Implementation of Programming Languages

Stefan Holdermans

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: stefan@cs.uu.nl
Web pages: http://people.cs.uu.nl/stefan/

November 12, 2007
1. Introduction
Course overview

§1

Computer programs are usually written in high-level programming languages such as C, Java, or Haskell.

Execution of such programs requires either a compiler or an interpreter for the language.

In this course, we study the workings of compilers and interpreters.
Themes

- Abstract syntax.
- Operational semantics.
- Code generation.
- Type checking/inference.
- Tools for compiler construction.
- Tree-oriented programming.
Sessions

Lectures:

- Monday, 9am–11am (BBL-426),
- Wednesday, 9am–11am (BBL-426).

Lab sessions:

- Monday, 11am–13pm (BBL-461, BBL-468),
- Wednesday, 11am–13pm (BBL-461, BBL-468).
Material

Web page:


Study material:

- slides (available shortly after the lectures),
- lecture notes (from previous years),
- handouts,
- a series of toy compilers (written in Haskell).
Additional material


Additional material (cont’d)


Grading

- Written exam (80%):
  - Wednesday, January 30, 2008, 9am–12pm (BBL-471, BBL-430).

- Lab assignments (20%):
  - tuples: Friday, November 30, 2007,
  - static-link optimization: Friday, December 21, 2007,
  - type reconstruction: Friday, January 25, 2008.

- To pass the course, both components should be no less than 5 (and the weighted sum no less than 5.75),
- To qualify for a second chance, the original result should be no less than 4.
1.1 Compilation and Interpretation
A compiler performs a meaning-preserving translation from one language into another language:

source program \[ \rightarrow \text{compiler} \rightarrow \text{target program}. \]

The meaning of a program is called its semantics.
Often—but not always—the compilation target is a machine-executable program:

\[
\text{input} \rightarrow \text{target program} \rightarrow \text{output}.
\]

- Lowest-level representation of the original program.
- Consists of instructions that can be directly executed by a computer’s CPU.
- Execution can be very fast.
Target platforms

§1.1

Different executables are required for different combinations of CPU and operating system:

▶ each type of CPU has its own instruction set,
▶ each operating system has its own conventions for dealing with executables.

Typically, a compiler is targeted at the same platform it runs on.

Cross compiler: generates code for a platform other than the one it runs on.
Building compilers

§1.1

Implementation language: the programming language in which a compiler is written.

So, to deploy a compiler, we first need a compiler or interpreter for the implementation language.

Often, the implementation language and the source language are the same (self-hoisting language).

Bootstrapping: use a compiler to compile new versions of itself.
The target of a compiler need not be an executable; any target language will do.

Often code is produced for further processing (e.g., assembling, linking).

Source-to-source compiler: translates a high-level language into another high-level language.
Interpreters

§1.1

An interpreter directly runs a source program on user-supplied input:

- Source code is available when the program is run.
- Often better run-time error reporting.
- Code typically runs slower than if it had been compiled.
Hybrid approaches

§1.1

An alternative approach: high-level source code is translated into code for a virtual machine (implemented in an interpreter).

source program → compiler → intermediate program → virtual machine → output

Just-in-time compilation: an interpreter first translates the source code into machine code and then runs the resulting executable.
1.2 Relevance of Compiler Construction
Very few people will have to write a compiler for a full-scale programming language.

Still, compiler construction is taught in almost all computer-science curricula in academia.
Compiler construction is one of the success stories of computer science.

Compilers are prime examples of

- well-designed, structured software with well-understood architectures,
- the application of formal methods in software development.
Wide applicability

Lots of issues that arise in compiler construction are relevant to other areas of software engineering as well.

Many problems are naturally expressed as translation problems.

Some problems even ask for special (domain-specific) programming languages to be developed.
1.3 History
A-0 system

The first electronic computers were programmed in machine language and, later, in assembly language.

Compilation was introduced by Grace Hopper in the A-0 system (1952):

- A-0 programs were subroutines identified by numeric codes.
- Calls to routines were denoted by juxtaposing the numeric code and call arguments.
- Today, A-0 would be considered a linker.
The first compiler for a higher language was the FORTRAN compiler by John Backus and his team at IBM (1957).

Initially, the attitude towards higher languages was sceptical: they were not expected to compete, performancewise, with assembly languages.

However, the FORTRAN compiler carried out heavy optimizations, resulting in impressively efficient code.

Moreover: a typical FORTRAN program was about 20 times smaller than the corresponding assembly program.
COBOL was the first language that could be compiled to multiple platforms (1960).

In 1962, Timothy Hart and Michael Levin created the Lisp 1.5 compiler, which was the first bootstrapping compiler.

During the 1960s and 1970s, the number of proposed programming languages increased rapidly; focus shifted from generation of fast code towards tools and techniques for implementing compilers and interpreters.
In 1977, Niklaus Wirth wrote *Compilerbau*, an influential textbook on compiler construction, in which he presented the stepwise implementation of a compiler for PL/0.

Notable features were

- the use of a recursive descent parser for syntactic analysis,
- portable P-code as a target of code generation,
- use of T-diagrams as a means for describing the bootstrapping problem.
Recent years are characterized by the emergence of new programming paradigms (OO, functional programming). These rely on run-time facilities that exceed the capabilities of typical hardware architectures. Challenge for implementors: mapping advanced high-level language concepts onto native machine languages.
1.4 A Tour of Trinity
Trinity is a small programming language designed by Ralf Hinze and Andres Löh to illustrate various concepts in programming-language design.

Different paradigms:

- value-oriented (functional),
- effect-oriented (imperative),
- object-oriented.

In this course, we use a particular subset, dubbed Featherweight Trinity.
Numeric expressions

\[ 2 + 3 \]

\[ 5 \times (7 + 11) \]

Numeric operators: \(+, -, \times, /, \%\).
Boolean expressions

\[ \text{false} \]

\[ \text{true} \]

\[ 17 \leq 19 \]

\[ \text{if } 2 \equiv 3 \text{ then false else true} \]

Relational operators: \( \preceq, \preceq, \equiv, \neq, \succeq, \succeq \).
String expressions

"Featherweight"

"Featherweight\nTrinity"

String concatenation: .

"Featherweight " . "Trinity"

Literals may span multiple lines.
Functions

§1.4

fun \( x \Rightarrow x \)

fun \( m \Rightarrow \text{fun } n \Rightarrow m + n \)

(fun \( x \Rightarrow \text{fun } y \Rightarrow x \)) 2 3

fun \( \text{rec } n \Rightarrow \text{if } n \equiv 0 \text{ then } 1 \text{ else } n \ast \text{self} \ (n - 1) \)
let val \( n = 2 \)
in  \( n + n \)
end

let val \( \text{fac} = \text{fun rec } n \Rightarrow \) if  \( n \equiv 0 \)
then 1
else  \( n \ast \text{self } (n - 1) \)
val \( \text{fib} = \text{fun rec } n \Rightarrow \) if  \( n \not\leq 2 \)
then  \( n \)
else  \( \text{self } (n - 2) + \text{self } (n - 1) \)
in  \( \text{fib } (\text{fac } 3) \)
end
Mutable references

```ml
let val r = ref 2
in  ! r
end

r := 3

Sequencing: .;

let val r = ref 2
    val s = ref 3
in  r := ! r + ! s;
    ! r
end
```
Comments

C-style nonnested comments:

// Featherweight

/* Trinity */
1.5 The Structure of a Compiler
Conceptually, the compilation process is divided into several stages.

Stages are grouped:

- front end (source-language specific),
- middle end (independent from source and target language),
- back end (target-language specific).
Lexical analysis

§1.5

source program \rightarrow \text{scanner} \rightarrow \text{tokens}

Tasks:

- breaking up the source program in lexical units (tokens): keywords, identifiers, literals, etc.,
- disposal of white space, line breaks, and comments,
- labeling tokens with line and column numbers (for error reporting).
Syntactic analysis

Tasks:

- inferring the hierarchical structure of the program,
- grouping lexical units in syntactic patterns (terms): declarations, statements, expressions, etc.,
- constructing the abstract syntax tree,
- reporting/correcting syntactic errors.
Semantic analysis

Tasks:

▶ semantics checks: asserting that no undefined identifiers are used, type checking, type inference, etc.,
▶ reporting semantic errors.

Programs that pass the semantic analyses are considered correct.
Translation into intermediate code

Intermediate code:

- independent from source and target language,
- low-level, restricted syntax that facilitates further analysis.

Tasks:

- simplifying source-language constructs,
- control-flow analysis,
- data-flow analysis,
- . . .
Tasks:

- dead-code elimination,
- constant propagation,
- common subexpression elimination,
- inlining,
- partial evaluation,
- rewriting,
- 
-
Code generation

![Diagram: intermediate code → code generator → target program]

Tasks:

- generating a target-language program,
- optimizations specific to the target language: e.g., instruction selection, register allocation.