Compiler Construction

Lecture 1: Introduction

Winter Semester 2018/19

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-1819/cc/

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Outline of Lecture 1

People

- Lectures:
  - Thomas Noll (noll@cs.rwth-aachen.de)
- Exercise classes:
  - Philipp Berger (berger@cs.rwth-aachen.de)
  - Matthias Volk (matthias.volk@cs.rwth-aachen.de)
- Student assistant:
  - Justus Fesefeldt

Target Audience

- BSc Informatik:
  - Wahlpflicht Theoretische Informatik
- MSc Informatik:
  - Wahlpflicht 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

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, stacks, trees, ...)
  - formal languages and automata theory (regular and context-free languages, finite and pushdown automata, ...)

Organisation

- Schedule:
  - Lecture Mon 12:30–14:00 AH 6 (starting 8 Oct)
  - Lecture Thu 12:30–14:00 AH 5 (starting 11 Oct)
  - Exercise class Mon 15:15–16:45 AH 5 (starting 15 Oct)
  - Two special lectures in mid-December (itestra)
  - see overview at https://moves.rwth-aachen.de/teaching/ws-1819/cc/

Organisation

- Schedule:
  - Lecture Mon 12:30–14:00 AH 6 (starting 8 Oct)
  - Lecture Thu 12:30–14:00 AH 5 (starting 11 Oct)
  - Exercise class Mon 15:15–16:45 AH 5 (starting 15 Oct)
  - Two special lectures in mid-December (itestra)
  - see overview at `https://moves.rwth-aachen.de/teaching/ws-1819/cc/`
- Exercises:
  - 1st assignment sheet next week, presented 22 Oct
  - Work on assignments in groups of ~~three~~four people
  - Recap of foundations (finite automata, ...) on 15 Oct

Organisation

- Schedule:
  - Lecture Mon 12:30–14:00 AH 6 (starting 8 Oct)
  - Lecture Thu 12:30–14:00 AH 5 (starting 11 Oct)
  - Exercise class Mon 15:15–16:45 AH 5 (starting 15 Oct)
  - Two special lectures in mid-December (itestra)
  - see overview at https://moves.rwth-aachen.de/teaching/ws-1819/cc/
- Exercises:
  - 1st assignment sheet next week, presented 22 Oct
  - Work on assignments in groups of ~~three~~four people
  - Recap of foundations (finite automata, ...) on 15 Oct
- Exam:
  - Written exams (2 h, 6 Credits) 21.02.2019 and 25.03.2019
  - Registration between 3 Dec and 10 Jan
  - Admission requires at least 50% of the points in the exercises

Organisation

- Schedule:
  - Lecture Mon 12:30–14:00 AH 6 (starting 8 Oct)
  - Lecture Thu 12:30–14:00 AH 5 (starting 11 Oct)
  - Exercise class Mon 15:15–16:45 AH 5 (starting 15 Oct)
  - Two special lectures in mid-December (itestra)
  - see overview at https://moves.rwth-aachen.de/teaching/ws-1819/cc/
- Exercises:
  - 1st assignment sheet next week, presented 22 Oct
  - Work on assignments in groups of ~~three~~four people
  - Recap of foundations (finite automata, ...) on 15 Oct
- Exam:
  - Written exams (2 h, 6 Credits) 21.02.2019 and 25.03.2019
  - Registration between 3 Dec and 10 Jan
  - 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

Outline of Lecture 1

What Is a Compiler?

What Is It All About?

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

"A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language)... The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program."

What Is It All About?

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

"A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language)... The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program."

Compiler vs. interpreter

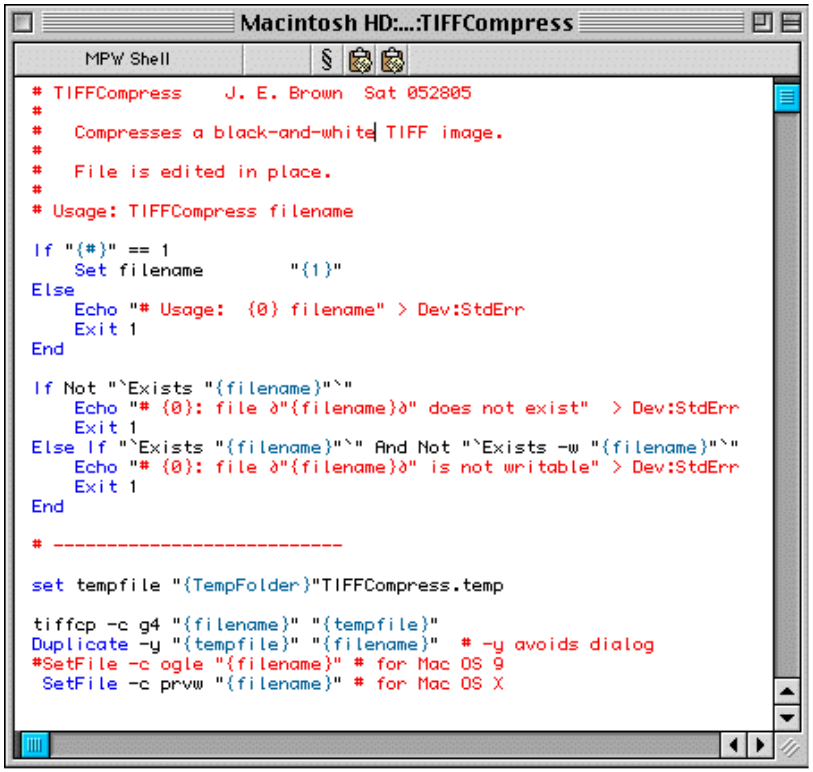
Compiler: translates an executable program in one language into an executable program in another language (possibly applying "improvements")

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

Web browsers

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

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML
2  <html>
3      <head>
4          <title>Example</title>
5          <link href="screen.css" rel="sty
6      </head>
7      <body>
8          <h1>
9              <a href="/">Header</a>
10         </h1>
11         <ul id="nav">
12             <li>
13                 <a href="one/">One</a>
14             </li>
15             <li>
16                 <a href="two/">Two</a>
17             </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}
```

Outline of Lecture 1

Expected Properties of a Compiler I

Correctness of translation

Goals:

syntactic correctness: conformance to source and target language specifications

- accept all (and only) syntactically valid input programs
- produce correct target code

semantic correctness: “equivalence” of source and target code

- behaviour of target code “corresponds to” (expected) behaviour of source code

Expected Properties of a Compiler I

Correctness of translation

Goals:

syntactic correctness: conformance to source and target language specifications

- accept all (and only) syntactically valid input programs
- produce correct target code

semantic correctness: “equivalence” of source and target code

- behaviour of target code “corresponds to” (expected) behaviour of source code

Techniques:

- compiler validation and verification
- proof-carrying code, ...
- cf. course on *Semantics and Verification of Software* (WS 2017/18, SS 2019)

Expected Properties of a Compiler II

Efficiency of generated code

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

Expected Properties of a Compiler II

Efficiency of generated code

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

Techniques:

- program analysis and optimisation
- cf. course on *Static Program Analysis* (SS 2018)

Expected Properties of a Compiler II

Efficiency of generated code

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

Techniques:

- program analysis and optimisation
- cf. course on *Static Program Analysis* (SS 2018)

Efficiency of compiler

Goal: translation process as fast and/or memory efficient as possible (for input programs of arbitrary size)

Expected Properties of a Compiler II

Efficiency of generated code

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

Techniques:

- program analysis and optimisation
- cf. course on *Static Program Analysis* (SS 2018)

Efficiency of compiler

Goal: translation process as fast and/or memory efficient as possible (for input programs of arbitrary size)

Techniques:

- fast (linear-time) algorithms
- sophisticated data structures

Expected Properties of a Compiler II

Efficiency of generated code

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

Techniques:

- program analysis and optimisation
- cf. course on *Static Program Analysis* (SS 2018)

Efficiency of compiler

Goal: translation process as fast and/or memory efficient as possible (for input programs of arbitrary size)

Techniques:

- 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, ...)

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

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-Qualität*, 2nd ed., Springer 2013)

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-Qualität*, 2nd ed., Springer 2013)
2. How often is the following loop traversed?

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

   FORTRAN IV: once
   Pascal: never

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-Qualität*, 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

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

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*

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`

Outline of Lecture 1

Compiler Phases

Lexical analysis (Scanner):

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

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

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

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

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

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

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:

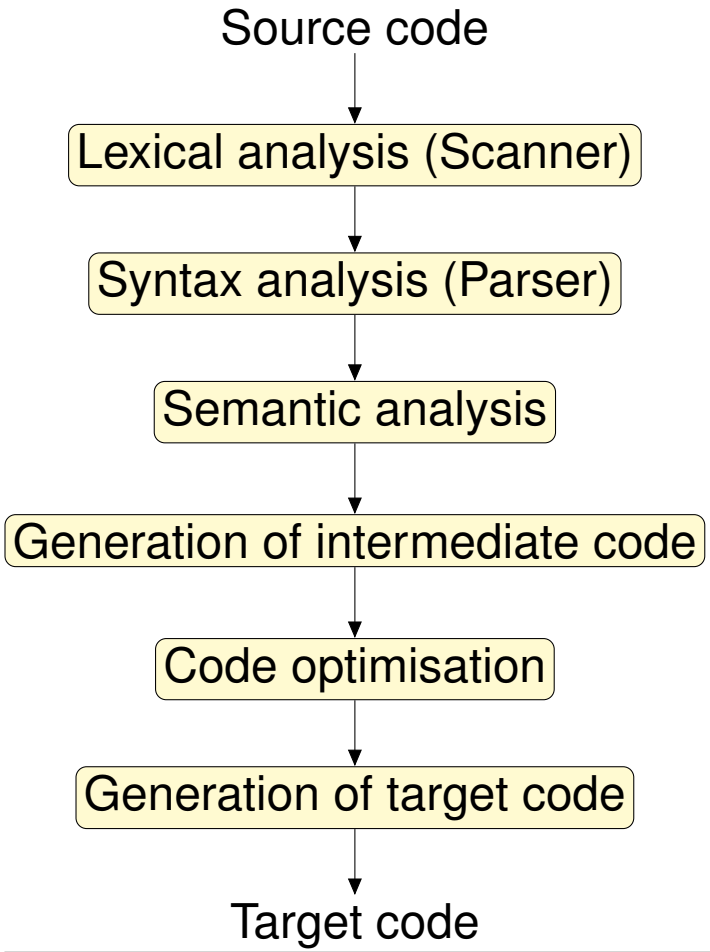
- translation into (target-independent) intermediate code
- by tree translations

Code optimisation: to improve runtime and/or memory behavior

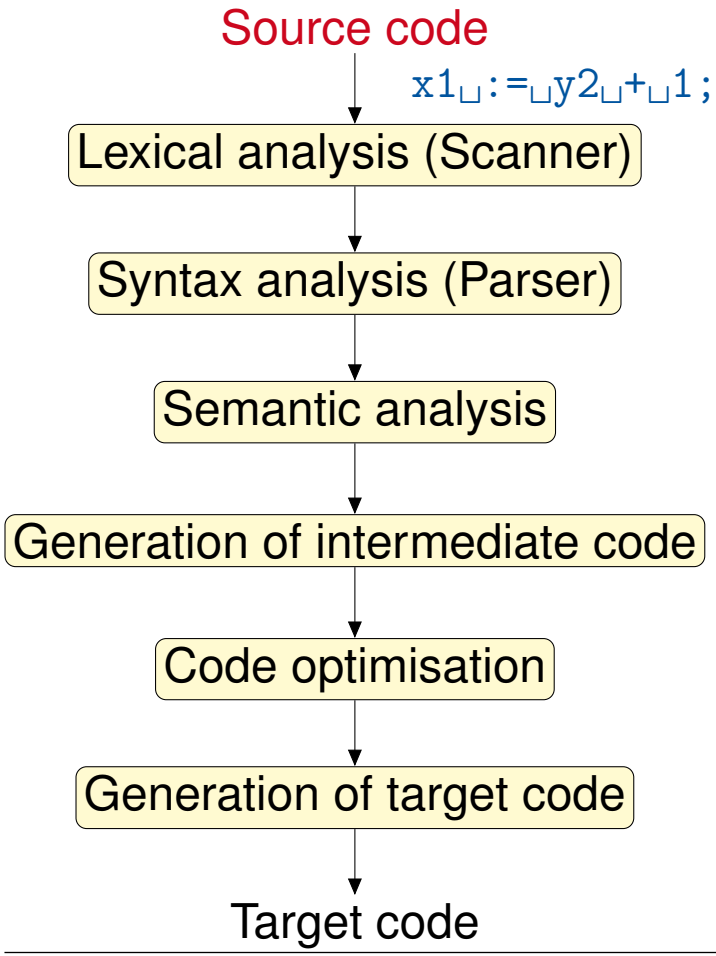
Generation of target code: tailored to target system

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

Conceptual Structure of a Compiler

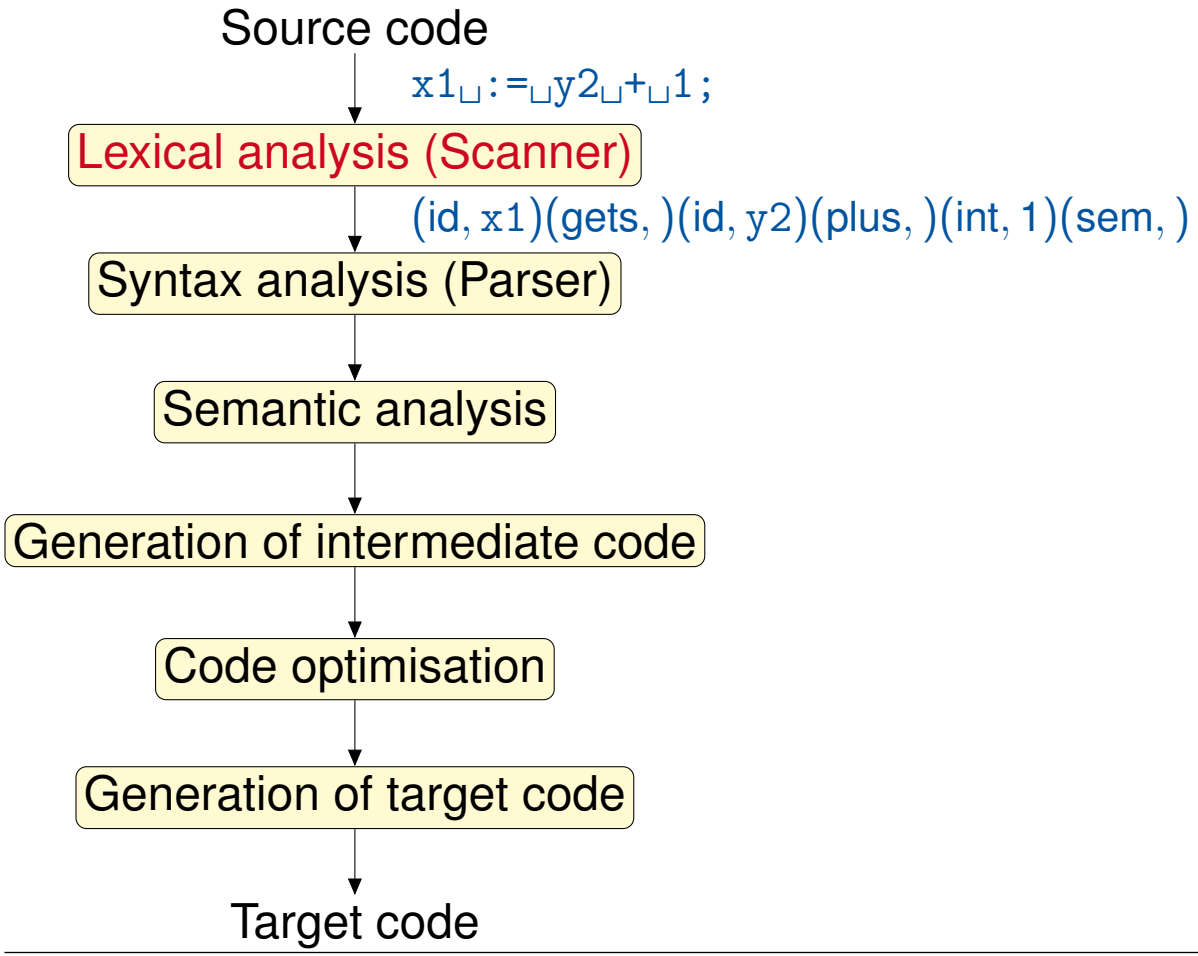


Conceptual Structure of a Compiler



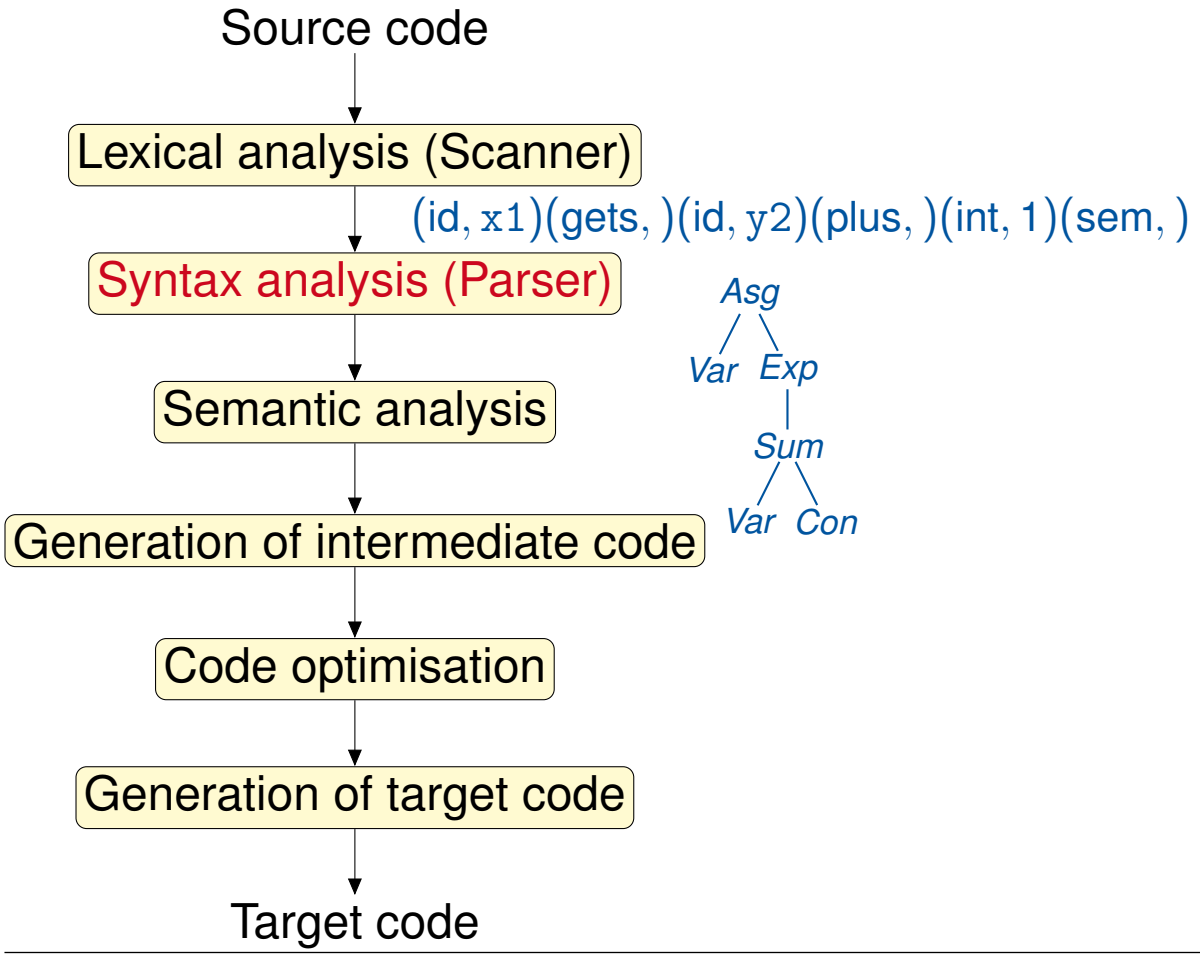
The High-Level View

Conceptual Structure of a Compiler



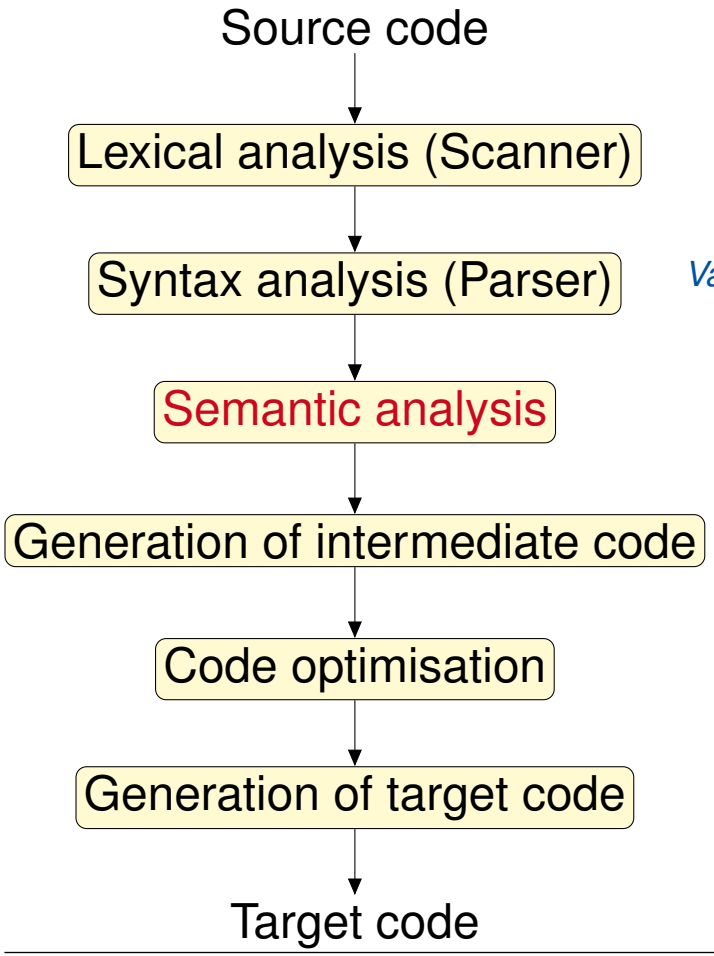
regular expressions/
finite automata

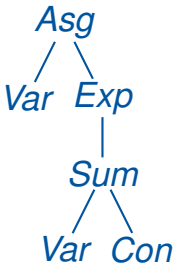
Conceptual Structure of a Compiler

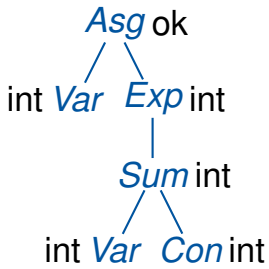


context-free grammars/
pushdown automata

Conceptual Structure of a Compiler



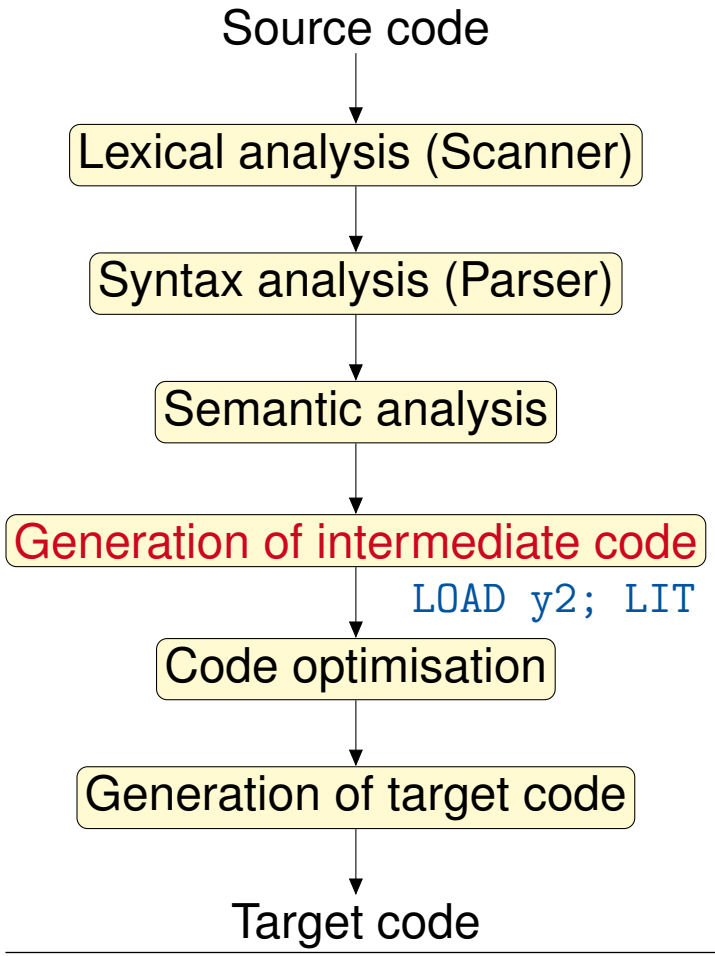


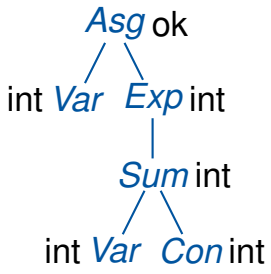


attribute grammars

The High-Level View

Conceptual Structure of a Compiler

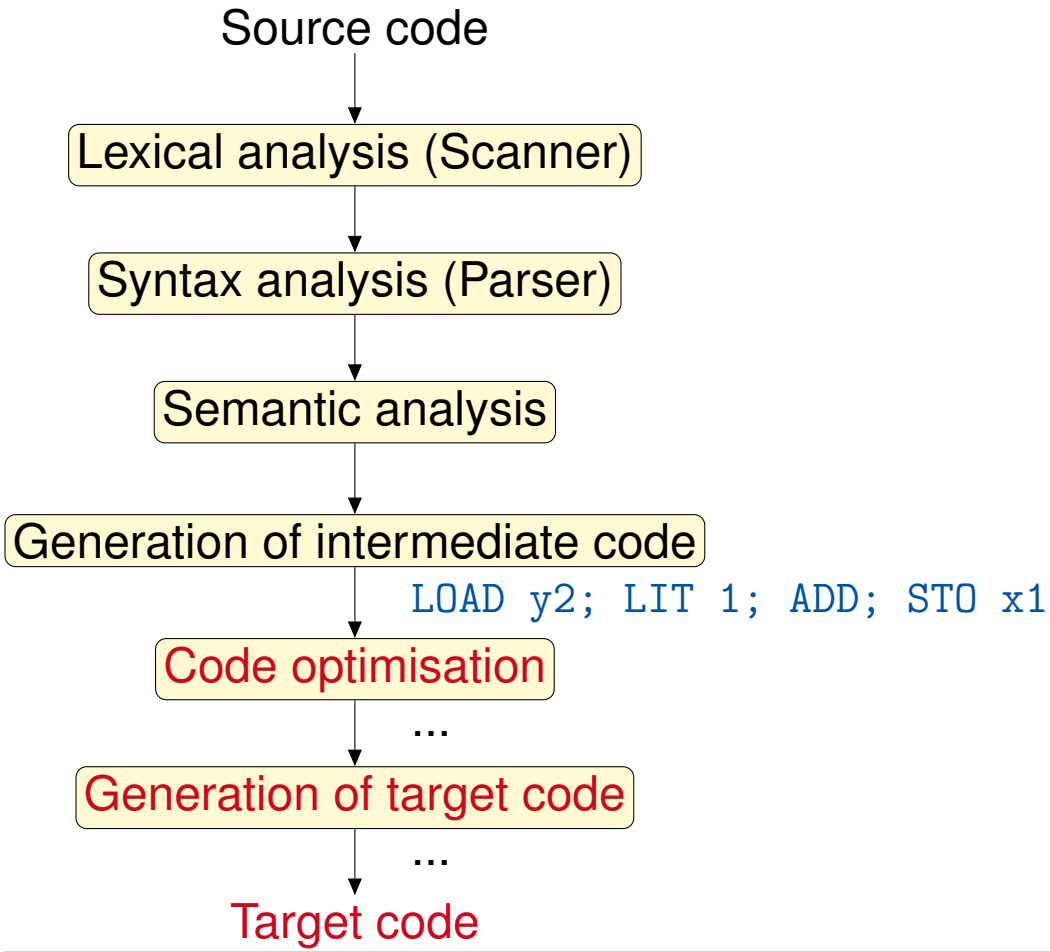




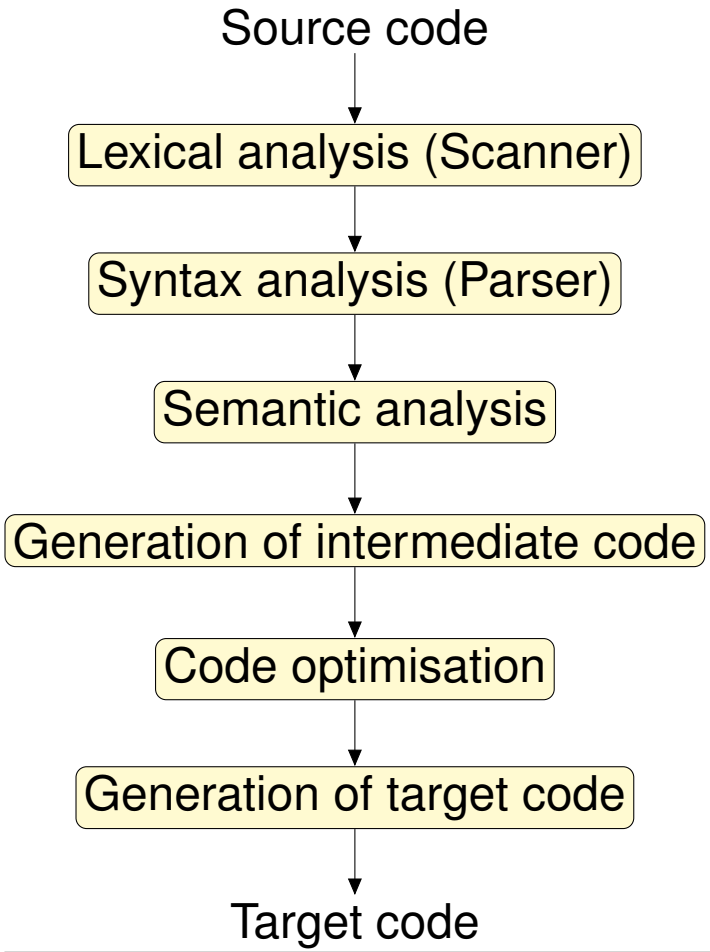
tree translations

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

Conceptual Structure of a Compiler



Conceptual Structure of a Compiler



[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

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)

Outline of Lecture 1

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

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