Was ist ein Compiler ?

- Was ist ein Compiler – und worum geht es?
- Wie ist ein Compiler aufgebaut?
- Warum beschäftigen wir uns mit Compilerbau?
- Wie ist die Veranstaltung organisiert?
- Was interessiert Sie besonders?

Literatur:

[MCD] **Grune et al., *Modern Compiler Design***

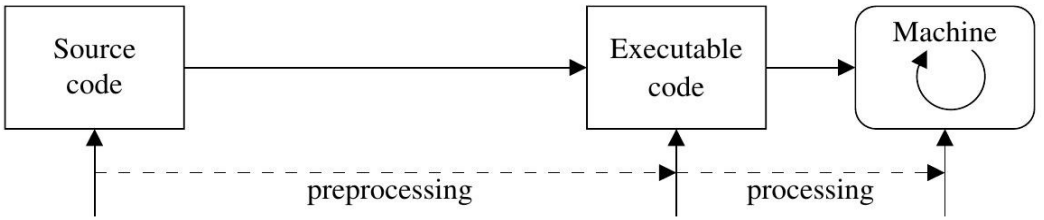
[MCIJ] Appel, *Modern Compiler Implementation in Java*

[EAC] Cooper et al., *Engineering a Compiler*

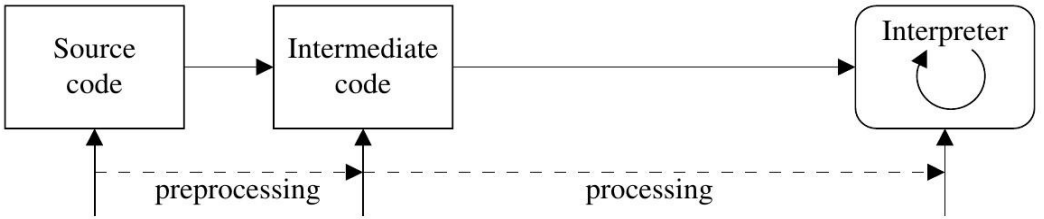
[CPT] Sethi et al., *Compilers: Principles Techniques, and Tools*

[SICP] Abelson et. al., *Structure and Interpretion of Computer Programs*

Was ist ein Compiler / Interpreter



Compilation



Interpretation

(Abbildung aus MCD)

- Umwandlung Quelltext nach Zielcode zwecks Ausführung.
- Der Quelltext kann mehr oder weniger nahe am Menschen sein.
- Das Ergebnis kann mehr oder weniger nahe an der Ausführung sein.
- Lässt sich der Übersetzungsprozess modularisieren?
- Gibt es erprobte Algorithmen?

Was ist ein Compiler ?

Eingabe:

```
63 6c 61 73 73 20 4d 61 69 6e 7b 0a 20 20 70 75
62 6c 69 63 20 73 74 61 74 69 63 20 76 6f 69 64
20 6d 61 69 6e 28 53 74 72 69 6e 67 5b 5d 20 61
29 20 7b 0a 20 20 20 20 53 79 73 74 65 6d 2e 6f
75 74 2e 70 72 69 6e 74 6c 6e 28 22 68 65 6c 6c
6f 22 29 3b 0a 20 20 7d 0a 7d 0a
```

Ausgabe:

```
ca fe ba be 00 00 00 34 00 1d 0a 00 06 00 0f 09
00 10 00 11 08 00 12 0a 00 13 00 14 07 00 15 07
00 16 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 29
56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e
75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 69
6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67
2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75
…
```

oder umgekehrt?

Ein Compiler erfordert Abstraktionsstufen:

- Bytes werden als Zeichen verstanden
- Zeichenfolgen werden als Symbole (Token) verstanden
- Symbolfolgen werden syntaktisch analysiert
- Aus der erkannten Struktur wird eine abstrakte Darstellung erzeugt
- Aus der abstrakten Darstellung wird eine Ausführung (Code) erzeugt

Jede Abstraktionsstufe erfordert eigene Algorithmen !
Compiler sind in großen Teilen deklarativ aufgebaut, da sich nur so erkennen lässt, ob sie korrekt sind.

Das Entwickeln eines Compilers macht auch auf Probleme von Programmiersprachen aufmerksam!

Die Übersetzung eines Programms

```
class Data(b: => Int) {
  lazy val x = b
}

def unclear: List[Data] = {
    val lst = ListBuffer[Data]()
    for (i <- 1 to 4) {
      lst += new Data(i)
    }
    lst.toList
}
```

Frage:

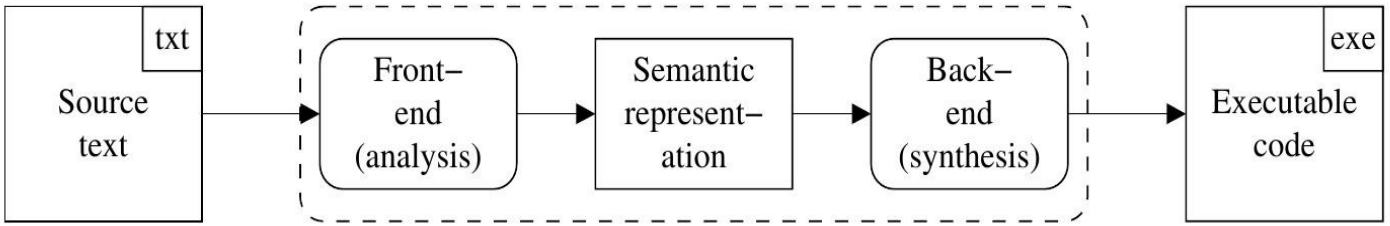
1) Gehört die Variable i in den Körper der for-Schleife?

2) Hängt das Ergebnis der Funktion davon ab?

3) Welche Konsequenzen hat das für den Compiler?

4) (Wie sieht das in Java-8 aus?)

Die grobe Struktur eines Compilers



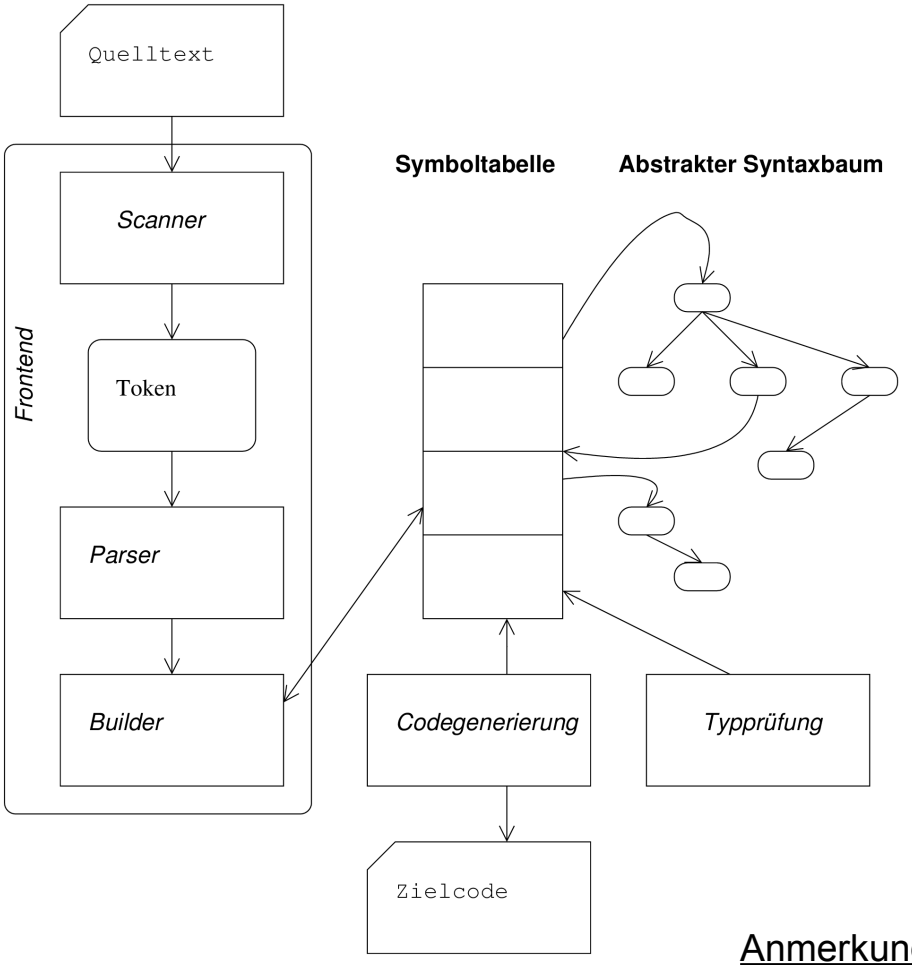
(Abbildung aus MCD)

- **Frontend:** Syntaktische Analyse des Eingabetextes
  z.T. auch semantische Analyse
- **Interne Darstellung** (Symboltabelle, Abstrakter Syntaxbaum):
  Analysen und Transformationen
- **Backend:** Generierung von Code für den Zielrechner / Interpretation

Anmerkung: Modular aufgebaute Compiler gibt es erst, seit Computer über genügend Speicher verfügen.

Ältere Programmiersprachen sind so entworfen, dass der Compiler einfach ist (z.B. C)

Der Compiler in größerem Detail



- **Frontend:** Analyse (Scanner, Parser) wird durch Theorie und Werkzeuge unterstützt. Der Builder erstellt die interne Darstellung.
- Die Bedeutung des Programms wird intern strukturiert dargestellt: Abstrakter Syntaxbaum (AST)
  Die Bezeichner werden in Symboltabellen erfasst. Quell- und Zielsprach-unabhängige Aktionen finden auf dem AST statt.
- **Backend:** Basierend auf dem AST wird die Ausgabe generiert.
  (manchmal wird hier zunächst ein interner Zwischencode erzeugt).
- Teile des Modells können auch in anderen Anwendungen auftreten (vgl. XML- oder oder JSON-Parser.

Anmerkung: Der Java-Bytecode kann als *externer* Zwischencode angesehen werden. Meines Wissens wurde dieses Konzept erstmals in den 60ern bei der Sprache BCPL verwendet.

Wozu Compilerbau ?

- Compilerbau ist ein erfolgreicher (und einer der ältesten Zweige) der Informatik
- Da es viele strukturierte Daten gibt, findet es über Compiler hinaus viele Anwendungen
- Compiler machen großen Gebrauch von Werkzeugen (Programmgeneratoren)
- Compiler basieren auf einer erprobten modularen Struktur
- Im Compiler lernt man viele nützliche Algorithmen in einer realistischen Umgebung kennen
- Die Kenntnis der Übersetzung vertieft das Verständnis von Programmiersprachen und der Ausführung von Computerprogrammen

Wie läuft die Veranstaltung ab ?

- Ziel: durch praktische Beschäftigung Zugang zur Theorie des Compilerbaus finden
- Teilweise Vorlesung, wo möglich Beispiele und praktische Übung
- Idealerweise: Gemeinsam eine einfache Programmiersprache definieren und bis hin zur Ausführung implementieren
- Projekte / Vorträge dienen der Bewertung. Sie können Teile des Gesamtziels unterstützen oder nicht behandelte Aspekte vertiefen
- Schwerpunkt (von meiner Seite her) weniger die perfekte Generierung von optimalem Code als die exemplarische Untersuchung moderner Techniken

Was interessiert Sie besonders ?

- ???
- ...

Ein erstes einfaches Beispiel

Beispiele für eine einfache interaktive Sprache für arithmetische Ausdrücke

```
let a = 3 * 4              Definition von Variablen

let b = a + 2

let a = a + 3

a + 4 * (2 + b)            Berechnung von Ausdrücken
```

let dient der Vereinfachung der Syntax (ein Beispiel für die Orientierung am Compiler)

Wie beschreibt man eine Sprache?

Wie schreibt man den Übersetzer?

Wie sieht die interne Darstellung aus?

Wie führen wir den Code aus?

Erste Annäherung

Programmiersprachen basieren auf Sätzen die aus Wörtern bestehen.

Wir haben 2 Sprachen

- die lexikalische Sprache der Wörter
- die Sprache der Anweisungen

Wörter

- Zahlen
- Schlüsselwörter
- Bezeichner:
- Sonderzeichen:

Wie beschreiben wir dies?

Wie beschreiben wir die Anweisungen?

Gehen wir dabei imperativ oder deklarativ vor?

Theoretische Informatik / Grammatik

Eine Sprache ist die Menge der erlaubten Sätze (= Zeichenfolge). Sprachen können durch Grammatiken beschrieben werden.

Eine Grammatik besteht aus:

- Terminalsymbolen: Zeichen der Sprache
- Nichterminalsymbolen: Namen zur Formulierung von Regeln
- Produktionsregeln: Symbolfolge $\rightarrow$ Symbolfolge
- Startsymbol: Nichtterminalsymbol

Eine Zeichenfolge ist ein Satz einer durch eine Grammatik G beschreibenen Sprache, wenn die Zeichen folge aus Terminalsymbolen besteht und wenn die Zeichenfolge aus dem Startsymbol mittels der Produktionsregeln abgeleitet werden kann.

Ableiten heißt: Ersetzen (innerhalb einer Satzform) einer Symbolfolge der linken Seite einer Regel durch die rechte Seite dieser Regel.

Hört sich kompliziert an, Was hilft das?

Vereinfachte Grammatiken (Chomsky-Hierarchie)

Noam Chomsky führte eine Abstufung der Komplexität von Grammatiken ein:

- Ebene 0: entspricht der Definition, alles ist erlaubt
- Ebene 1: kontextsensitive Grammatik.
  Form: α N $\beta \rightarrow \alpha \gamma \beta$
  α, β und γ sind Folgen von beliebigen Symbolen.
- Ebene 2: kontextfreie Grammatik.
  Form: N $\rightarrow \alpha$
  N ist ein Nichtterminalsymbol, α eine beliebige Symbolfolge
- Ebene3: reguläre Grammatik.
  Form: N $\rightarrow \alpha$ M
  α ist ein Terminalsymbol (oder auch eine Folge), N, M sind Nichterminalsymbole

Bedeutung:

- **Kontextfreie Grammatiken** lesen sich wie Definitionen. Sie können durch Stapelautomaten erkannt werden. **Anwendung: Syntax von Programmiersprachen**
- Reguläre Grammatiken sind äquivalent zu regulären Ausdrücken. Sie können durch endliche Automaten erkannt werden. **Anwendung Beschreibung von Wörtern von Programmiersprachen.**
- **Konsequenz: lesbare Beschreibung und eine effiziente Erkennung der Sprache.**

Beispiel: kontexfreie Grammatik

Terminalsymbole: **`let IDENTIFIER NUMBER + * ( ) =`**

Nichterminalsymbole: *Stmt Expr*

Startsymbol: *Stmt*

Produktionsregeln:

Stmt → **let IDENTIFIER =** *Expr*

Stmt → *Expr*

Expr → *Expr* **+** *Expr*

Expr → *Expr* ***** *Expr*

Expr → **(** *Expr* **)**

Expr → **NUMBER**

Expr → **IDENTIFIER**

Anmerkung:
Das ist eine einfache und auch korrekte Grammatik für die Beispielsprache.
Sie hat aber wesentliche Nachteile.
Welche?

Beispiel: lexikalische Grammatik für Zahl

Terminalsymbole: **0 ... 9**

Nichterminalsymbole: *N*

Startsymbol: *N*

Produktionsregeln:

N → **0** *N*

...

N → **9** *N*

N → **0**

...

N → **9**

Vereinfachter regulärer Ausdruck:

```
N = [0-9]+
```

Anmerkung: für kontextfreie Grammatiken gibt es eine Darstellung, bei der die rechte Seite einer Regel reguläre Ausdrücke enthält.
(erweiterte Backus-Naur-Form EBNF)

Beispiel: eine praktikable EBNF

Produktionsregeln:

Stmt → **let IDENTIFIER =** *Expr*

| *Expr*

Expr → *Term* { **+** *Term* }

Term → *Factor* { ***** *Factor* }

Factor → **(** *Expr* **)** | **NUMBER** | **IDENTIFIER**

Die Theorie ist nicht mehr gut zu erkennen. Aber: diese Form lässt sich einfach per Hand in ein Programm umsetzen (rekursiver Abstieg).

Anmerkung: Der rekursive Abstieg ist eine einfache und beliebte Technik für einfache Grammatiken. Die Menge der Grammatiken, die erkannt werden kann, ist jedoch stark eingeschränkt. Das ist auch der Grund, dass ich hier das **let** eingeführt habe.