1 Preliminaries

2 Introduction
Lectures:
- Thomas Noll (noll@cs.rwth-aachen.de)
- Uwe Naumann (naumann@stce.rwth-aachen.de)

Exercise classes:
- Friedrich Gretz (fgretz@cs.rwth-aachen.de)
- Christina Jansen (christina.jansen@cs.rwth-aachen.de)
Evaluation of exercises
Organizational support
12 hrs/week contract
Previous CC lecture not a prerequisite (but of course helpful)
BSc Informatik:
- Wahlpflichtfach Theorie

MSc Informatik:
- Theoretische Informatik

MSc Software Systems Engineering:
- Theoretical Foundations of SSE (was: Theoretical CS)

Diplomstudiengang Informatik:
- Theoretische (+ Praktische) Informatik
- Vertiefungsfach **Formale Methoden, Programmiersprachen und Softwarevalidierung**
- Combination with Katoen, Thomas, Vöcking, ...; Kobbelt, Seidl, ...
What you can expect:

- how to implement (imperative) programming languages
- application of theoretical concepts
- compiler = example of a complex software architecture
- gaining experience with tool support

What we expect: basic knowledge in

- imperative programming languages
- algorithms and data structures
- formal languages and automata theory
Organization

**Schedule:**
- Lecture Wed 10:00–11:30 AH 6 (starting 4 April)
- Lecture Thu 15:00–16:30 AH 5 (starting 5 April)
- Exercise class Mon 10:00–11:30 AH 2 (starting 16 April)
- see overview at [http://www-i2.informatik.rwth-aachen.de/i2/cc12/](http://www-i2.informatik.rwth-aachen.de/i2/cc12/)

**1st assignment sheet** next week, presented 16 April

Work on assignments in **groups of three**

**Written exams** (2 h) on Thu 12 July/Mon 24 September
- for BSc/MSc candidates (6 credits)
- for Diplom candidates (Übungsschein)

**Admission** requires at least 50% of the points in the exercises

Written material in **English**, lecture and exercise classes in **German**, rest up to you
1 Preliminaries

2 Introduction
What Is It All About?

Compiler = Program: Source code → Target code

Source code: in high-level programming language, tailored to problem
- imperative vs. declarative (functional, logic) vs. object-oriented
- sequential vs. concurrent

Target code: usually byte/assembly/machine code, tailored to machine
- architecture dependent (RISC/CISC/parallel)
Usage of Compiler Technology I

Programming language interpreters

- Ad-hoc implementation of small programs in scripting languages (perl, bash, ...)
- Programs usually interpreted, i.e., executed stepwise
- Moreover: many non-scripting languages involve interpreters (e.g., JVM as byte code interpreter)

```
# TIFFCompress  J. E. Brown  Sat 02/28/05
#
# Compresses a block-and-white TIFF image.
#
# File is edited in place.
#
# Usage: TIFFCompress filename
#
if "($*)" == 1
  Set filename "($*)"
else
  Echo "Usage: {0} filename" > Dev:StdErr
  Exit 1
end

if Not "Exists '{filename}'"
  Echo "{0}: file '{filename}' does not exist" > Dev:StdErr
  Exit 1
else if "Exists '{filename}'" And Not "Exists -w '{filename}'"
  Echo "{0}: file '{filename}' is not writable" > Dev:StdErr
  Exit 1
end

# ---------------------
set tempfile "$(TopFolder)"TIFFCompress.tmp
lifscp -c g4 "($filename)" "($tempfile)"
Duplicate -y "$tempfile" "$filename" # -y avoids dialog
Setfile -c dgses "$filename" # for Mac OS 8
Setfile -c p+w "$filename" # for Mac OS 9
```
Web browsers

- Receive **HTML (XML)** pages from web server
- Analyse (**parse**) data and **translate** it to graphical representation

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML
<html>
  <head>
    <title>Example</title>
    <link href="screen.css" rel="sty
  </head>
  <body>
    <h1>
      <a href="/">Header</a>
    </h1>
    <ul id="nav">
      <li>
        <a href="one/">One</a>
      </li>
      <li>
        <a href="two/">Two</a>
      </li>
    </ul>
  </body>
</html>
```
Text processors

- \LaTeX = “programming language” for texts of various kinds
- Translated to DVI, PDF, ...

```latex
\documentclass[12pt]{article}
\title{This is the title}
\author{Author One \& Author Two}
\date{\today}
\begin{document}
\maketitle
This is the content of this document.
This is the 2nd paragraph.
Here is an inline formula:
$V = \frac{4}{\pi r^3} \{3\}$
And appearing immediately below
is a displayed formula:
$\$$V = \frac{4}{\pi r^3} \{3\}\$$
\end{document}
```
Properties of a Good Compiler

Correctness

**Goals:** conformance to source and target language specifications; “equivalence” of source and target code
- compiler validation and verification
- proof-carrying code, ...

Efficiency of generated code

**Goal:** target code as fast and/or memory efficient as possible
- program analysis and optimization

Efficiency of compiler

**Goal:** translation process as fast and/or memory efficient as possible (for inputs of arbitrary size)
- fast (linear-time) algorithms
- sophisticated data structures

Remark: mutual tradeoffs!
Aspects of a Programming Language

Syntax: “How does a program look like?”
- hierarchical composition of programs from structural components

Semantics: “What does this program mean?”
- “Static semantics”: properties which are not (easily) definable in syntax (declaredness of identifiers, type correctness, ...)
- “Dynamic semantics”: execution evokes state transformations of an (abstract) machine

Pragmatics
- length and understandability of programs
- learnability of programming language
- appropriateness for specific applications
- ...
Motivation for Rigorous Formal Treatment

Example

1. From NASA’s Mercury Project: FORTRAN DO loop
   - DO 5 K = 1,3: DO loop with index variable K
   - DO 5 K = 1.3: assignment to (real) variable D05K

2. How often is the following loop traversed?
   
   for i := 2 to 1 do ...

   FORTRAN IV: once
   PASCAL: never

3. What if p = nil in the following program?
   
   while p <> nil and p^.key < val do ...

   Pascal: strict Boolean operations ✖
   Modula: non-strict Boolean operations ✓
Historical Development

**Code generation:** since 1940s
- ad-hoc techniques
- concentration on back-end
- first FORTRAN compiler in 1960

**Formal syntax:** since 1960s
- LL/LR parsing
- shift towards front-end
- semantics defined by compiler/interpreter

**Formal semantics:** since 1970s
- operational
- denotational
- axiomatic
- see course *Semantics and Verification of Software*

**Automatic compiler generation:** since 1980s
- [f]lex, yacc, ANTLR, action semantics, ...
- see [http://catalog.compilertools.net/](http://catalog.compilertools.net/)
Compiler Phases

Lexical analysis (Scanner):
- recognition of symbols, delimiters, and comments
- by regular expressions and finite automata

Syntax analysis (Parser):
- determination of hierarchical program structure
- by context-free grammars and pushdown automata

Semantic analysis:
- checking context dependencies, data types, ...
- by attribute grammars

Generation of intermediate code:
- translation into (target-independent) intermediate code
- by tree translations

Code optimization: to improve runtime and/or memory behavior
Generation of target code: tailored to target system
Additionally: optimization of target code, symbol table, error handling
Conceptual Structure of a Compiler

Source code

Lexical analysis (Scanner)

Syntax analysis (Parser)

Semantic analysis

Generation of intermediate code

Code optimization

Generation of machine code

Target code

x1 := y2 + 1

regular expressions/finite automata

(id, x1)(gets, )(id, y2)(plus, )(int, 1)

context-free grammars/pushdown automata

attribute grammars

attribute grammars

tree translations

LOAD y2; LIT 1; ADD; STO x1

... [omitted: symbol table, error handling]
Classification of Compiler Phases

**Analysis vs. synthesis**

- **Analysis**: lexical/syntax/semantic analysis
  (determination of syntactic structure, error handling)
- **Synthesis**: generation of (intermediate/machine) code + optimization

**Front-end vs. back-end**

- **Front-end**: machine-independent parts
  (analysis + intermediate code + machine-independent optimizations)
- **Back-end**: machine-dependent parts
  (generation + optimization of machine code)

**Historical: $n$-pass compiler**

- $n = \text{number of runs through source program}$
- nowadays mainly one-pass
## Literature

(CS Library: “Handapparat Softwaremodellierung und Verifikation”)

### General

### Special
- D. Brown, R. Levine T. Mason: *lex & yacc*, O'Reilly, 1995

### Historical
- N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, 1996