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/cc10/

Winter semester 2010/11

Outline

1. Preliminaries

2. Introduction

People

- Lectures: Thomas Noll
  - Lehrstuhl für Informatik 2, Room 4211
  - E-mail `noll@cs.rwth-aachen.de`
  - Phone (0241)80-21213
- Exercise classes: Christina Jansen
  - E-mail `christina.jansen@cs.rwth-aachen.de`
- Student assistants:
  - Stefan Breuer
  - Ernst Wrtal

Target Audience

- BSc Informatik: V3 Ü2, 6 credits
  - Wahlpflichtfach Theorie
- MSc Informatik: V3 Ü2, 6 credits
  - Theoretische Informatik
- MSc Software Systems Engineering: V4 Ü2, 8 credits
  - Theoretical CS
  - Specialization *Formal Methods, Programming Languages and Software Validation*
- Diplomstudiengang Informatik: V4 Ü2
  - Theoretische (+ Praktische) Informatik
  - Vertiefungsfach *Formale Methoden, Programmiersprachen und Softwarevalidierung*
  - Combination with Katoen, Thomas, Vöcking, ...; Kobbelt, Seidl, ...

Expectations

- 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
  - formal languages and automata theory

Organization

- Schedule:
  - Lecture Tue 14:00–15:30 AH 2 (starting October 19)
  - Lecture Thu 13:30–15:00 AH 1 (starting October 14)
  - Exercise class Wed 10:00–11:30 AH 2 (starting October 20)
  - see overview at `http://www-i2.informatik.rwth-aachen.de/i2/cc10/`
- 1st assignment sheet next week, presented October 27
- Work on assignments in groups of three
- Written exam on Tue February 1
  - for BSc/MSc candidates (6/8 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

Outline

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 machine code

- architecture dependent (RISC/CISC/parallel)

More applications of compiler techniques:

- Parsing of structured data (HTML, XML, ...)
- Cross-compiling: Java → C
- File conversion: LaTeX → PDF
- PostScript interpreters
- ...

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 `DO5K`
2. How often is the following loop traversed?

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

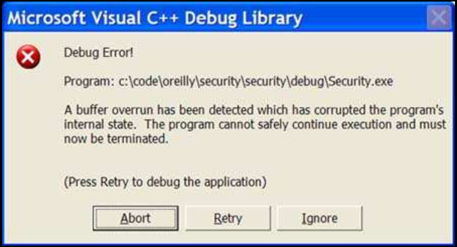
   FORTRAN IV: once

   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 👍



Debug Error!

Program: c:\code\oreilly\security\security\debug\Security.exe

A buffer overrun has been detected which has corrupted the program's internal state. The program cannot safely continue execution and must now be terminated.

(Press Retry to debug the application)

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/`

Compiler Phases

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:

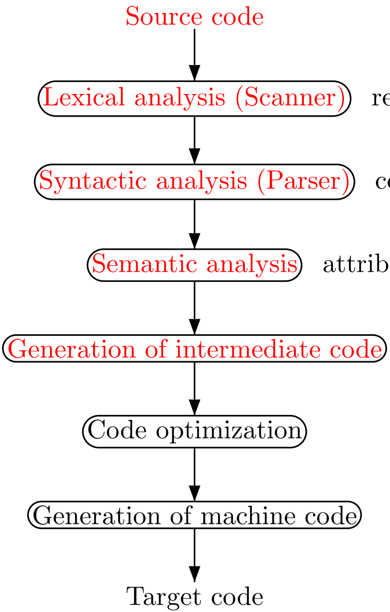
- translation into (target-independent) intermediate code
- by tree translations

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



Source code

Lexical analysis (Scanner)

Syntactic analysis (Parser)

Semantic analysis

Generation of intermediate code

Code optimization

Generation of machine code

Target code

x1 := y2 + 1

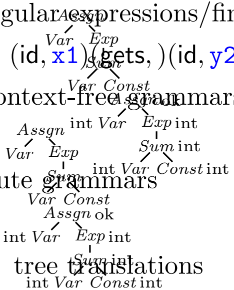
regular expressions/finite automata

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

context-free grammars/pushdown automata

attribute grammars

tree translations



LOAD y2; LIT 1; ADD; STO x1

...

... [omitted: symbol table, error handling]

Classification of Compiler Phases

Analysis vs. synthesis

Analysis: lexical/syntactic/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 = number of runs through source program
- nowadays mainly one-pass

Literature

(CS Library: “Handapparat *Programmiersprachen und Verifikation*”)

General

- A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman: *Compilers – Principles, Techniques, and Tools; 2nd ed.*, Addison-Wesley, 2007
- 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
- R. Wilhelm, D. Maurer: *Übersetzerbau, 2. Auflage*, Springer, 1997

Special

- O. Mayer: *Syntaxanalyse*, BI-Wissenschafts-Verlag, 1978
- D. Brown, R. Levine T. Mason: *lex & yacc*, O’Reilly, 1995
- T. Parr: *The Definite ANTLR Reference*, Pragmatic Bookshelf, 2007

Historical

- W. Waite, G. Goos: *Compiler Construction, 2nd edition*, Springer, 1985
- N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, 1996