Compiler Construction

Lecture 1: Introduction

Summer Semester 2017

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-17/cc/

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

People

- Lectures:
  - Thomas Noll (noll@cs.rwth-aachen.de)
- Exercise classes:
  - Sebastian Junges (sebastian.junges@cs.rwth-aachen.de)
  - Christoph Matheja (matheja@cs.rwth-aachen.de)
  - Matthias Volk (matthias.volk@cs.rwth-aachen.de)
- Student assistants:
  - Justus Fesefeldt
  - Louis Wachtmeister

Target Audience

- BSc Informatik:
  - Wahlpflicht Theoretische Informatik
- MSc Informatik:
  - Theoretische Informatik
- MSc Software Systems Engineering:
  - Theoretical Foundations of SSE
- ...

Expectations

- What you can expect:
  - how to implement (imperative) programming languages
  - application of theoretical concepts (scanning, parsing, static analysis, ...)
  - 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 (queues, trees, ...)
  - formal languages and automata theory (regular and context-free languages, finite and pushdown automata, ...)

Organisation

- Schedule:
  - Lecture Tue 14:15–15:45 AH 6 (starting 2 May)
  - Lecture Thu 14:15–15:45 AH 1 (starting 4 May)
  - Exercise class Tue 12:15–13:45 AH 2 (starting 9 May)
  - Special: 27/29 June (itestra)
  - see overview at https://moves.rwth-aachen.de/teaching/ss-17/cc/
- Exercises:
  - 1st assignment sheet next week, presented 16 May
  - Work on assignments in groups of three people
- Exam:
  - Written exams (2 h, 6 Credits) on 4 August/19 September
  - Registration by 19 May
  - Admission requires at least 50% of the points in the exercises
- Written material in English (including exam), lecture and exercise classes in German, rest up to you

What Is a Compiler?

What Is It All About?

https://en.wikipedia.org/wiki/Compiler

"A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. The most common reason for converting source code is to create an executable program."

Compiler vs. interpreter

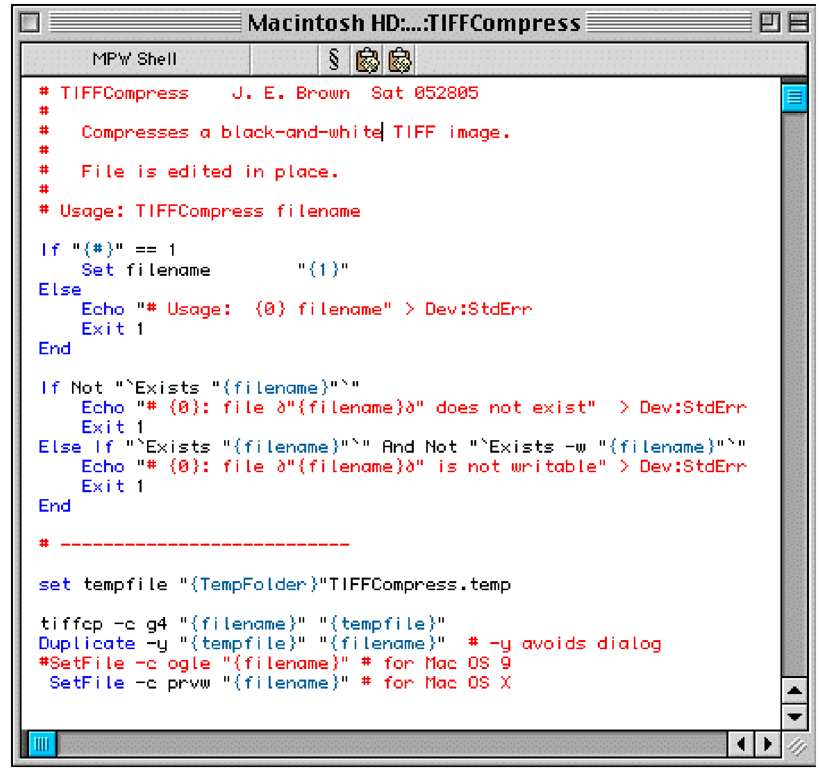
Compiler: translates an executable program in one language into an executable program in another language (possibly applying "improvements")

Interpreter: directly executes an executable program, producing the corresponding results

Usage of Compiler Technology I

Programming language interpreters

- Ad-hoc implementation of small programs in scripting languages (JavaScript, Perl, Ruby, 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

```
<!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>
```

Web browsers

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

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

Correctness

Goals:

syntactic correctness: conformance to source and target language specifications
semantic correctness: "equivalence" of source and target code

Techniques:

- compiler validation and verification
- proof-carrying code, ...
- cf. course on *Semantics and Verification of Software* (SS 2015, WS 2017/18)

Properties of a Good Compiler II

Efficiency of generated code

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

- program analysis and optimisation
- cf. course on *Static Program Analysis* (WS 2016/17)

Efficiency of compiler

Goal: translation process as fast and/or memory efficient as possible (for input programs 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 (keywords, identifiers, expressions, statements, ...)

Semantics: "What does this program mean?"

- "Static semantics": properties which are not (easily) definable in syntax (declaredness of identifiers, type correctness, ...)
- "Operational 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 1.1

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`

   (cf. Dirk W. Hoffmann: *Software-Qualitt*, 2nd ed., Springer 2013)
2. How often is the following loop traversed?

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

   FORTRAN IV: once
   Pascal: never
3. What if value of `p` is `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
- cf. course on *Semantics and Verification of Software*

Automatic compiler generation: since 1980s

- `[f]lex`, `yacc/bison`, ANTLR, ...
- cf. `https://www.thefreecountry.com/programming/compilerconstruction.shtml`

The High-Level View

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 optimisation: to improve runtime and/or memory behavior

Generation of target code: tailored to target system

Additionally: optimisation of target code, symbol table, error handling

Conceptual Structure of a Compiler

Source code

`x1␣:=␣y2␣+␣1;`

Lexical analysis (Scanner)

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

Syntax analysis (Parser)

Semantic analysis

Generation of intermediate code

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

Code optimisation

...

Generation of target code

...

Target code

Asg
Var *Exp*
Sum
Var *Con*

Asg ok
int *Var* *Exp* int
Sum int
int *Var* *Con* int

regular expressions/
finite automata

context-free grammars/
pushdown automata

attribute grammars

tree translations

[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/target) code + optimisation

Front-end vs. back-end

Front-end: machine-independent parts
(analysis + intermediate code + machine-independent optimisations)

Back-end: machine-dependent parts (generation + optimisation of target code)

- instruction selection
- register allocation
- instruction scheduling

Role of the Runtime System

- Memory management services
  - allocation (on heap/stack)
  - deallocation
  - garbage collection
- Run-time type checking (for non-“strongly typed” languages)
- Error processing, exception handling
- Interface to the operating system (input and output, ...)
- Support for parallelism (communication and synchronisation)

Literature

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

Specific

- 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