Curriculum for

Certified Professional for Software Architecture (CPSA)®
Advanced Level

Module
DSL

Domain-Specific Languages

2023.1-RC1-EN-20230323

Table of Contents

List of Learning Goals . . . 2
Introduction: General information about the iSAQB Advanced Level . . . 3
What is taught in an Advanced Level module? . . . 3
What can Advanced Level (CPSA-A) graduates do? . . . 3
Requirements for CPSA-A certification . . . 3
Essentials . . . 4
What does the module “DSL” convey? . . . 4
Curriculum Structure and Recommended Durations . . . 4
Duration, Teaching Method and Further Details . . . 5
Prerequisites . . . 5
Structure of the Curriculum . . . 5
Supplementary Information, Terms, Translations . . . 5
1. Intro and Motivation . . . 7
1.1. Terms and Principles . . . 7
1.2. Learning Goals . . . 7
1.3. References . . . 7
2. Syntax . . . 8
2.1. Terms and Principles . . . 8
2.2. Learning Goals . . . 8
2.3. References . . . 8
3. Semantics . . . 9
3.1. Terms and Principles . . . 9
3.2. Learning Goals . . . 9
3.3. References . . . 9
4. Language Design . . . 10
4.1. Terms and Principles . . . 10
4.2. Learning Goals . . . 10
4.3. References . . . 11
5. Tools . . . 12
5.1. Terms and Principles . . . 12
5.2. Learning Goals . . . 12
5.3. References . . . 13
6. Examples . . . 14
6.1. Terms and Principles . . . 14
References . . . 15

List of Learning Goals

- LG 1-1: Requirements and Architectural Relevance
- LG 1-2: Definitions
- LG 1-3: Basics of modelling
- LG 1-4: Embedded vs. Stand-Alone DSLs
- LG 2-1: Syntax and language design
- LG 2-2: Formal grammars
- LG 2-3: Concrete and abstract syntax
- LG 3-1: Basics of semantics
- LG 3-2: Expressiveness
- LG 3-3: Design of semantics
- LG 4-1: General Design Issues
- LG 4-2: Type systems
- LG 4-3: Compositional Domain Modeling
- LG 4-4: Effects
- LG 5-1: Syntax tools
- LG 5-2: Semantics tools
- LG 5-3: Languages
- LG 5-4: Development Environments

Introduction: General information about the iSAQB Advanced Level

What is taught in an Advanced Level module?

- The iSAQB Advanced Level offers modular training in three areas of competence with flexibly designable training paths. It takes individual inclinations and priorities into account.
- The certification is done as an assignment. The assessment and oral exam is conducted by experts appointed by the iSAQB.

What can Advanced Level (CPSA-A) graduates do?

CPSA-A graduates can:

- Independently and methodically design medium to large IT systems
- In IT systems of medium to high criticality, assume technical and content-related responsibility
- Conceptualize, design, and document actions to achieve quality requirements and support development teams in the implementation of these actions
- Control and execute architecture-relevant communication in medium to large development teams

Requirements for CPSA-A certification

- Successful training and certification as a Certified Professional for Software Architecture, Foundation Level® (CPSA-F)
- At least three years of full-time professional experience in the IT sector; collaboration on the design and development of at least two different IT systems
  - Exceptions are allowed on application (e.g., collaboration on open source projects)
- Training and further education within the scope of iSAQB Advanced Level training courses with a minimum of 70 credit points from at least three different areas of competence
  - existing certifications (for example: Sun/Oracle Java architect, Microsoft CSA) can be credited upon application
- Successful completion of the CPSA-A certification exam

Essentials

What does the module “DSL” convey?

Many application domains use complex, user-defined rules, flows or entities. The primary implementation language of a project is not always appropriate for expressing these aspects. This may be the case when:

- these aspects are user-configurable
- the implementation language is too expressive to guarantee properties such as safety, run time or termination statically
- the implementation language is not sufficiently expressive to describe these aspects readably and comprehensibly.

In such cases, devising a custom language for these aspects helps manage complexity. The resulting language is a *domain-specific language* (or *DSL* for short). Well-designed DSLs

- contribute to separation of concerns by decoupling description from implementation,
- empower users to solve problems using the software that would otherwise require developers to extend the software first,
- enable low-code approaches that allow solving problems with less code than would be possible using the implementation language of the project, and
- lay the foundations for some architectural quality goals, such as adaptability, modifiability, analysability, and security.

Architects designing and implementing DSLs can draw from a large body of both scholarly material and practical experience in programming language design and compiler construction. This module gives an introduction to the most important aspects of this material. It enables them to

- understand where a DSL fits into an overall architecture,
- design useful and user-friendly DSL in a systematic fashion, and
- consider DSLs as an integral technique in architecture design.

Curriculum Structure and Recommended Durations

Content	Recommended minimum duration (minutes)
1. Intro and Motivation	60
2. Syntax	180
3. Semantics	300
4. Language Design	300
5. Tools	180
6. Examples	60
Total	1080 (18h)

Duration, Teaching Method and Further Details

The times stated below are recommendations. The duration of a training course on the DSL module should be at least **3** days, but may be longer. Providers may differ in terms of duration, teaching method, type and structure of the exercises, and the detailed course structure. In particular, the curriculum provides no specifications on the nature of the examples and exercises.

Licensed training courses for the DSL module contribute the following credit points towards admission to the final Advanced Level certification exam:

Methodical Competence:	**10** Points
Technical Competence:	**10** Points
Communicative Competence:	**10** Points

Prerequisites

Participants **should** have the following prerequisite knowledge:

- basic computer-science education related to languages, i.e.
  - formal grammars
  - regular expressions
  - difference between interpreters and compilers
- the role of types in programming

Knowledge in the following areas may be **helpful** for understanding some concepts:

- familarity with the Chomsky hierarchy
- compiler construction
- tactical domain-driven design
- combinator-library design
- functional softare architecture

Structure of the Curriculum

The individual sections of the curriculum are described according to the following structure:

- **Terms/principles**: Essential core terms of this topic.
- **Teaching/practice time**: Defines the minimum amount of teaching and practice time that must be spent on this topic or its practice in an accredited training course.
- **Learning goals**: Describes the content to be conveyed including its core terms and principles.

This section therefore also outlines the skills to be acquired in corresponding training courses.

Supplementary Information, Terms, Translations

To the extent necessary for understanding the curriculum, we have added definitions of technical terms to

the iSAQB glossary and complemented them by references to (translated) literature.

1. Intro and Motivation

Duration: 60 min	Practice time: 0 min

1.1. Terms and Principles

parser, compiler, interpreter, syntax, semantics

1.2. Learning Goals

LG 1-1: Requirements and Architectural Relevance

- Know possible requirements and conditions that might make a DSL a sensible architectural decision, such as an insufficiently expressive implementation language, the need for complex user-defined rules, the need to restrict behavior of parts of the software, or the need to make parts of the software analysable in non-trivial ways.
- Understand that the design of a DSL - including its syntax - may have far-reaching consequences for a project, as it will impact its intended quality goals, including its usability, and will be harder to change the more code is written in the DSL.
- Know advantages and disadvantages of using a custom DSL over an existing programming or configuration language.

LG 1-2: Definitions

- Know the basic terms in the context of domain-specific languages: parser, compiler, interpreter, syntax, lexical syntax, abstract syntax, semantics, run-time, compile-time.
- Know the taxonomy of DSLs regarding types, embedded/stand-alone, interpreted/compiled.
- Understand the difference between a general-purpose programming language and a domain-specific language.

LG 1-3: Basics of modelling

- Understand the connection between the business domain and the design of a domain-specific language.
- Know the basic principles of Domain-Driven Design and how they relate to domain-specific languages.
- Know how to identify possible applications of domain-specific languages in a given business domain.

LG 1-4: Embedded vs. Stand-Alone DSLs

- Understand that there is a spectrum between library design and language design, with no clear boundaries.
- Understand that designing and implementing a DSL does not have to be a singular decision, but can progress gradually along that spectrum until an economic and organizational sweet spot is reached.

1.3. References

[Aho et al. 2006], [Evans 2003], [Nystrom 2021], [Wąsowski and Berger 2023], [Ghosh 2010]

2. Syntax

Duration: 120 min	Practice time: 60 min

2.1. Terms and Principles

host language, grammar, parsing, abstract syntax, Lisp

2.2. Learning Goals

LG 2-1: Syntax and language design

- Understand that syntax is a matter of both ergonomics and taste, and thus the subject of requirements.
- Know that it is possible to adopt the syntax of an existing language.
- Understand that it is possible to embed a DSL into a host language, avoiding (part of) syntax design altogether.
- Understand the role of macros in designing domain-specific syntax in Lisp-like languages.

LG 2-2: Formal grammars

- Understand the difference between lexical and structural syntax.
- Understand regular and context-free grammars.
- Understand the role of parsers, and restrictions on grammar structure imposed by parsers.
- Understand the difference between LL and LR parsing algorithms.
- Know the Chomsky hierarchy.

LG 2-3: Concrete and abstract syntax

- Understand the role of abstract syntax in processing DSL language elements.
- Understand the difference between concrete and abstract syntax.
- Understand the difference between traditional syntax of text-based languages along with the associated tooling, and projectional editing.
- Understand the stratified design of syntax in Lisp-like languages.

2.3. References

[Wilhelm et al. 2021], [Leermakers 1993], [Culpepper et al. 2019], [Völter et al. 2014], [Wąsowski and Berger 2023], [Ghosh 2010]

3. Semantics

Duration: 180 min	Practice time: 120 min

3.1. Terms and Principles

specification, implementation, denotational semantics, operational semantics, big-step semantics, small-step semantics, Lambda Calculus, Turing machines, rewriting systems, finite state machines

3.2. Learning Goals

LG 3-1: Basics of semantics

- Understand the difference between compilers and interpreters.
- Understand the independence of a language's semantics and its (multiple) implementations.
- Understand the difference between a specification and an implementation.
- Know the basic functionality of a JIT compiler that can work both as compiler and interpreter, and how it can be leveraged for DSLs, for example GraalVM.

LG 3-2: Expressiveness

- Know various computation models, such as Lambda Calculus, Turing machines, rewriting systems, and finite state machines.
- Understand the principle of least power, in which the least expressive computation model should be chosen to represent the business domain; and when to deviate from the principle.
- Understand the connection between expressiveness and compilation (or interpretation).
- Can isolate static and dynamic parts of computation.

LG 3-3: Design of semantics

- Know various types of semantics, such as denotational and operational semantics, as well as big- and small-step semantics.
- Understand the connection between the semantics of the DSL and the semantics of the target languages (compilation) and/or host languages (interpretation).
- Know how to define a semi-formal semantics of a DSL using a prose representation.

3.3. References

[Nipkow and Klein 2014], [Baader and Nipkow 1998]

4. Language Design

Duration: 180 min	Practice time: 120 min

4.1. Terms and Principles

types, overloading, compositionality, combinator library, effect, continuation

4.2. Learning Goals

LG 4-1: General Design Issues

- Know the difficulty of good language design, and its impact on architectural decisions, in particular whether to implement an embedded or stand-alone DSL, and whether to re-use design or implementation elements from existing languages.
- Understand the impact of changes in language design on a software system, and how they relate to changes elswhere in an existing architecture.

LG 4-2: Type systems

- Understand the basics of type systems, including type inference.
- Know various design criteria for a type system and practical examples, such as Java and Standard ML.
- Know how to leverage a host or target language's type system to build a type system for a DSL, and alternatively know the basic principles behind designing a dedicated type system.
- Understand the trade-offs between a typed and untyped DSL.
- Understand the concept of overloading, specifically as it applies to primitive values (overloading integers, etc.), and its role in designing embedded DSLs.

LG 4-3: Compositional Domain Modeling

- Understand the concept of compositionality: that the meaning of a compound entity only depends on the meaning of its components, not their internal structure.
- Understand the value of compositionality for domain modelling.
- Understand the relationship between the ideas of "closure of operations" in Domain-Driven Design and compositionality.
- Know construction principles for (compositional) combinator libraries.
- Understand how compositionality (or its absence) manifests itself in DSL design.
- Know examples from real projects where compositionality is a crucial aspect of DSL design.

LG 4-4: Effects

- Understand the concept of effects.
- Know examples for reader, writer, state, and control effects.
- Understand the concept of a continuation.

- Know typical examples of effects in typical DSLs.
- Know implementation techniques for effects, specifically some that allow dependency injection.
- Understand the relationship between effects and hexagonal architecture.

4.3. References

[Pierce 2002], [Peyton Jones and Eber 2003], [Elliott and Hudak 1997], [Pretnar 2015], [Alama 2020], [Wąsowski and Berger 2023], [Ghosh 2010]

5. Tools

Duration: 120 min	Practice time: 60 min

5.1. Terms and Principles

interpretation, code generation, macro, staging, hygiene

5.2. Learning Goals

LG 5-1: Syntax tools

- Understand the mode of operation of parser and scanner generators.
- Know at least one tool for generating parsers, or an embedded DSL for parsers, such as Bison, Essence, ANTLR, or Parsec.

LG 5-2: Semantics tools

- Understand the relationship between interpretation and code generation.
- Know how to establish an informal connection between the desired semantics of a DSL and the actual implementation, such as via automated and systematic testing, or tools provided by a host language.
- Know at least one tool for specifying and executing the formal semantics of a DSL using modelling and specification languages such as PLT Redex or Ott.
- Know at least one tool for supporting the design of a type system such as PLT Redex or Typesystem Trace.
- Know at least one tool for writing interpreters of a DSL in a general-purpose host language, such as Xtext or GraalVM

LG 5-3: Languages

- Understand the properties of programming languages that facilitate implementing DSLs: abstraction, overloading, extensible syntax.
- Understand the role of macros in DSLs design.
- Know the concepts of staging and hygiene relevant for the use of macros.
- Know at least one host language conducive to implementing embedded DSLs such as Racket, Haskell, Ruby or Kotlin.

LG 5-4: Development Environments

- Understand the relationship between syntax and IDE functionality.
- Understand what requirements IDE tooling has of the language infrastructure, in order to provide syntax support, code navigation, and refactoring.
- Know of the existence of tools to automate the creation on IDE functionality such as DrRacket, Xtext, MPS or Spoofax.

5.3. References

[Bettini 2016], [Steinberg et al. 2008], [Völter 2013], [Felleisen et al. 2009], [Erdweg at el. 2015], [Humer et al. 2014], [Parr 2010], [Kleppe 2008]

6. Examples

Duration: 60 min	Practice time: 0 min

This section is not examinable.

6.1. Terms and Principles

In every licensed training session, at least one example for a DSL must be presented.

Type and structure of the examples presented may depend on the training and participants' interests. The are not prescribed by iSAQB.

References

This section contains references that are cited in the curriculum.

A

- [Aho et al. 2006] Aho, Alfred, Lam, Monica, Sethi, Ravi, Ullman, Jeffrey: Compilers: Principles, Techniques, and Tools. Addison Wesley, 2006.
- [Alama 2020] Alama, Jesse: Language-Oriented Programming in Racket - A Cultural Anthropology. Gumroad, 2020. https://jessealama.gumroad.com/l/lop-in-racket-cultural-anthro

B

- [Baader and Nipkow 1998] Baader, Franz, Nipkow, Tobias: Term Rewriting and All That. Cambridge University Press, 1998.
- [Bettini 2016] Bettini, Lorenzo: Implementing Domain-Specific Languages with Xtext and Xtend - Second Edition. Packt, 2016.

C

- [Culpepper et al. 2019] Culpepper, Ryan, Felleisen, Matthias, Flatt, Matthew, Krishnamurthi, Shriram: From Macros to DSLs: The Evolution of Racket. Summit on Advances in Programming Languages, 2019. https://cs.brown.edu/~sk/Publications/Papers/Published/cffk-macros-to-dsls/

E

- [Elliott and Hudak 1997] Elliott, Conal, Hudak, Paul: Functional reactive animation. In ACM SIGPLAN International Conference on Functional Programming (ICFP '97). pages 263-273. ACM, Amsterdam.
- [Evans 2003] Evans, Eric J.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.
- [Erdweg at el. 2015] Sebastian, Erdweg et al.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. In Computer Languages, Systems & Structures, 44:24–47, 2015.

F

- [Felleisen et al. 2009] Felleisen, Matthias, Finder, Robert Bruce, Flatt, Matthew: Semantics Engineering with PLT Redex. MIT Press, 2009.

G

- [Ghosh 2010] Ghosh Debasish: DSLs in Action. Manning, 2010.

H

- [Humer et al. 2014] Humer, Christian, et al.: A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE'14), 2014.

K

- [Kleppe 2008] Kleppe, Anneke: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels . Addison-Wesley, 2008.

L

- [Leermakers 1993] Leermakers, René: The Functional Treatment of Parsing. Springer Science, 1993.

M

- [Metsker 2001] Metsker, Steven John: Building Parsers With Java. Addison Wesley, 2001.

N

- [Nipkow and Klein 2014] Nipkow, Tobias, Klein, Gerwin: Concrete Semantics. Springer, 2014.
- [Nystrom 2021] Nystrom, Robert: Crafting Interpreters. Genever Benning, 2021.

P

- [Parr 2010] Parr, Terrence: Language Implementation Patterns. O'Reilly, 2010.
- [Pierce 2002] Pierce, Benjamin C.: Types and Programming Languages. MIT Press, 2002.
- [Peyton Jones and Eber 2003] Peyton Jones, Simon, Eber, Jean-Marc: How to write a financial contract. Chapter in "The Fun of Programming", ed. Gibbons and de Moor, Palgrave Macmillan 2003.
- [Pretnar 2015] Matija Pretnar: An Introduction to Algebraic Effects and Handlers. Electronic Notes in Theoretical Computer Science 319 (2015) 19-35.

S

- [Steinberg et al. 2008] Steinberg, Dave, Budinsky, Frank, Paternostro, Marcelo, Merks, Ed: EMF: Eclipse Modeling Framework, 2nd edition. Addison-Wesley, 2008.

V

- [Völter 2013] Völter, Markus: DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. 2013
- [Völter et al. 2014] Völter, Markus, Siegmund, Janet, Berge, Thorsten, Kolb, Bernd: Towards user-friendly projectional editors. In International Conference on Software Language Engineering 2014. 41-61. Springer.

W

- [Wąsowski and Berger 2023] Wąsowski, Andrzej, Berger Thorsten: Domain-Specific Languages - Effective Modeling, Automation, and Reuse. Springer 2023.
- [Wilhelm et al. 2021] Wilhelm, Reinhard, Seidl, Helmut, Hack, Sebastian: Compiler Design - Syntactic and Semantic Analysis. Springer-Verlag Berlin Heidelberg, 2013.