

# Compiler: Einführung

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik  
Hochschule München

Letzte Änderung: 10.05.2017 15:49

## Inhaltsverzeichnis

Ein Compiler ...	2
Ein Compiler ...	2
Das Studium vom Compilerbau ...	2
Konzeptionelle Roadmap	2
Programmiersprachen	3
Compiler	3
Interpreter	3
Kombinationen	4
oft als JIT	4
Warum sollten Sie sich mit Compilerdesign beschäftigen...	4
Theorie und Praxis	4
Die elementaren Prinzipien der Compilierung	5
Erstes Prinzip	5
Zweites Prinzip	5
Compilerstruktur	5
Compilerstruktur ...	5
Optimierer	6
<b>Überblick über die Compilierung</b>	<b>6</b>
Frontend	6
Scanner	7
Parser	7
Typüberprüfung (type checking)	7
Zwischenrepräsentation	8
Zwischenrepräsentation: Beispiel Graph	8
Zwischenrepräsentation: Beispiel lowlevel, sequentieller Code	8

Der Optimierer . . . . .	9
Beispiel . . . . .	9
Das Backend . . . . .	10

## Ein Compiler ...

- ist ein Computerprogramm
- das ein Programm
  - geschrieben in einer Sprache
- in ein Programm
  - geschrieben in einer anderen Sprache
- übersetzt

## Ein Compiler ...

- ist ein großes Softwaresystem
- mit vielen
  - internen Komponenten,
  - Algorithmen und
  - komplexen Interaktionen zwischen beiden

## Das Studium vom Compilerbau ...

- ist eine Einführung in Techniken für die
  - Übersetzung und
  - Optimierung von Programmen
- und eine praktische Übung in
  - Software Engineering

## Konzeptionelle Roadmap

- ein Compiler muss die
  - **Syntax** und die
  - **Bedeutung**

der Eingabesprache sowie die

- **Syntax** und die
- **Bedeutung**

der Ausgabesprache verstehen

- und ein Compiler benötigt ein
    - **Abbildungsschema**
- von der Quell- zur Zielsprache

## Programmiersprachen

- formale Sprachen
- haben rigide Eigenschaften und Bedeutungen
- sind so entworfen, dass sie keine Mehrdeutigkeiten enthalten

## Compiler

- Quellsprachen
  - z.B. Java, C, C++, ...
- Zielsprachen
  - z.B. Maschinsprache, Bytecode für die JVM oder .NET
- es gibt aus *source-to-source-translators*
  - z.B. Übersetzung einer neuen Programmiersprache nach C
- und andere, z.B.
  - LaTeX to PostScript
  - Markdown to HTML

## Interpreter

- führen ein Programm aus
- benötigen aber auch Scanner, Parser, Zwischenrepräsentationen, ...
- z.B. Python, Perl, Scheme, ...

## Kombinationen

- Compiler nach Bytecode
- Bytecode wird von einem Interpreter ausgeführt

### oft als JIT

- Compiler der zur Laufzeit übersetzt
- *just-in-time compiler*

## Warum sollten Sie sich mit Compilerdesign beschäftigen...

- Ein Compiler enthält einen Informatik-Mikrokosmos
  - gierige (greedy) Algorithmen (register allocation)
  - heuristische Suchtechniken (list scheduling)
  - Graphalgorithmen (dead-code elimination)
  - dynamische Programmierung (instruction selection)
  - endliche Automaten und Kellerautomaten (scanning and parsing)
  - Fixpunktalgorithmen (data-flow analysis)
- Ein Compiler muss mit folgenden Problemen umgehen können
  - dynamischer Allokation
  - Synchronization
  - Namensgebung
  - Lokalität
  - hierarchischem Speichermanagement
  - Pipelinescheduling

## Theorie und Praxis

Compiler demonstrieren den erfolgreiche Anwendung der Theorie auf praktische Probleme

- Werkzeuge zum Generieren von Scannern und Parsern
  - wenden Ergebnisse aus der formalen Sprachtheorie an
  - die selben Werkzeuge werden für die Textsuche, das Filtern von Websites, die Textverarbeitung und Kommandozeileninterpreter verwendet
- Typüberprüfung und statische Analyse
  - wenden Ergebnisse der Verbandstheorie, der Zahlentheorie und anderer mathematischer Teilgebiete an

- Codegeneratoren
  - benutzen Algorithmen für die Mustererkennung in Bäumen, das Parsen, die dynamische Programmierung und Textvergleiche

## Die elementaren Prinzipien der Compilierung

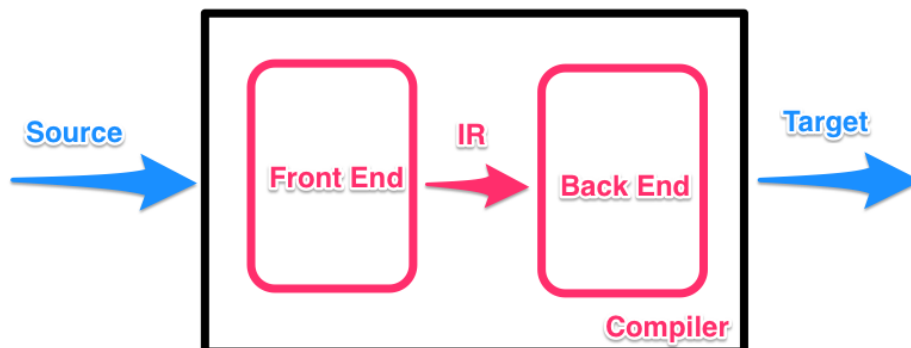
### Erstes Prinzip

Der Compiler muss die Bedeutung des zu compilierenden Programms erhalten.

### Zweites Prinzip

Der Compiler muss das Eingabeprogramm auf wahrnehmbare Art und Weise verbessern.

## Compilerstruktur



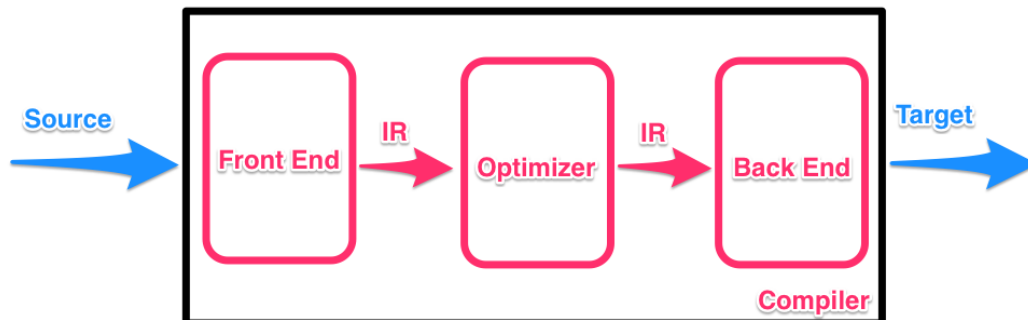
Ein 2-Phasen-Compiler

## Compilerstruktur ...

- das Frontend muss seine Kenntnis der Quellsprache nutzen

- muss sicherstellen, dass das Quellprogramm wohlgeformt ist
- die Zwischenrepräsentationen (intermediate representation, IR) ist die maßgebliche Repräsentation des Codes für den Compiler
  - es kann mehrere Zwischenrepräsentationen für verschiedene Compilerphasen geben
- das Backend muss die IR abbilden auf
  - die Befehlsmenge und
  - die endlichen Ressourcen

## Optimierer



Ein 3-Phasen-Compiler

## Überblick über die Compilierung

### Frontend

- muss die Programmstruktur mit deiner **Sprachdefinition** vergleichen
- formale Definition = endliche Menge von Regeln = **Grammatik**, z.B.  
*Satz* → *Subjekt* *Verb* *Objekt* *Satzendezeichen*

- **Verb** und **Satzendezeichen** sind Teile der Sprache
- *Satz*, *Subjekt* und *Objekt* sind syntaktische Variablen
- das Symbol  $\rightarrow$  heisst “leitet her”

## Scanner

- Beispiel: “Compiler sind technische Objekte.”
- der Scanner
  - nimmt diesen Zeichenstrom und
  - wandelt ihn um in einen Strom von klassifizierten Worten
- Beispiel:  
(Substantiv, “Compiler”), (Verb, “sind”),  
(Adjektiv, “technische”), (Substantiv, “Objekte”),  
(Satzendezeichen, “.”)

## Parser

- Regeln
  1. *Satz*  $\rightarrow$  *Subjekt* *Verb* *Objekt* *Satzendezeichen*
  2. *Subjekt*  $\rightarrow$  **Substantiv**
  3. *Subjekt*  $\rightarrow$  *Modifikator* **Substantiv**
  4. *Objekt*  $\rightarrow$  **Substantiv**
  5. *Objekt*  $\rightarrow$  *Modifikator* **Substantiv**
  6. *Modifikator*  $\rightarrow$  **Adjektiv**
  - ...
- wenn es möglich ist eine Herleitung zu finden, die
  - mit dem Nichtterminal *Satz* startet und
  - endet mit **Substantiv** **Verb** **Adjektiv** **Substantiv** *Satzendezeichen*dann gehört der Satz “Compiler sind technische Objekte.” zur Sprache
- der Prozess der automatisch nach so einer Herleitung sucht heisst “Parsen”

## Typüberprüfung (type checking)

- das Frontend ist außerdem dafür zuständig Typkonsistenz zu überprüfen
- z.B.

$a = a * 2 * b * c * d$

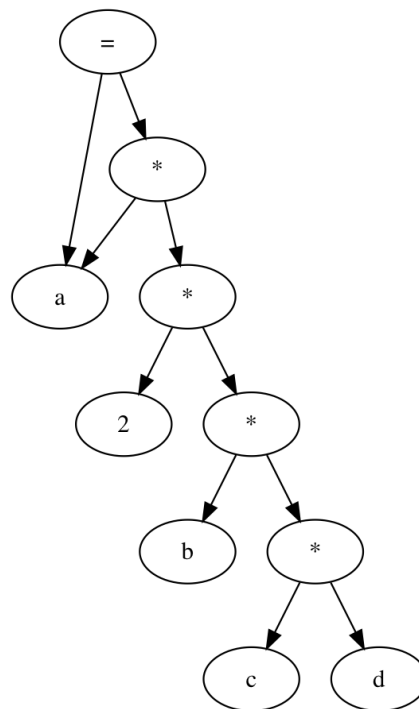
mag syntaktisch wohlgeformt sein

aber wenn b und d Zeichenketten sind, ist der Ausdruck ungültig

## Zwischenrepräsentation

- Ziel des Frontends ist eine Zwischenrepräsentation des Codes zu erzeugen
- verschiedene Formen
  - Graphen
  - Assemblerprogramme

## Zwischenrepräsentation: Beispiel Graph



DAG für  $a = a * 2 * b * c * d$

## Zwischenrepräsentation: Beispiel lowlevel, sequentieller Code

Lowlevel, sequentieller Code für  $a = a * 2 * b * c * d$ :



```
t0 = a * 2
t1 = t0 * b
t2 = t1 * c
t3 = t2 * d
a = t3
```

## Der Optimierer

- analysiert die Zwischenrepräsentation
- und schreibt den Code so um, dass er das selbe berechnet, aber in einer effizienteren Art und Weise
  - reduziert die Laufzeit der Anwendung oder
  - reduziert die Größe des generierten Codes oder
  - reduziert die Energie die der Prozessor verbraucht
  - oder ...

## Beispiel

- ursprünglicher Code

```
b = ...
c = ...
a = 1
for i = 1 to n
  read d
  a = a * 2 * b * c * d
end
```

- verbesserter Code

```
...
t = 2 * b * c
for i = 1 to n
  read d
  a = a * d * t
end
```

- besser aber als ProgrammiererIn nicht darauf verlassen, sondern selbst schon “guten” Code schreiben

## Das Backend

- durchläuft die Zwischenrepräsentation
- gibt Code für die Zielplattform aus

**Instruction Selection** wählt die Maschinenbefehle aus die der jeweiligen IR-Operation entsprechen

**Register Allokation** entscheidet welche Werte in Registern und welche im Hauptspeicher gehalten werden

**Instruction Scheduling** ermittelt die Reihenfolge in der die Befehle effizient ausgeführt werden können