

Compiler Construction

Lecture 1: Introduction

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

<http://www-i2.informatik.rwth-aachen.de/i2/cc08/>

Summer semester 2008

1 Preliminaries

2 Introduction

- Lectures: **Thomas Noll**
 - Lehrstuhl für Informatik 2, Room 4211
 - E-mail noll@cs.rwth-aachen.de
 - Phone (0241)80-21213
- Exercise classes: **Daniel Klink**
 - Lehrstuhl für Informatik 2, Room 4205
 - E-mail klink@cs.rwth-aachen.de
 - Phone (0241)80-21210
- Student assistants:
 - **Johanna Nellen** (johanna.nellen@rwth-aachen.de)
 - **Maximilian Odenbrett** (maximilian.odenbrett@rwth-aachen.de)

Target Audience

- Bachelor program (**Informatik**): V3 Ü2
 - Wahlpflichtfach Theorie
- Master programs (**Software Systems Engineering** [, **Informatik**]): V4 Ü2
 - Theoretical (+ Practical) CS
 - Specialization *Formal Methods, Programming Languages and Software Validation*
- Diplomstudiengang (**Informatik**): V4 Ü2
 - Theoretische (+ Praktische) Informatik
 - Vertiefungsfach *Formale Methoden, Programmiersprachen und Softwarevalidierung*
- In general:
 - interest in implementation of (imperative) programming languages
 - application of theoretical concepts
 - compiler = example of a complex software architecture
 - gaining experience with tool support
- Expected: basic knowledge in
 - imperative programming languages
 - formal languages and automata theory

- Schedule:
 - Lecture Mon 10:00–11:30 AH 2 (starting April 14)
 - Lecture Thu 15:00–16:30 AH 5
 - Exercise class Wed 13:30–15:00 AH 3 (starting 23.10.2006)
- (see overview at
<http://www-i2.informatik.rwth-aachen.de/i2/cc08/>)
- Today: 0th assignment sheet, presented next Wednesday
- Work on assignments in groups of three
- Examination:
 - oral for BSc candidates (6 ECTS credit points)
 - otherwise (8 ECTS credit points) depending on number of candidates
- Admission requires at least 50% of the points in the (non-Diplom) exercises
- Written material in English, lecture and presentation of assignments in German, rest up to you

1 Preliminaries

2 Introduction

What Is It All About?

Compiler = Program: Source code → Machine code

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

Machine code: architecture dependent
(von Neumann; RISC/CISC/parallel)

Important issues:

Correctness: “equivalence” of source and machine code
(\Rightarrow compiler verification, proof-carrying code, ...)

Efficiency of generated code: machine code as fast and/or memory
efficient as possible
(\Rightarrow program analysis and optimization)

Efficiency of compiler: translation process as fast and/or memory
efficient as possible
(\Rightarrow sophisticated algorithms and data structures;
bootstrapping)

Efficiency depends on system environment (mutual **tradeoff**)

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
- ...

- **Formal syntax** since 1960s (LL/LR parsing);
semantics defined by compiler/interpreter
- **Formal semantics** since 1970s
(operational/denotational/axiomatic)
- **Automatic compiler generation** since 1980s
([f]lex, yacc, action semantics, ...)

Examples:

① How often is the following loop traversed?

```
for i := 2 to 1 do ...
```

FORTRAN IV: once

PASCAL: never

② What if $p = \text{nil}$ in the following program?

```
while p <> nil and p^.key < val do ...
```

Pascal: strict Boolean operations ↓

Modula: non-strict Boolean operations ↑

Lexical analysis (Scanner):

- recognition of symbols, delimiters, and comments
- by regular expressions and finite automata

Syntactic 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:

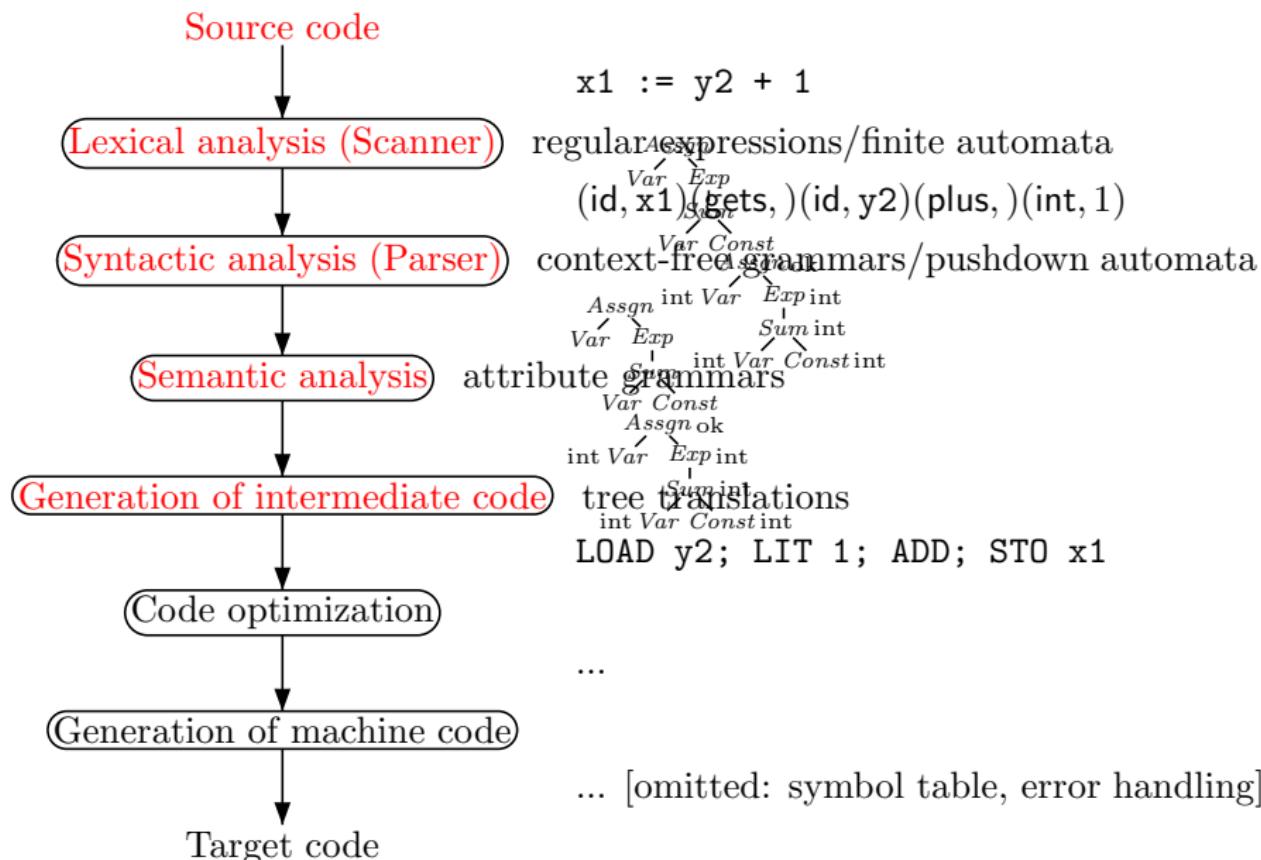
- translation into (target-independent) intermediate code
- by tree translations

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



Classification of Compiler Phases

Analysis: lexical/syntactic/semantic analysis
(determination of syntactic structure, error handling)

Synthesis: generation of (intermediate/machine) code + optimization

Alternatively:

Frontend: machine-independent parts
(analysis + intermediate code + machine-independent optimizations)

Backend: machine-dependent parts
(generation + optimization of machine code)

Another classification: *n*-pass compiler

(number of runs through source program; nowadays mainly one-pass)

(also see the collection [“Handapparat”] at the CS Library)

- A. Aho, R. Sethi, J. Ullman: *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, 1988
- W. Waite, G. Goos: *Compiler Construction*, 2nd edition, Springer, 1985
- R. Wilhelm, D. Maurer: *Übersetzerbau*, 2. Auflage, Springer, 1997
- N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, 1996
- J.R. Levine et al.: *lex & yacc*, O'Reilly, 1992
- A.W. Appel, J. Palsberg: *Modern Compiler Implementation in Java*, Cambridge University Press, 2002
- D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen: *Modern Compiler Design*, Wiley & Sons, 2000
- O. Mayer: *Syntaxanalyse*, BI-Verlag, 1978