Reflections on Courses for Software Language Engineering

Anya Helene Bagge[1], Ralf Lämmel[2], and Vadim Zaytsev[3]

[1] Bergen Language Design Laboratory, University of Bergen, Norway
[2] Software Languages Team, University of Koblenz-Landau, Germany
[3] Institute of Informatics, Universiteit van Amsterdam, The Netherlands

Abstract. Software Language Engineering (SLE) has emerged as a field in computer science research and software engineering, but it has yet to become entrenched as part of the standard curriculum at universities. Many places have a compiler construction (CC) course and a programming languages (PL) course, but these are not aimed at training students in typical SLE matters such as DSL design and implementation, language workbenches, generalized parsers, and meta-tools.

We describe our experiences with developing and teaching software language engineering courses at the Universities of Bergen and Koblenz-Landau. We reflect on lecture topics, assignments, development of course material, and other aspects and variation points in course design.

1 Introduction

Software Language Engineering (SLE) is concerned with design, implementation, composition, evolution, and other aspects of software languages and language-aware software components. SLE uses language models which break down into definitions of syntax, well-formedness, semantics, etc. Language implementation is concerned with parsing, semantic analysis, translation, IDE support, etc.

SLE is yet to become entrenched as part of the standard curriculum at universities. Many places have a compiler construction (CC) course and a programming languages theory (PLT) course, but these are not aimed at training students in typical SLE matters such as DSL design and implementation, language workbenches, generalized parsers, and meta-tools. In this paper, we describe our experiences with developing and teaching software language engineering courses at our universities, reflect on course material, and discuss variations on course design.

Here is a summary of lessons learned:

- SLE scenarios involve various artifacts (grammars, programs, rewrite systems) and data flows (parser generation, parser compilation, parser application), which needs comprehension support. We found *megamodeling* to be effective in this context.
- SLE shares much of its foundations with model-driven engineering (MDE) and CC. Further, SLE depends on foundations from PLT. Of course, SLE also assumes foundations from general software engineering (SE). Managing these dependencies in actual course designs may require substantial efforts.

Topic	**Learning objective** [2]
Introduction	List principles of SLE; summarize features of a language
Syntax, grammars, parsing	Understand syntax; engineer grammars; (re)use parsers
Domain-specific languages	Recognize a DSL in a realistic setting; recognize its domain
Scannerless parsing, layout	Understand tradeoffs between different technologies
Evaluator-based semantics	Apply language engineering; recognize language semantics
Scoping and environments	Understand scoping; implement it
Type checking	Understand and implement semantic analysis
Tree traversal and rewriting	List different approaches to language processing
Term project introduction	Design and plan an SLE experiment
Formal semantics	Recall basics of formal semantics
Term project presentations	Present, discuss and analyse projects

Table 1. Condensed lecture plan for the Bergen course (Autumn semester 2013). Some of the topics ran over multiple lectures.

- SLE, as a teaching area, is more obviously addressed by 'presence courses'; it is less obvious how to apply eLearning or MOOC in this context because the engineering aspect of SLE and richness of related technologies and methods calls for project work including student presentations and discussion.

Roadmap. The rest of the paper provides details on our SLE course experiences. §2 contains facts about two SLE courses in which the authors were involved as teachers. §3 discusses the use of megamodels in SLE courses. §4 makes explicit a number of variation points for SLE course designs. §5 synthesizes guidelines from our experiences. §6 discusses related work in the context of CS education.

2 Data points on SLE courses

2.1 SLE in Bergen

Position in curriculum. The course fills the slot of the old traditionalistic CC course in Bergen, keeping the same title (*Introduction to Program Translation*), but being more of an introduction to software languages and software language engineering. The course is 10 ECTS credits, being optional but available to students at all levels. In terms of dependencies, we recommend that the students have a basic computer science background, including a course on programming paradigms. Unfortunately, there is currently no course offered on PLT or semantics, which could given the students useful background for the SLE course. The course runs every second year (two actual runs so far: 2011 and 2013).

Overall format. The course is centered around lectures and lab exercises, per week we have two 90 minute lectures and one 90 minute lab; the rest is self-study. Rascal [11] is used as the main programming language throughout, apart from the term project where students are free to pick their own technologies.

The course plan is loosely structured, and is adapted to the needs and interests of the students. Given the students' weak background in programming languages, we usually spend a fair amount of time filling this gap.

Topic	Technologies
Small imperative/OO language with interpreter	Handwritten Visual C++
C−− with ARM backend	Handwritten C
Language operational semantics and prototyping	Prolog
Language implementation for LLVM	OCaml
Forth-like language	Haskell
Code formatting	Java and PGF [3]
Code formatting	C
Code arena game	Xtext, Java, libGDX
Natural-like language for car navigation	F# + homemade parser combinators

Table 2. Some of the students' self-selected term projects for the Fall 2013 course.

Lectures. The lectures contain a mix of theory and practice, with theory explained on the blackboard and in discussions, and then illustrated by example in live coding sessions. The code is then made available to the students, as a basis for or complement to exercises.

The 2013 lecture plan (sketched in Table 1) includes 22 lectures, of which six are dedicated to recap and term project presentations and discussions. In addition to covering fundamental topics, the first few lectures also showcase language-based modeling and engineering using invited speakers.

The blackboard is used for theory and smaller examples, and for giving big picture overviews of concepts and relationships (megamodels). Live coding is semi-interactive, with feedback and suggestions from the students. Seeing a theoretic concept immediately implemented in practice seems helpful to students' understanding — though perhaps an even better approach would be to do this in a lab setting where the students could experiment on their own rather than watching the lecturer.

Assignments. Lab work consists of a mix of non-compulsory weekly exercises, designed to help understand and retain the topics discussed in the lectures, and a compulsory term project towards the end of the semester, where the students typically end up with a full small-scale software language implementation.

The weekly exercises include topics such as regular expressions, grammars and parsing, and language implementation — usually fitting together with whatever is covered in the lectures.

The term project is completely open-ended (in the 2013 edition), with a free choice of topic and technology. Table 2 sketches some of the projects the 2013 students chose. Roughly a third of the course was spent on the project.

Grading. We have a 45–60 minute oral exam, beginning with a presentation and discussion of the term project, and concluding with theoretical questions. With the open-ended term project, assessing and comparing the students' work proved quite difficult, so rather than grading the end result, we focused more on the students' insight into their language engineering efforts and their ability to reflect on design choices.

Material. For the 2011 edition of the course, we used Scott's *Programming Language Pragmatics* [18] as well as lecture notes. Scott's book was not a perfect fit for the course, particularly with regard to technology choices, so the 2013

Topic	**Learning objective** [2]
Introduction	Recognize SLE in the computer science context
An SLE bibliography	Summarize landmark SLE literature
A DSL primer	Carry out modeling of DSLs
Grammars and parsing	Implement parsers / frontends of language processors
Megamodeling	Recognize and navigate the linguistic view on technologies
Attribute grammars	Implement semantic analysis for languages
Rewriting & strategies	Implement software transformations in language processors
Automated refactoring	Recall challenges of refactoring across languages
Grammar-based testing	Understand grammar-aware testing of language processors
Code generation	Summarize an architecture of a compiler backend

Table 3. Lectures of the SLE course in Koblenz (Winter semester 2013/14)

edition was based entirely on the lectures and associated notes. Additionally, we have developed a glossary of SLE terms, which was also made into flash cards for exam preparation. All course resources are available online: `http://bitbucket.org/anyahelene/inf225public/wiki`.

2.2 SLE at Koblenz

The course — `http://softlang.wikidot.com/course:sle` — has been held 4 times by now, but in its current, now stabilized form it has been held only twice.

Position in curriculum. The SLE course is a 6 ECTS optional module for CS students. In Koblenz, optional modules are open to BSc and MSc students, but MSc students are better prepared for this course, as they would have completed a compulsory BSc module on PLT, which is typically taken only towards the end of the BSc. The SLE course was designed to replace a conservative CC course. A modern CC course would still be a good complement, but there are currently no additional resources available for it.

Overall format. The 6 ECTS of the course are evenly distributed over lectures and labs. Thus, there is one lecture and one lab per week over the duration of a semester with 60–90 minutes for each meeting. There are three to four assignments over the duration of the course; two involve software development, one or two are reading assignments. The results of all assignments are presented by the students and discussed in depth at the lab meetings. The development-oriented assignments use a Scrum-like format (again scheduled for the lab meetings) to discuss plans, intermediate results, and open issues. There is no exam; instead the assignment results are used for grading. The interactive format works best for smaller numbers of students: 10-15. If we were taking more students, the amount of student presentations would be hard to handle and the necessary number of variations on a given assignment would be hard to design.

Lectures. In Table 3, the topics of the lectures and a short indication of the associated learning objectives are shown for the most recent edition of the course. A few things are worth pointing out. The lecture titled *An SLE bibliography* surveys SLE literature briefly to introduce the students to the SLE area in a literature-based manner. The lecture is based on annotated SLE bibliogra-

Classifier	**Topic**	**Duration**
Reading	Seminal literature on SLE	2 week
Development	DSL implementation	3 weeks
Reading	Grammar-based testing	1 week
Development	Grammar-based testing	3 weeks

Table 4. Assignments of the SLE course in Koblenz (Winter semester 2013/14)

phy [13]. Students drill into the literature on the grounds of a reading assignment. The lecture titled *Megamodeling* has a rich function in the SLE course: i) Megamodels help understanding language definitions and language processors in terms of constituents, dependencies, data flows, and other relationships. ii) Megamodeling notation gives rise to an advanced example of a domain-specific modeling language, which can then also be studied in itself in the course.

Assignments. In Table 4, the assignments are briefly characterized for the most recent edition of the course. Students work individually or they team up with one partner. The idea is that the assignments do not just help with technical skills in SLE, but they also contribute to soft and research skills more generally. Thus, reading, presentation, and discussion are important elements of the course. The feedback by the students regarding this format has been consistently positive. An assignment solution for a test generator of the recent course edition was eventually refined and could be published at the SLE 2014 conference [10].

Grading. As mentioned before, there is no exam. The assignment results are used for grading. In our experience, the students highly appreciate this form of course completion because it motivates them to invest into the assignments and they do not face another exam at the end of semester. Given the breadth of topics in the course, a conservative exam would be a design challenge anyhow.

Material. The course does not leverage any specific textbook or small set thereof. Instead, the course leverages a small annotated bibliography [13], as mentioned before. The corresponding papers are also available online. Further, the course leverages course material (mainly slides), as collected from colleagues who run and ran courses with SLE relevance. These resources have been organized online so that others can take advantage of them as well at `http://slecourse.github.io/slecourse`. Contributions are highly welcome.

3 Megamodels in SLE courses

We have found that the architecture of language processors and the underlying technologies as well as some SLE processes can be described at a high level of abstraction by means of megamodels [4]. This finding is partially based on our own investigations into megamodeling [8,20] in the context of SLE and the broader context of software technologies. Informally, we use megamodels to impose structure on SLE matters in a broad sense in a manner similar to the use of tombstone diagrams in CC courses.

In Figure 1 and Figure 2, we show two megamodels as they play a role in the SLE courses. The model at the top appears in the parsing lecture of the Koblenz course. It is used to explain the basic data flow around parser generation and

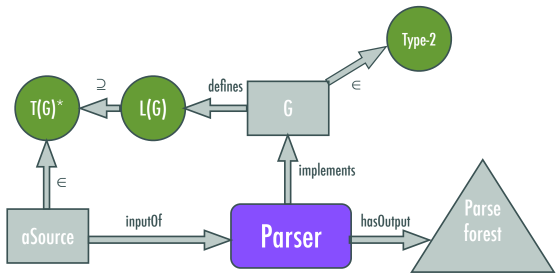

Fig. 1. A megamodel used in an SLE course: basic notion of a grammar-derived parser

invocation. It also adds a few conceptual characteristics such as the expectations that we start from a context-free (type 2) grammar. We also expresss that the input does at least agree with the terminal set of the grammar. Practically, this assumption may not hold, but the megamodel provides a good abstraction for discussing such issues. Likewise, the indication of a parse forest is merely meant to allow for the discussion of ambigious grammars and the different possible strategies of parsing technologies.

The megamodel at the bottom provides a highly conceptualized view on parsing, unparsing, and related operations or parts thereof. Various representation levels (such as strings, tokens, ASTs) and various phases of parsing are called out. In a given language implementations, several nodes and edges in the model are irrelevant, but the megamodel allows for a systematic discussion of options in language implementation and associated properties such as bidirectionality.

4 Variation points for SLE courses

Over the timeline and over the different courses, we experienced some variation points for SLE courses that we discuss here.

Overlapping with MDE and CC. Conceptually, SLE is close to CC and MDE. For instance, the concepts of parsing, semantic analysis, intermediate representations, and code generation are integral parts of a typical CC course, which should also play a role in an SLE course. Likewise, the concepts of metamodeling, model transformation, and traceability are integral parts of the MDE body of knowledge, which should also be covered in an SLE course.

However, it is practically impossible that an SLE course would cover both CC and MDE. Historically, CC courses cover grammar classes and associated parsing algorithms in much detail. They also cover non-trivial foundations of optimization (e.g., flow analysis) and code generation (e.g., BURS). In contrast, an SLE course should favor one or two state-of-the-art practical parsing approaches (such as LL(*) and generalized LL/LR parsing) without in-depth coverage of the involved algorithms. Likewise, an SLE course should only have cursory coverage, if any, of compiler optimization and code generation. Also, an SLE course can-

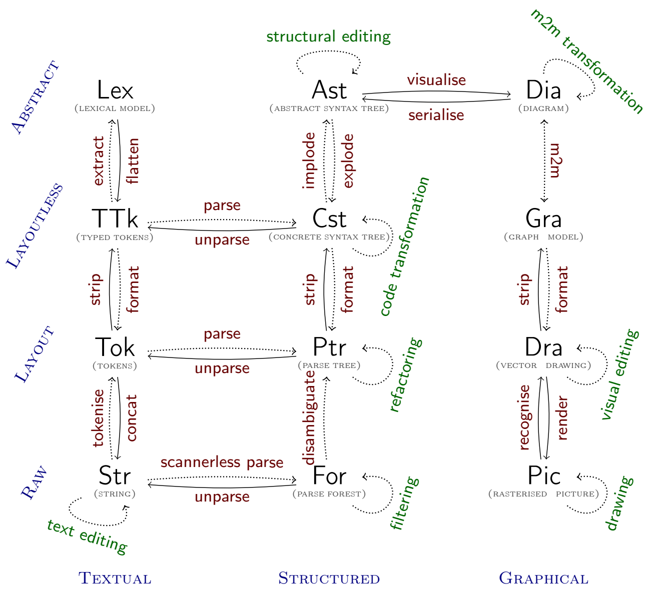

Fig. 2. A megamodel used in an SLE course: parsing, unparsing and friends [20]

not include an introduction to MDE, but MDE technologies could be leveraged in an SLE course, if students have some prior understanding of metamodeling. For instance, MDE-based technologies for language implementation, e.g., Xtext, could be used in addition to technologies from the CC and PLT communities.

Less versus more technologies. The Koblenz course has used multiple technologies. In particular, for the development assignments, students were given technology options and they could even suggest alternative routes. The Bergen course uses now Rascal [11] as the main technology in lectures.

Staying with a single or a small set of technologies has a number of advantages: (1) the technologies are (hopefully) well-known by the teaching staff, who can then guide the students in their use; (2) the students have a chance to gain a deep enough understanding to actually use the selected technologies properly; (3) lab setup and exercise design becomes somewhat easier.

Covering a wider range of technologies also has its appeal. Firstly, students become aware of the broad offering of SLE tools and technologies. Secondly, if the students use different tools, they can exchange experiences and insights — perhaps broadening the horizon of the lecturer as well.

Less versus more theory prerequisites. A typical Koblenz student has taken a course on PLT and is familiar with functional programming (FP). A typical Bergen student has taken a programming paradigms course which includes a brief introduction to functional and logic programming, but there is no counterpart for syntax, semantics, and type systems of programming languages.

SLE relies on concepts such as context-free grammars, scoping, types, stores, and interpreters. Thus, if students lack such background, then an SLE course must definitely spend some time on these concepts. However, an applied and selective approach is applicable here. For instance, a pragmatic approach to type checking and operational semantics-inspired interpretation is possibly sufficient.

Exercises versus projects. By *exercises* we mean weekly assignments complementing one another and expanding on the topics covered in the lectures. By *projects* we mean substantially bigger assignments that may run over several weeks and involve extra milestones such as intermediate and final presentation.

Both in Bergen and Koblenz, we mainly use projects. In fact, exercises are not used at all in Koblenz, but reading assignments provide another category. The reasoning is here that an SLE course is similar to a practical SE class, where actual designs and implementations must be developed, the underlying technologies selected, and students must familiarize themselves with the technologies. This scope typically implies projects, except for the more basic notions such as interpretation, which could be covered with the exercise format.

Less versus more constrained assignments. Should assignments specify a specific SLE problem? An example of a more constrained situation is the first development-oriented assignment of the Koblenz course (§2.2): students had to implement a given DSL FSML—a language for finite state machines [12]. A detailed specification including a well-defined list of language processing components was used as a requirement specification. An example of a largely unconstrained situation is an open term project that was used in the latest edition of the Bergen class. Students could suggest an SLE-related project that they wanted to work on.

According to the evaluation, Bergen students were happy with the freedom given, and felt that they learned a lot. In particular, several of the students learned hard lessons about the value of using specialized tools and languages rather than implementing things by hand. We know that open assignments are harder to complete and to grade, since they pose greater strain on the evaluative skills [17] and do not always map teacher's tasks to students' goals and tactics [15]. It also seems that open assignments come with a higher probability of failure — a student may have picked an over-ambitious project or an ill-suited technology. This aspect can be somewhat addressed through supervision and it has not been a serious issue in the end for all the course editions we have run.

Less versus more conservative exams. We take a hands-on approach to SLE in our courses. The lectures do provide conceptual knowledge, but we consider it most important that students can design and implement language processors. Thus, a conservative exam which would be focusing on definitions, syntax, type systems, semantics, programming in the small, and engineering methods or process would be ill-suited for the hands-on orientation. Thus, project assignments are to be part of the grading scheme. In Koblenz, the course has evolved to the point that projects are the sole input for the final grade. To this end, we place high priority on the presentations per assignment, where students need to make an effort to connect to general concepts from the lecture and discuss their approach at a high level of abstraction.

5 SLE course design guidelines

- The content of a modern SLE course spans over a range of technologies and approaches. This places higher demands on both teaching staff and students. We are beginning to cope with this complexity by leveraging students' curiosity and letting them choose, prepare, and present study units.
- Special techniques are needed for explaining the state of the art in SLE, without overwhelming students with the complexity of modern technologies, languages, and frameworks. We found megamodeling and megamodel renarration to be useful tools for comprehension and comparison.
- SLE is an advanced course, and thus its design should be context-aware. For example, in a curriculum which covers model-driven or model-based topics, a modelware-centric instantiation of SLE is appropriate. In another curriculum which covers compiler construction, a grammarware-centric instantiation is appropriate.
- As typical for software engineering, certain SLE problems manifest easier on a large scale and may require significant investments by the students to master the involved methods and technologies. Hence, an SLE course should favor bigger and open-ended projects over strict-portioned exercises.
- Grading in an SLE course should not focus on the deliverables developed in project work, especially when the students may explore different tasks and technologies, as assessment would be hard and possibly unfair. Instead, the focus should be on assessing the gained insights and acquired skills.
- An SLE course should not be seen or designed as replacing or subsuming PLT, CC, and MDE courses. Instead, these different courses may complement each other, while the SLE course may even function as a meeting point where those domains can be connected in a broader sense.
- The most basic SLE methods are parsing, analysis, interpretation, translation, and generation. Curricula are ideally tuned so that some elements of these methods can be effectively prepared by courses on formal languages and programming language theory or paradigms, if not introduction to computer science, so that an SLE course can focus on engineering aspects, e.g., conflict or ambiguity resolution in parsing.

6 Concluding remarks

Some modern CC courses successfully persist in following the "let's make a compiler" paradigm [1], but most are structured around learning progression [14,16] (performing the same educational tasks several times on increasingly difficult languages) or project-based education [7,9] (students participate in a project spanning over multiple courses providing scopes to complete its components). In these projects, the notion of a compiler may be generalized: it is used as an incidental well-studied software system [6]; diverse and broad examples of translators may be offered [5]; students may also pick a compiler/translator problem themselves [19].

There are currently no textbooks that can form a sole basis for an SLE course. We have made an attempt to systematically explore the methods and technologies of SLE in order to create a designated SLE course that could replace or complement courses for CC, PLT, and MDE in a CS/SE curriculum. Course designs elsewhere show the blurring boundaries between the subjects [14], but none so far have explicitly converged to SLE as a meeting point for these domains.

References

1. A. V. Aho. Teaching the Compilers Course. *SIGCSE Bulletin*, 40:6–8, Nov. 2008.
2. L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, P. R. Pintrich, J. Raths, and M. C. Wittrock. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2000.
3. A. H. Bagge and T. Hasu. A Pretty Good Formatting Pipeline. In *SLE*, volume 8225 of *LNCS*, pages 177–196. Springer, October 2013.
4. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *OOPSLA/GPCE: Best Practices for MDSD*, 2004.
5. R. Bodik. Small Languages in an Undergraduate PL/Compiler Course. *SIGPLAN Notices*, 43(11):39–44, Nov. 2008.
6. S. Debray. Making Compiler Design Relevant for Students Who Will (Most Likely) Never Design a Compiler. *SIGCSE Bulletin*, 34(1):341–345, Feb. 2002.
7. A. Demaille. Making Compiler Construction Projects Relevant to Core Curriculums. In *ITiCSE*, pages 266–270. ACM, 2005.
8. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *MoDELS 2012*, pages 151–167. Springer, 2012.
9. W. G. Griswold. Teaching Software Engineering in a Compiler Project Course. *Journal on Educational Resources in Computing*, 2, Dec. 2002.
10. J. Härtel, L. Härtel, and R. Lämmel. Test-data Generation for Xtext. In *SLE 2014*, volume 8706 of *LNCS*, pages 342–351. Springer, 2014.
11. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *GTTSE 2009*, volume 6491 of *LNCS*, pages 222–289. Springer, Jan. 2011.
12. R. Lämmel. Another DSL primer. Technical Documentation. Version 0.00003 as of 25 December 2013. Available online at `http://git.io/fsml`, 2013.
13. R. Lämmel. Yet another annotated SLEBOK bibliography. Available online `https://github.com/slebok/yabib`, 2013.
14. J. Mead. A Compiler Tutorial Scaled for the Programming Languages Course. In *SIGCSE'06*, pages 32–36. ACM, 2006.
15. D. J. Nicol and D. Macfarlane-Dick. Formative Assessment and Self-regulated Learning: a Model and Seven principles of Good Feedback Practice. *Studies in Higher Education*, 31(2):199–218, Apr. 2006.
16. D. Rayside. A Compiler Project with Learning Progressions. In *ICSE Companion*, pages 392–399. ACM, 2014.
17. D. R. Sadler. Formative Assessment and the Design of Instructional Systems. *Instructional Science*, 18(2):119–144, 1989.
18. M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2009.
19. W. M. Waite, A. Jarrahian, M. H. Jackson, and A. Diwan. Design and Implementation of a Modern Compiler Course. *SIGCSE Bulletin*, 38(3):18–22, 2006.
20. V. Zaytsev and A. Bagge. Parsing in a Broad Sense. In *MoDELS*, 2014.