Compiler Construction

Lecture 1: Introduction

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH AACHEN UNIVERSITY

noll@cs.rwth-aachen.de

http://moves.rwth-aachen.de/teaching/ss-14/cc14/

Summer Semester 2014

Outline

1. Preliminaries
2. Introduction

People

- Lectures:
  - Thomas Noll (noll@cs.rwth-aachen.de)
- Exercise classes:
  - Friedrich Gretz (fgretz@cs.rwth-aachen.de)
  - Souymodip Chakraborty (chakraborty@cs.rwth-aachen.de)
- Student assistant:
  - Philipp Berger
  - Samiro Discher

Target Audience

- BSc Informatik:
  - Wahlpflicht Theoretische Informatik
- MSc Informatik:
  - Theoretische Informatik
- MSc Software Systems Engineering:
  - Theoretical Foundations of SSE (was: Theoretical CS)

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
  - algorithms and data structures
  - formal languages and automata theory

Organization

- Schedule:
  - Lecture Mon 14:15–15:45 AH 6 (starting 14 April)
  - Lecture Wed 10:15–11:45 AH 6 (starting 9 April)
  - Exercise class Fri 08:15–09:45 AH 2 (starting 16 April)
  - Special: 16 April (exercise), 2/4 June (itestra)
  - see overview at http://moves.rwth-aachen.de/teaching/ss-14/cc14/
- 1st assignment sheet next week, presented 25 April
- Work on assignments in groups of 2-3 people
- Written exams (2 h, 6 Credits) on 25 July/3 September
- 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

What Is It All About?

Compiler = Program: Source code $\rightarrow$ Target code

Source code: in high-level programming language, tailored to problem

- imperative vs. declarative (functional, logic) vs. object-oriented
- sequential vs. concurrent

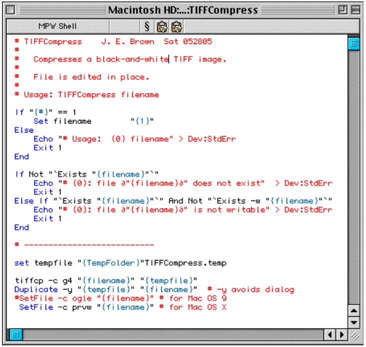
Target code: low-level code, tailored to machine

- platform-independent byte code (for virtual machine such as JVM)
- platform-dependent assembly/machine code (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 also involve interpreters (e.g., JVM as byte code interpreter)



Usage of Compiler Technology II

Web browsers

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

```
<!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>
```

Usage of Compiler Technology III

Text processors

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

```
\documentclass[12pt]{article}
%options include 12pt or 11pt or 10pt
%classes include article, report, book, letter, thesis
\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 I

Efficiency of generated code

Goal: target code as fast and/or memory efficient as possible

- program analysis and optimization
- cf. course on Static Program Analysis (WS 2012/13, 2014/15)

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

Properties of a Good Compiler II

Correctness

Goals: conformance to source and target language specifications; "equivalence" of source and target code

- compiler validation and verification
- proof-carrying code, ...
- cf. course on Semantics and Verification of Software (SS 2013, 2015)

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 $\lightning$
   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
- cf. course on *Semantics and Verification of Software*

Automatic compiler generation: since 1980s

- `[f]lex`, `yacc`, ANTLR, action semantics, ...
- cf. 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) — regular expressions/finite automata

`x1␣:=␣y2␣+␣1;`

↓

$(\mathrm{id}, x1)(\mathrm{gets},)(\mathrm{id}, y2)(\mathrm{plus},)(\mathrm{int}, 1)$

Syntax analysis (Parser) — context-free grammars/pushdown automata

Assgn [*Var* , *Exp* [*Sum* [*Var* , *Const*]]]

↓

Semantic analysis — attribute grammars

Assgn ok [int *Var* , *Exp* int [*Sum* int [int *Var* , *Const* int]]]

↓

Generation of intermediate code — tree translations

`LOAD y2; LIT 1; ADD; STO x1`

↓

Code optimization — ...

↓

Generation of machine code — ... [omitted: symbol table, error handling]

↓

Target code

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 = number of runs through source program
- nowadays mainly one-pass

Literature

(CS Library: "Handapparat *Softwaremodellierung 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