

# Programming Language Concepts for Software Developers

Peter Sestoft  
IT University of Copenhagen, Denmark  
sestoft@itu.dk  
Draft version 2008-05-18

## Abstract

This note describes and motivates our current plans for an undergraduate course on programming language concepts for software development students. We describe the *competences* we expect students to acquire as well as the *topics* covered by the course. We plan to use C# and Scheme as instruction languages, and will study the implementation of Java and C# with their underlying platforms, the Java Virtual Machine and .NET Common Language Runtime, and emphasize implementation exercises. This comes at the expense of classical compiler course subjects such as register allocation, and optimization of imperative loop-intensive code. One purpose of this white paper is to solicit comments and advice on the choice of planned competences and topics.

## 1. A software developer's main tools

A software developer must have a wide range of competences: the ability to work with customers to elicit requirements; the ability to work with other software developers (and users) to design, build, document, debug, test and maintain software; and much more.

Here we focus on the competences needed to design and construct software that is efficient (in space and time), reliable, and maintainable. To have this competence, a software developer must be able to choose appropriate tools and use them well. Among the most prominent tools are programming languages, such as Java, C#, C, Scheme, and related libraries and development environments. For this reason *a solid knowledge of programming language concepts is essential for the construction-oriented part of a software developer's work* — even if he or she will never write a compiler at work.

## 2. Background

The IT University of Copenhagen recently created a new undergraduate program in software development. In the fifth semester of this program, to be given first time in the fall of 2009, two sister courses are taken in parallel: One course on programming language concepts (called “programs as data”) and another one on operating systems and machine-oriented programming (in C). This note de-

scribes our current thoughts on the goals, contents and form of the programs as data course.

Prior the fifth semester, students will have taken courses in object-oriented programming (using Java and C#), algorithms and data structures, distributed systems, data bases, architecture, and more; see table 1.

## 3. Competences versus topics

We present the planned course by explaining both what the student can *do* after the course, and in terms of a traditional list of topics. To illustrate the difference, consider the competence-oriented description:

“the student can *design* a concrete syntax and an abstract syntax for a subset of Icon and *use* a lexer generator and a parser generator to *implement* a continuation-based interpreter for a subset of Icon”

versus the topic-oriented description:

“the course presents LL-parsing, lexer and parser generators, abstract syntax trees, interpretation, continuations and backtracking”

The latter description does not make it clear what the student can *do* after the course: *Recite* “A grammar is a four-tuple ...”? *Identify* a context free grammar when presented with one? *Use* the tools? *Prove* that a language is in  $LL(k)$  for some  $k$ ? *Compare* the theories and tools to other theories and tools? *Generalize* them?

We stress construction and implementation in the course:

*Learning takes place through the active behavior of the student: it is what he does that he learns, not what the teacher does.* (Ralph W. Tyler 1949)

Time-wise, there is no room for a separate (compiler) project within the course, so we plan a string of bi-weekly assignments in which the students develop or extend a simple compiler, type checker, program generator, and so on.

## 4. Desired student competences

The course should give students these competences:

- Students must be able to *analyse* and *explain* the performance (time and space) characteristics of a program written in Java, C# or C, when executed on modern hardware, based on an understanding of the implementation of these languages.

Semester	25% workload	25% workload	50% workload
6	Reflection on IT	Elective	Bachelor project
5	Machine-oriented prog.	<b>Programs as data</b>	Business and organization, guided internship
4	Software processes	Elective	Project, programming tools, 6-groups
3	Concurrency	Data bases	Analysis and design, w project
2	OO programming	Algorithms and data struct.	Project, algorithmic, 4-groups
1	User interfaces	Communication	Introductory programming, w project

**Table 1.** Bachelor’s programme in software development, IT University of Copenhagen. This paper is about the fifth semester course “programs as data”, and partially about its sister course machine-oriented programming.

- Based on the machine model, the students must be able to *estimate* whether one language construct is preferable to another (for instance, reference type versus value type in C#) for reasons of time or space consumption.
- Students must be able to *use* their tools at a higher level of sophistication. For instance, to *perform* recursive manipulation of (abstract syntax) trees, to *use* higher-order functions or their equivalents, and so on.
- Students must be able to *use* well-established tools for regular expression matching, lexing and parsing, for text input and for building abstract syntax representations.
- Students must be able to *design* abstract syntax representations, and *use* recursion to process such representations, e.g. for type analysis, for compilation, and for simplification of logical or arithmetic expressions.
- Students must be able to *compare* the expressiveness, designs and performance characteristics of programming languages (such as Java and C#), and *explain* how their features follow from design decisions and implementation techniques. For instance, why is `new Pair<Integer,Integer>[42]` legal in C# but illegal in Java, when `Pair<T,U>` is a generic type?
- Students must be able to *view* programs both as creators of behaviors, and as data that can be analysed, transformed and generated by other programs.
- Students must be able to critically *evaluate* “new” technologies that are marketed to them, and *relate* those technologies to academic principles and existing technologies. This requires a historical perspective; it took two or three decades for garbage collection (1960), object-oriented programming (1967) and polymorphic types and type inference (1978) to become mainstream — in Java 5.0 and C# 2.0.

## 5. Course topics

To give students the desired competences, the course will cover the following topics:

- Lexing, regular expressions, finite state machines, NFAs and DFAs, lexer generators.
- Parsing, top-down versus bottom-up, LL versus LR, parser generators, Coco/R.
- A stack machine, stack-based evaluation of expressions, Postscript. In the sister course, students may implement a bytecode-based stack machine in C.
- Compilation of a small subset (`*p`, `&x`, pointer arithmetics, arrays, `arr[i]`) of C to stack machine code.
- Type checking, statically typed versus dynamically typed languages.

- The stack and heap machine model of Java and C#, and their implementation by rather simplistic compilation Java-to-JVM and C#-to-.NET.
- The JVM and .NET bytecode languages and their surprising similarity. Simplified register machines and part of the x86 instruction set will be covered in the sister course.
- The JVM and .NET execution platforms and just-in-time generation of register machine code. (But little about classic compiler techniques such as register allocation, reduction in strength, common subexpression elimination). The diversity of the just-in-time compilers, e.g. Sun Hotspot Client, Server, IBM JVM, BEA JRockit, and their properties.
- Garbage collection techniques. In the sister course, students may implement a simple garbage collector in C.
- Scheme, mostly-functional programming, dynamically typed languages, abstract syntax as concrete syntax.
- Continuations, exceptions, an interpreter for a subset of Icon.
- Syntax-directed analysis and transformation of expressions.
- Program generation with Scheme backquote and comma, and runtime program generation using `eval`; maybe runtime program generation with Java or C# bytecode.

## 6. Why the strong focus on Java and C#?

As the reader will have noted, we put much emphasis on Java and C#. There are two reasons for this. First, those languages are likely to be the main tools for our students in their work life, so it is particularly valuable to throw light on their strengths, weaknesses, and properties, and how these follow from the implementation techniques used. Second, the students are familiar with those languages before the course and will appreciate the significance of improved mastery. Third, studying those languages will familiarize the student with a wide range of implementation techniques — type inference, stack machines, bytecode, just-in-time compilation, garbage collection, reflection — that are highly relevant also in other languages. For these reasons, we prefer to study the implementations of Java and C# rather than those of potentially neater or more powerful languages such as Scheme, ML and Haskell.

In addition to the implementation techniques mentioned above, we will illustrate continuations, by studying an interpreter for a subset of Icon. This also stresses the utility of tail calls (which are available on the .NET platform, although not in C#).

## 7. Why Scheme as instruction language?

Scheme is an ideal vehicle for program generation because it is dynamically typed and because its abstract and concrete syntax coincide. Also, it is an advertisement for simplicity, much expressive power flows from just a few good concepts, and hence it provides an antidote to the designs of Java, C# and many other languages. Scheme is useful for functional programming, and hence program-

ming with continuations, although it lacks a type system to guide the use higher-order functions.

## 8. Why C# as instruction language?

We plan to use C# (as well as Scheme) as the instruction language: the language in which checkers, compilers and so on are implemented. Scheme or Standard ML would seem better suited for this purpose, but we do not want to give students the impression that their new competences are relevant only in those languages, which typically do not interface well with standard execution platforms. The languages F# and Scala do use standard execution platforms and libraries. However, they use an unfamiliar syntax and are still evolving, but some day they may be better choices for instruction languages.

In the past, “academic” languages such as Scheme, ML, Haskell and Smalltalk had many attractive features not shared by “practical” mainstream languages such as C, C++, Pascal and Fortran. These attractive features include automatic memory management, well-defined semantics, portability across several platforms, exception handling or first-class continuations, reasonably meaningful static or dynamic type system, and higher-order functions.

Today, the “practical” mainstream languages Java and C# do have all those attractive features. (And note that these language are semantically *very* different from C, C++, Pascal and Fortran). Granted, they also have a cumbersome and misleading syntax, for instance for function types such as `delegate bool pred(int x)`, which is a definition of the name `pred` with the actual type scattered around it. Nevertheless, using C# as instruction language also allows us to show that it can be used in a more advanced ways than are usually promoted in object-oriented programming courses.

Clearly, there are many language constructs and concepts not illustrated by Java and C#:

- Dynamic types, which we intend to illustrate with Scheme.
- Co-routines, which could be studied in the context of Smalltalk; but this is not a high priority.
- Concurrency mechanisms such as the Ada rendez-vous, join patterns, and transactional memory. However, it is not yet clear what language constructs for concurrency and distribution will become dominating, and how they will be efficiently implemented on shared-memory processors and on systems with lower communication bandwidth and higher latencies.
- Pattern matching, which we intend to illustrate with ML, F#, or Scala. This enables students to see the essence of the visitor pattern, and to easily perform structural transformations on terms, as in query optimization, or when computing disjunctive normal form.
- Traits, which could be illustrated with Scala.
- Lazy evaluation, which could be illustrated using Haskell, but it is unlike we will do that. We believe that in practice the time and space properties lazy programs are very difficult to predict for most software developers — and a developer should be able to assess the properties of things he builds.
- Runtime code generation, which we intend to illustrate with Scheme (backquote, comma and `eval`), and possibly with runtime bytecode generation on the JVM or .NET platform.

## 9. Teaching method and materials

In the past we have taught a somewhat similar course using Standard ML as instruction language to build interpreters, compilers and so on. This approach has worked for several batches of students (from 2001 to 2006), but it hasn’t worked well: students find Stan-

dard ML’s syntax and concepts difficult, even without any mention of structures, signatures and functors. Recently we have been using C# as a metalanguage, to write simple interpreters, compilers and type checkers. The language is considerably more verbose but still this seems to work better with our students.

For the Standard ML-based past course we developed lecture notes [2] whose table of contents is shown in table 2, but those notes would require serious rework and extension to cover the “programs as data” course described here. Hence we need to find or develop suitable materials; recommendations and advice are warmly welcomed.

Chapter	Contents
2	Interpreters and compilers
3	From concrete syntax to abstract syntax
4	A first-order functional language
5	A higher-order functional language
6	Imperative languages, micro-C interpretation
7	Compiling micro-C to a stack machine
8	Continuations, interpretation of micro-Icon
9	Backwards compilation, on-the-fly optimization
10	Real-world abstract machines, JVM, .NET
11	The heap and garbage collection
12	An object-oriented language
13	Reflection, the Java and .NET APIs
14	Runtime code generation, Scheme, Java bytecode
A	Standard ML crash course

**Table 2.** Contents, old lecture notes using Standard ML as presentation language [2].

## 10. To do: Competence levels

We intend to use the SOLO taxonomy (Structure of Observed Learning Outcomes) of John Biggs [1] to make the description of intended competences more precise. The SOLO taxonomy classifies students’ competences into five levels, shown in figure 1 along with some characteristic verbs at each level. We also intend to create a mapping from topics to competences.

In the broader context of the entire undergraduate curriculum, the proposed course attempts to raise the students’ programming competences to SOLO level 5, by giving them the skills, tools and theories to reason about program behavior and performance.

Level	Student competences
1 pre-structural	has no understanding, uses irrelevant information
2 uni-structural	recite, identify, name, follow simple instructions
3 multi-structural	enumerate, describe, classify, combine, structure, execute
4 relational	compare, relate, analyze, apply, explain
5 extended abstract	generalize, hypothesize, theorize

**Figure 1.** The SOLO taxonomy of Biggs [1].

## References

- [1] John Biggs. *Teaching for quality learning at university: what the student does*. Open University Press, United Kingdom, second edition, 2003.
- [2] Peter Sestoft. Programming language concepts. Draft lecture notes, version 0.34, February 2006. At <http://www.itu.dk/people/sestoft/papers/plc-0.34-2up.pdf>.