Advanced compiler construction

Michel Schinz

2007–03–16

General course information

Teacher & assistant

Teacher:

Michel Schinz

`Michel.Schinz@epfl.ch`

Assistant:

Iulian Dragos

INR 321, ☎ 368 64

`Iulian.Dragos@epfl.ch`

Course goals

The goal of this course is to teach you:

1. how to compile high-level functional and object-oriented programming languages, and
2. how to optimise the generated code.

To achieve these goals, the course is split in two parts:

1. a part covering virtual machines, memory management, closure conversion, etc.
2. a part covering data-flow analysis, SSA form, etc.

Evaluation

Evaluation will be based on:

- a project, made in groups of two persons at most, and
- an individual exam – oral or written – at the end of the semester.

Notice that the exam will take place **during the last week of the semester**, not after it.

Grading scheme

The grade will be attributed according to the following scheme:

Part	**Weight**
Project 1: garbage collector	15%
Project 2: closure conversion	15%
Project 3: tail call elimination	10%
Advanced project	30%
Exam	30%

Project overview

You will have to improve a compiler and a virtual machine (VM) for minischeme a tiny dialect of Scheme, itself a dialect of Lisp.

For example, the `map` function is minischeme is written:

```
(define map
  (lambda (f l)
    (if (null? l)
        nil
        (cons (f (head l))
              (map f (tail l))))))
```

The compiler is written in Scala, the VM in C.

Project parts

The project is split in two parts:

1. a common part, during which all groups have to complete the same “simple” tasks (*e.g.* add garbage collection to the VM),
2. an individual part, during which all groups have to choose two advanced tasks, try to complete them and describe their work in a short report (*e.g.* implement a JIT compiler for the VM).

Resources

The course has a Web page:

`http://lamp.epfl.ch/teaching/advanced_compiler`

Moreover, to handle project submissions we will use moodle – please register to this course on the system!

`http://moodle.epfl.ch/`

Course name: Advanced Compiler Construction

Enrolment key: ACT

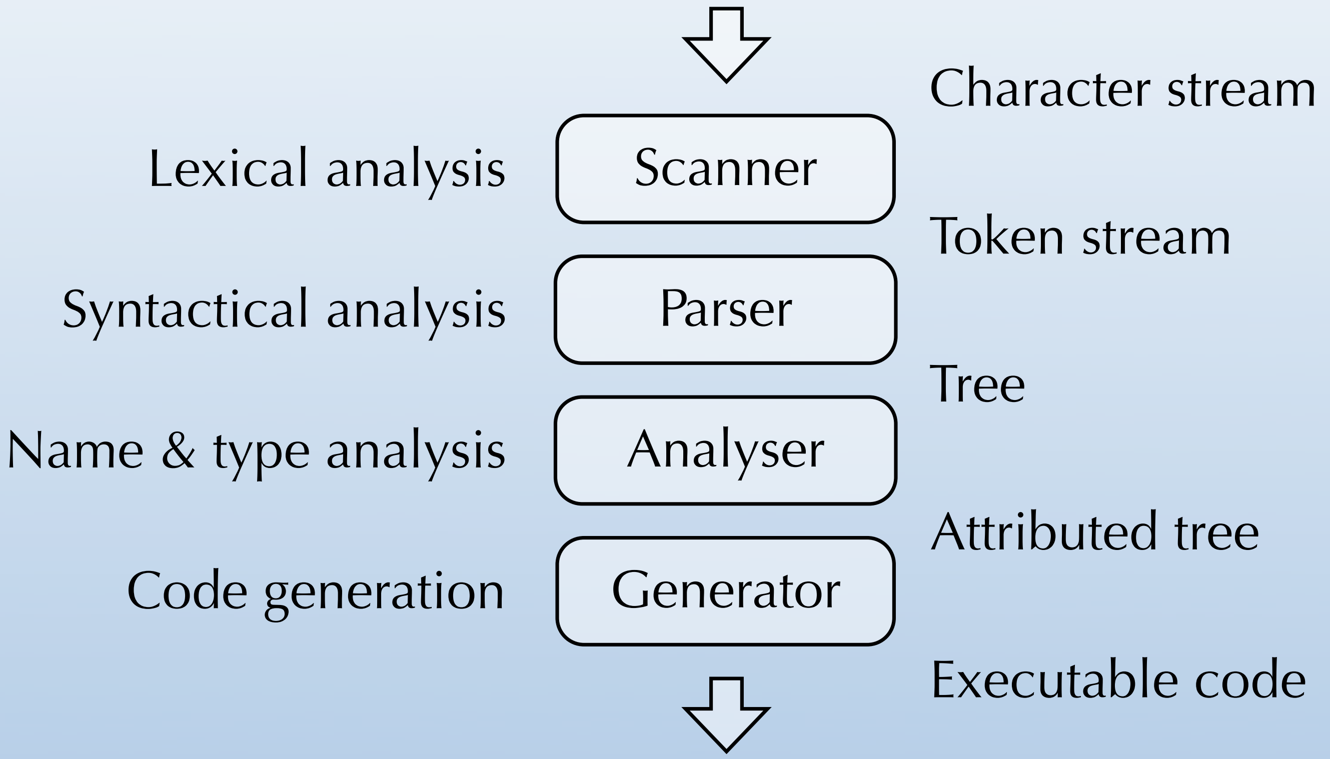
Questions can either be asked during the exercise sessions, or through the course's newsgroup:

`epfl.ic.cours.act`

Course overview

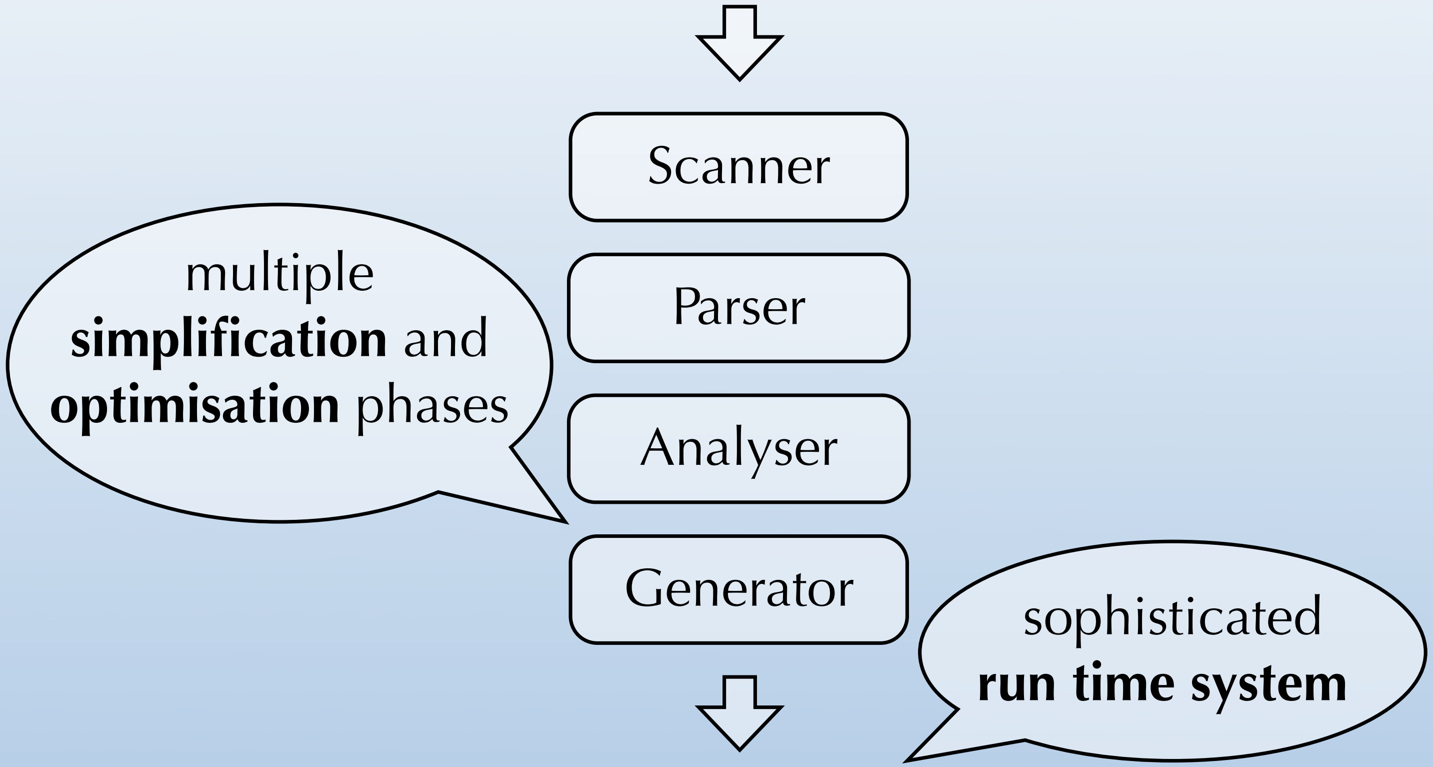
What is a compiler?

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



What is a compiler, really?

Real compilers are often more complicated…



Scanner
Parser
Analyser
Generator
multiple **simplification** and **optimisation** phases
sophisticated **run time system**

Additional phases

Simplification phases transform the program so that complex concepts of the language (*e.g.* pattern matching, anonymous functions, …) are translated using simpler ones.

Optimisation 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 simplification phase: Java compilers have a simplification 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();
  }
}
```

Optimisation phases

Example of optimisation phase: Java compilers optimise 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 optimisation 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.

Kinds of IRs

Intermediate representations can broadly be split in three categories:

- **graphical IRs**, which represent the program as a graph,
- **linear IRs**, which represent the program as a linear sequence of instructions, and
- **hybrid IRs**, which are partly graphical, partly linear.

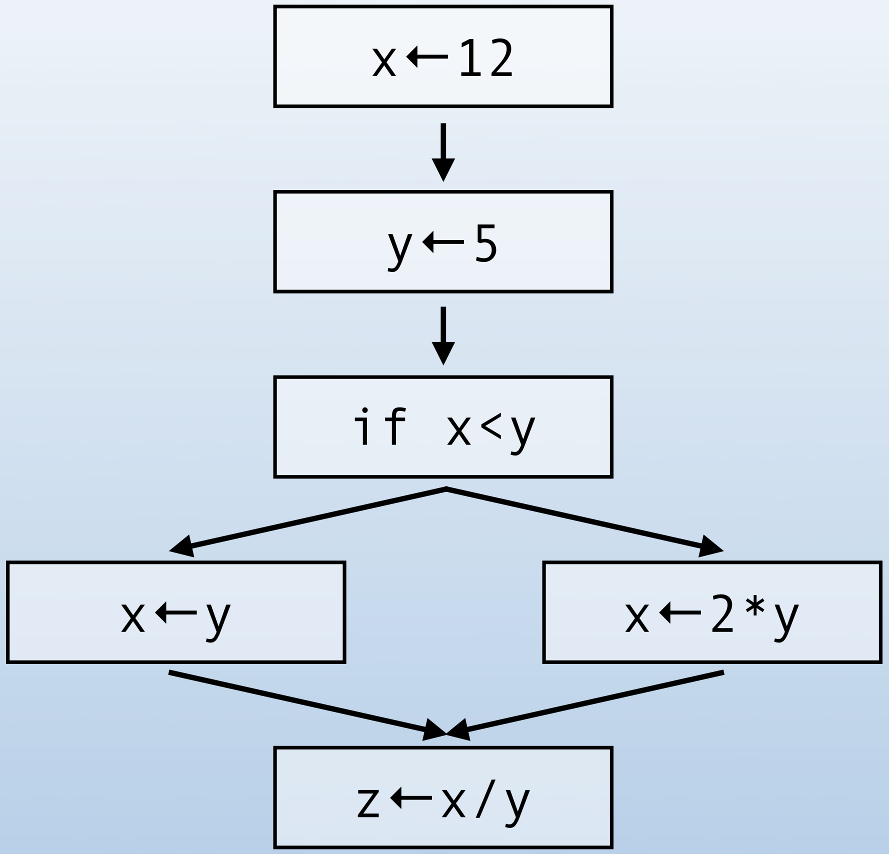
Graphical IRs

Graphical intermediate representations represent the program as a graph.

They are often used in the initial phases of the compiler. In particular, the AST produced by the parser is a graphical IR.

Examples: ASTs, some kinds of control-flow graphs, etc.

Graphical IR example



This is an example of a **control-flow graph** (**CFG**). Nodes are instructions, and edges represent the possible flow of control: if there is an edge from n_1 to n_2, then control can flow directly from n_1 to n_2.

Linear IRs

Linear intermediate representations represent the program as a sequence of instructions.

They are often used in the final phases of the compiler, since machine code itself is linear.

Examples: three-address code, stack languages, etc.

Linear IR example

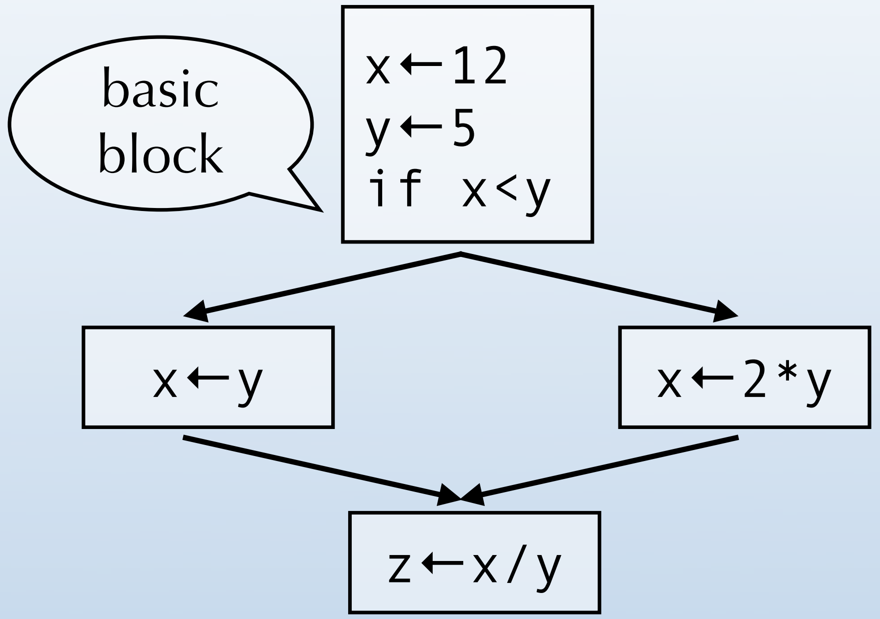
```
     x←12
     y←5
     if x<y goto L1
     x←y
     goto L2
L1:  x←2*y
L2:  z←x/y
```

Hybrid IRs

Hybrid intermediate representations have graphical and linear components.

For example, most control-flow graphs are hybrid, unlike the one presented before: the nodes in the CFG are linear sequences of instructions – called **basic blocks** – but the CFG itself is a graph.

Hybrid IR example



This CFG is equivalent to the one presented before, but its nodes are basic blocks and not single instructions. It therefore contains fewer nodes.

A **basic block** is a maximal sequence of instructions such that control always enters at the top and leaves at the bottom.

SSA form

Static single-assignment (**SSA**) form is an intermediate representation with an important characteristic: all “variables” are assigned exactly once.

This characteristic makes a lot of optimisations easier. For example, identifying common sub-expressions is trivial.

Transforming an imperative program to SSA form implies the introduction of so-called Φ-functions.

SSA form example

```
        $x_1 \leftarrow 12$
        $y_1 \leftarrow 5$
        if $x_1 < y_1$ goto L1
        $x_2 \leftarrow y_1$
        goto L2
L1:     $x_3 \leftarrow 2 * y_1$
L2:     $x_4 \leftarrow \Phi(x_2, x_3)$
        $z_1 \leftarrow x_4 / y_1$
```

The Φ-function "magically" selects the correct `x`, depending on the flow of control

Intermediate languages

Intermediate representations that can be represented textually as a program are often called **intermediate languages**.

Intermediate languages are similar to normal programming languages, but designed with different goals. For example, simplicity is usually preferred to conciseness.

Some intermediate languages are typed. This can help debugging the compiler, as the result of each phase can be type-checked.

Run time system

Implementing a high-level programming language usually means more than just writing a compiler!

A complete **run time system** (**RTS**) must be written, to assist the execution of compiled programs by providing various services like memory management, threads, etc.

For example, the Java Virtual Machine is the run time system for Java, Scala and many other programming languages. It handles (lazy) class loading, byte-code verification and interpretation, just-in-time compilation, threading, garbage collection, etc. and provides a debugging interface.

A Java Virtual Machine is actually more complex to develop than a Java compiler!

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.

VM pros and cons

Virtual machines are interesting for several reasons:

- the compiler can target a single, usually high-level architecture,
- the program can easily be monitored during execution, *e.g.* to prevent malicious behaviour, 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 produced optimised code.

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