# Course introduction

Advanced Compiler Construction

Michel Schinz – 2021–02–25

# General information

# Course goals

The goal of this course is to teach you:

- how to compile high-level functional and object-oriented languages,
- how to optimize the generated code, and
- how to support code execution at run time.

To achieve this, the course is split in three parts:

1. compilation of high-level concepts (e.g. closures),
2. intermediate languages and optimizations,
3. virtual machines and garbage collection.

# Prerequisite skills

To complete the project successfully, you need:

- excellent knowledge of functional programming, ideally in Scala,
- good knowledge of (relatively) low-level programming in C.

Beware: acquiring these skills during the course can be challenging.

# Evaluation

The grade will be based on three aspects:

1. several group projects, to be completed in groups of at most two people,
2. an individual oral exam.

Note: the course is evaluated during the semester, which has two important consequences:

- there is no retake exam,
- the oral exam will take place during the last week of the semester, not during the official exam period.

# Grading scheme

The final grade will be based on your results in:

- the various project parts, spread over 11 weeks, which contribute to 80% of the grade,
- the final exam, which contributes to 20% of the grade.

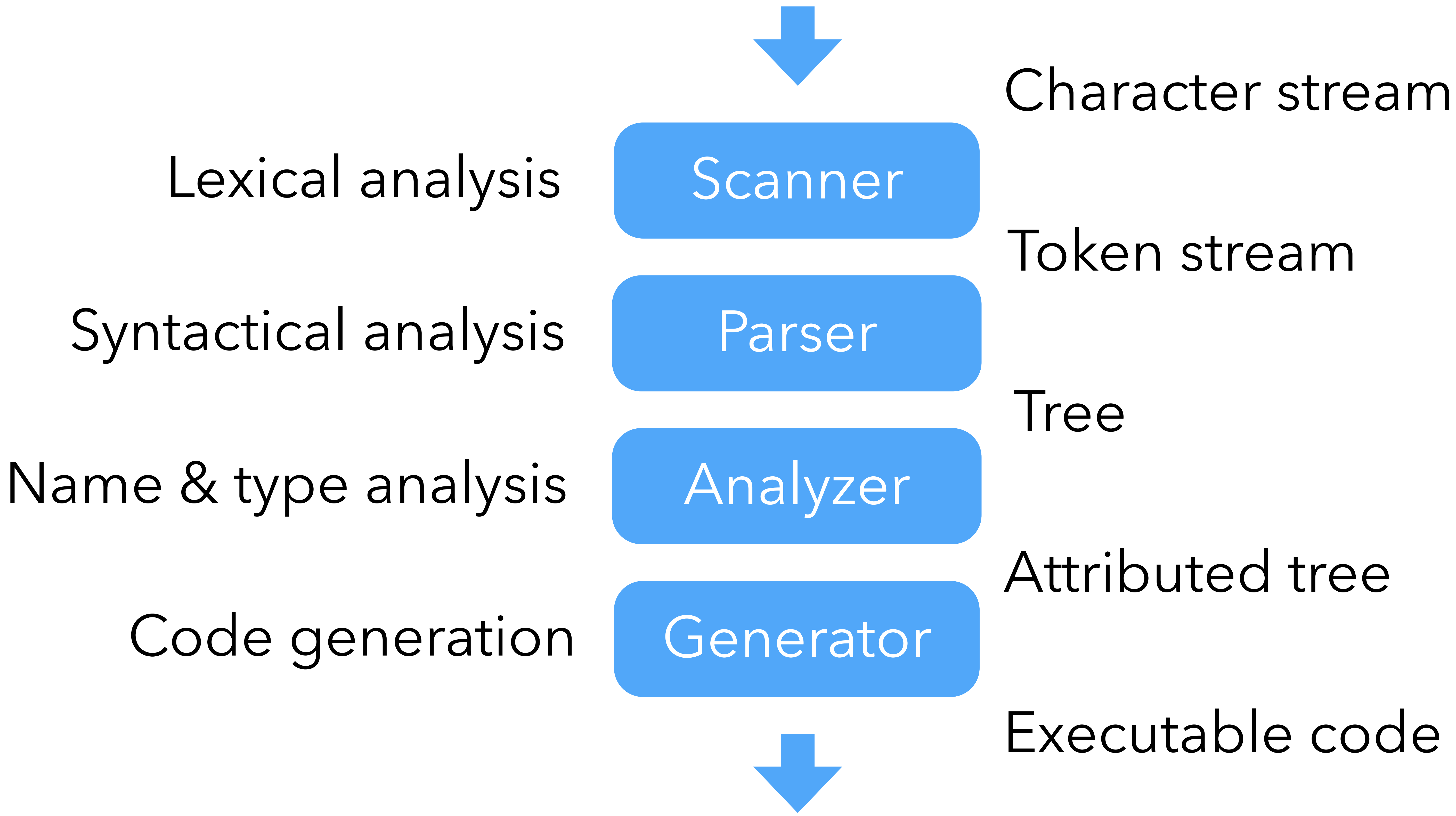
# Resources

Lecturer:
Michel Schinz
Assistants:
Ergys Dona, Julie Giunta
Web page:
`https://cs420.epfl.ch`
Forum:
`piazza.com/epfl.ch/spring2021/cs420`

# Course overview

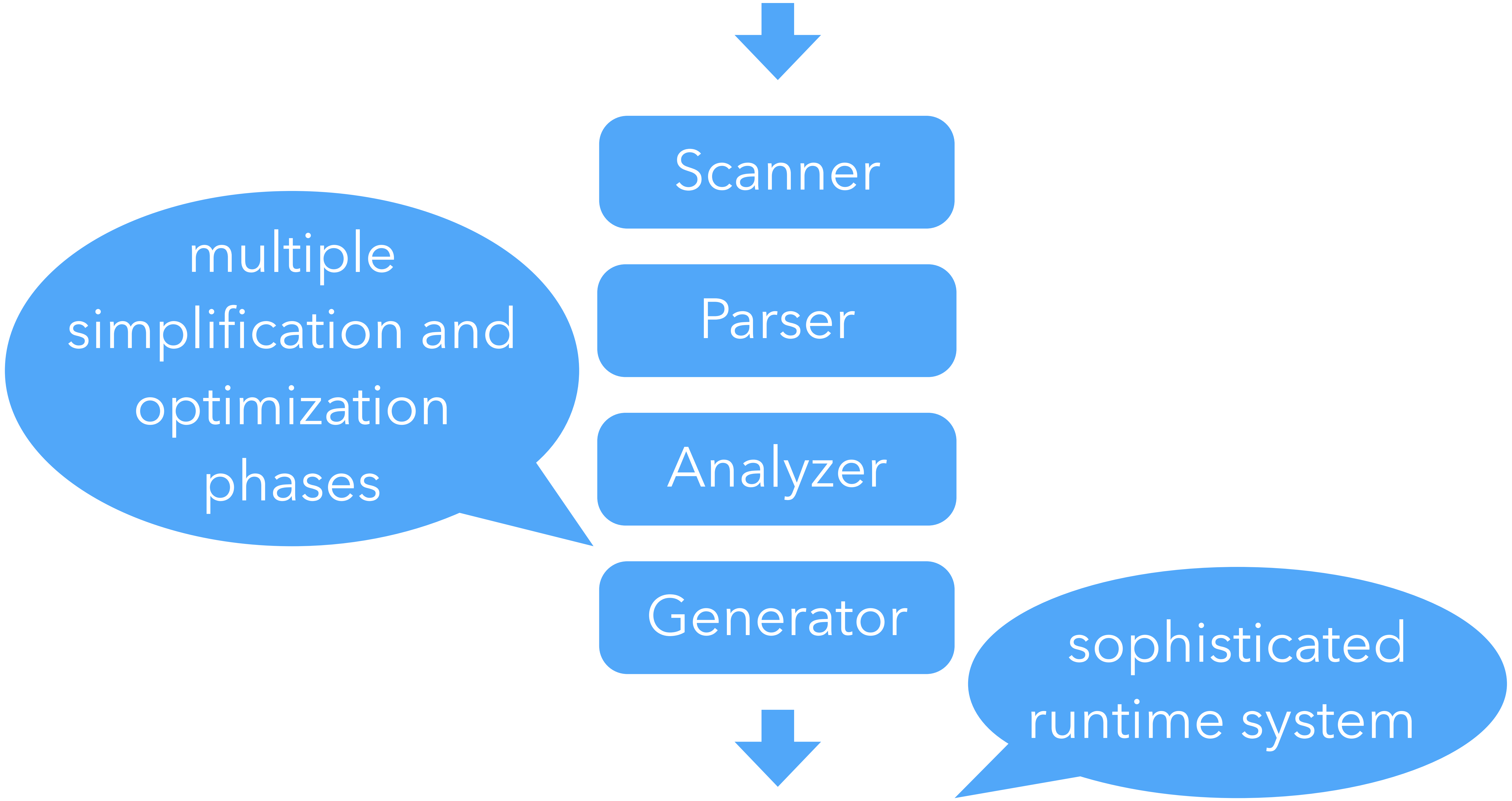
# What is a compiler?

Your current view of a compiler must be something like this:



# What is a compiler, really?

Real compilers are often more complicated…



# Additional phases

**Simplification** (or **lowering**) **phases**

translate complex concepts of the language (e.g. pattern matching) into simpler ones.

**Optimization phases**

try to improve the program's usage of some resource (e.g. CPU time, memory).

# Simplification phases

Example of a simplification phase in Java compilers: transformation of nested classes into top-level ones.

```
class Out {
  void f1() { }
  class In {
    void f2() {
      f1();
    }
  }
}
```

```
class Out {
  void f1() { }
}
class Out$In {
  final Out this$0;
  Out$In(Out o) {
    this$0 = o;
  }
  void f2() {
    this$0.f1();
  }
}
```

# Optimization phases

Example of an optimization phase in Java compilers:
removal of **dead code**, i.e. code that can never be executed.

```
class C {
  public final static boolean debug = !true;
  int f() {
    if (debug) {
      System.out.println("C.f() called");
    }
    return 10;
  }
}
```

# Optimization phases

Example of an optimization phase in Java compilers: removal of **dead code**, i.e. code that can never be executed.

```
class C {
  public final static boolean debug = !true;
  int f() {
    if (debug) {
      System.out.println("C.f() called");
    }
    return 10;
  }
}
```

dead code, removed during compilation

# Intermediate representations

To manipulate the program, simplification and optimization phases must represent it in some way. Options:

- use the abstract syntax tree (AST),
- use another **intermediate representation** (**IR**).

Sophisticated compilers usually use several different IRs.

# Run time system

Apart from the compiler, a complete **run time system** (**RTS**) must be written, to provide various services to executing programs, like:

- code loading and linking,
- code interpretation, compilation and optimization,
- memory management (garbage collection),
- concurrency,
- etc.

That's a lot, and Java RTSs, for example, are often more complex than Java compilers!

# Memory management

Most modern programming languages offer **automatic memory management**: the programmer allocates memory explicitly, but deallocation is performed automatically.
The deallocation of memory is usually performed by a part of the run time system called the **garbage collector** (**GC**).
A garbage collector periodically frees all memory that has been allocated by the program but is not reachable anymore.

# Virtual machines

Instead of targeting a real processor, a compiler can target a virtual one, usually called a **virtual machine** (**VM**). The produced code is then interpreted by a program emulating the virtual machine.

Virtual machines have many advantages:

- the compiler can target a single architecture,
- the program can easily be monitored during execution, e.g. to prevent malicious behavior, or provide debugging facilities,
- the distribution of compiled code is easier.

The main (only?) disadvantage of virtual machines is their speed: it is always slower to interpret a program in software than to execute it directly in hardware.

# Dynamic (JIT) compilation

To make virtual machines faster, **dynamic**, or **just-in-time** (**JIT**) **compilation** was invented.
The idea is simple: Instead of interpreting a piece of code, the virtual machine translates it to machine code, and hands that code to the processor for execution.
This is usually faster than interpretation.

# Summary

Compilers for high-level languages are more complex than the ones you've studied, since:

- they must translate high-level concepts like pattern-matching, anonymous functions, etc. to lower-level equivalents,
- they must be accompanied by a sophisticated run time system, and
- they should produce optimized code.

This course will be focused on these aspects of compilers and run time systems.