Ralf Lämmel

Software Languages

Syntax, Semantics, and Metaprogramming

Chapter 1
The Notion of a Software Language

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.

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].

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork DMT, acryl, 2008 by Matt Sheehy with the artist’s permission. This work also quotes https://commons.wikimedia.org/wiki/File:Vincent_van_Gogh_-_Zeegezicht_bij_Les_Saintes_Maries_de_la_Mer_-_Google_Art_Project.jpg subject to the attribution “Vincent van Gogh: Seascape near Les Saintes-Maries-de-la-Mer (1888) [Public domain], via Wikimedia Commons.” This work artistically morphs an image, https://www.flickr.com/photos/eelcovisser/4772847104 showing the person honored, subject to the attribution “Permission granted by Eelco Visser for use in this book.”
Chapter 2
A Story of a Domain-Specific Language

Abstract In this chapter, several fundamental concepts and engineering techniques for software languages are explained by means of an illustrative domain-specific language. In particular, we exercise the internal and external styles of DSL implementation, textual and visual syntax, parsing, interpretation, and code generation. As a running example, we deal with a DSL for finite state machines FSML (FSM Language). In addition to implementing FSML with mainstream languages and technologies, we discuss design and implementation options and concerns overall and we describe a number of “recipes” for DSL development.

1 There is no “Greek” in Martin Fowler’s textbooks on refactoring [4] and DSLs [5], both addressing important topics in software language engineering. These accessible textbooks triggered research on these topics and connected research better with “mainstream” software development. Martin Fowler was again visionary when he asked in 2005 “Language Workbenches: The Killer-App for Domain Specific Languages?” (https://www.martinfowler.com/articles/languageWorkbench.html), thereby fueling the development of and research on language workbenches [2, 3, 16, 15, 17, 13].

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork DMT, acrylic, 2006 by Matt Sheehy with the artist’s permission. This work also quotes https://en.wikipedia.org/wiki/Wheat_Field_with_Cypresses#/media/File:Wheat-Field-with-Cypresses—(1889)—Vincent-van-Gogh-Met.jpg, subject to the attribution Vincent van Gogh: Wheat Field with Cypresses (1889) [Public domain], via Wikipedia. This work artistically morphs an image https://en.wikipedia.org/wiki/Martin_Fowler showing the person honored, subject to the attribution By Welbyshery Nunes - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=39594469.
Chapter 3
Foundations of Tree- and Graph-Based Abstract Syntax

Abstract A software language can be regarded as a set of structured elements with some associated meaning. A language’s syntax defines its elements and their structure. We may speak of string, tree, and graph languages — to convey the nature of the elements’ structure. One may distinguish two forms of syntax: concrete versus abstract syntax. The former is tailored towards processing (reading, writing, editing) by humans who are language users; the latter is tailored towards processing (parsing, analyzing, transforming, generating) by programs that are authored by language implementers. In this chapter, we cover the foundations of abstract syntax. This includes the notion of conformance of terms (trees) or models (graphs) to signatures or metamodels. The proposed notations for signatures and metamodels correspond to proper software languages in themselves, giving rise to a metametalevel that we develop as well. We defer implementation aspects of abstract syntax, coverage of concrete syntax, and semantics of languages to later chapters.

1 The software language engineering community aims to integrate more specialized communities. Richard Paige is a modelware “stronghold”; he has contributed to pretty much everything modelware, for example, model merging and composition [6][1], model evolution [2], model to text and vice versa [10][3], and visual syntax [7]. Richard Paige definitely advances community integration in his work, as exemplified by his tutorial on metamodeling for grammar researchers [8] or his Twitter persona.

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork DMT, acrylic, 2006 by Matt Sheehy with the artist’s permission. This artwork also quotes https://commons.wikimedia.org/wiki/File:Irises-Vincent_van_Gogh.jpg subject to the attribution “Vincent van Gogh: Irises (1889) [Public domain], via Wikimedia Commons.” This work artistically morphes an image https://www.cs.york.ac.uk/people/paige showing the person honored, subject to the attribution “Permission granted by Richard Paige for use in this book.”
Chapter 4
Representation of Object Programs in Metaprograms

Abstract This chapter discusses different representation options for abstract syntax in the context of implementing programming languages or language-based software components. This is an important foundation for metaprogramming. That is, we assume that one language – the metalanguage – is used for writing programs that analyze, manipulate, translate, generate, or otherwise consume or produce programs in another language – the object language. In this context, abstract syntax thus plays the role of defining the object-program representation in metaprograms. This chapter also discusses other implementation aspects of abstract syntax: conformance checking, serialization, and resolution (AST-to-ASG mapping).

1 Technological spaces are not dictated by natural laws, but once they are observed, named, and promoted, they add structure to the computer science landscape. Jean Bézivin has been prominent in observing, characterizing, and promoting the move from objects and components to models [3]. His projects have been aimed at practical and relevant languages and tools, for example, ATL [13]. He has mediated between academia and practice (such as OMG) in the field of model-driven engineering/architecture (MDE/MDA) [3]. He has helped to give birth to the very notion of technological space, explained the MDE instance [13,4], integrated it into the broader software language engineering community, and pushed MDE to a more macroscopic level [7,6,2].

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork DMT, acrylic, 2006 by Matt Sheehy with the artist’s permission. This work also quotes https://commons.wikimedia.org/wiki/File:Vincent_van_Gogh_-_Almond_Blossom-\_\_-_VGA_F671.jpg, subject to the attribution “Vincent van Gogh: Almond Blossom (1890) [Public domain], via Wikimedia Commons.” This work artistically morphs an image, https://plus.google.com/+JeanBezivin, showing the person honored, subject to the attribution “Permission granted by Jean Bézivin for use in this book.”
Chapter 5
A Suite of Metaprogramming Scenarios

Abstract This chapter is a basic introduction to metaprogramming. A metaprogram is a program that processes (i.e., takes as input or produces as output) programs. Metaprogramming is at the heart of software language implementation and processing. The processed programs or artifacts are also referred to as object programs. The language in which the metaprograms are written is referred to as the metalanguage. The language of the processed programs or artifacts is referred to as the object language. The following are all important scenarios of metaprogramming: interpretation, compilation, transformation, analysis, and code generation. In this chapter, we exercise several metaprogramming scenarios using Haskell as the metalanguage.

1 At its heart, this book focuses on metaprogramming in the sense of source-code analysis and manipulation (as opposed to run-time reflection or adaptive systems). James Cordy may be regarded as a representative of the discipline – he has developed languages and systems for metaprogramming (notably TXL [12]), and he has carried out or overseen major industrial projects, important case studies, or surveys in many application areas of metaprogramming [13, 45]. James Cordy started his career with influential work on language design and compiler technology, focused eventually on legacy systems [14], and is nowadays an authority on program comprehension, software transformation, code analysis (e.g., clone detection), and various other areas in empirical and automated software engineering.

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork DMT, acrylic, 2006 by Matt Sheehy with the artist’s permission. This work also quotes https://commons.wikimedia.org/wiki/File:Sunset_at_Montmajour_1888_Van_Gogh.jpg subject to the attribution “Vincent Van Gogh: Sunset at Montmagnour (1888) [Public domain], via Wikimedia Commons.” This work artistically morphs an image, https://en.wikipedia.org/wiki/James_Cordy showing the person honored, subject to the attribution “By Cordyj (talk) (Uploads) - Own work, CC BY-SA 3.0, https://en.wikipedia.org/w/index.php?curid=31348050.”
Chapter 6
Foundations of Textual Concrete Syntax

Abstract In this chapter, we consider the notion of concrete syntax of software languages thereby complementing the earlier discussion of abstract syntax (Chapters 3 and 4). Concrete syntax is tailored towards processing (reading, writing, editing) by humans who are language users, while abstract syntax is tailored towards processing by programs that are authored by language implementers. In this chapter, we focus on the concrete syntax of string languages as defined by context-free grammars (CFGs). In fact, we cover only textual concrete syntax; we do not cover visual concrete syntax. We introduce the algorithmic notion of acceptance for a membership test for a language. We also introduce the algorithmic notion of parsing for recovering the grammar-based structure of input. We defer the implementation aspects of concrete syntax, including actual parsing approaches, to the next chapter.

1 There is clearly nothing wrong with the notion of a Turing machine – after all it is Turing-complete, but the way it is described and discussed is clearly very reminiscent of how we think of actual (early) computing machines working operationally, if not mechanically. Personally, I have always felt more attracted to the lambda calculus, with its high level of abstraction, much more focused on computation than on operation. Likewise, I admire the Chomsky hierarchy [4], as it defines grammars in a fundamental manner, including a semantics that makes no operational concessions. There is a need for well-engineered grammar forms, such as parsing expression grammars [5], but all such work stands on the shoulders of Chomsky.
Chapter 7
Implementation of Textual Concrete Syntax

Abstract This chapter discusses implementation aspects of textual concrete syntax: parsing, abstraction, formatting, and the use of concrete as opposed to abstract object syntax in metaprograms. We focus on how parsers, formatters, etc. are actually implemented in practice, subject to using appropriate libraries, tools, and metaprogramming techniques.

1 Paul Klint’s contributions to computer science are not limited to the implementation (or the practice or the application) of concrete syntax, but this is an area in which he has continuously driven the state of the art over the years. Some of his work on concrete syntax has focused on supporting it in interactive programming environments and language workbenches such as ASF+SDF and Rascal [21, 55, 31, 30]. In other work, he has been addressing practical challenges regarding parsing, for example, in terms of scannerless parsing and ambiguity detection in generalized (LR) parsing [13, 5]. Paul Klint loves grammars [29].

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork DMT, acrylic 2006 by Matt Sheehy with the artist’s permission. This work also quotes https://commons.wikimedia.org/wiki/File:Sunset_at_Montmajour_1888_Van_Gogh.jpg subject to the attribution “Vincent van Gogh: Sunset at Montmajour (1888) [Public domain], via Wikimedia Commons.” This work artistically morphes an image http://homepages.cwi.nl/~paulk showing the person honored, subject to the attribution “Permission granted by Paul Klint for use in this book.”

201
Chapter 8
A Primer on Operational Semantics

Abstract The semantics of a software language assigns meanings to the elements of the language. The field of *programming language theory* provides rigorous techniques for the definition of semantics which are based on mathematical and logical tools. In this chapter, we introduce the method of *operational semantics*: inference rules are used to model the stepwise computation of a program. We do not go into the details of the underlying theoretical underpinnings, but the level of formality may help in developing and reasoning about interpreters and other semantics-aware language processing components (e.g., analyzers, optimizers, or refactorings) more systematically. We demonstrate the implementation of operational semantics in Haskell.

1 The 2004 tsunami took Isabelle Attali and her sons Ugo and Tom from her family and friends. She was in Sri Lanka at the time. Isabelle Attali may be credited with helping launch the field of software languages as she was working on making formal and declarative language definitions – in particular, attribute grammars and operational semantics – practically useful by addressing issues of scalability, tool support, integration, and case studies [1, 2, 3, 4, 5]. She was involved in WAGA (Workshop on Attribute Grammars and Applications) and LDTA (Language Descriptions, Tools, and Applications) – both predecessors of the SLE conference. Isabelle, you are missed (http://www.labri.fr/perso/chaumett/attalicaromel/).

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork *DMT, acrylic, 2016* by Matt Sheehy with the artist's permission. This work also quotes [https://commons.wikimedia.org/wiki/File:Wheat_Stacks_with_Reaper_1888_Vincent_Van_Gogh.jpg](https://commons.wikimedia.org/wiki/File:Wheat_Stacks_with_Reaper_1888_Vincent_Van_Gogh.jpg) subject to the attribution “Vincent Van Gogh: Wheat Stacks with Reaper (1888) [Public domain], via Wikimedia Commons.” This work artistically morphs an image [http://www.sdp.fr/ oasis/personnel/Isabelle.Attali/images/ia2002.jpg](http://www.sdp.fr/oasis/personnel/Isabelle.Attali/images/ia2002.jpg) showing the person honored, subject to the attribution “Permission granted by Isabelle Attali’s husband Denis Caromel for use in this book.”
Chapter 9
A Primer on Type Systems

Abstract Types are semantic properties of program phrases. For instance, the type of an expression may model what type of value the expression would be evaluated to eventually, for example, the type of natural numbers or of Boolean values in an expression language. Types may be assigned to program phrases statically by means of a type system – this is a formal system consisting of inference rules, very much like a semantics definition. Assigned types (“properties”) must predict runtime behavior in a sound manner, i.e., the properties should never be violated by the actual runtime behavior. This is also referred to as type safety (or soundness). The rules making up a type system are easily implemented as type checkers, for example, in Haskell, as we will demonstrate. In this chapter, we provide a (very) basic introduction to type systems.

1 I bet that if you were to ask a programming language researcher for recommendations for textbooks on “programming language theory” with good coverage of “type systems”, most lists would start with Benjamin C. Pierce’s “Types and Programming Languages” [2]. Modest insiders just call it the “TAPL” book. This book and, even more so, the more specialized book “Advanced . . .” [3] capture an incredibly thorough and comprehensive discussion of the broad topic of (mostly static) types, taking advantage of Pierce’s distinguished academic career in the programming language field.

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork [DMT, acrylic, 2006 by Matt Sheehy] with the artist’s permission. This work also quotes [https://en.wikipedia.org/wiki/File:The_Mulberry_Tree_by_Vincent_Van_Gogh.jpg] subject to the attribution “Vincent Van Gogh: The Mulberry Tree (1889) [Public domain], via Wikipedia.” This work artistically morphs an image, [https://www.cis.upenn.edu/~bcpierce/bio.html] showing the person honored, subject to the attribution “Permission granted by Benjamin C. Pierce for use in this book.”
Chapter 10
An Excursion into the Lambda Calculus

**Abstract** The lambda calculus is an idealized programming language which captures the core of functional programming and serves as a notion of computability. The lambda calculus is also a good foundation for studying programming language concepts generally by means of adding dedicated extensions to the basic calculus. Our excursion into the lambda calculus is meant here to let us briefly visit a number of language concepts and aspects of semantics and typing that are of general interest in language design, definition, and implementation. This includes the notions of substitution, fixed-point computation, encoding, and type variance.

---

1 The lambda calculus (or, in fact, the lambda calculi or the lambda cube 1) is a beautiful pillar (a formal tool) of theoretical computer science, logic, and programming language theory 2, 3. It has taken much more than its initial introduction by Alonzo Church to remain relevant to today’s research and to have become a standard tool in programming language design and type theory. Henk Barendregt deserves much credit in this regard.

---

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork [DMT, acrylic, 2006 by Matt Sheehy](https://commons.wikimedia.org/wiki/File:Van_Gogh_-_Starry_Night_-_Google_Art_Project.jpg) with the artist’s permission. This work also quotes [Van Gogh: The Starry Night (1889) (Public domain), via Wikimedia Commons](https://en.wikipedia.org/wiki/Henk_Barendregt) showing the person honored, subject to the attribution “By Jan Benda - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=19135230”
Chapter 11
An Ode to Compositionality

Abstract In this chapter, we complement the earlier development of operational semantics with another approach to defining semantics, namely the higher-order functional approach of denotational semantics. We focus here on compositionality, which is a structuring principle for interpreters, analyses, and yet other functionality for languages. We discuss two styles of denotational semantics: the simpler “direct” style and the more versatile “continuation” style capable of dealing with, for example, nonbasic control flow constructs. Denotational semantics can be implemented easily as interpreters, for example, in Haskell, as we will demonstrate.

1 Twenty-five years after his death, two papers by Christopher Strachey appeared [13, 14]: one on his lectures on programming language semantics and another (coauthored with Christopher P. Wadsworth) on continuations. Domain theory would probably not exist without Strachey [11]. My supervisor’s generation would have known the work of Strachey (and Scott) through Joseph E. Stoy’s textbook [14] and Peter D. Mosses’ thesis [5]. I would fall in love with denotational style also, thanks to its applications to parallel and logic programming [6, 2]. Every software language engineer, in fact, every software engineer, should understand and leverage “compositionality” [1].

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork [DM], acrylic, 2006 by Matt Sheehy with the artist’s permission. This work also quotes [Vincent Van Gogh: Vaas met tuingladiolen en Chinese asters (1886) [Public domain], via Wikimedia Commons]. This work artistically morphs an image [Camphill Village Trust], subject to the attribution “Permission granted by Camphill Village Trust for use in this book.”
A Suite of Metaprogramming Techniques

Abstract Metaprogramming may be done with just a few programming techniques: an object-program representation (to capture the syntactical structure of object programs), pattern matching or accessors (to take apart object programs or to select suitable parts thereof), pattern building or constructors (to construct or compose object programs), and a computational model for tree walking (e.g., visitors in OO programming or possibly just recursion). In this chapter, we describe some metaprogramming techniques on the basis of which many metaprogams can be written in a more disciplined style. That is, we describe term rewriting, attribute grammars, multi-stage programming, partial evaluation, and abstract interpretation.

1 Mastery of semantics-based techniques, type-system acrobatics, over-the-head functional programming – these labels pretty reliably map to Oleg Kiselyov without too much risk of hash-code collision. The photo shows him while he was talking about “typed final (tagless-final) style” [http://okmij.org/ftp/tagless-final/] – an advanced topic of metaprogramming not included in this book. One may wonder what a textbook would look like if Oleg was ever to write down a good part of his operational knowledge.

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork JMT, acrylic, 2006 by Matt Sheehy with the artist’s permission. This work also quotes https://en.wikipedia.org/wiki/File:Roses_-_Vincent_van_Gogh.JPG subject to the attribution “Vincent van Gogh, Roses (1890) [Public domain], via Wikipedia.” This work artistically morphs an image, http://www.cs.ox.ac.uk/projects/gip/school/kiselyov.JPG subject to the attribution “Permission granted by Oleg Kiselyov for use in this book.”