

# Course introduction

Advanced Compiler Construction  
Michel Schinz – 2016-02-25

# **General information**

# Course goals

The goal of this course is to teach you:

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

To achieve these goals, the course is roughly split in three parts of unequal length:

- a part covering the compilation of high-level concepts (e.g. closures),
- a part covering intermediate languages and optimizations,
- a part covering virtual machines and garbage collection.

# Prerequisite skills

To complete the project successfully, you need:

- excellent knowledge of functional programming 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:

- several group projects, to be completed in groups of at most two people,
- one or more (depending on class size) individual projects, to be completed alone,
- an individual oral exam.

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

1. there is no retake exam,
2. 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:

Dmitry Petrashko

Manos Koukoutos

Web page:

`lamp.epfl.ch/teaching/advanced_compiler`

Forum:

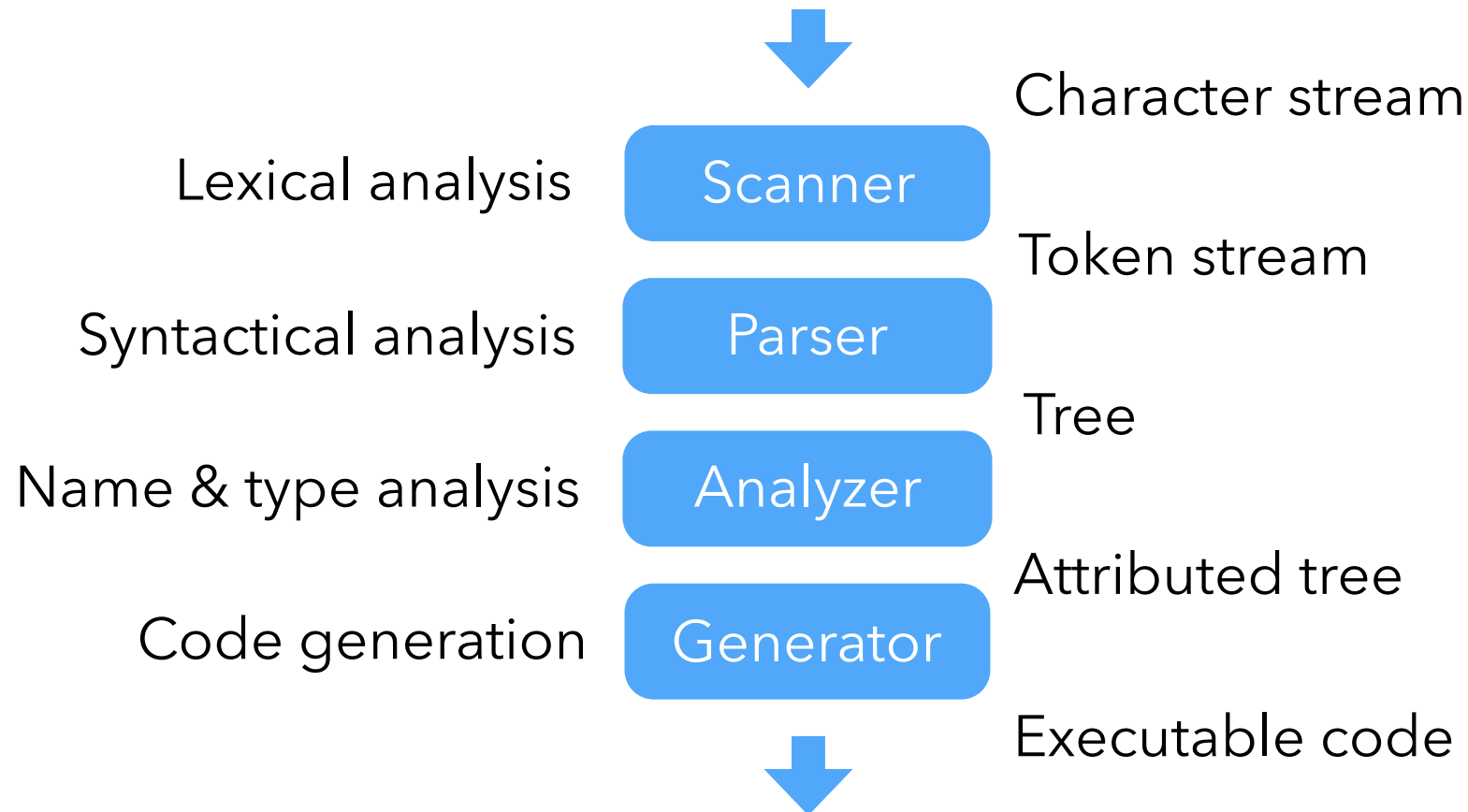
`piazza.com/epfl.ch/spring2016/cs420/home`

# Course overview



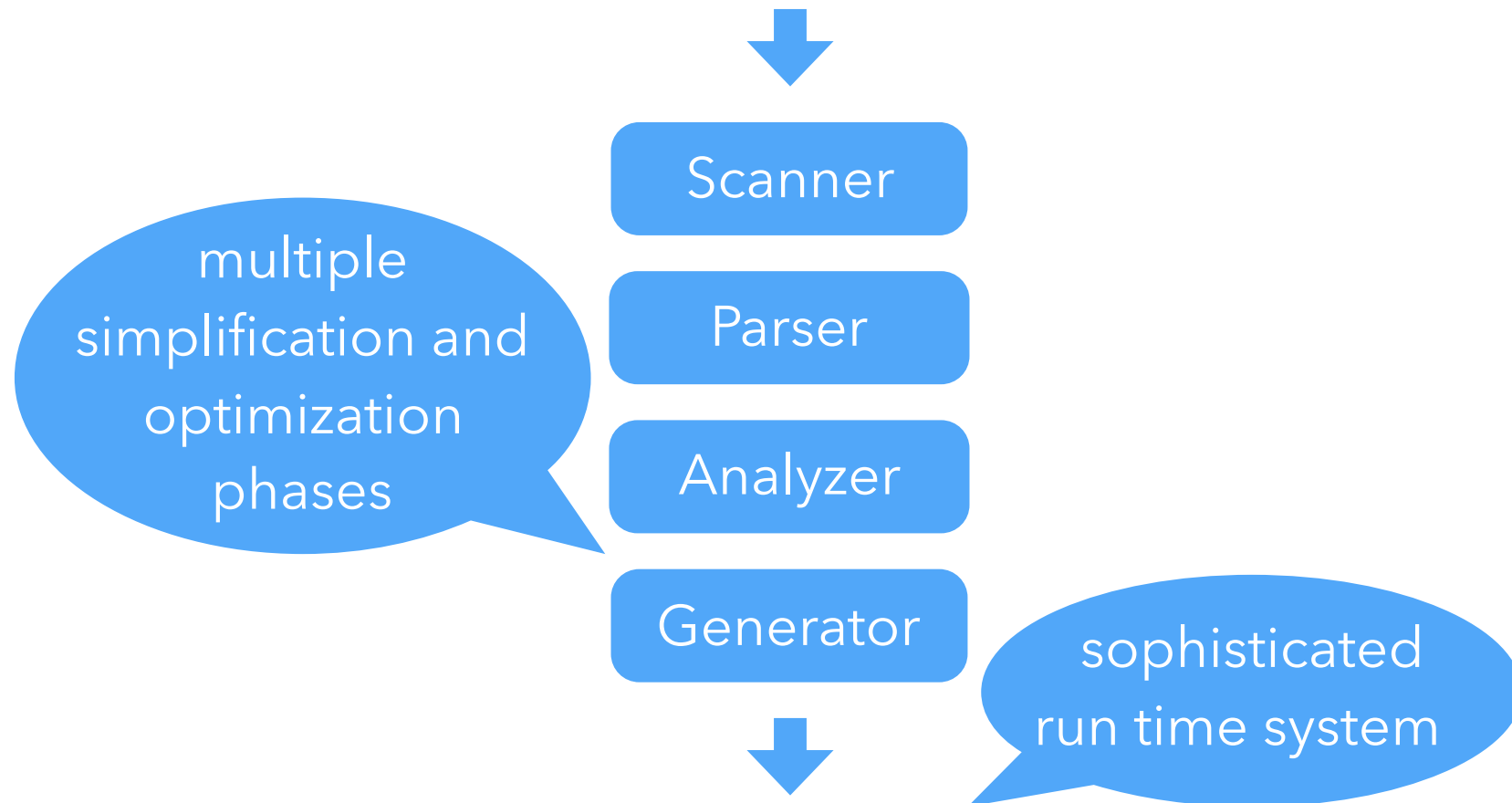
# What is a compiler?

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** transform the program so that complex concepts of the language – pattern matching, anonymous functions, etc. – are translated using simpler ones.

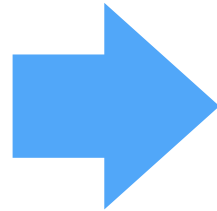
**Optimization phases** transform the program so that it hopefully makes better use of some resource – e.g. CPU cycles, memory, etc.

Of course, all these phases must preserve the meaning of the original program!

# Simplification phases

Example of a simplification phase: Java compilers have a phase that transforms nested classes to 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: Java compilers optimize expressions involving constant values. That includes removing **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: Java compilers optimize expressions involving constant values. That includes removing **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.

One possibility is to use the representation produced by the parser – the abstract syntax tree (AST).

The AST is perfectly suited to certain tasks, but other **intermediate representations** (IR) exist and are more appropriate in some situations.

# Run time system

Implementing a high-level programming language usually means more than just writing a compiler. A complete **run time system** 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.

This is quite a lot, and modern Java Virtual Machines, for example, are 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.