Course introduction

Michel Schinz
Advanced Compiler Construction – 2009-02-20

General information

Course goal

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 optimize the generated code.

To achieve these goals, the course is split in two parts (of unequal length):

1. a part covering virtual machines, memory management, closure conversion, etc.
2. a part covering register allocation, standard optimizations, etc.

Evaluation

The grade will be based on three factors:

- three "basic" projects, to be done in groups of at most two people,
- an advanced project, to be done individually,
- an oral exam.

Warning: 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 semester – in the last week.

Grading scheme

Part	Weight
Basic project #1: garbage collector	15 %
Basic project #2: closure conversion	15 %
Basic project #3: tail-call elimination	10 %
Advanced project	30 %
Oral exam	30 %

Project overview

You will have to improve a compiler and a virtual machine 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 virtual machine in C.

Project parts

The project is split in two parts:

1. a basic part, in which all groups have to complete the same three tasks,
2. an advanced part, for which every student must choose one task among those proposed, complete it and write a short report about the result.

Resources

Lecturer:
  Michel Schinz, `Michel.Schinz@epfl.ch`
Assistant:
  Lukas Rytz, INR 321, `Lukas.Rytz@epfl.ch`
Web page:
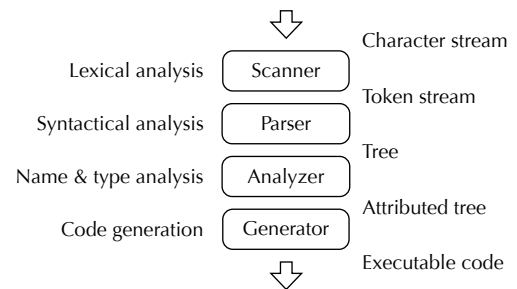 `http://lamp.epfl.ch/teaching/advanced_compiler/`
Moodle site:
  `http://moodle.epfl.ch/course/view.php?id=173`
  (enrolment key: ACT)

Course overview

What is a compiler?

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

	⇩	Character stream
Lexical analysis	Scanner	
		Token stream
Syntactical analysis	Parser	
		Tree
Name & type analysis	Analyzer	
		Attributed tree
Code generation	Generator	
	⇩	Executable code

What is a compiler, really?

Real compilers are often more complicated…

⇩

Scanner

Parser

Analyzer

Generator

⇩

multiple simplification and optimization phases

sophisticated run time system
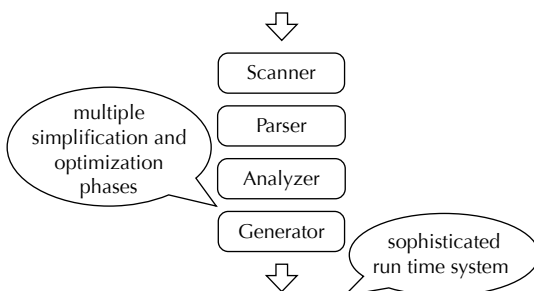
Additional phases

Simplification 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();
  }
}
```

13

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;
  }
}
```

14

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

14

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.

15

Intermediate representations

Many intermediate representations have been used in compilers. They can be differentiated according to several criteria:
- imperative (*i.e.* with a notion of mutable variables) or functional,
- typed or untyped,
- structured as a control-flow graph (CFG), or as an abstract syntax tree,
- lazy or strict,
- etc.

Let's look at a few popular examples…

16

RTL / CFG

RTL (for **Register Transfer Language**), also known as **quadruples**, is a kind of intermediate representation commonly found in compilers for imperative languages.

It is imperative, and can be structured either as a linear sequence of instructions, or as a **control-flow graph** (**CFG**) in which instructions are nodes and edges represent the possible flow of control.

The defining characteristic of RTLs is that most instructions compute a single operation on a few registers (*i.e.* variables) and put the result back in a register.
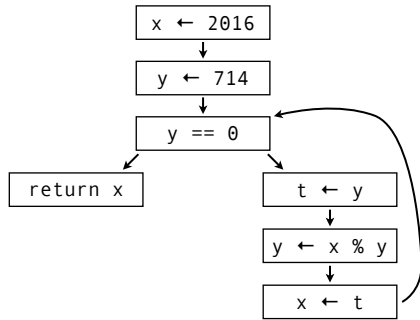
Example:

```
x ← y + z
```

17

RTL / CFG

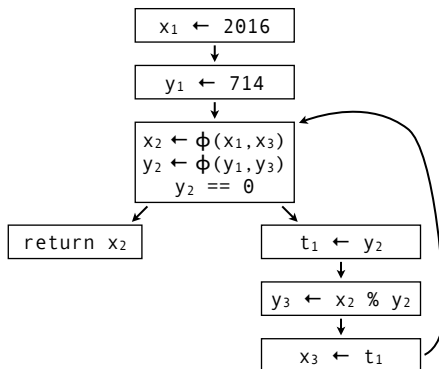Example: computing the greatest common divisor of 2016 and 714 in an RTL structured as a CFG.

```
x ← 2016
    ↓
y ← 714
    ↓
y == 0
```

```
return x        t ← y
                  ↓
                y ← x % y
                  ↓
                x ← t
```

SSA form

An RTL program is said to be in **static single assignment** (**SSA**) form if all variables are assigned at most once. This property simplifies the analysis of the program.

When several versions of a source variable reach a given node in the CFG, so-called ɸ-functions are used to reconcile them. These ɸ-functions are translated before the final code is generated.

RTL / CFG in SSA form

```
x₁ ← 2016
```
$x_1 \leftarrow 2016$

$y_1 \leftarrow 714$

$x_2 \leftarrow \phi(x_1, x_3)$
$y_2 \leftarrow \phi(y_1, y_3)$
$y_2 == 0$

$\text{return } x_2$

$t_1 \leftarrow y_2$

$y_3 \leftarrow x_2 \% y_2$

$x_3 \leftarrow t_1$

Functional IRs

Functional intermediate representations are basically variants of the λ-calculus. They are tiny functional languages and, for that reason, they are often called **intermediate languages**.

Such representations are popular in compilers for functional languages, but could also be used in compilers for imperative languages.

Functional IR

let f(x,y) =
    if y == 0 **then**
       x
    else
       let t = x % y **in** f(y, t)
in f(2016, 714)

Note: internally, such IRs are represented as a tree.

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, Groovy 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 good Java Virtual Machine is actually more complex 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.

24

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, usually high-level 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.

25

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.

26

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 optimized code.

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

27