SOFTWARE LANGUAGE ENGINEERING

CREATING DOMAIN-SPECIFIC LANGUAGES USING METAMODELS

ANNEKE KLEPPE

FOREWORD BY
JEAN-MARIE FAVRE
SOFTWARE LANGUAGE ARCHAEOLOGIST
AND SOFTWARE ANTHROPOLOGIST, LIG, ACONIT,
UNIVERSITY OF GRENOBLE, FRANCE

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Kleppe, Anneke G.
Software language engineering : creating domain-specific languages using metamodels / Anneke Kleppe. — 1st ed.
p. cm.
Includes bibliographical references and index.
ISBN: 0-321-55345-4 (paper back : alk. paper) 1. Programming languages (Electronic computers) 2. Software engineering. 3. Computer software—Development. I. Title.
QA76.7.K576 2008
005.1—dc22

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-55345-4
ISBN-10: 0-321-55345-4

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, December 2008

Preface

This book started out as my PhD thesis, a work I started rather late in my career. This book offers a condensed report of the knowledge I have gained over the years on designing languages for the software industry. I hope that this book will be helpful for you in learning about languages, their constituting elements, and how they can be defined. First, however, I would like to explain why this book is not a thesis.

Not a Thesis

As it turned out, it was a wrong move for me to go from industry to academia. Certain requirements to a PhD thesis, or the way these requirements are upheld, prohibit truthful expression of my ideas and insights in the subject matter. These requirements are: the work should be new; the work should be judged by a jury of peers; and the work should be based on a sound theory. So, you might ask, what's wrong with that? These requirements certainly appear to be very good ones. Let me elaborate on these for a bit.

For a scientist to be sure that his or her work is new requires a large investment in studying what other people have been doing. With the increasing number of both scientists and publications per scientist, this requirement can in practice not be verified, only falsified. The limited life span of the average scientist is simply too short in view of the large body of scientific publications. This is the reason that the work of most scientists resembles the work being done by a worm: dissecting a single square yard of the garden of knowledge completely and ever so thoroughly. Only on this small area can the scientist be sure that he or she knows all related work. Unfortunately, the work I wanted to do better resembles that of a bee: visiting the

many wonderful flowers in the garden of knowledge and bringing ideas, concepts, and techniques from the one to the other.

Furthermore, one can question this requirement even more fundamentally: What is newness? How do we decide what is new and what is not? The following quote[1] expresses beautifully how deceptive the difference between old and new is: "(to) make something that was not there before, . . . is deceptive, because the separate elements already existed and floated through history, but they were never before assembled in this manner. Joining, assembling, the new will always consist of that."

For example, when Jos Warmer and I created the Object Constraint Language, did we create something new? In one way we did not, because every element of the language was already there: set theory, object orientation, the notation used in Smalltalk, pre- and postconditions, you name it. Yet somehow, we managed to put all these things together like ingredients in a soup, stir them, and end up with something that most people consider to be a new language.

The second scientific requirement is that the work should be judged by a jury of peers. In practice, this means that the life of a scientist is all about judging (doing reviews) and being judged (getting reviewed). I have found that this is not beneficial for such qualities as open-mindedness and creativity, which every scientist is assumed to possess and cherish. Nor does it encourage cooperation, for who takes the credit? This life full of judgment, combined with the pressure to publish regularly, simply does not provide the right breeding ground for innovative research.

Furthermore, who are these peers who should do the judging? When you are a wormlike scientist, it is usually possible to find a few others working on the same square yard of the garden: people who are able to understand the work presented. Still, many scientists complain about the quality of the reviewers assigned to judge their work. Even worse is the case for a beelike scientist. Suppose that the bee takes an idea from patch A and applies it to patch B. The people in A will not be interested, whereas the people in B will have trouble understanding this strange idea. In a world full of worms, the bee has no peers. Please note that I do not object to having worms in the garden of knowledge. The work of the worms is necessary to keep the soil fertile. I just wish that there were more bees, because they are just as necessary; without their work, the garden cannot bear fruit.

---

1. Translated from Dutch by AK: "wanneer je zelf iets maakt wat er niet eerder was, . . . is dat bedrieglijk, want de afzonderlijke elementen bestonden wel en zweefden in de geschiedenis, maar ze werden nooit eerder op deze wijze samengebracht. Samenbrengen, bijeenvoegen, het nieuwe zal altijd daaruit bestaan" (Connie Palmen, *Lucifer*, p. 261).

Let's proceed to the last requirement, which is that the work should be based on a sound theory. In fact, I have found that this requirement collides with the first. If you want to do innovative research, which I do, it appears that in computer science, the advances in theory no longer precede the advances in practice. Most of the major programming languages (Java, C#), modeling languages (UML), combined with model-driven software development, application architectures (SOA), and network architectures (.Net, J2EE) of the past two decades were conceived in industry. Academic computer science is in danger of slowly becoming an empirical field that studies the advances being made in industry.

Furthermore, most theory in academic computer science is mathematics based. Because only the very basic elements of mathematics are taught to students, most of whom end up working in industry, the gap between theory and practice is growing larger and larger. That leaves us with innovative research on one side of the gap, in industry, and sound theory on the other, in academia.

Widening the gap even further is the existence of a certain disdain for applied research in academia. When faced with making their theories applicable to industry, most scientists react like the mathematician in a well-known joke. This mathematician wakes up to see a fire starting in his bedroom. He looks at the fire and thinks: "Well, I have a bucket in kitchen, water in the bathroom, and I know how to apply these to put out the fire. So, all's well." And he goes back to sleep. In my opinion, progress in computer science is not only about knowing how to do things but also about doing them, especially because doing things always poses new challenges, thus providing you with extra knowledge about how to do (or not to do) these things. On the other hand, a different type of disdain exists in industry. Scientists are regarded as being strange creatures who speak gibberish and are extremely impractical. Therefore, people from industry are rather reluctant to use new ideas from academia, even when they are good ones.

As is, this book is not a thesis. It presents some things that are new but that are based on existing concepts, as well as some things that are old but still relevant for software language design. This book has not been judged by peers in detail, although many parts have been published in reviewed conferences. Finally, I am certain that the theory on which it is based leaves a lot of room for wormlike scientists to work on and improve what is presented here.

In short, the goal of this book is to illuminate fundamental issues about the design of software languages—some relating to syntax and some relating to meaning. Instead of describing a fixed step-by-step plan of how to create a new software language, the book shows what the ingredients of a language specification

are and how each of these can be best expressed, thus providing the language engineer with maximum flexibility.

For this purpose, the use of grammars is contrasted with the use of metamodeling, showing that metamodeling is the best choice. In this book, the metamodel is used as a common notation; and the theory of graphs and graph transformations, an underlying foundation.

ABOUT THIS BOOK

A software language, unlike jargon and other natural-language phenomena, does not develop spontaneously from another, more commonly used language. Instead, a software language needs to be created artificially. For instance, when creating a domain-specific language (DSL), smart people who know the domain that the DSL is targeting need to accumulate the concepts from the domain and question themselves on the relationships between these concepts. Furthermore, the DSL designers need to invent a (concrete) syntax to identify these concepts and relationships. Next, tools, such as editors and code generators, need to be created. In short, the definition of a DSL takes a rather large effort. Therefore, the question of how to define languages for describing software in a consistent, coherent, comprehensible, yet easy manner is of great importance.

Many areas of expertise can contribute to your knowledge of software languages. This book visits the following areas of expertise:

- Computational linguistics
- Fundamental mathematics
- Grammars and compiler technology
- Graphs and graph transformations
- Model-driven software development
- Modeling and metamodeling
- Natural-language research
- Philosophy
- Visual-language Design

Not all these areas are as stand-alone as they appear to be. Studying the wealth of knowledge that these areas comprise, you can find many connections between them and similarities in the concepts used. Examples are grammar versus metamodel, model versus graph, and textual versus visual (graphical) language. Building on these connections and similarities, this book offers you:

- A way to define languages (textual and visual), including their semantics, based on object-oriented (meta)modeling and graph transformations
- An insight into the differences and similarities of the theories of textual and visual languages
- Additions to the integration of object-oriented modeling with graph theory
- A method for generating a grammar from a metamodel and the tools for supporting this
- Insights in to how to built a code generator for a new language
- A way to create support for multilanguage models/programs

ORGANIZATION OF THIS BOOK

When writing this book, I assumed that the reader would be an IT professional or student who is interested in software language design, domain-specific languages, and metamodeling. This reader has knowledge of UML, as well as some knowledge of OCL, but not necessarily of the topics from the areas mentioned earlier. To aid you in understanding these topics, I have included separate sections containing background information on these topics. When you are familiar with the topic, you can skip over these and continue reading the normal text flow. You can find the topics and their page numbers in a separate table of contents in the beginning of this book.

Many people do not sit down to read a book from the first page to the last, certainly not when it's a textbook. Therefore, I have tried to write this book in the style of a Web site, with relatively stand-alone chapters but with links to other chapters. You will find multiple references to other sections. I hope this style helps you to find the parts of the book that are really interesting to you.

Following are short descriptions of the chapters and reasons why you should read or skip any of them.

- Chapter 1 explains the growing importance of software languages and the role of language engineering.
- Chapter 2 describes who is responsible for what within the area of software languages and their use. It also explains the types of tools used by people with various roles. The chapter has an introductory level.
- Chapter 3 explains the concepts used in this book—like language, language specification, linguistic utterance, and mogram—as well as the differences between general-purpose languages and domain-specific ones. The chapter has an introductory level and can be very useful for managerial people.
- Chapter 4 describes in much more detail what a language and a mogram are. It defines the concept of language specification and explains the differences between abstract and concrete forms. Furthermore, some formalisms for creating language specifications are described and compared. The chapter is of interest to the reader who wants to know more about the theoretical background of languages. It has an intermediate level.
- Chapter 5 describes the underlying mathematical theory of metamodels, models, and instances of models. The chapter is of interest to the reader who wants to know more about the theoretical background of metamodeling. It has an advanced level.
- Chapter 6 explains why an abstract syntax model is the central element of a language specification. The chapter also describes Alan, a software language that will be used in a number of chapters. The chapter has an intermediate level.
- Chapter 7 explains how concrete and abstract syntax relate, giving a framework for the recognition process of raw text or pixels into an instance of the language's metamodel or abstract syntax graph. Skip this chapter if you are familiar with scanning, parsing, binding, and so on, and are not interested in seeing how these concepts differ for textual and graphical concrete syntaxes. This chapter has an introductory level.
- Chapter 8 explains how a textual concrete syntax can be derived from a metamodel. Read this chapter if you want to know more about the coupling

of BNF grammars to metamodels. Skip it if you are interested only in graphical syntaxes. This chapter has an advanced level.

- Chapter 9 describes what semantics is and how it can be expressed. Read this chapter if you want to know why understanding is personal. Skip it if you think that a compiler or a code generator explains the meaning of a language well enough. This chapter has an introductory level.
- Chapter 10 dives into the details of specifying semantics by building a code generator. The chapter has an advanced level.
- Chapter 11 explains how languages can be created in such a way that expressions in multiple languages can relate to each other. Skip this chapter if you focus on working with a single language. It has an intermediate level.

ACKNOWLEDGMENTS

Over the years, many people have helped me gain the knowledge that I am sharing with you in this book, and I am very grateful to them. However, a number of people on this occasion deserve special attention. First and foremost, I want to thank Arend Rensink for giving me the opportunity to experience life in academia. We cooperated on the work for Chapter 5, and together with Harmen Kastenberg, we worked on the topic of operational semantics for software languages, which you can find in Chapter 9. In all this, Arend proved to be much more than a wormlike scientist: open and broad-minded, while remaining critical on details and underlying theory.

I would also like to thank another former colleague from the University of Twente: Theo Ruys. Our discussions on the grammar generator where very helpful. My thanks go to David Akehurst for working with me on the comparison of textual and graphical syntaxes, to Jos Warmer for his cooperation on multilanguage support, and to my students for bringing some fun into my research.

Certainly, the reviewers of early versions of this book deserve to be mentioned: Barney Boisvert, Tony Clark, Dave Hendricksen, Anders Hessellund, Ron Kersic, and Markus Völter. I am grateful for their remarks, which I hope made this book a better read for you. Finally, I want to thank the Netherlands Organization for Scientific Research (NWO), which funded the Grasland project at the University of Twente of which I have been part.

FINALLY

As I wrote in the first paragraph of this preface, I hope that this book will be helpful for you in learning about software languages. Like the author of a thesis, I too feel that I have gained some good insights into this topic. I have written this book simply to offer my knowledge to you, its reader. Whether or not you agree with what I write, I hope that it will improve your knowledge of the subject.

Soest, Netherlands, June 2008

Foreword

"Humankind is defined by language;
but civilization is defined by writing."

Computer science is defined
by computer languages.

Will the information age be defined
by the web of software languages?[2]

Since the invention of writing and then of printing, many books have been written on many different topics. So one should always ask, why another book? In particular, why did the author spend so much time and effort in writing *this* book? Should we read it? Is there something new and important here? The answer is *yes*. My recommendation is clear: Don't miss this book!

I've spent a couple of decades trying to understand software, which is certainly the most complex and versatile artefact ever built by humans. Instead of reading the future, I've dedicated a lot of time and effort to reading the past, following Winston Churchill's advice: "The farther back you can look, the farther forward you are likely to see." I realized that getting a deep understanding of software required understanding the history of not only computer science but also information technology and, ultimately, going back to prehistory, when everything started.

Nobody knows for sure when articulated language first appeared, but it was in prehistoric times, during the Stone Age. There is a general agreement on the fact that Homo sapiens is the only species with such elaborated language. Other species

2. The first two sentences are the first two sentences of the book *The world's writing systems*, co-edited by P. T. Daniels and W. Bright. These three sentences attempt to summarize in a very concise way the logical progression from Stone Age and natural languages toward Information Age and software languages.

use different communication means, but they remain very limited. For instance, bees use a "domain-specific language" to indicate to other bees the direction of food. By contrast, human language distinguishes itself for being general-purpose and reflexive. This latter property describes a language's capability to describe facts about itself. This book, which throughout talks about languages, is a perfect example of the metalinguistic capacity of humans. By reading this book, you will learn how metalanguages can be used to define languages. This task is not only extremely enjoyable for the intellect but also has strong practical implications.

Writing is, without any doubt, one of the first and most important information technologies ever invented. You could not read this book without it. As a matter of fact, the invention of writing marks the shift from *pre-history* to *history*. Writing did not pop up all at once, though. Very rudimentary techniques, such as notched bones or pebbles, were used in prehistoric times for information recording and basic information processing. But with the *neolithic* revolution, this situation started to change.

The domestication of plants and animals led to agriculture and livestock farming. Social life organized around settlements enabled (1) the separation of activities, (2) the specialization of knowledge and skills, (3) the appearance of "domain experts," and (4) the diversification of artifacts produced. While some men worked in fields, other specialized as shipmen, butchers, craftsmen, and so on. The notions of property and barter appeared, bringing new requirements for information recording and processing. In Mesopotamia, protowriting appeared in the form of clay tablets; in Egypt, in the form of tags. Domain-specific languages were used to record the number of sheep, grain units, or other goods. Other kinds of tablets were used to organize tasks.

As villages grew into cities and then into kingdoms, the complexity of social structures increased in an unprecedented manner. Protowriting turned into w*riting,* and *civilizations* appeared in Egypt and Mesopotamia. Civilization is based on writing. Without writing, rulers could not establish laws, orders could not be sent safely to remote troops, taxes could not be computed, lands and properties could not be administrated, and so on. Early protowriting systems were used locally and evolved in an ad hoc way. The development of writing, by contrast, was somehow controlled and managed by states, via the caste of scribes and the establishment of schools and then libraries to keep bodies of texts or administrative records. In other words, one of the characteristics of civilizations is that written languages are "managed." As you will see, this book provides important insights about various roles in *language management* and *language engineering,* which is one step fur-

ther. Note that many kinds of notations, either graphical or textual, have been devised. Almost all scientific and engineering disciplines use some domain-specific languages: chemical notations in chemistry, visual diagrams in electronics, architectural drawings, all kinds of mathematical notations, to name just a few.

All the languages and technologies mentioned so far were based on the use of symbols produced and interpreted by humans. By contrast, Charles Babbage invented the concept of an "information machine." The whole idea was to replace "human computers" using ink and paper to perform repetitive calculations. For this purpose, the machine would have to *read* and *write* symbols in some automated fashion. In some sense, computer science is "automatic writing."

In the previous century, computers became a reality. The first computers, such as the Colossus, ENIAC, or Mark I, were truly technological monsters. Operators had to continuously feed the machines. Humans were dominated by the machines. This period, which I call Paleo informatics, is similar to the Paleolithic, when cavemen were dominated by nature. Whereas the Paleolithic is characterized by small groups of individuals performing a single activity—namely, finding food—Paleo informatics was characterized by small groups of computer shamans focusing on programming. The term *computer language* was associated with computer science, just as if computers really cared about languages humans invented to deal with them. I guess this is one of the biggest misunderstandings in computer science.

The Paleo informatics age is over. The shift to *neoinformatics*, though unnoticed, is not recent. In the past decades, the relationships between humans and computers have dramatically changed in favor of humans. Just as the neolithic revolution marked the domestication of nature by humans, humans have now domesticated computers. Even children can use computing devices. This new relation between our society and computers leads to the (re)emergence of the term *informatics.* Simply put, informatics is computer science in context. Informatics provides the whole picture, including both humans and computers; computer science corresponds to the narrow view focusing on computers.

What is interesting is that with the shift from Paleo informatics to neoinformatics, the size of social structures in the software world has been increasing: *Single programmers* writing *programs* have been replaced by *teams of engineers* developing and evolving *software*. Just like the neolithic period, neoinformatics is characterized by (1) the separation of activities, (2) the specialization of knowledge and skills, (3) the appearance of "domain experts," and (4) the diversification of (software) artifacts. Programming is no longer the unique activity in the software world. Neoinformatics is characterized by a multiplication of languages,

including programming languages, but also specification languages (Z), requirement languages, modeling languages (UML), architecture description languages (Wright), formatting languages (LaTeX), business-process languages (BPEL), model-transformation languages (ATL), metalanguages (BNF), query language (SQL), and so on. It would be easy to cover a whole page with such a list. Moreover, the concept of language can take many different incarnations and names: metamodels, grammars, ontologies, schemas, logics, models, calculus, and so forth.

Since all these languages are used in the context of software production, I fully agree with the author that they should be called *software languages*. I know that many people would have suggested the use of the term *computer language*, as it initially referred to programming languages but later became more vague. I strongly disagree with that solution. I even believe that this term should be banned! First, it should be banned because of the unbalanced and inappropriate emphasis on computers; second, it should be banned because it does not fit with the trend in informatics toward the “disappearing computer,” or the “invisible computer.”

So far, software languages have been developed in a rather ad hoc way. But this should change in the future. Many experts predict that ubiquitous computing and ultra-large-scale systems will be part of the future of informatics. This will mean building software-intensive systems of incredible complexity. This will be achievable only through the collaboration of many experts from many disciplines. What is more, these systems will certainly have to evolve over centuries. It is likely that their continuous evolution will involve complex consortiums gathering many organizations, governmental or not.

I predict that the need to manage software languages explicitly will become more obvious each day in the future. On the one hand, the implication of more actors means more communication and hence the need for *vernacular software languages* (i.e., interchange languages). On the other hand, specialization can be achieved by domain-specific languages. They act as *vehicular software languages* (i.e., local languages). Ultra-large-scale systems will therefore imply controlling and managing complex *networks of software languages*. The heterogeneity of the Internet and of the information on it will lead to what could be called the *web of (software) languages*.

As a matter of fact, this vision is not new. If you closely observe what is going on in some fields, some human activities are based on a similar scenario. Consider, for instance, very complex artifacts, such as planes. Designing and building Airbuses requires the collaboration of many companies in many countries. Many experts in many different fields are supposed to design, realize, or test parts of the

whole system. These experts may use different natural languages for internal documents and many different technical or scientific languages. For instance, the graphical formalism used to design the cockpit is certainly totally different from the language used to model the deformations of wings, which is itself totally different from the language used to describe electronic circuits. Building complex systems, such as planes, also implies complex language networks.

Software languages are already there and so must be considered from a scientific point of view. I call *software linguistics* the subset of linguistics (i.e., the science of languages) that will be devoted to the study of software languages. Software linguistics should be contrasted both with natural linguistics—its counterpart for natural languages—and with computational linguistics, or applying computational models in the context of (natural) linguistics. Software linguistics is just the other way around: linguistics applied to software.

Languages crossed the ages—from the Stone Age to the information age—but their diversity and variety are ever increasing. By contrast to the large body of books that describe very specific language-oriented techniques, such as parsing or implementing visual editors, this book provides the first global view on software language engineering, while also providing all the necessary technical details for understanding how these techniques are interrelated. If you want to know more about software language engineering, you could not be luckier: You have exactly the right book in your hands! Just read it.

Jean-Marie Favre
Software Language Archaeologist and Software Anthropologist
LIG, ACONIT, University of Grenoble, France

Chapter 1

Why Software Language Engineering?

An invasion of armies can be resisted,
but not an idea whose time has come.

—Victor Hugo
French dramatist, novelist,
and poet (1802–1885)

This chapter explains why languages are becoming more and more important in computer science and why language engineering, the art of creating languages, is an important topic for computer scientists. This chapter also gives some background knowledge on several theories regarding languages and language engineering.

1.1 An Increasing Number of Languages

In the past few years, an increasing number of languages used for designing and creating software have been created. Along with that has come an increasing interest in language engineering: in other words, the act of creating languages.

First, *domain-specific languages* (DSLs) have become more and more dominant. DSLs—languages for describing software—are focused on describing either a certain aspect of a software system or that software system from a particular viewpoint. One can compare a DSL to a certain jargon, for example, in stockmarket jargon terms like "going short" and "bear market" have specific meaning.

Second, modeling languages, specifically domain-specific modeling languages (DSMLs)—a special type of DSL—have become very important within the field of model-driven development (MDD), also known as model-driven architecture (MDA). Each model is written in a specific language, and the model is transformed into another model written in (in most cases) yet another language. Without the existence of multiple modeling and programming languages, model-driven development would be certainly less relevant. However, multiple languages do exist and, judging by the number of (programming) language debates on the Internet, many people are very happy with that. They are glad that they can use their favorite language because, in their opinion, the other languages are all so very worthless.

Another driving force behind the increase in the number of languages used in software development is the recognition of various categories of software systems. Each category of software systems, or architecture type, defines its own set of languages. For example, Service Oriented Architecture (SOA) has brought a large number of languages into being:

- Semantic Web Services Language (SWSL)
- Web Services Business Process Execution Language (WS-BPEL)
- Web Services Choreography Description Language (WS-CDL)
- Web Services Description Language (WSDL)
- Web Services Flow Language (WSFL, from IBM)
- Web Services Modeling Language (WSML, from ESSI WSMO group)

The coming years will bring more and more languages describing software into this world. Without a doubt, there will be developments similar to the ones we have witnessed in the creation of programming languages. Some languages will be defined and owned by private companies, never to be used outside that company. Some languages will be defined by standardization committees (e.g., OMG, CCITT, DOD, ISO); others will be created by private companies and standardized later on. Some languages will become popular, and some will find an early death in oblivion. But the one thing we can be certain of is that information technology (IT) professionals will be spending their time and efforts on creating languages and the tooling to support their use.

1.2 SOFTWARE LANGUAGES

What is the type of the languages that are we talking about? For instance, the acts of modeling software and programming are traditionally viewed to be different. Likewise, the characteristics of the languages used for either modeling or programming are perceived to be different, as shown in Table 1-1. However, the similarities between modeling and programming languages are larger than the differences. In essence, both describe software, and ultimately—after transformation and/or compilation—both need to be executable. Differences between modeling and programming that traditionally seemed very important are becoming less and less distinctive. Models are becoming precise and executable—see, for instance, Executable UML [Mellor and Balcer 2002]—and graphical; high-level syntaxes for programming languages are becoming popular—see, for example, the UML profile for Java [OMG-Java Profile 2004]. Furthermore, model-driven development has taught us that models must not be regarded as rather inconsequential artifacts of software development but instead as the core products.

In the remainder of this book, I use the term *software language* to indicate any language that is created to describe and create software systems. Note that many different languages target software development: programming and modeling languages, as well as query, data-definition, and process languages. In fact, all the languages mentioned in Section 1.1 are software languages.

Table 1-1 *Perceived Differences between Modeling and Programming Languages*

Modeling Language	Programming Language
Imprecise	Precise
Not executable	Executable
Overview	Detailed view
High level	Low level
Visual	Textual
Informal semantics	Execution semantics
By-product	Core product

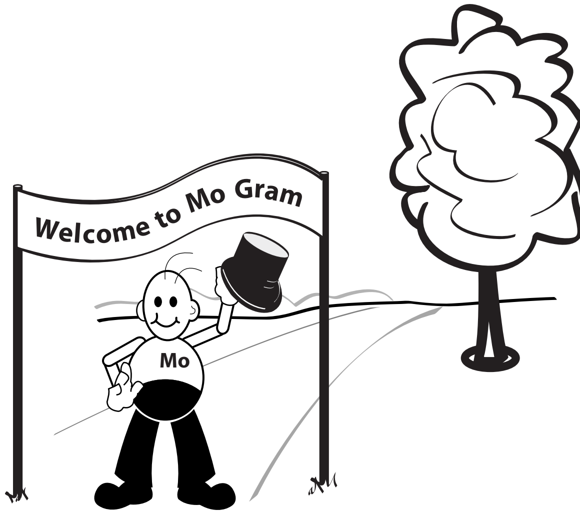

Figure 1-1 *Mogram: a concept that includes programs, models, and other things*

But how should the product written in a software language be named? Should it be called *model, program, query*, or *data definition* or any other word that is currently in use for an artifact created with a software language? To avoid any form of bias, I introduce the term *mogram* (Figure 1-1), which can be a model or a program, a database schema or a query, an XML file, or any other thing written in a software language (see Section 3.1.1).

1.3 THE CHANGING NATURE OF SOFTWARE LANGUAGES

When you want to create software languages, you need to have a clear picture of their character. Over the past two decades, the nature of software languages has changed in at least two important ways. First, software is increasingly being built using graphical (visual) languages instead of textual ones. Second, more and more languages have multiple syntaxes.

1.3.1 Graphical versus Textual Languages

There is an important difference between graphical and textual languages. In the area of sound, a parallel to this difference is clearly expressed in the following quote contrasting the use of spoken word with music.

> It is helpful to compare the linear structure of text with the flow of musical sounds. The mouth as the organ of speech has rather limited abilities. It can utter only one sound at a time, and the flow of these sounds can be additionally modulated only in a very restricted manner, e.g., by stress, intonation, etc. On the contrary, a set of musical instruments can produce several sounds synchronously, forming harmonies or several melodies going in parallel. This parallelism can be considered as nonlinear structuring. The human had to be satisfied with the instrument of speech given to him by nature. This is why we use while speaking a linear and rather slow method of acoustic coding of the information we want to communicate to somebody else. [Bolshakov and Gelbukh 2004]

In the same manner, we can distinguish between textual and graphical software languages. An expression in a textual language has a linear structure, whereas an expression in a graphical language has a more complex, parallel, nonlinear structure. Each sentence in a textual language is a series of symbols—or *tokens* as they are called in the field of parser design—juxtaposed. In graphical languages, symbols, such as a rectangle or an arrow, also are the basic building blocks of a graphical expression. The essential difference between textual and graphical languages is that in graphical languages, the symbols can be connected in more than one way.

To exemplify this, I have recreated a typical nonlinear expression in a linear manner. Figure 1-2(a) shows what a Unified Modeling Language (UML) class diagram would need to look like when being expressed in a linear fashion. The normal, nonlinear way of expressing the same meaning is shown in Figure 1-2(b). The problem with the linear expression is that the same object (Company) needs to appear twice because it cannot be connected to more than one other element at the same time. This means that you need to reconcile the two occurrences of that object, because both occurrences represent the same thing.

Note that many languages have a hybrid textual/graphical syntax. For instance, the notation for attributes and operations in a UML class diagram is a textual syntax embedded in a graphical one.

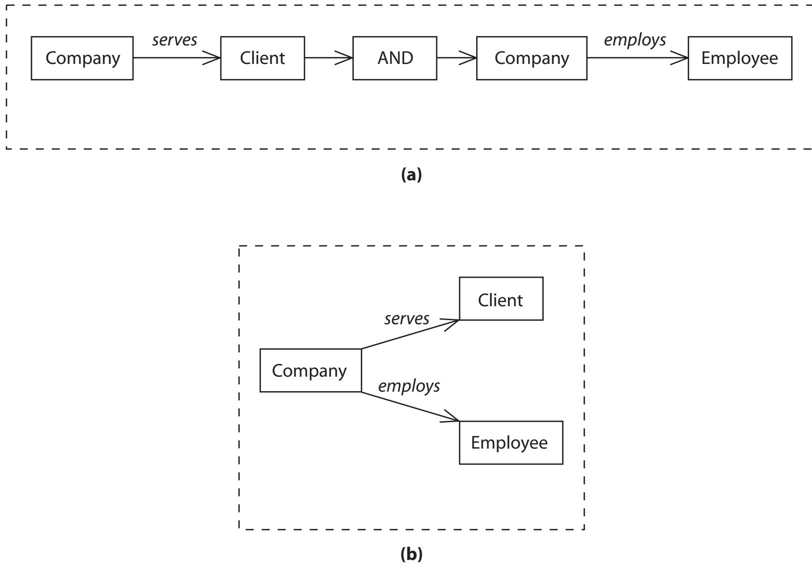

Figure 1-2 *A linear and a nonlinear expression*

The traditional theory of computer languages—compiler technology—is focused on textual languages. Therefore, without losing the valuable knowledge gained in this area, we need to explore other paths that lead toward the creation of graphical languages.

1.3.2 Multiple Syntaxes

Another aspect of current-day software languages is the fact that they often have multiple (concrete) syntaxes. The mere fact that many languages have a separate interchange format (often XML based) means that they have both a normal syntax and an interchange syntax. At the same time, there is a growing need for languages that have both a graphical and a textual syntax, as shown by such tools as TogetherJ, which uses UML as a graphical notation for Java. The UML itself is a good example of multisyntax language. There is the well-known UML diagram notation [OMG-UML Superstructure 2005], which is combined with the Human-readable UML Textual Notation (HUTN) [OMG-HUTN 2004] and the interchange format called XMI [OMG-XMI 2005].

A natural consequence of multiple syntaxes is that (concrete) syntax cannot be the focus of language design. Every language must have a common representation of a language expression independent of the outward appearance in which it is entered by or shown to the language user. The focus of language design should be on this common representation, which we call *abstract syntax*.

1.4 The Complexity Crisis

One reason to invest in software language engineering is that over the past decades, the nature of not only software languages but also software development has changed. In fact, it is the nature of the software applications that has changed and with it, software development.

Software applications are becoming more and more complex. As Grady Booch once mentioned in a presentation on UML 1.1, computer scientists today do not create the type of application from the 1960s or 1970s in less time and with less effort. Instead, they create far more complex applications.

The number of languages, frameworks, and so on, that an ordinary programmer nowadays needs to deal with is exceedingly large. For example, in the development of an average Web application based on Java, one needs to have knowledge of the Java language and some of its APIs, XML, XSD, XSLT, JSP, Struts or JSF, Enterprise JavaBeans or Hibernate/Spring, SQL, UML, Web Services, and, lately, Ajax. All these bring with them their own complexity, not to mention the complexity that arises from the combination. The time when printing “Hello, World” on a screen was a simple exercise for novices is long gone. Nowadays, you need to know how the graphical user interface (GUI) library works, because this text needs to be shown in a separate window with fitting fonts and colors. Furthermore, user demands are such that the sentence must be personalized into “Hello Mrs. Kleppe,” for which data must be retrieved from the database. Even better would be to print the text “Goodmorning, Mrs. Kleppe,” for which you have to find out what time of day it is at the user’s site. In short, building software is becoming more and more challenging for all of us, not only for novices.

Unfortunately, all this means that the problem, which was called the *software crisis* by Bauer in 1968 and Dijkstra in 1972 [Dijkstra 1972], has grown worse: Many software projects take more time and budget than planned. However, this is not because of the fact that there has been no progress in computer science.

Rather, there has been so much progress that we are able to take up increasingly complex assignments. This means that time and again, we are facing a new, unknown, and uncertain task, which in its turn leads to an inability to take full control over the development process.

An interesting and worrying consequence of the increasing size and complexity of software applications is that it is no longer possible to know an application inside out. Much expert knowledge from various areas is needed to create an application. In many projects, experts on one topic—say, XML—are creating parts of the application without intimate knowledge of what other experts—for instance, on user interface design—produce. It is no longer humanly possible to know all the things that need to be known about an application.

A common way to tackle the complexity is to revert to using frameworks and patterns. And although in the short term this approach does help, in the long term we need our software languages to be at a higher level of abstraction if we want to be able to keep understanding what it is that we are producing. A higher level of abstraction means that we keep manual control over the parts of the development process in which new problems are tackled (variability), while relying on automation for the simpler parts (standardization). The stuff that we have created time and again and thus know thoroughly can be created automatically by mapping the higher-level mogram to a lower-level target language. This is the essence of model-driven development.

Furthermore, we are faced with an increase of interest in creating domain-specific languages. When these are not well designed, the applications created with these languages will not be of good quality.

The challenge for language engineers is that the software languages that we need to create must be at a higher level of abstraction, a level we are not yet familiar with. They must be well designed, and software developers must be able to intermix the use of multiple languages. Here too, we are facing a new, unknown, and uncertain task.

1.5 WHAT WE CAN LEARN FROM ...

Luckily, a large amount of existing knowledge is helpful in the creation of software languages. In this book, I draw mainly from three sources: natural-language studies,

traditional computer language theories, and graph grammars. And although this field does not have a great influence in this book, I also include a short description of the work from the visual languages community, because it is rather similar to traditional computer language theories but aimed at graphical, or visual, languages.

1.5.1 Natural-Language Studies

When someone who is not a computer scientist hears the word *language*, the first thing that comes to mind is a natural language, such as the person's mother tongue. Although software languages are artificially created and natural languages are not, we can still learn a lot from the studies of natural languages. (See Background on Natural-Language Studies.) Of course, we cannot use every part of it. For example, software languages do not have a sound structure, so there is no need to study phonology.[1] However, all other fields of study are as relevant to software languages as they are to natural languages. It must be noted that these fields of study are, of course, interrelated. One cannot reasonably study one aspect of a

Background on Natural-Language Studies

These studies are divided into the following five categories.

1. Phonology, which is concerned with the sound structure of language
2. Morphology, which is concerned with the structure of the words in the language, such as inflections or conjugations
3. Syntaxis, which is concerned with the structure of sentences, or how words are combined
4. Discourse, which is concerned with the structure of conversations or other compositions of sentences
5. Semantics, which is concerned with the meaning of words, sentences, and discourses

Introductions to these categories can be found in Fromkin et al. [2002] and Tserdanelis and Wong [2004].

With regard to software languages, we are interested mostly in syntaxis and semantics.

1. However, it is interesting to know that the graphical language UML has been adapted for use by blind people and that this adaptation does use some form of sound [Petrie et al. 2002].

language, such as morphology, without being at least aware of the other aspects, such as syntaxis.

It is interesting to see that with the advent of mobile phones, the phenomenon of multiple syntaxes is also emerging in natural language. A second morphological and syntactical structuring of natural language expressions has developed; for example, everybody understands the next two phrases as being the same: *4u* and *for you*.

1.5.2 Traditional Language Theory

In the late 1950s, the fundamentals of current-day theory for textual software languages were laid down by such people as Chomsky [1965] and Greibach [1965, 1969]. In the 1970s and 1980s, these fundamentals were used by, among others, Aho and Ullman to develop the theory of compiler construction [Hopcroft and Ullman 1979, Aho et al. 1985]. In this research, grammars were used to specify textual languages.

The original motivation for the study of grammars was the description of natural languages. While linguists were studying certain types of grammars, computer scientists began to describe programming languages by using a special notation for grammars: Backus-Naur Form (BNF). This field has brought us a lot of knowledge about compiler technology. For more information, see Background on Grammars (p. 48), Background on Compiler Technology (p. 96), and Background on Backus-Naur Format (p. 116).

1.5.3 Graph Theory

Graphs have long been known as mathematical constructs consisting of objects—called nodes or vertices—and links—called edges or arcs—between them. Over the ages, mathematicians have built up a large body of theory about graphs: for instance, algorithms to traverse all nodes, connectivity theorems, isomorphisms between two graphs. All this has been brought to use in graph grammars.

Graphs grammars specify languages, usually graphical (visual) languages, by a set of rules that describe how an existing graph can be changed and added to. Most graph grammars start with an empty graph, and the rules specify how the expressions in your language can be generated. For more information, see Background on Graphs and Trees (p. 50).

1.5.4 The Visual-Languages Community

Visual-language design is our last background area. Although it started later than textual languages, the advance of visual languages has also been investigated for a few decades now. This area can be roughly divided into two parts: one in which grammars are based on graphs and one that does not use graph grammars.

In this book, I do not use *visual language*, the term commonly used in this community. Instead, I use *graphical language*, a phrase I find more appropriate because textual languages are also visual, while at the same time most nontextual languages are denoted by some sort of nodes with edges or connectors between them: in other words, denoted like a graph.

Non-Graph-Grammar Based

The non-graph-grammar-based research is concerned mostly with scanning and parsing visual-language expressions: in other words, diagrams. Several formalisms for denoting the rules associated with scanning and parsing have been proposed: Relational Grammars [Weitzman and Wittenburg 1993], Constrained Set Grammars [Marriott 1995], and (Extended) Positional Grammars [Costagliola and Polese 2000]. For a more extensive overview, see Marriott and Meyer [1997] or Costagliola et al. [2004].

Virtually all work in this area focuses on the recognition of basic graphical symbols and grouping them into more meaningful elements, although some researchers stress the fact that more attention needs to be paid to language concepts, which in this field is often called *semantics*. Often, these semantics are added in the form of attributes to the graphical symbols. This is very different from the metamodeling point of view. For instance, although UML is a visual language, its metamodel does not contain the notions of box, line, and arrow.

Common to all formalisms is the use of an alphabet of graphical symbols. These graphical symbols hold information on how to materialize the symbol to our senses, such as rendering info, position, color, and border style. Next to the alphabet, various types of spatial relationships are commonly defined, based on position (left to, above), attaching points (end point of a line touches corner of rhombus), or attaching areas by using the coordinates of the symbol's bounding box or perimeter (surrounds, overlaps). Both the alphabet and the spatial relationships are used to state the grammar rules that define groupings of graphical symbols.

Although the work in this area does not use graph grammars, the notion of graphs is used. The spatial relationships can be represented in the form of a graph, a so-called spatial-relationship graph [Bardohl et al. 1999], in which the graphical symbols are the nodes, and an edge between two nodes represents a spatial relationship between the two symbols.

Graph-Grammar Based

The graph-grammar community has also paid attention to visual-language design, most likely because the graph formalism itself is a visual one. In this field, more attention is paid to language concepts. In fact, the graph-grammar handbook states that the graphical symbols needed to materialize the language concepts to the user are attached as attribute values to the graph nodes representing the language concepts, an approach that is certainly different from the non-graph-grammar-based field.

Another distinctive difference is that the non-graph-grammar-based approach uses grammar rules with only one nonterminal on the left-hand side: that is, grammar rules in a context-free format. Graph grammars, on the other hand, may contain rules that have a graph as the left-hand side: that is, grammar rules in a context-sensitive format.

A large number of tools are able to create development environments for visual languages. These include, among others, DiaGen [Minas and Viehstaedt 1995], GenGed [Bardohl 1999], AToM3 [de Lara and Vangheluwe 2002], and VL-Eli [Kastens and Schmidt 2002].

1.6 SUMMARY

More and more languages are used for building software. Almost always, multiple languages are needed to create one single application. I use the name *software language* to indicate any of them: a modeling language, a query language, a programming language, or any other kind of language.

The nature of software languages has changed over the years. The most important software languages used to be textual, but currently many software languages have a graphical syntax. Furthermore, many languages today have two or more syntaxes: for instance, both a textual and a graphical, or a normal and an interchange syntax.

Because the applications that we create are becoming more and more complex, the languages that we use must reside at a higher level of abstraction. Otherwise, we will never be able to meet the ever-rising demands and expectations of our customers.

All software languages must be artificially created. Therefore, language specification—both the act and the result of the act—becomes very important. Luckily, this is not a completely new field. We can draw on the knowledge gained in natural-language research, compiler technology, graph theory, and visual languages.

Index

A

Abstract form
- code generation and, 154–156
- defined, 185
- of a mogram, 41, 78
- template language targeting, 157–158

Abstract machines
- modeling with operational semantics, 139–140
- runtime environment of, 141
- semantics and, 138–139
- states in, 142–143
- transitions in, 143–144

Abstract syntax, 75–91
- abstract form of a mogram, 78
- as focus of language design, 7
- as gateway to semantics, 76–77
- hidden, underlying, unifying nature of, 75–76
- mogram/language relationship and, 77
- summary, 90–91
- syntactic vs. semantic correctness, 79
- syntactically incorrect mograms, 80

Abstract syntax graphs. *See* ASGs (abstract syntax graphs)

Abstract syntax models. *See* ASM (abstract syntax model)

Abstract syntax trees
- abstract form of a mogram and, 41
- building, 49
- compilers and, 96
- defined, 185
- textual languages and, 98

Abstraction, 27–29
- abstract vs. incomplete, 29
- defined, 29
- expressiveness and, 29
- hardware level and, 28–29
- raising the level of, 29–31
- ranges of, 32–33
- semantic analysis, 98

Abstraction levels
- code generation and, 150
- defined, 185
- hardware level, 28–29
- raising, 29–31
- ranges of, 32–33

Accidental complexity, software construction and, 113–114

Active languages, combining multiple languages and, 174–175

Acyclic constraint, 64

Alan (**a language**)
- abstract syntax model, 86–90
- concrete syntax model, 106–110, 118–120
- GCSM example, 106–110
- generic types in, 85
- Observer pattern in, 84–85
- overview of, 83
- standard library, 86
- TCSM example, 118–120

Analyzers, in tool set of language user, 17–18

APIs (application programming interfaces), 37

Application-centered processes, vs. language-centered processes, 15

Application developers. *See* Language users
ASGs (abstract syntax graphs)
  binding and, 98–99
  defined, 185
  forming, 98
  static checking and, 99–100
  transforming a parse tree into, 115
ASM (abstract syntax model)
  abstract-to-concrete transformation, 117–121
  creating, 80–81
  defined, 78, 185
  domain modeling for vertical DSLs, 81–82
  Grasland generator and, 115
  language specification parts, 41
  pattern finding for horizontal DSLs, 82–83
  summary, 90–91
ASM (abstract syntax model), Alan example
  expressions, 87–89
  Observer pattern in, 89–90
  overview of, 86
  standard library, 90
  types, 86–87
asm2tcsm transformations, 117–121
  Alan example, 118–120
  designer input to, 120–121
  reference handling and, 117–118
Attributed grammars, language specification, 49–51
Attributes
  Alan and, 86
  context-free grammars and, 49

B

Backus-Naur Form. *See* BNF (Backus-Naur Form)
Bidirectionality constraint, 62–63
Bidirectionality, in code generation, 163–164
Binding
  complex, 127–128
  overview of, 126
  semantic analysis and, 98–99
  simple, 127
BNF (Backus-Naur Form)
  background on, 116
  context-free grammars and, 47
  creating BNF rule set in textual language creation, 114
  study of grammars and, 10
Business expectations, gap with IT development, 31–32

C

C# partial classes, techniques for extending or overriding generated code, 162
CASE (computer-aided software engineering), 101
Change management, combining multiple languages and, 180–181
COBOL (COmmon Business-Oriented language), 30
Code-completion editors, 100
Code generation, 149–170
  abstraction levels and, 150
  bidirectionality in, 163–164
  building code generator, 151
  combining multiple languages and, 181–182
  concrete or abstract form of target, 154–156
  CreateOrFind pattern, 159–160
  extension points in generated code, 161–163
  flexibility of off the shelf software vs. custom, 167, 169
  hard-coded or model-transformation rules, 151–153
  language design and, 149–150
  mappers, 160–161
  modularization of, 167–168
  multiple semantics, 150–151
  patterns, 158
  source-driven or target-driven translation, 153–154
  summary, 169–170
  target platforms and, 164–167
  template language targeting abstract form, 157–158
  treewalker or visitor pattern, 158–159
Code generators
  accidental complexity and, 114
  compilers compared with, 161–162

concrete or abstract form of target, 154–156
hard-coded or model-transformation rules, 151–153
multiple semantics and, 150–151
overview of, 151
source-driven or target-driven translation, 153–154
template language targeting abstract form, 157–158
Commercial off the shelf (COTS) software, 167, 169
COmmon Business-Oriented language (COBOL), 30
Compilers
background of compiler technology, 96–97
code generators compared with, 161–162
multiple semantics and, 150–151
recognition process and, 94
scanning and parsing, 97
Complex binding, 127–128
Complexity crisis, software languages and, 7–8
Computer-aided software engineering (CASE), 101
Computer experts, vs. domain experts, 33–34
Concepts, in abstract syntax models, 76–77
Conceptual models. *See* ASM (abstract syntax model)
Concrete form
defined, 78, 185
of a mogram, 78–79
Concrete syntax, 93–111
Alan and, 83
editors, 100
of graphical languages, 107–109. *See* GCSMs (graphical concrete syntax models)
optical recognition, 94–96
overview of, 93
recognition process and, 94
scanning and parsing, 97
semantic analysis, 98–100
summary, 110–111
textual-graphical mismatches (blind spots), 101–103
of textual languages. *See* TCSMs (textual concrete syntax models)
tool support, 93–94
Concrete syntax model. *See* CSMs (concrete syntax model)
Concrete targets, code generator and, 154–156
Constraints, 62–66
overview of, 62
types of, 62–66
UML class diagrams, 58
Context-free grammars, 47–49
COTS (commercial off the shelf) software, 167, 169
CreateOrFind pattern, code generation, 159–160
CSMs (concrete syntax models)
abstract-to-concrete transformation, 117–121
Alan example, 106–110
creating, 105–106
defined, 78, 186
language specification parts, 41
overview of, 104–105
summary, 110–111
transforming to ASM, 109–110
Cycles, graphs and trees and, 50

D

Denotational semantics
defined, 135
overview of, 134–135
Denotations, 135
Derivation graphs, 51
Derivation trees
defined, 186
underlying structure of context-free grammar, 47
Derivations, in formal language theory, 25
Derived elements, OCL, 121
Design engineers. *See* Language engineers
Directed graphs, 50, 58–59
disobserve operation, Alan, 83
Display format, 186
Domain-driven design, 81
Domain experts, vs. computer experts, 33–34
Domain modeling, for vertical DSLs, 81–82
Domain-specific modeling languages (DSMLs), 2

DSLs (domain-specific languages)
distinguishing from other software languages, 37–38
domain experts vs. computer experts, 33–34
domain modeling for vertical DSLs, 81–82
frameworks and APIs compared with, 37
vs. general languages, 33
horizontal vs. vertical, 35–37
increasing dominance of, 1
large user group vs. small user group, 34–35
overview of, 33
pattern finding for horizontal DSLs, 82–83
summary, 38
DSMLs (domain-specific modeling languages), 2

E
Editors
concrete syntax, 100
graphical, 102–103
in tool set of language user, 17–18
Empty subclasses, techniques for extending or overriding generated code, 162–163
End states, transitions and, 144
Essential complexity, in software construction, 113–114
Executers, in tool set of language user, 17–18
Expressions
Alan ASM, 87–89
confusion regarding terminology and, 24
Expressiveness, abstraction and, 29
Extension points, in generated code, 161–163

F
Flexibility, off the shelf vs. custom software, 167, 169
Forests, 50
Formal language theory, 25
Formalisms
attributed grammars, 49–51
context-free grammars, 47–49
defined, 186
graph grammars, 51–52
language workbench and, 19–20
languages developed by meta-language engineers, 16
metamodeling, 53–54
overview of, 47
UML profiling, 52–53
Forms, creating mograms with, 102
Formula Translating System (Fortran), 34
Fortran (Formula Translating System), 34
Frame graphs, 143
Frameworks
DSLs compared with, 37
tackling complexity with, 8
target platforms with/without, 164–167
techniques for extending or overriding generated code, 162
Free-format editors (symbol-directed), 100

G
GCSMs (graphical concrete syntax models), 107–109
Generic types, in Alan, 85
Grammars
attributed, 49–51
background on, 48
context-free, 47–49
defined, 186
formal language theory and, 25
graph, 51–52
metamodels compared with, 53–54, 125–126
traditional language theory and, 10
visual-languages and, 11–12
Graph grammars, 51–52
Graph transformations, 144
Graphical concrete syntax model (GCSM), 107–109
Graphical editors, 102–103
Graphical languages
defined, 186
graph grammars and, 12, 51–52
graph theory and, 10
non-graph-grammar based, 11–12
overview of, 11
textual-graphical mismatches (blind spots), 101–103
vs. textual languages, 5–6
Graphs
background on, 50
labeled, 58–59

mapping UML class and object diagrams to, 67–68
metamodeling and, 58–61
operational semantics using, 138–139
type graphs, 60
Grasland generator, 113–116
accidental complexity and, 113–114
binding and, 126–128
generating IDE from a language specification, 115–116
implementation of asm2tcsm transformation, 120–121
implementation of tcsm2bnf transformation, 122–126
keywords and, 125
look-aheads and, 126
need for SSA generator, 114
static checking and, 128–129
Guarded blocks, techniques for extending or overriding generated code, 163

H

Hard-coded rules, code generators, 151–153
Hard references, combining multiple languages and, 173–174
Hardware level, abstraction and, 28–29
Hidden descriptor, abstract syntax, 75–76
Hiding elements, 175–176
Hybrid syntax transformations, 73

I

IDEs (Integrated Development Environments)
change handling, 180–181
elements of, 17
generating from a language specification, 115–116
language workbench compared with, 19–20
multilanguages and, 179
structure editors and, 101
In-place transformations, 72
Incompleteness, abstraction compared with, 29
Instances, of models, 58, 61–62
Invariant constraint, 62

J

JavaCC parser, 115
JET templates, 73

K

Keywords
Grasland generator and, 125
vs. primitive elements, 26

L

Labeled graphs, 58–59
LALR parsing, 97
Language-centered processes, vs. application-centered processes, 15
Language design, code generation and, 149–150
Language-design environment, 186. *See also* language specification
Language engineers
defined, 186
role of, 15–16
tasks of, 19
tool generators, 20–21
tool set of, 19–20
Language interfaces
combining multiple languages and, 176–177
language specification parts, 42
offered or required language interfaces, 177–179
Language specification, 39–46
attributed grammars, 49–51
context-free grammars, 47–49
defined, 39, 186
example of, 43–46
formalisms, 47
generating IDE from, 115–116
graph grammars, 51–52
metamodeling, 53–54
overview of, 39–40
parts of, 41–42
process of creating, 42–43
rules for structuring mograms, 40–41
steps in, 43
summary, 54–55
UML profiling, 52–53
Language users
defined, 186
role of, 15–16
tasks of, 16–17
tool set of, 17–19

Language workbench
  defined, 187
  language engineering and, 19
Languages
  abstract vs. incomplete, 29
  abstraction levels and expressiveness, 27–29
  active, 174–175
  business expectations and, 31–32
  defined, 23, 186
  DSLs (domain-specific languages) compared with general languages, 33
  vs. formalisms, 16
  mograms or linguistic utterances and, 24–26
  passive, 174–175
  primitives (predefined elements) and libraries, 26–27
  raising the level of abstraction, 29–31
  ranges of abstraction and, 32–33
  summary, 38
Libraries
  Alan standard, 86, 90
  primitive and standard, 26–27
Linear structures, textual languages and, 5–6
Linguistic utterances
  deciding when it belongs to a language, 79
  defined, 187
  in definition of a language, 23
  language interfaces and, 176
  mogram as term for, 25–26
  semantics and, 133
LL parsing, 97
Look-aheads, 126
LR parsing, 97

M

Mappers, code generation and, 160–161
Mapping
  semantic, 134–135, 137
  syntactic, 42, 188
  UML class and object diagrams to graphs, 67–68
Mathematical expression language, 43–46
MDD (model-driven development), 69
  code generators for dealing with accidental complexity, 114
  domain-driven design compared with, 81
  DSMLs (domain-specific modeling languages) and, 2
  platforms and, 69–70
  tacking complexity and, 8
  transformations and, 70–73
Meaning triangle
  applying to software, 134
  overview of, 131–132
  translational semantics and, 137
Meanings, 135. *See also* Semantics
Meta-language engineer, 16
Metaclass
  binding and, 126–127
  defined, 68, 187
Metamodels, 57–74
  constraint types, 62–66
  defined, 187
  formalisms for language specification, 53–54
  foundations of, 57–58
  grammars compared with, 125–126
  graphs and, 58–61
  metamodel-to-metamodel transformation, 121–126
  models and instances, 61–62
  models conforming to, 77
  overview of, 57, 68–69
  platforms and, 69–70
  summary, 73–74
  transformations, 70–73
  UML diagrams as notation, 66–68
Microsoft Visio, 102–103
Model-driven development. *See* MDD (model-driven development)
Model transformations. *See* Transformations
Modeling languages, programming languages compared with, 3
Models
  conforming to its metamodel, 77
  defined, 61, 187
  instances of, 61–62
  metamodel as a model to specify a language, 68
  metamodel-to-metamodel transformation, 121–126
  platform independence and, 70
  vs. programs, 24–25

semantics and, 139–140
type graphs and, 58
Modularization, of code generation, 167–168
Mograms
abstract form, 78
advantages of multiple, 172
Alan CSM example, 106–107
as alternative term for linguistic utterance, 25–26
code generation and, 149
combining multiple, 171
concept of, 4
concrete form, 78–79
defined, 187
forms for creation of, 102
intermogram references, 173
model transformation and, 70
mogram/language relationship, 77
multiple mograms in one or more languages, 172–173
passive and active languages and, 174–175
role of concrete syntax and, 93
rules for structuring, 40–41
runtime system and semantic mapping, 134–135
syntactically incorrect, 80
Multiple languages, combining, 171–183
change management, 180–181
code generation, 181–182
hard and soft references, 173–174
information hiding, 175–176
intermogram references, 173
language interfaces, 176–177
multiple mograms for one application, 171–173
offered or required language interfaces, 177–179
passive and active languages, 174–175
resolving/checking references, 179–180
summary, 183
support for language evolution, 182
tool support and, 179
Multiple mograms
advantages of, 172
intermogram references, 173
for one or more languages, 172–173
passive and active languages and, 174–175
Multiple semantics, code generation, 150–151
Multiplicities, constraints and, 62–63

N

Namespaces, binding and, 126–127
Natural languages
abstract syntax and, 75
defined, 187
development of software languages and, 9–10
syntactically, but not semantically correct, 79
NLP (neurolinguistic programming), 133
Nodes (vertices), graphs composed of, 58
Nonlinear structures, graphical languages and, 5–6
Notation, guidelines for, 106
Notifications, techniques for extending or overriding generated code, 162

O

Object Management Group (OMG), 69–70
Object-oriented modeling, 81
observe operation, Alan, 83
Observer pattern, Alan, 84–85, 89–90
OCL
Alan and, 83
derived elements, 121
Octopus tool and, 157
required language interface, 178–179
simple binding and, 127
well-formedness rules, 128–129
OCR (optical character recognition), 94
Octel, code generator, 157–158
Octopus, 157
Offered language interfaces
combining multiple languages and, 177–179
parts of language specification, 42
OMG (Object Management Group), 69–70
openArchitectureWare, 179
Operational semantics
defined, 135
graphs for, 138–139
modeling, 139–140
overview of, 137–138
semantic data model, 142
semantic process model, 143

Operational semantics, *continued*
transitions, 143–144
Von Neumann architecture, 141–142
Operations, Alan, 86
Optical character recognition (OCR), 94
Optical recognition, concrete syntax, 94–96
Ordering constraint, 63

P
Parse graph, 97–98
Parser generators, 114
Parsing
comparing grammars and metamodels and, 125–126
recognition processes and, 97
tokens and, 5
transforming a parse tree into a ASG, 115
Passive languages, combining multiple languages and, 174–175
Patterns
finding for horizontal DSLs, 82–83
tacking complexity with, 8
Patterns, code generation
CreateOrFind pattern, 159–160
Mappers, 160–161
overview of, 158
Treewalker or Visitor pattern, 158–159
Platforms, metamodels and, 69–70
Plug-ins, techniques for extending or overriding generated code, 162
Posyms, 95
Pragmatic semantics
defined, 135
overview of, 135–136
Primitive libraries, 27
Primitives (predefined elements), 26–27
Process support tools, 22
Production rules. *See also* Grammars
formal language theory, 25
grammars as sets of, 48
Programming languages, modeling languages compared with, 3
Programs
defined, 187
vs. models, 24–25

Q
QVT (Query/View/Transformation), 73, 151

R
Recognition processes, concrete syntax
optical, 94–96
overview of, 94
scanning and parsing, 97
semantic analysis, 98–100
Redefinition constraint, 65
References
asm2tcsm transformations and, 117–118
binding and, 126–127
checking, 180
hard vs. soft, 173–174
information hiding and, 175–176
intermogram, 173
pragmatic semantics and, 135–136
resolving, 179–180
Required language interfaces
combining multiple languages and, 177–179
parts of language specification, 42
Roles, in language engineering, 15–22
different processes requiring different actors, 15–16
language engineer, 19–21
language user, 16–19
overview of, 15
summary, 21–22
Round-trip engineering, 163
Rules
BNF rule set in textual language creation, 114
hard-coded or model-transformation rules, 151–153
production rules. *See* Production rules
for structuring mograms, 40–41
well-formedness, 128–129
Runtime system
executing mograms and, 134–135
operational semantics and, 140–141

S
Scanning process, concrete syntax, 97
Semantic analysis, 98–100. *See also* SSA (static semantic analyzer)

Semantic data model, 142
Semantic domain model
  applying, 42
  defined, 187
  of object-oriented language, 145
  overview of, 140
  semantic data model as component of, 142
Semantic mapping
  executing mograms and, 134–135
  translational semantics and, 137
Semantic process model, 143
Semantics, 131–148
  abstract syntax as gateway to, 76–77
  code generation. *See* Code generation
  denotational, 134–135
  modeling and, 139–140
  multiple semantics in code generation, 150–151
  nature of a semantics description, 133–134
  operational, 137–139
  personal (subjective) nature of, 131–133
  pragmatic, 135–136
  semantic data model, 142
  semantic process model, 143
  simple example, 144–148
  software languages and, 134–135
  summary, 148
  symbol attributes and, 11
  syntactic vs. semantic correctness, 79
  transitions, 143–144
  translational, 136–137
  Von Neumann architecture and, 141–142
Semantics description
  defined, 187
  nature of, 133–134
Sentences, confusion regarding terminology and, 24
Serialisation syntax, 188
Service Oriented Architecture (SOA), 2
Set of positioned symbols, 95
Snapshots, operational semantics as series of, 138
SOA (Service Oriented Architecture), 2
Soft references, combining multiple languages and, 173–174
Software applications, complexity crisis and, 7–8
Software languages
  changes in, 4
  complexity crisis and, 7–8
  defined, 3, 188
  graph theory, 10
  graphical vs. textual languages, 5–6
  increasing number of, 1–2
  multiple languages. *See* Multiple languages, combining
  multiple syntaxes, 6–7
  natural-language studies and, 9–10
  semantics and, 134–135
  summary, 12–13
  support for language evolution, 182
  traditional language theory, 10
  visual-languages community, 11–12
Source-driven transformation, code generator and, 153–154
Spatial relationship graphs, 12, 97
SSA (static semantic analyzer), 126–129
  analysis with, 98
  binding and, 126–128
  checking and, 128–129
  creating generator for, 114
Standard libraries
  Alan, 86
  Alan ASM, 90
  defined, 27
Start states, transitions and, 144
State transition system, 135, 138
Statements, confusion regarding terminology and, 24
States, in abstract machines, 142–143
Static checking
  Grasland generator and, 128–129
  semantic analysis, 99–100
Static semantic analyzer. *See* SSA (static semantic analyzer)
Stream-based transformation, 72
Structure editors (syntax-directed), 100–101
Structure transformations, 72–73
Subset constraint, 66
Symbol-directed editors, 100

Symbol table
  compilers and, 96
  defined, 188
  derivation trees and, 47
Symbolic identifiers, labeled graphs, 59
Symbols
  building blocks of graphical expressions, 5
  set of positioned, 95graphical, 11
Syntax
  multiple syntaxes, 6–7
  syntactic vs. semantic correctness, 79
  syntactically incorrect mograms, 80
Syntax-directed editors, 100
Syntax mapping
  defined, 188
  parts of language specification, 42

T

Target-driven transformation, code generator and, 154
Target languages, in translations
  code generation and, 164–167
  concrete or abstract form of target, 154–156
  source-driven or target-driven translation, 153–154
  template language targeting abstract form, 157–158
Target metaclass, 126
Target platforms, code generation, 164–167
TCSMs (textual concrete syntax models), 113–129
  abstract-to-concrete transformation, 117–121
  Alan example, 118–120
  Grasland generator, 113–116
  metamodel-to-metamodel transformation, 121–126
  static semantic analyzer, 126–129
  summary, 126–129
tcsm2bnf transformation
  designer input to, 122–126
  generating BNFset, 121–122
  overview of, 115, 121–126
Template languages
  hard-coded or model-transformation rules, 151–153
  targeting abstract form, 157–158
Text editors, 100
Textual concrete syntax model. *See* TCSM (textual concrete syntax model)
Textual-graphical mismatches (blind spots), 101–103
Textual languages
  context-free grammars and, 49
  vs. graphical languages, 5–6
  textual-graphical mismatches (blind spots), 101–103
  traditional language theory, 10
TogetherJ, 6
Tokens, parser design, 5
Tool generators, language engineers and, 20–21
Tool support, multiple languages and, 179
Traditional language theory, 10
Transformations, 70–73
  asm2tcsm and tcsm2bnf, 115
  building code generator and, 151–153
  CMS to ASM, 109–110
  code generation. *See* Code generation
  formal notion of a model and, 70–71
  graph transformations, 144
  metamodel-to-metamodel, 121–126
  taxonomy of, 71–72
Transformers
  tool generators and, 20–21
  in tool set of language user, 17–18
Transitions, semantics, 143–144
Translational semantics
  code generation as form of. *See* Code generation
  defined, 135
  overview of, 136–137
Traversal algorithms, 159
Trees, as directed graphs, 50
Treewalker pattern, code generation, 158–159
Treewalkers, creating textual languages and, 114–115
Type graphs
  defined, 60
  models as, 58, 61–62
Types
  Alan, 85
  Alan ASM, 86–87
TypeVar class, Alan ASM, 86

U
UML (Unified Modeling Language)
Alan and, 83
concrete representation of UML association class, 104
example of language interface, 177
examples of multiple syntax languages, 6
generating Java application from, 160
mapping class diagrams to graphs, 67
mapping object diagrams to graphs, 67–68
metamodeling and, 53
OCL requiring language interface for, 178–179
Octopus tool and, 157
profiling, 52–53
Underlying descriptor, abstract syntax, 76
Unified Modeling Language. *See* UML (Unified Modeling Language)
Unifying descriptor, abstract syntax, 76
Union constraint, 66
Uniqueness constraint, 63–64
Unshared constraint, 65
User groups, DSLs, 34–35

V
Value graphs, 140, 142
Values, semantic data model, 142
Vector graphics editors, 100
Velocity templates, 73
Vertices (nodes), graph, 58
View transformations, 72
Visio, 102–103
Visitor pattern, code generation, 158–159
Visual languages. *See* Graphical languages
Von Neumann architecture, 141–142

W
Web applications, complexity crisis and, 7
Well-formedness rules, 128–129. *See also* Static checking

X
xText, 179