML, MOF, EMF, class diagram, modeling, metamodeling, model transformation, MDE, ...
SQLware table, SQL, relational model, relational algebra, ...
RDFware resource, triple, Linked Data, WWW, RDF, RDFS, OWL, SPARQL, ...
Objectware objects, object graphs, object models, state, behavior, ...
Javaware Java, Java bytecode, JVM, Eclipse, JUnit, ...

We refer to [40] for a rather detailed discussion of one technological space – modelware (MDE). We refer to [89] for a discussion of multiple technological spaces with focus on Modelware and RDFware centric and cursory coverage of grammarware, Javaware, and XMLware and the interconnections between these spaces.

Technological spaces are deeply concerned with software languages:

Data models The data in a space conforms to some data model, which can be viewed as a "semantic domain" in the sense of semantics in the context of language definition. For instance, the data model of XML is defined by a certain set of trees, according to the XML infoset [274]; the data model JSON is a dictionary format that is a simple subset of Javascript objects; and the data model of SQLware is the relational model [67].

Schema languages Domain- or application-specific data can be defined by appropriate schema-like languages. Schemas are to tree- or graph-based data what (context-free) grammars are to string languages [149]. For instance, the schema language of JSON is JSON Schema [208]; the schema language of grammarware is EBNF [137] in many notational variations [277]; and the schema languages of

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-