

# Einführung in den Compilerbau

Niclas Kusenbach

LaTeX version:  SCHOUTER

## Table of Contents

### Contents

<b>1 Einführung</b>	<b>2</b>	3.1.2 Blockstrukturen . . . . .	8
1.0.1 Wirkung und Bedeutung . . . . .	2	3.2 Attribute . . . . .	8
1.0.2 Motivation . . . . .	2	3.3 Identifikation . . . . .	9
1.1 Aufbau eines Compilers . . . . .	2	3.4 Typprüfung . . . . .	9
1.1.1 Syntaxanalyse . . . . .	2	3.4.1 Typüberprüfung – Prinzip . . . . .	9
1.1.2 Kontextanalyse . . . . .	2	3.5 Implementierung der Kontextanalyse . . . . .	9
1.1.3 Codeerzeugung . . . . .	2	3.6 Standardumgebung . . . . .	10
1.1.4 Optimierung . . . . .	3	3.7 Typäquivalenz . . . . .	10
1.2 Syntax und Grammatik . . . . .	3	3.7.1 Ansätze . . . . .	10
1.2.1 Begrifflichkeiten . . . . .	3	3.7.2 Komplexe Typen . . . . .	10
1.3 (Mini-)Triangle . . . . .	3	3.8 Algorithmus der Kontextanalyse . . . . .	10
1.3.1 Syntaxbäume . . . . .	3	3.9 Zusammenfassung . . . . .	11
1.4 Kontextuelle Einschränkungen . . . . .	4		
1.5 Semantik . . . . .	4		
1.5.1 Operationelle Sicht . . . . .	4		
1.5.2 Beispiele . . . . .	4		
1.6 Zusammenfassung . . . . .	4		
<b>2 Syntaktische Analyse/Lexparse</b>	<b>5</b>		
2.1 Compilerstruktur . . . . .	5		
2.2 Syntaxanalyse – Überblick . . . . .	5		
2.3 BNF und EBNF . . . . .	5		
2.4 Grammatiktransformationen . . . . .	5		
2.5 Parsing-Grundlagen . . . . .	6		
2.6 Top-Down Parsing . . . . .	6		
2.7 Rekursiver Abstieg . . . . .	6		
2.8 Starter- und Folgemengen . . . . .	6		
2.9 AST (Abstract Syntax Tree) . . . . .	6		
2.10 Scanner (Lexikalische Analyse) . . . . .	7		
2.11 Automatisierung . . . . .	7		
2.12 Typische Fehler . . . . .	7		
2.13 Zusammenfassung – Wichtigste Punkte .	7		
<b>3 Kontextanalyse</b>	<b>8</b>		
3.1 Geltungsbereiche und Symboltabellen .	8		
3.1.1 Grundlagen . . . . .	8		

# 1 Einführung

## Was ist ein Compiler?

Ein **Compiler** ist die **Schnittstelle zwischen Programmiersprache und Maschine**. Er übersetzt menschenlesbaren Quellcode in maschinennahe Instruktionen.

- **Programmiersprachen:** gut handhabbar für Menschen (z.B. Java, C++)
- **Maschine:** optimiert auf Geschwindigkeit, Energieeffizienz, Fläche

### 1.0.1 Wirkung und Bedeutung

- Compiler beeinflussen direkt die **effektive Rechenleistung**.
- Beispiel: Unterschiedliche Compiler erzeugen unterschiedlich effizienten Code.
- Spezialisierte Prozessoren erfordern angepasste Compiler (DSPs, GPUs, FPGAs).

## Paralleles Rechnen

Trend von Ein-Prozessor-Systemen hin zu **Mehrkern- und heterogenen Systemen**.

- OpenMP – Mehrkern-CPUs
- CUDA – GPUs
- OpenCL – Kombination aus CPUs und GPUs

### 1.0.2 Motivation

- Compilerbau kombiniert Theorie, Architektur und Softwaretechnik.
- Zentrale Themen: Parsing, Codegenerierung, Optimierung.

## 1.1 Aufbau eines Compilers

### Compilerphasen

1. **Front-End:** Lexikalische, syntaktische und kontextuelle Analyse
2. **Middle-End:** Optimierung der Zwischendarstellung (IR)
3. **Back-End:** Codeerzeugung für Zielarchitektur

### 1.1.1 Syntaxanalyse

- Überprüfung der Syntaxregeln ⇒ **Abstrakter Syntaxbaum (AST)**

### 1.1.2 Kontextanalyse

- Variablenbindung, Typprüfung, Scope-Überprüfung
- Ergebnis: **Dekorierter AST (DAST)**

### 1.1.3 Codeerzeugung

- Zuweisung von Speicher, Übersetzung von AST zu Maschinencode

#### 1.1.4 Optimierung

- Ziel: effizienterer Code bei gleicher Semantik
- Beispiele:
  - **Constant Folding:**  $x = (2 + 3) * y \Rightarrow x = 5 * y$
  - **Common Subexpression Elimination**
  - **Strength Reduction**
  - **Loop-Invariant Code Motion**

### 1.2 Syntax und Grammatik

#### Syntax

Beschreibt die **Struktur korrekter Programme**.

#### Formalisierungsmethoden

- **Reguläre Ausdrücke (RE)** – beschreiben Tokens, aber nicht Programmsyntax.
- **Kontextfreie Grammatiken (CFG)** – Basis für Programmiersprachen.
- **BNF / EBNF** – Notation zur Beschreibung von CFGs.

#### 1.2.1 Begrifflichkeiten

- **Terminale:** konkrete Symbole
- **Nichtterminale:** syntaktische Kategorien
- **Produktionen:** Regeln der Grammatik
- **Startsymbol:** Ausgangspunkt der Herleitung

#### Mehrdeutigkeit

Eine Grammatik ist **mehrdeutig**, wenn ein Satz mehrere Ableitungsbäume hat. Für Compiler sind nur eindeutige CFGs sinnvoll.

### 1.3 (Mini-)Triangle

#### Mini-Triangle

- Pascal-artige Beispiel-Sprache
- Enthält Variablen, Konstanten, Schleifen, Bedingungen
- Keine Unterprogramme
- Beispielhafte CFG-Definitionen für:
  - **Command, Expression, Declaration, Type-denoter**

#### 1.3.1 Syntaxbäume

- **Konkrete Syntax:** enthält alle syntaktischen Details
- **Abstrakte Syntax:** reduziert auf semantisch relevante Struktur (AST)

## AST als IR

- **Vorteile:** maschinenunabhängig, gut für Analysen
- **Nachteile:** weniger geeignet für hardwarenahe Optimierungen

## 1.4 Kontextuelle Einschränkungen

### Geltungsbereiche (Scopes)

- Jede Variable muss **vor ihrer Verwendung** deklariert sein.
- Deklaration = *bindendes Auftreten*, Verwendung = *verwendendes Auftreten*.

### Typprüfung

- Jede Operation verlangt passende Operandentypen.
- Beispielregeln:

$E_1 > E_2$  : liefert bool, wenn  $E_1, E_2 : \text{int}$

$V := E$  : nur erlaubt, wenn Typen äquivalent

$\text{while } E \text{ do } C$  : nur erlaubt, wenn  $E : \text{bool}$

## 1.5 Semantik

### Semantik

Beschreibt die **Bedeutung von Programmen zur Laufzeit**.

### 1.5.1 Operationelle Sicht

- **Anweisungen:** verändern Zustand (z.B. Variablen, I/O)
- **Ausdrücke:** werden evaluiert und liefern Werte
- **Deklarationen:** binden Namen an Speicherbereiche

### 1.5.2 Beispiele

- **AssignCmd:**  $V := E$ 
  1. Evaluiere  $E \Rightarrow v$
  2. Weise  $v$  an Variable  $V$  zu
- **BinaryExp:**  $E_1 \text{ op } E_2$ 
  1. Evaluiere  $E_1, E_2 \Rightarrow v_1, v_2$
  2. Führe Operation  $\text{op}(v_1, v_2)$  aus

## 1.6 Zusammenfassung

- Compiler übersetzen Hochsprache → Maschinencode.
- Bestehen aus: Front-End, Middle-End, Back-End.
- Zentrale Themen: Syntax, Semantik, Typen, Optimierung.
- Mini-Triangle dient als Lehrsprache zur Umsetzung der Konzepte.

## 2 Syntaktische Analyse/Lexparse

### Übersetzung und Phasen

- **Syntaxanalyse** → Struktur des Programms (AST)
- **Kontextanalyse** → Bedeutungsprüfung (Typen, Gültigkeit)
- **Codegenerierung** → Übersetzung in Zielcode

### 2.1 Compilerstruktur

Ein-Pass-Compiler:

- Führt alle Phasen gleichzeitig aus
- Keine echte Zwischendarstellung (IR)
- Typisch für kleine Sprachen (z. B. Pascal)

Multi-Pass-Compiler:

- Arbeitet mit mehreren Durchgängen über Quelltext/IR
- Datenweitergabe über IR (AST)
- Bessere Modularität und Optimierung

### 2.2 Syntaxanalyse – Überblick

- **Scanner (Lexer)**: Wandelt Zeichenfolge → Tokenfolge
- **Parser**: Wandelt Tokenfolge → Abstract Syntax Tree (AST)
- Token = atomares Symbol des Quellprogramms

### Kontextfreie Grammatik (CFG)

Eine CFG ist ein 4-Tupel  $(N, T, P, S)$  mit:

- $N$ : Nichtterminale
- $T$ : Terminale
- $P$ : Produktionen
- $S$ : Startsymbol

### 2.3 BNF und EBNF

- **BNF**: Grundform zur Definition von Grammatiken
- **EBNF**: Erweiterung mit regulären Ausdrücken, optionalen und wiederholten Konstrukten
- Beispiel:  
Expression ::= primary-Expression (operator primary-Expression)\*

### 2.4 Grammatiktransformationen

- **Gruppierung**: Zusammenfassen gleicher LHS
- **Linksausklammern**: Gemeinsame Präfixe auslagern
- **Linksrekursion beseitigen**:  $N ::= X|NY \Rightarrow N ::= X(Y)^*$

- **Ersetzung von Nicht-Terminalen:** falls nur eine Regel existiert

## 2.5 Parsing-Grundlagen

---

### Parsing

Entscheidung, ob Eingabe zur Grammatik gehört und Aufbau des Syntaxbaumes.

- **Top-Down (z. B. rekursiver Abstieg):** Von Startsymbol zu Terminalen
- **Bottom-Up (z. B. Shift/Reduce):** Von Terminalen zur Wurzel

## 2.6 Top-Down Parsing

---

- Aufbau des Syntaxbaums von oben nach unten
- Expandiere jeweils das linke Nichtterminal
- LL(k): Grammatik, bei der mit  $k$  Lookahead-Tokens eindeutig entschieden werden kann
- **LL(1)** → wichtigster Fall für rekursiven Abstieg

## 2.7 Rekursiver Abstieg

---

### Rekursiver Abstieg

Jedes Nichtterminal erhält eine Prozedur `parseN()`, deren Aufrufstruktur dem Parsebaum entspricht.

- `accept(t)` prüft aktuelles Token
- `acceptIt()` akzeptiert aktuelles Token ohne Prüfung
- `currentToken`: vom Scanner geliefert

## 2.8 Starter- und Folgemengen

---

### `starters[[X`

]] Menge aller Tokens, die am Anfang einer aus  $X$  ableitbaren Zeichenkette stehen können.

### `follow[[X`

]] Menge aller Tokens, die in der Grammatik direkt nach  $X$  folgen können.

### LL(1)-Bedingungen:

- Für jede Alternative  $X|Y$ :  

$$\text{starters}[[X]] \cap \text{starters}[[Y]] = \emptyset$$
- Wenn  $\varepsilon$ -Produktionen vorkommen:  

$$\text{starters}[[X]] \cap (\text{starters}[[Y]] \cup \text{follow}[[X|Y]]) = \emptyset$$

## 2.9 AST (Abstract Syntax Tree)

---

- Strukturierte Repräsentation des Programms
- Parser erzeugt AST-Knoten beim rekursiven Abstieg

- Jede Grammatikregel entspricht einer AST-Unterklasse

### AST-Aufbau

- **Abstrakte Basisklasse:** AST
- **Subklassen:** Command, Expression, Declaration, TypeDenoter
- Terminalknoten (z. B. Identifier, Operator) speichern tatsächlichen Text

## 2.10 Scanner (Lexikalische Analyse)

### Scanner

Wandelt Zeichen → Tokens anhand regulärer Ausdrücke (REs).

#### Aufgaben:

- Entfernt Whitespace, Kommentare
- Liefert Token(*kind*, *spelling*, *position*)
- Nutzt endlichen Automaten oder rekursiven Abstieg

#### Beispiel EBNF Mini-Triangle:

```

Identifier ::= Letter (Letter — Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= +| - | * | / | < | > | =

```

## 2.11 Automatisierung

- Scanner-Generatoren: **JLex**, **JFlex**
- Parser-Generatoren: **ANTLR (LL\*)**, **JavaCC (LL(k))**

## 2.12 Typische Fehler

- Linksrekursion nicht entfernt
- Linksausklammern vergessen
- Anfangs-/Folgemengen überschneiden sich → nicht LL(1)
- Schlüsselwörter nicht vom Identifier getrennt

## 2.13 Zusammenfassung – Wichtigste Punkte

- CFG-Grundlagen (BNF/EBNF)
- Transformationen: Gruppierung, Linksausklammern, Rekursionsbeseitigung
- LL(1)-Parsing und Bedingungen
- Rekursiver Abstieg: Struktur und Methoden
- AST-Struktur: Klassenhierarchie, Terminal- und Nichtterminalknoten
- Scanner: REs, endliche Automaten, Schlüsselworterkennung

## 3 Kontextanalyse

### Ziel der Kontextanalyse

Überprüfung der **kontextuellen Einschränkungen**:

- Geltungsbereiche von Bezeichnern
- Typregeln (Typüberprüfung)
- Konsistenzregeln (z. B. Funktionsaufrufe, Parameteranzahl, etc.)

### 3.1 Geltungsbereiche und Symboltabellen

#### 3.1.1 Grundlagen

- Jede Deklaration **bindet** einen Namen.
- Jede Verwendung eines Bezeichners muss einer gültigen **Bindung** zugeordnet sein.
- Fehlende Bindung ⇒ **Fehler**.

#### Symboltabelle

Datenstruktur zur **Zuordnung von Namen zu Attributen**.

- Schnelles Nachschlagen (z. B. Hash-Tabelle)
- Enthält Art, Typ, Sichtbarkeit, evtl. Adresse
- Hierarchische Struktur für verschachtelte Blöcke

#### 3.1.2 Blockstrukturen

- **Monolithisch:** Alle Deklarationen global (BASIC, COBOL)
- **Flach:** Global + lokal (FORTRAN)
- **Verschachtelt:** Beliebige Tiefe (Pascal, Ada, Java)

#### Geltungsbereichsregeln

- Kein Bezeichner mehrfach innerhalb eines Blocks.
- Verwendung nur mit Deklaration im lokalen oder umschließenden Block.
- Lokale Deklarationen überdecken äußere.

### 3.2 Attribute

#### Attribute

Informationen, die zu einem Bezeichner gespeichert werden:

- Art (Konstante, Variable, Funktion)
- Typ (int, char, boolean, array, record, ...)
- Sichtbarkeit, ggf. Speicheradresse

#### Verwendung:

- Überprüfung von Geltungsbereichs- und Typregeln
- Vorbereitung auf Code-Generierung

## Speicherung:

- Einfach: explizite Speicherung in Klassen
- Besser: Referenz auf **AST-Knoten** der Deklaration

## 3.3 Identifikation

### Identifikation

Erste Phase der Kontextanalyse. Ordnet jede **Verwendung** einer **Definition** zu.

**Ziel:** Aufbau einer Symboltabelle durch AST-Durchlauf.

**Kombination mit:** Typprüfung (zweite Phase).

## 3.4 Typprüfung

### Typ

Einschränkung der möglichen Interpretationen eines Speicherbereiches oder Ausdrucks.

- **Statische Typisierung:** zur Compile-Zeit (z. B. Pascal, C)
- **Dynamische Typisierung:** zur Laufzeit (z. B. Python)

### 3.4.1 Typüberprüfung – Prinzip

1. Bestimme Typen von Teilausdrücken (*Bottom-Up* im AST)
2. Prüfe, ob Typanforderungen des Kontextes erfüllt sind

### Typregeln:

- **if E then ... → E** muss vom Typ Boolean sein.
- Zuweisung  $x := E \rightarrow$  Typ von  $x =$  Typ von  $E$ .
- Binäre Operatoren:  $E_1 \text{ op } E_2$  typkorrekt, wenn  $E_1 : T_1, E_2 : T_2$  und  $\text{op} : T_1 \times T_2 \rightarrow R$ .

## 3.5 Implementierung der Kontextanalyse

### AST-Dekoration

**Dekorierter AST** speichert zusätzliche Informationen:

- Typ jedes Ausdrucks
- Verweis auf bindende Deklaration

### Vorgehensweisen:

- OO-Ansatz: Jede AST-Klasse implementiert `check()`.
- Funktionaler Ansatz: Separate Typprüfungsfunction.
- **Visitor-Pattern:** Modularer Ansatz zur Trennung von Struktur und Verhalten.

### Visitor-Pattern

Ermöglicht neue Operationen auf AST-Knoten, ohne deren Klassen zu verändern.

### Beispiele:

- `visitAssignCommand`: prüft Typkompatibilität von LHS und RHS.
- `visitLetCommand`: öffnet/schließt Scope in Symboltabelle.
- `visitIfCommand`: prüft Boolean-Bedingung.

## 3.6 Standardumgebung

### Standardumgebung

Vordefinierte Typen, Konstanten und Operationen, die zur Analyse vorhanden sein müssen.

**Beispiele:**

- Typen: `Integer`, `Boolean`, `Char`
- Konstanten: `true`, `false`
- Operatoren: `+`, `-`, `*`, `<`, `=`

Implementierung:

- Als AST-Einträge vor der Analyse eingefügt.
- Liegen auf äußerster Scope-Ebene (Ebene 0/1).

## 3.7 Typäquivalenz

### Typäquivalenz

Regel, wann zwei Typen als "gleich" gelten.

#### 3.7.1 Ansätze

- **Strukturelle Äquivalenz**: gleiche Struktur  $\Rightarrow$  äquivalent.
- **Namenäquivalenz**: nur identische Typdefinitionen sind äquivalent.

In Triangle: strukturelle Typäquivalenz.

**Beispiele:**

- `record n: Integer; c: Char end`  $\neq$  `record c: Char; n: Integer end`
- `array 8 of Char` = `array 8 of Char`

#### 3.7.2 Komplexe Typen

- Verweis auf Typbeschreibung im AST (`TypeDenoter`)
- Struktureller Vergleich von Sub-ASTs zur Typprüfung

## 3.8 Algorithmus der Kontextanalyse

### Algorithmus

Ein kombinierter AST-Durchlauf:

1. Tiefensuche (DFS) über AST
2. Identifikation (Symboltabelle aufbauen)
3. Typprüfung (Dekoration des ASTs)

**Voraussetzung:** Bindungen erscheinen im Code **vor** Verwendungen.

**Ergebnis:** Dekorierter AST mit allen Typ- und Scopeinformationen.

### 3.9 Zusammenfassung

---

- Kontextanalyse = Identifikation + Typprüfung
- Symboltabelle verwaltet Geltungsbereiche
- Typprüfung sichert korrekte Operationen
- Visitor-Pattern modularisiert Implementierung
- Standardumgebung liefert primitive Typen
- Triangle nutzt strukturelle Typäquivalenz

#### Prüfungsrelevante Kernbegriffe

Symboltabelle, Geltungsbereich, Attribut, Typüberprüfung, AST-Dekoration, Visitor, Standardumgebung, Typäquivalenz