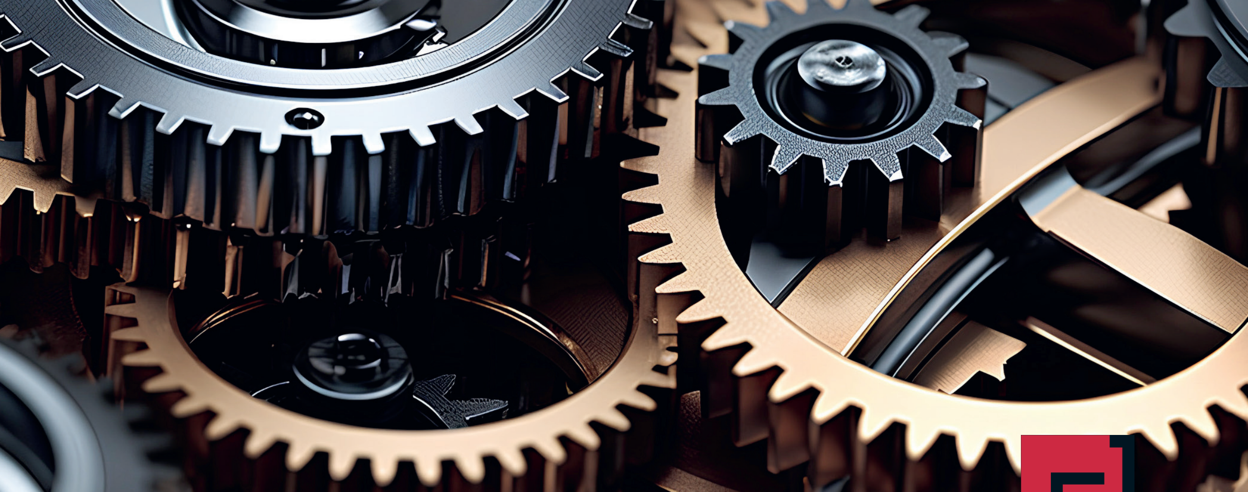

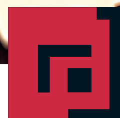

Hanspeter Mössenböck

# Compilerbau

## Grundlagen und Anwendungen

dpunkt.verlag

**Hanspeter Mössenböck** ist Professor für Informatik an der Johannes Kepler Universität Linz und beschäftigt sich seit vielen Jahren mit Programmiersprachen und Compilern. Er war Mitarbeiter von Professor Niklaus Wirth an der ETH Zürich, einem der Pioniere des Compilerbaus, der unter anderem die Programmiersprache Pascal entwickelt hat. Seit über 20 Jahren kooperiert er mit Oracle Labs auf dem Gebiet der dynamischen Compileroptimierung für Java und andere Programmiersprachen. Viele der an seinem Institut entwickelten Techniken werden heute weltweit in Java-Systemen eingesetzt. Hanspeter Mössenböck ist Autor von Büchern über Java, C#, .NET sowie über compilererzeugende Systeme.

Coypright und Urheberrechte:
Die durch die dpunkt.verlag GmbH vertriebenen digitalen Inhalte sind urheberrechtlich geschützt. Der Nutzer verpflichtet sich, die Urheberrechte anzuerkennen und einzuhalten. Es werden keine Urheber-, Nutzungs- und sonstigen Schutzrechte an den Inhalten auf den Nutzer übertragen. Der Nutzer ist nur berechtigt, den abgerufenen Inhalt zu eigenen Zwecken zu nutzen. Er ist nicht berechtigt, den Inhalt im Internet, in Intranets, in Extranets oder sonst wie Dritten zur Verwertung zur Verfügung zu stellen. Eine öffentliche Wiedergabe oder sonstige Weiterveröffentlichung und eine gewerbliche Vervielfältigung der Inhalte wird ausdrücklich ausgeschlossen. Der Nutzer darf Urheberrechtsvermerke, Markenzeichen und andere Rechtsvorbehalte im abgerufenen Inhalt nicht entfernen.

Hanspeter Mössenböck

# Compilerbau

## Grundlagen und Anwendungen

dpunkt.verlag

Hanspeter Mössenböck
*hanspeter.moessenboeck@jku.at*

Lektorat: Christa Preisendanz
Lektoratsbüro: Julia Griebel, Friederike Demmig
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Hanspeter Mössenböck
Herstellung: Stefanie Weidner, Frank Heidt
Umschlaggestaltung: Eva Hepper, Silke Braun

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-98889-008-5
PDF 978-3-98890-145-3
ePub 978-3-98890-146-0

1. Auflage 2024
Copyright © 2024 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

*Schreiben Sie uns:*
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *hallo@dpunkt.de*.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

# Geleitwort

Programmiersprachen spricht man nicht – es sind formale Systeme. Programme, die die Texte, die in Programmiersprachen formuliert sind, in Folgen von Computerinstruktionen übersetzen, nennt man Compiler. Es handelt sich dabei um komplexe Programme. Am Anfang der Compilertechnologie standen die Sprachen Fortran (1957) und Algol (1960). Die Fertigung ihrer Compiler beschäftigte große Teams von Programmierern über Jahre. Durch die Systematisierung des Compilerbaus, wie sie in diesem Buch vorgestellt wird, konnte diese Arbeit für Einzelpersonen jedoch auf wenige Monate reduziert werden. Dies war ein gewaltiger Fortschritt.

Grundlegend neue Programmiersprachen gibt es heute eher selten und völlig neue Computerarchitekturen ebenfalls. Compilerbau scheint daher eine spezielle Sparte zu sein, die wenigen Spezialisten in großen Firmen vorbehalten bleibt. Wozu also dieses Buch?

Der Grund ist einfach. Jedes Programm, jede Anwendung, ist nach Regeln aufgebaut, die spezifizieren, wie Anweisungen und Datendeklarationen auszusehen haben. Wenn diese Regeln formalisiert werden, erhöht das die Klarheit und Verständlichkeit der Anwendung. Der Schlüssel dazu beruht auf Formalisierung, also auf der Spezifikation einer Syntax für die Eingabe. Dadurch wird eine Syntaxanalyse ermöglicht, die die Grundlage des Compilerbaus darstellt. Die Syntaxanalyse und weitere in diesem Buch beschriebene Techniken sind aber nicht nur zur Verarbeitung von Programmiersprachen nützlich, sondern lassen sich auch auf viele andere Probleme anwenden, bei denen es um die systematische Verarbeitung strukturierter Eingaben geht. Diese Techniken tragen maßgeblich zur Korrektheit und zum Verständnis solcher Anwendungen bei.

Möge dieses Buch bei dieser vielversprechenden Entwicklung behilflich sein!

*Prof. em. Dr. Niklaus Wirth*
Zürich, im Dezember 2023[1]

1. Prof. Dr. Niklaus Wirth verstarb völlig unerwartet eine Woche nach Abfassung dieses Geleitworts im 90. Lebensjahr.

# Vorwort

Compiler bauen? Das machen doch nur große Firmen wie Microsoft, Google oder Oracle. Das stimmt, aber fast alle Informatikerinnen und Informatiker scheinen irgendwann einmal den Wunsch zu verspüren, eine eigene Programmiersprache zu entwerfen, und sei es nur eine domänenspezifische Sprache oder eine Kommandosprache für spezifische Zwecke. Natürlich möchte man dann auch einen Compiler dafür schreiben. Dieser Wunsch scheitert oft daran, dass das nötige Compilerbau-Wissen fehlt oder die in einschlägigen Büchern beschriebenen Techniken zu kompliziert sind und sich vor allem mit fortgeschrittenen Themen wie Optimierung, Registerallokation oder den Details der Codeerzeugung beschäftigen.

Dabei sind die Grundlagen des Compilerbaus einfach – jeder kann sie erlernen und zu seinem Methodenrepertoire hinzufügen. Während Compiler für Programmiersprachen wie Java oder C/C++ tatsächlich nur von großen Firmen entwickelt werden, gibt es viele Aufgaben (auch außerhalb des eigentlichen Compilerbaus), die sich mithilfe elementarer Techniken einfach und elegant lösen lassen. Im Prinzip kann man diese Techniken immer dann anwenden, wenn eine strukturierte Eingabe vorliegt, die durch eine Grammatik beschrieben werden kann. Beispiele dafür sind einfache Kommandosprachen, aber auch die Verarbeitung von Konfigurationsdateien, Logdateien, Stücklisten oder Messdatenreihen.

Dieses Buch zeigt, wie es geht. Es behandelt die praxisrelevanten Grundlagen des Compilerbaus, von der lexikalischen Analyse über die Syntaxanalyse bis zur Semantikverarbeitung und zur Codeerzeugung. Weitere Themen sind die Beschreibung von Übersetzungsprozessen durch attributierte Grammatiken sowie der Einsatz eines Compilergenerators zur automatischen Erzeugung der Kernteile eines Compilers. Gerade diese letzten beiden Themen sind in der Praxis höchst relevant, obwohl man sie in vielen Compiler-Büchern nicht findet.

Zur Syntaxanalyse wird in diesem Buch der rekursive Abstieg verwendet, ein einfaches Top-down-Verfahren, das auch per Hand (d.h. ohne Werkzeuge) implementiert werden kann. Zur Abrundung wird allerdings am Ende des Buches auch die Bottom-up-Syntaxanalyse vorgestellt, die zwar mächtiger, aber auch aufwendiger ist als der rekursive Abstieg.

Techniken versteht man erst so richtig, wenn man sie auf ein konkretes Beispiel anwendet. Daher wird im Buch als durchgängiges Fallbeispiel ein Compiler

für *MicroJava* – eine einfache Java-ähnliche Programmiersprache – entwickelt, der ausführbaren Bytecode – ähnlich dem Java-Bytecode – erzeugt. Der vollständige Quellcode dieses Compilers kann von [Download] heruntergeladen und studiert werden. Als Implementierungssprache für den Compiler sowie für alle Beispiele in diesem Buch wird Java verwendet.

Als Zielmaschine des Compilers wird eine vereinfachte *Java Virtual Machine* (JVM) verwendet, nämlich die *MicroJava Virtual Machine* (μJVM), die einfach genug ist, um nicht in Details zu ersticken, aber auch realistisch genug, um damit die Techniken der Codeerzeugung zu erlernen. Die μJVM besitzt als Instruktionssatz einen Bytecode, der sich an den Bytecode der JVM anlehnt. Ein Interpreter für diesen Bytecode wird ebenfalls zur Verfügung gestellt. Durch das Studium der Codeerzeugung lernt man auch viel über die Funktionsweise eines Rechners, was ein weiterer Grund ist, sich mit Compilerbau zu beschäftigen.

Am Ende jedes Kapitels gibt es Übungsaufgaben zu den behandelten Themen. Musterlösungen dazu sind unter [Download] zu finden. Man sollte versuchen, die Übungen zu bearbeiten, weil man dadurch die behandelten Techniken besser versteht. Aber selbst wenn man nicht die Zeit findet, die mehr als 70 Übungsaufgaben selbst zu lösen, sollte man sich zumindest die Musterlösungen ansehen, weil sie zusätzliche Beispiele zu den einzelnen Kapiteln darstellen.

Das Buch entstand aus einer Compilerbau-Vorlesung, die ich seit vielen Jahren an der Johannes Kepler Universität Linz sowie an der Oxford Brookes University in England halte. Es kann als begleitende Unterlage zu einer einführenden Compilerbau-Vorlesung verwendet werden, an die sich dann eine fortgeschrittene Vorlesung mit Themen wie Optimierung oder Registerallokation anschließen kann. Die Powerpoint-Folien der diesem Buch zugrunde liegenden Vorlesung werden unter [Download] zur Verfügung gestellt. Das Buch kann aber auch zum Selbststudium verwendet werden, da es alle Techniken beschreibt, die für den Bau compilerähnlicher Werkzeuge in der Praxis benötigt werden.

Ich möchte an dieser Stelle meinem ehemaligen Lehrer und Kollegen Prof. Niklaus Wirth (ETH Zürich) für das Geleitwort zu diesem Buch danken. Er ist ein Meister des Compilerbaus, von dem ich viele Techniken übernommen habe. Ebenfalls bedanken möchte ich mich beim dpunkt.verlag für die wie immer hervorragende Begleitung dieses Buchprojekts und den ausgezeichneten Lektoratsservice.

*Hanspeter Mössenböck*
Linz, im Dezember 2023

[Download] *https://ssw.jku.at/CompilerBuch/*

- Vorlesungsfolien
- Quelltext des MicroJava-Compilers
- Musterlösungen zu den Übungsaufgaben
- Weitere Materialien

# Inhaltsverzeichnis


**1 Überblick 1**

1.1 Geschichte des Compilerbaus . . . . . 2
1.2 Dynamische Struktur eines Compilers . . . . . 5
1.3 Statische Struktur eines Compilers . . . . . 10
1.4 Grammatiken . . . . . 10
1.5 Syntaxbäume . . . . . 19
1.6 MicroJava . . . . . 22
1.7 Übungsaufgaben . . . . . 23

**2 Lexikalische Analyse 27**

2.1 Reguläre Grammatiken und endliche Automaten . . . . . 28
2.2 Der Scanner als endlicher Automat . . . . . 32
2.3 Implementierung des Scanners . . . . . 34
2.4 Übungsaufgaben . . . . . 39

**3 Syntaxanalyse 41**

3.1 Kontextfreie Grammatiken und Kellerautomaten . . . . . 41
3.2 Rekursiver Abstieg . . . . . 46
3.3 LL(1)-Eigenschaft . . . . . 57
3.4 Syntaxfehlerbehandlung . . . . . 63
3.4.1 Fehlerbehandlung im Panic Mode . . . . . 64
3.4.2 Fehlerbehandlung mit allgemeinen Fangsymbolen . . . . . 64
3.4.3 Fehlerbehandlung mit speziellen Fangsymbolen . . . . . 72
3.5 Übungsaufgaben . . . . . 76

**4 Attributierte Grammatiken 79**

4.1 Bestandteile . . . . . 80
4.2 Anwendungsbeispiele . . . . . 82
4.3 Übungsaufgaben . . . . . 89

5 **Symbolliste** 93

5.1 Objektknoten . . . 94
5.2 Scopeknoten . . . 99
5.3 Strukturknoten . . . 104
5.4 Typprüfungen . . . 106
5.5 Lösen von LL(1)-Konflikten mittels der Symbolliste . . . 109
5.6 Initialisierung der Symbolliste . . . 111
5.7 Übungsaufgaben . . . 112

6 **Codeerzeugung** 115

6.1 Die MicroJava-VM . . . 117
6.1.1 Speicherbereiche . . . 118
6.1.2 Instruktionssatz . . . 121
6.2 Codespeicher . . . 132
6.3 Operanden der Codeerzeugung . . . 133
6.4 Laden von Werten . . . 136
6.5 Ausdrücke . . . 141
6.6 Zuweisungen . . . 145
6.7 Sprünge und Marken . . . 147
6.8 Ablaufkontrollstrukturen . . . 153
6.8.1 while-Anweisung . . . 153
6.8.2 if-Anweisung . . . 154
6.8.3 break-Anweisung . . . 155
6.8.4 Kurzschlussauswertung boolescher Ausdrücke . . . 156
6.9 Methoden . . . 159
6.10 Objektdatei . . . 165
6.11 Übungsaufgaben . . . 166

7 **Der Compilergenerator Coco/R** 169

7.1 Scannerbeschreibung . . . 173
7.2 Parserbeschreibung . . . 177
7.3 Fehlerbehandlung . . . 182
7.4 LL(1)-Konflikte . . . 185
7.5 Beispiele . . . 188
7.5.1 Lesen eines Binärbaums . . . 188
7.5.2 Fragebogengenerator . . . 191
7.5.3 Abstrakte Syntaxbäume . . . 195
7.6 Übungsaufgaben . . . 205

**8 Exkurs: Bottom-up-Syntaxanalyse 209**

8.1 Arbeitsweise eines Bottom-up-Parsers . . . . . 209
8.2 LR-Grammatiken . . . . . 214
8.3 LR-Tabellenerzeugung . . . . . 217
8.4 LR-Tabellenverkleinerung . . . . . 225
8.5 Semantikanschluss . . . . . 228
8.6 LR-Fehlerbehandlung . . . . . 232
8.7 Übungsaufgaben . . . . . 239

**A Die Sprache MicroJava 241**

A.1 Lexikalische Struktur . . . . . 241
A.2 Syntax . . . . . 241
A.3 Semantik . . . . . 242
A.4 Kontextbedingungen . . . . . 243
A.5 Implementierungsbeschränkungen . . . . . 246

**B Der MicroJava-Compiler 247**

B.1 Überblick . . . . . 247
B.2 Schnittstellen der Compilerklassen . . . . . 248

**Literatur 253**

**Index 255**

# 1 Überblick

Ein Compiler ist ein Werkzeug, das ein *Quellprogramm* (z.B. in Java, Pascal oder C) in ein *Zielprogramm* übersetzt. Die Sprache des Zielprogramms ist meist Maschinencode (z.B. Instruktionen eines Prozessors oder Java-Bytecode). Das Übersetzungsziel kann aber auch eine andere Quellsprache sein. In diesem Fall spricht man von einem *Cross-Compiler.*

Neben Compilern im engeren Sinne gibt es auch compilerähnliche Werkzeuge, die eine beliebige syntaktisch strukturierte Eingabe in etwas übersetzen, das keine Sprache im engeren Sinne ist. So ein Werkzeug könnte zum Beispiel eine Logdatei in eine Tabelle übersetzen, die Informationen aus der Logdatei in aggregierter Form darstellt.

Man mag sich vielleicht fragen, wozu Compilerbau-Kenntnisse eigentlich nützlich sind. Schließlich entwickeln nur große Firmen wie Microsoft, Google oder Oracle Compiler. Das ist zwar richtig, aber es sprechen mehrere Gründe dafür, Compiler zu studieren:

- Compiler gehören zu den am häufigsten benutzten Werkzeugen der Softwareentwicklung. Daher sollten wir verstehen, wie sie aufgebaut sind und wie sie funktionieren.
- Wenn wir und mit Compilern beschäftigen, lernen wir auch, wie ein Computer auf Maschinenebene funktioniert. Wir müssen uns mit seinem Instruktionssatz, seinen Registern, seinen Adressierungsarten sowie mit seinen Datenbereichen wie dem Methodenkeller oder dem Heap befassen. Das schafft eine Brücke zwischen Hardware und Software.
- Wer weiß, in welche Instruktionen bestimmte Sprachkonstrukte übersetzt werden, bekommt ein besseres Gefühl für die Effizienz von Programmen.
- Im Compilerbau müssen wir uns mit Grammatiken beschäftigen. Strukturierte Daten durch Grammatiken zu beschreiben, gehört so wie das Programmieren zum Handwerkszeug der Softwareentwicklung.

Darüber hinaus sind Compilerbau-Kenntnisse aber auch im allgemeinen Software Engineering nützlich. Während nur wenige Leute Compiler im engeren Sinne bauen, sind die meisten von ihnen im Laufe ihres Berufslebens irgendwann einmal mit der Aufgabe konfrontiert, compilerähnliche Werkzeuge zu entwickeln. Dies

reicht von der Auswertung von Kommandozeilenparametern über die Verarbeitung von Stücklisten oder Dokumentbeschreibungssprachen (z.B. PDF oder Postscript) bis hin zur statischen Programmanalyse, die aus einem Quellprogramm Kennzahlen wie zum Beispiel seine Komplexität berechnet. In all diesen Fällen sind Compilerbau-Techniken nützlich.

Das vorliegende Buch beschäftigt sich mit dem Compilerbau im engeren Sinne, obwohl die vermittelten Techniken auch im allgemeinen Software Engineering eingesetzt werden können. Es beschreibt die vollständige Implementierung eines Compilers für eine einfache Java-ähnliche Programmiersprache (*MicroJava*), der ausführbaren Bytecode (ähnlich dem Java-Bytecode) erzeugt. Dabei richtet es sich an Fachleute aus der Praxis. Formale Grundlagen werden nur so weit behandelt, wie sie zum Verständnis der angewandten Techniken nötig sind. Das Buch deckt auch bewusst nicht alle Feinheiten des Compilerbaus ab, sondern nur jene Techniken, die in der Praxis benötigt werden. Insbesondere geht es kaum auf Optimierungsverfahren ein, die ein eigenes Buch füllen würden und nur für Leute relevant sind, die Produkt-Compiler schreiben. Ferner behandelt es lediglich Compilerbau-Techniken für imperative Sprachen. Funktionale oder logische Programmiersprachen benötigen zumindest teilweise andere Techniken, die aber im praktischen Software Engineering auch seltener zum Einsatz kommen.

Bücher über fortgeschrittene Techniken des Compilerbaus gibt es in großer Anzahl (z.B. [ALSU08], [Appe02], [Coop22], [FCL09], [Much97]). Sie behandeln vor allem Techniken der statischen und dynamischen Codeanalyse, das umfangreiche Gebiet der Compiler-Optimierung sowie Feinheiten der Codeerzeugung und der Registerallokation. Sie sind allerdings nur für Leute relevant, die Produkt-Compiler für Sprachen wie Java oder C++ entwickeln wollen. Für Einsteiger sind sie oft eher verwirrend.

Das vorliegende Buch entstand aus einer zweistündigen Vorlesung über die Grundlagen des Compilerbaus und kann als Lehrbuch für diese Zwecke eingesetzt werden. Es richtet sich aber auch an Leute aus der Praxis, die die Arbeitsweise eines Compilers besser verstehen und ihr Methodenrepertoire erweitern wollen.

## 1.1 Geschichte des Compilerbaus

Die ersten Compiler entstanden Ende der 1950er-Jahre. Damals war der Compilerbau eine Geheimwissenschaft, ja ein Hype, den man durchaus mit dem heutigen Interesse an künstlicher Intelligenz vergleichen kann. Nur wenige wussten darüber Bescheid, und es gab auch noch kaum Techniken zur Übersetzung von Programmiersprachen. Die Entwicklung der ersten Compiler kostete viele Personenjahre. Heute ist der Compilerbau eines der am besten erforschten Gebiete der Informatik. Es gibt ausgereifte Techniken für die Syntaxanalyse, die Optimierung und die Codeerzeugung, sodass Studierende heute einen (einfachen) Compiler in einem einzigen Semester implementieren können.

Dieses Kapitel gibt einen kurzen (und notwendigerweise unvollständigen) Abriss der Geschichte des Compilerbaus, die gleichzeitig auch eine Geschichte der Programmiersprachen und ihrer Konzepte ist. Dabei werden die Erkenntnisse und Fortschritte an exemplarischen Sprachen festgemacht, die als Meilensteine gelten und die Entwicklung der Compilerbau-Techniken in diesem Zeitfenster repräsentieren.

**1957: Fortran.** Eine der ersten höheren Programmiersprachen war Fortran [Back56]. John Backus, der damals für IBM arbeitete, leitete ein Team, das sich zum Ziel gesetzt hatte, die damals übliche Assembler-Programmierung durch eine höhere Programmiersprache zu ersetzen und einen Compiler dafür zu implementieren, der Maschinencode erzeugte, der es mit der Effizienz handgeschriebener Assembler-Programme aufnehmen konnte. Viele der Grundtechniken des Compilerbaus wurden hier entwickelt. So war es anfangs alles andere als klar, wie man arithmetische Ausdrücke (z.B. `a + b * c`) so in Maschinenbefehle übersetzen konnte, dass die Vorrangregeln der Operatoren beachtet werden. Auch die Übersetzung von Verzweigungen und Schleifen in Maschinenbefehle musste erst erarbeitet werden, ebenso wie die Mechanismen zum Aufruf von Prozeduren oder zum Zugriff auf Arrays.

**1960: Algol.** Algol60 [Naur64] war ein Meilenstein in der Geschichte der Programmiersprachen. Es wurde von einem Konsortium von Experten aus Europa und den USA entwickelt, die teilweise von Universitäten kamen, teilweise aber auch aus der Industrie. Wichtige Neuerungen waren die Blockstruktur mit lokalen und globalen Variablen sowie das Konzept der Rekursion, das es ermöglichte, dass Prozeduren ihre Variablenwerte über rekursive Aufrufe hinweg beibehielten. Algol60 war aber auch deshalb bemerkenswert, weil es die erste Programmiersprache war, deren Syntax formal durch eine Grammatik definiert wurde. John Backus und Peter Naur entwickelten dafür die Backus-Naur-Form (BNF), die heute in zahlreichen Varianten zum Rüstzeug jedes Compilerbauers gehört.

**1970: Pascal.** Ein Nachfolger von Algol60 war die Sprache Pascal [JW75], hinter der Pioniere wie Niklaus Wirth und Tony Hoare standen. Obwohl Pascal als einfache Unterrichtssprache konzipiert war, führte sie wichtige Neuerungen ein, die auch im Compilerbau ihren Niederschlag fanden. Während ältere Sprachen nur vordefinierte Datentypen wie `integer` oder `real` kannten, konnte man in Pascal eigene Datentypen definieren, die man dann zur Deklaration von Variablen verwenden konnte. Neben Arrays als Tabellen gleichartiger Elemente wurden Records (auch Structs genannt) eingeführt, die Daten unterschiedlichen Typs zu einer neuen Einheit zusammenfassten. Obwohl die Idee von Records bereits in Cobol auftauchte, wurden Records als Datentyp erst in Pascal eingeführt. Der Zugriff auf Elemente solcher strukturierten Datentypen erforderte neue Compilerbau-Techniken. Außerdem erzeugten die ersten Pascal-Compiler nicht Maschinencode, sondern sogenannten »P-Code«, der dem heutigen Bytecode von Java ähnelte. P-Code war

wesentlich kompakter als Maschinencode, wodurch Pascal-Programme auch auf Mikrocomputern mit ihren damals sehr beschränkten Speicherkapazitäten liefen. Das förderte die Verbreitung von Pascal, hatte allerdings den Nachteil, dass P-Code nicht direkt auf einer Maschine ausführbar war, sondern interpretiert werden musste, was Laufzeit kostete.

**1985: C++.** C++ [Stro85] ist ein Nachfolger der Sprache C, die etwa zur gleichen Zeit wie Pascal entstand und ebenfalls ein Nachfolger von Algol60 war. In C++ wurde das damalige Wissen im Bereich der Programmiersprachen und des Compilerbaus konsolidiert. Es wurden unter anderem Konzepte wie Objektorientierung, Ausnahmebehandlung oder generische (d.h. parametrisierbare) Datentypen eingeführt, die es zwar auch schon in anderen Sprachen gab, die aber erst mit C++ populär wurden. Insbesondere die Objektorientierung erforderte neue Techniken im Compilerbau, wie zum Beispiel die Darstellung von Vererbungshierarchien oder die Übersetzung dynamisch gebundener Prozeduraufrufe.

**1995: Java.** Auch Java [AGH00] war keine völlig neue Sprache, sondern fasste Konzepte existierender Sprachen zusammen. Ein neuartiges Prinzip war allerdings die *Just-in-time-Compilation* (JIT-Compilation), die Java-Programme portabel machte. Java-Programme werden in Bytecode-Instruktionen übersetzt (ähnlich dem P-Code von Pascal), die auf jedem Rechner ausgeführt werden können, auf dem es einen Bytecode-Interpreter gibt. Besonders häufig ausgeführte Programmteile werden dabei zur Laufzeit (just in time) in Maschinencode des jeweiligen Rechners übersetzt.

**2005: Scala.** Scala [Oder08] ist eine Weiterentwicklung von Java und benutzt als Ausführungsumgebung die Java-Plattform (d.h. einen Bytecode-Interpreter samt JIT-Compiler). Wie viele der heute gängigen Sprachen integriert Scala Konzepte funktionaler Sprachen mit imperativen Konzepten. Zu den funktionalen Konzepten gehören Lambda-Ausdrücke (Funktionen, die als Daten betrachtet werden und anderen Funktionen als Parameter mitgegeben werden können), Lazy Evaluation (Auswertung von Ausdrücken erst wenn diese benötigt werden) oder Pattern-Matching (Erkennen von Mustern in Folgen oder Baumstrukturen). Natürlich erfordern diese Mechanismen ebenfalls spezielle Compilerbau-Techniken.

Die Geschichte der Programmiersprachen und des Compilerbaus ist damit noch lange nicht zu Ende. Auch heute werden noch laufend neue Sprachkonzepte erfunden, die dann zu neuen Compilerbau-Techniken führen. Sie gehen aber weit über die Intention dieses Buches hinaus, nämlich eine Einführung in die grundlegenden und heute gängigen Techniken des Compilerbaus zu geben. Wer an der Geschichte von Programmiersprachen interessiert ist, findet Hintergrundinformationen in den Tagungsbänden der Konferenz »History of Programming Languages« ([HOPL-I], [HOPL-II], [HOPL-III]).

## 1.2 Dynamische Struktur eines Compilers

Als ersten Überblick über die Funktionsweise von Compilern sehen wir uns ihre dynamische Struktur an, also die Phasen, in denen eine Übersetzung abläuft (Abb. 1.1).

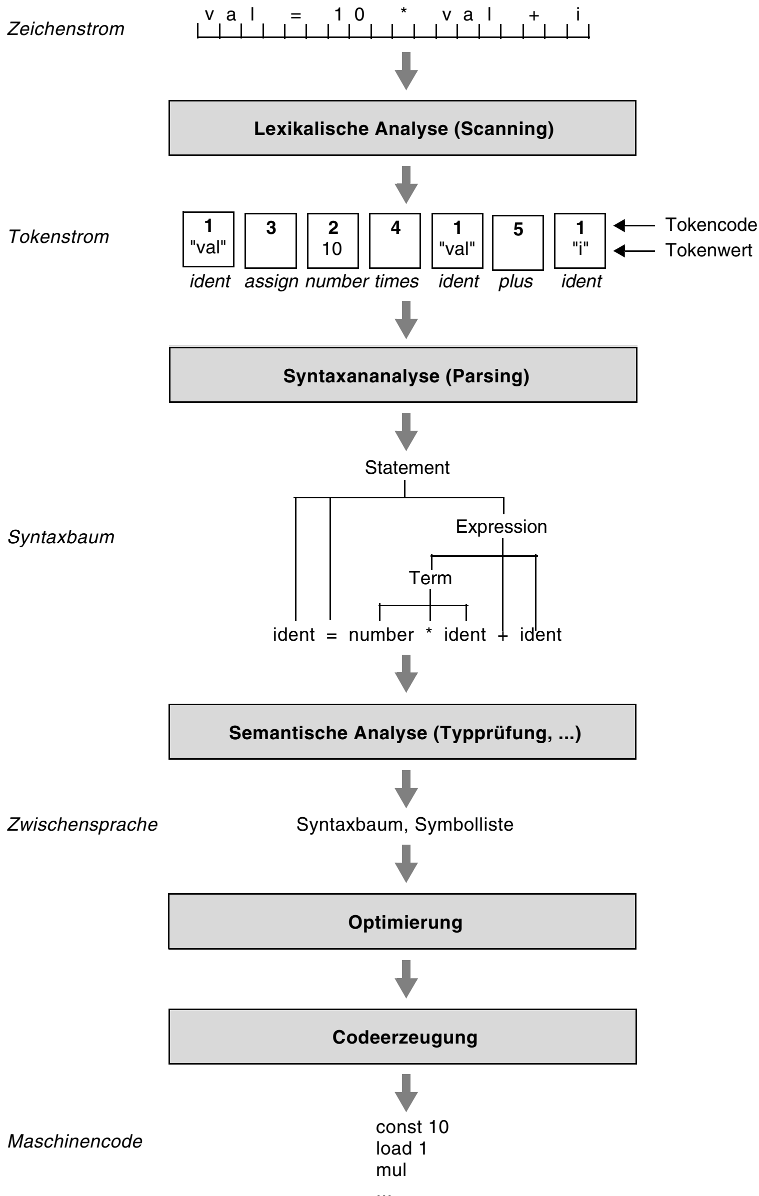


***Abb. 1.1*** *Dynamische Struktur eines Compilers*

**Lexikalische Analyse.** Die lexikalische Analyse ist die erste Phase eines Compilers. Sie transformiert den Zeichenstrom des Quellprogramms in einen Tokenstrom. Dabei eliminiert sie bedeutungslose Zeichen (z.B. Leerzeichen) und fasst andere Zeichen zu Symbolen zusammen (z.B. Namen, Zahlen oder Operatoren), die *Token* genannt werden. Jede Tokenart wird durch einen eindeutigen Tokencode identifiziert (z.B. 1 für Namen (*identifiers*), 2 für Zahlen etc.). Bei manchen Token gibt es nur eine einzige Ausprägung (der Operator "*" ist z.B. durch seinen Tokencode ausreichend identifiziert). Andere Token (z.B. Namen) können aber mehrere Ausprägungen haben (z.B. gibt es unterschiedliche Namen wie "val" oder "i"). Man braucht daher hier zusätzlich zum Tokencode (z.B. 1 = ident) auch einen Tokenwert (z.B. "val"). Jedes Token ist ein Objekt, das durch einen Tokencode und einen optionalen Tokenwert bezeichnet wird. Der Tokenstrom ist eine Abstraktion des Zeichenstroms und vereinfacht die nächsten Schritte des Compilers. Den Compilerteil, der die lexikalische Analyse übernimmt, nennt man den *Scanner*.

**Syntaxanalyse.** Die nächste Phase eines Compilers ist die Syntaxanalyse. Sie analysiert den Tokenstrom anhand der Grammatik der Quellsprache und baut einen Syntaxbaum auf, der die syntaktische Struktur des Quellprogramms widerspiegelt. Gelingt das, so ist das Programm syntaktisch korrekt, und es kann mit der nächsten Phase fortgefahren werden. Gelingt es nicht, liegt ein Syntaxfehler vor, der gemeldet wird und korrigiert werden muss, bevor das Programm erneut compiliert werden kann. Den Compilerteil, der die Syntaxanalyse übernimmt, nennt man den *Parser*.

**Semantische Analyse.** Die Syntaxanalyse prüft nur die syntaktische Korrektheit eines Programms. Weitere Kriterien wie die Bedingungen, dass alle verwendeten Namen deklariert und dass die Datentypen in Zuweisungen und Ausdrücken kompatibel sein müssen, werden in der semantischen Analyse geprüft. Dabei wird auch eine *Symbolliste* aufgebaut, die ein Verzeichnis aller deklarierten Namen und ihrer Eigenschaften darstellt. Der Syntaxbaum und die Symbolliste sind eine weitere Abstraktion des Quellprogramms und stellen eine Zwischenrepräsentation des Programms im Compiler dar.

**Optimierung.** An die semantische Analyse schließt sich üblicherweise die Optimierung an, bei der versucht wird, das Programm durch Transformationen schneller oder kompakter zu machen. Optimierungen sind ein umfangreiches Thema, das ganze Bücher füllt und für Produkt-Compiler essenziell ist. Im vorliegenden Buch gehen wir aber bewusst nicht auf Optimierungen ein, da es uns nur darum geht, die Grundtechniken einfacher Compiler oder compilerähnlicher Werkzeuge zu studieren.

**Codeerzeugung.** Die letzte Phase eines Compilers ist schließlich die Codeerzeugung, bei der aus der Zwischenrepräsentation des Programms der Zielcode erzeugt wird.

### Einpass- und Mehrpass-Compiler

Die Phasen eines Compilers können entweder miteinander verzahnt oder strikt nacheinander ablaufen. Im ersten Fall spricht man von einem Einpass-Compiler, im zweiten Fall von einem Mehrpass-Compiler.

In einem *Einpass-Compiler* (Abb. 1.2) liefert der Scanner das jeweils nächste Token, der Parser prüft, ob es an der aktuellen Stelle in die Grammatik passt, die semantische Analyse verarbeitet das Token, indem sie zum Beispiel Typprüfungen durchführt, bevor der Codegenerator Instruktionen der Zielmaschine erzeugt. Solange das Quellprogramm noch nicht zu Ende ist, beginnt dieser Zyklus anschließend von vorne.

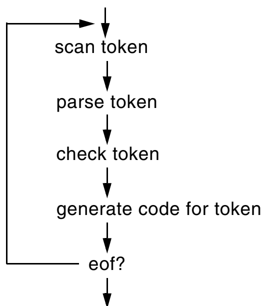


***Abb. 1.2*** *Einpass-Compiler*

In einem *Mehrpass-Compiler* (Abb. 1.3) sind die einzelnen Phasen Programmteile, die nacheinander ablaufen. Der Scanner liest die Quelldatei und erzeugt einen Tokenstrom, der Parser prüft dessen syntaktische Korrektheit und erzeugt einen Syntaxbaum, die semantische Analyse führt weitere Prüfungen durch und erzeugt eine Symbolliste, bevor der Codegenerator daraus den Zielcode erzeugt.

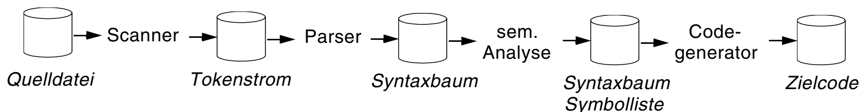


***Abb. 1.3*** *Mehrpass-Compiler*

Mehrpass-Compiler waren früher oft nötig, weil der Hauptspeicher klein war oder die Programmiersprache so komplex, dass eine Aufteilung in kleinere Programmteile nötig erschien. Beides ist heute irrelevant. Daher baut man heute oft Einpass-Compiler. Allerdings wird in optimierenden Produkt-Compilern meist ein Zweipass-Verfahren gewählt, bei denen ein sogenanntes *Frontend* den Scanner, den Parser, die semantische Analyse und einen einfachen Codegenerator enthält,

der eine Zwischensprache erzeugt, die anschließend vom *Backend* in den Zielcode übersetzt wird (Abb. 1.4).

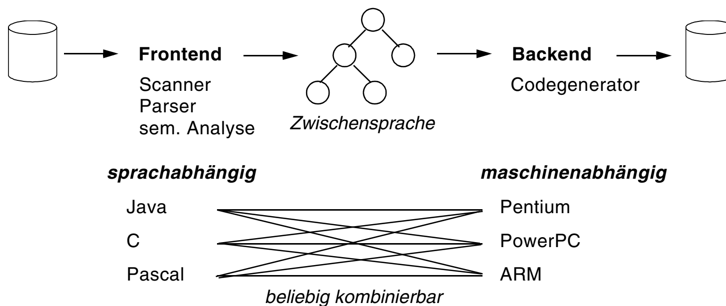


***Abb. 1.4*** *Zweipass-Compiler mit Zwischensprache*

Das Frontend ist sprachabhängig, denn für Sprachen wie Java, C oder Pascal müssen unterschiedliche Scanner und Parser implementiert werden. Das Backend ist hingegen maschinenabhängig, denn für Prozessoren wie Pentium, PowerPC oder ARM müssen unterschiedliche Codegeneratoren implementiert werden. Alle Frontends erzeugen aber dieselbe Zwischensprache und alle Backends transformieren dieselbe Zwischensprache in den jeweiligen Zielcode.

Diese Zerlegung bietet mehrere Vorteile. Zum einen erreicht man eine bessere Portierbarkeit. Wenn man einen Compiler für eine neue Sprache wie Python implementieren möchte, muss man lediglich ein Frontend dafür schreiben. Mit den verschiedenen Backends hat man dann sofort Python-Compiler für alle Zielmaschinen, für die es ein Backend gibt. Generell kann jedes beliebige Frontend mit jedem beliebigen Backend kombiniert werden, um Compiler unterschiedlicher Quellsprachen für unterschiedliche Zielmaschinen zu erhalten. Der größte Vorteil ist aber, dass Optimierungen auf der Zwischensprache wesentlich einfacher zu bewerkstelligen sind als auf der Quellsprache. Daher sind so gut wie alle optimierenden Compiler Zweipass-Compiler.

Der Nachteil dieser Zerlegung besteht darin, dass Zweipass-Compiler langsamer und speicheraufwendiger sind als Einpass-Compiler, weil eine Zwischensprache erzeugt und im Hauptspeicher verwaltet werden muss. Mit den heutigen schnellen Rechnern und großen Speichern ist das aber kein wirkliches Problem. Da wir allerdings im vorliegenden Buch nicht über Optimierungen sprechen, wird unser MicroJava-Compiler ein Einpass-Compiler.

## Compiler und Interpreter

Ein Compiler übersetzt ein Quellprogramm direkt in Maschinencode, der dann geladen und ausgeführt werden kann (Abb. 1.5).

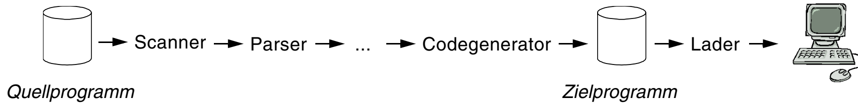


***Abb. 1.5*** *Compiler*

Ein Interpreter führt hingegen ein Programm »direkt« aus, ohne es vorher in Maschinencode zu übersetzen. Allerdings muss das Programm auch hier vorher analysiert werden, d.h., man braucht zumindest einen Scanner und einen Parser, der die Struktur des auszuführenden Programms erkennt. Sobald diese Struktur aber bekannt ist, wird das Programm ausgeführt, d.h. interpretiert (Abb. 1.6).

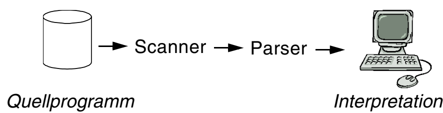


***Abb. 1.6*** *Interpreter*

Bei einem Interpreter spart man sich die vollständige Compilation. Benutzende haben den Eindruck, dass ihr Programm sofort ausgeführt wird. Allerdings ist die Interpretation wesentlich langsamer als die Ausführung eines compilierten Programms. Anweisungen in einer Schleife müssen zum Beispiel bei jedem Schleifendurchlauf erneut vom Scanner und vom Parser verarbeitet werden, bevor sie interpretiert werden können. Daher wird in vielen Sprachen (z.B. in Java und auch in MicroJava) ein Compiler eingesetzt, der allerdings keinen Maschinencode erzeugt, sondern Code einer »virtuellen Maschine« (VM). Java-Programme werden zum Beispiel in *Bytecode* übersetzt – ein einfaches Instruktionsformat, das dann von der Java-VM interpretiert werden kann (Abb. 1.7).

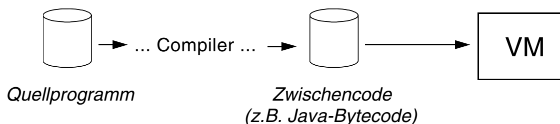


***Abb. 1.7*** *Interpretation von virtuellem Code*

Auf diese Weise muss jede Anweisung des Quellprogramms nur ein einziges Mal analysiert und übersetzt werden, während die Interpretation des Bytecodes effizienter abläuft als die Interpretation von Quellcode. Die VM »simuliert« dabei eine physische Maschine, indem sie Bytecode anstatt Maschinencode ausführt. Das hat auch den Vorteil, dass der Bytecode auf jeder Maschine ausgeführt werden kann, auf der es einen entsprechenden Interpreter gibt. Programme werden dadurch portabel.

## 1.3 Statische Struktur eines Compilers

Die statische Struktur eines Compilers beschreibt die Komponenten (Klassen), aus denen er besteht. In Produkt-Compilern gibt es zahlreiche solcher Komponenten, aber die wichtigsten, aus denen jeder Compiler (auch der MicroJava-Compiler) besteht, werden in Abb. 1.8 dargestellt.

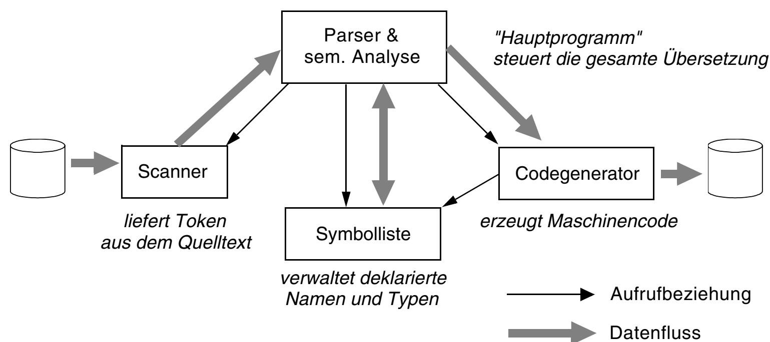


***Abb. 1.8*** *Statische Struktur eines Compilers*

Der Parser, in den auch die semantische Analyse integriert ist, übernimmt die Rolle des Hauptprogramms und steuert die gesamte Übersetzung. Wann immer er ein Token benötigt, ruft er den Scanner auf, der das nächste Token aus dem Quelltext schält und es an den Parser liefert, der es syntaktisch und semantisch analysiert. Deklarierte Namen und ihre Eigenschaften werden vom Parser in die Symbolliste eingetragen und bei ihrer Verwendung aus der Symbolliste abgerufen. Schließlich ruft der Parser Methoden des Codegenerators auf, um Instruktionen des Zielcodes zu erzeugen.

Auch der MicroJava-Compiler ist nach diesem Schema aufgebaut. In Kapitel 2 werden wir uns den Scanner ansehen, in Kapitel 3 den Parser, in Kapitel 5 die Symbolliste und schließlich in Kapitel 6 den Codegenerator.

## 1.4 Grammatiken

Programmiersprachen haben wie natürliche Sprachen eine syntaktische Struktur, die durch eine Grammatik beschrieben werden kann. Eine Grammatik besteht aus Regeln, die angeben, wie die einzelnen Sprachteile aufgebaut sind. Das folgende Beispiel zeigt eine Regel, die die Struktur einer while-Anweisung beschreibt:

```
WhileStatement = "while" "(" Condition ")" Statement.
```

Eine while-Anweisung beginnt also mit dem Schlüsselwort "while", gefolgt von einer geklammerten Bedingung (Condition) und einer Anweisung (Statement), die den

Rumpf der Schleife bildet. Im Allgemeinen bestehen Grammatiken aus folgenden vier Teilen:

- **Terminalsymbole**: Eine Menge von Symbolen (Token), die nicht weiter zerlegt werden und sozusagen die »Atome« der Sprache darstellen. Dazu gehören Schlüsselwörter wie "while" oder "if", Operatoren wie "+" oder "-", Sonderzeichen wie ";" oder "," und schließlich Symbole wie Namen oder Zahlen, die aus Sicht der Grammatik atomar sind.
- **Nonterminalsymbole**: Eine Menge von Symbolen, die größere Sprachteile darstellen und daher in weitere Terminal- oder Nonterminalsymbole zerlegt werden müssen. Dazu gehört zum Beispiel das Nonterminalsymbol WhileStatement aus dem obigen Beispiel, aber auch Condition oder Statement.
- **Produktionen**: Eine Menge von Grammatikregeln, die die Zerlegung von Nonterminalsymbolen in weitere Terminal- und Nonterminalsymbole beschreiben. Das obige Beispiel zeigt die Produktion (und somit die Zerlegung) des Nonterminalsymbols WhileStatement.
- **Startsymbol**: Das oberste Nonterminalsymbol, aus dem alles andere (also die gesamte Sprache) abgeleitet werden kann.

Terminal- und Nonterminalsymbole bilden zusammen das *Alphabet* der durch die Grammatik beschriebenen Sprache, also den Symbolvorrat, aus dem die Grammatik zusammengesetzt ist.

### EBNF-Schreibweise von Grammatiken

Für die Schreibweise von Grammatiken gibt es verschiedene Notationen. Wir verwenden in diesem Buch die EBNF (*Extended Backus Naur Form*) [Wirt77], die nach den Compiler-Pionieren John Backus und Peter Naur benannt ist. Sie ist eine Erweiterung der reinen BNF, auf die wir in Abschnitt 1.5 zurückkommen.

Eine EBNF-Produktion besteht aus einer linken und einer rechten Seite, die durch ein Gleichheitszeichen getrennt sind. Jede Produktion wird durch einen Punkt abgeschlossen:

```
WriteStatement = "write" ident "," Expression ";" .
```

Auf der linken Seite steht ein Nonterminalsymbol. Die rechte Seite besteht aus einer Folge von Terminal- und Nonterminalsymbolen. Terminalsymbole können Namen sein (z.B. ident) oder Literale (z.B. "write", ",", ";"), die sich selbst bedeuten. Nonterminalsymbole sind immer Namen. Per Konvention schreiben wir Terminalsymbole mit kleinem Anfangsbuchstaben (z.B. ident) und Nonterminalsymbole mit großem Anfangsbuchstaben (z.B. Expression).

Die rechte Seite einer Produktion kann in EBNF-Schreibweise außerdem Metasymbole enthalten, die Alternativen trennen, optionale oder wiederholbare Teile darstellen oder mehrere Alternativen durch Klammern gruppieren:

| Metasymbol | Zweck | Beispiel | Bedeutung |
|---|---|---|---|
| \| | trennt Alternativen | a \| b \| c | a oder b oder c |
| (...) | gruppiert Alternativen | a (b \| c) | ab \| ac |
| [...] | Option | [a] b | ab \| b |
| {...} | Wiederholung (0..unendlich oft) | {a} b | b \| ab \| aab \| aaab \| ... |

**Beispiel: Grammatik der arithmetischen Ausdrücke**

Als Beispiel betrachten wir eine EBNF-Grammatik der arithmetischen Ausdrücke, in der als Operanden Namen und Zahlen vorkommen können:

```
Expr   =  ["+" | "-"] Term {("+" | "-") Term} .
Term   =  Factor {("*" | "/") Factor} .
Factor =  ident | number | "(" Expr ")" .
```

Ein Ausdruck (Expr) beginnt mit einem optionalen Vorzeichen ("+" oder "-"), auf das ein oder mehrere Terme folgen können, die durch "+" oder "-" voneinander getrennt sind. Ein Term besteht aus einem oder mehreren Faktoren, die durch "*" oder "/" voneinander getrennt sind. Ein Faktor besteht schließlich aus einem Namen (ident), einer Zahl (number) oder einem geklammerten Ausdruck. Beachten Sie, dass die runden Klammern in Factor Terminalsymbole sind, die in der Eingabesprache vorkommen (deshalb stehen sie in Hochkommas), während die runden Klammern in den Produktionen für Expr und Term Metasymbole sind, die lediglich Alternativen zusammenfassen.

Man kann Grammatiken auch grafisch durch *Syntaxdiagramme* darstellen (Abb. 1.9), die die Symbole der Grammatik durch Linien verbinden, denen man folgen kann, um die aus einem Nonterminalsymbol ableitbaren Symbole zu ermitteln. Solche Diagramme sind zwar einfach zu lesen, nehmen aber viel Platz ein und sind außerdem schwer maschinell zu verarbeiten. Daher bleiben wir in diesem Buch bei der textuellen Grammatiknotation.

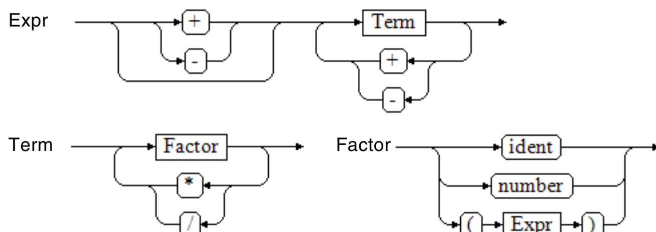


***Abb. 1.9*** *Grammatik als Syntaxdiagramme*

Die Nonterminalsymbole der obigen Grammatik sind Expr, Term und Factor. Bei den Terminalsymbolen unterscheiden wir zwischen *einfachen Terminalsymbolen* ("+", "-", "*", "/", "(", ")"), von denen es nur eine einzige Ausprägung gibt, und *Terminalklassen* (ident, number), von denen es mehrere Ausprägungen gibt (Namen können zum Beispiel x, y oder sum sein; Zahlen können zum Beispiel 1, 10 oder 355 sein). Das Startsymbol dieser Grammatik ist Expr.

Mit Grammatiken lassen sich auch *Vorrangregeln* von Operatoren ausdrücken. Wenn man zum Beispiel den Ausdruck

```
- a * 3 + b / 4 - c
```

nach obiger Grammatik analysiert, wird der Zeichenstrom zuerst in Terminalsymbole umgewandelt, die dann schrittweise zu Nonterminalsymbolen zusammengefasst werden (Abb. 1.10).

```
     -  ident  * number +  ident  / number -  ident
⇨    -  Factor * Factor  + Factor /  Factor  - Factor
⇨    -       Term        +       Term        - Term
⇨                        Expr
```

***Abb. 1.10*** *Analyse des Ausdrucks - a * 3 + b / 4 - c*

Wie man sieht, bezieht sich das Vorzeichen hier auf den gesamten Term (a * 3) und nicht nur auf die Variable a. Möchte man das ändern, muss das optionale Vorzeichen in die Produktion von Factor verschoben werden, sodass - ident zu einem Faktor zusammengefasst wird, bevor Faktor * Faktor als Term erkannt wird:

```
Expr   =  Term {("+" | "-") Term}.
Term   =  Factor {("*" | "/") Factor}.
Factor =  ["+" | "-"] (ident | number | "(" Expr ")").
```

Durch Ändern der Grammatik kann man also die Vorrangregeln beeinflussen. Allgemein gilt, dass Operatoren auf niedrigeren Grammatikebenen Vorrang vor Operatoren auf höheren Grammatikebenen haben. In der zuletzt angegebenen Grammatik hat also das unäre Vorzeichen in der Produktion von Factor Vorrang vor den binären Operatoren "*" und "/" in der Produktion von Term und diese wiederum haben Vorrang vor den binären Operatoren "+" und "-" in der Produktion von Expr.

#### Terminale Anfänge von Nonterminalsymbolen

Für die Syntaxanalyse ist es wichtig, die terminalen Anfänge von Nonterminalsymbolen zu kennen, also die Terminalsymbole, mit denen ein Nonterminalsymbol beginnen kann. Für unsere ursprüngliche Grammatik

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

stellen wir fest, dass die Produktion von Factor drei Alternativen hat. Die erste beginnt mit ident, die zweite mit number und die dritte mit "(". Die terminalen Anfänge von Factor sind also:

First(Factor) = **ident, number, "("**

Die Produktion von Term beginnt mit Factor, dessen terminale Anfänge wir bereits kennen. Es gilt also:

First(Term) = First(Factor) = **ident, number, "("**

Die Produktion von Expr kann schließlich mit einem Vorzeichen ("+" oder "-") beginnen. Weil das Vorzeichen aber optional ist, kann sie auch mit Term beginnen, dessen terminale Anfänge bereits bekannt sind:

First(Expr) = "+", "-", First(Term) = **"+", "-", ident, number, "("**

### Terminale Nachfolger von Nonterminalsymbolen

Ähnlich wie die terminalen Anfänge muss der Syntaxanalysator auch die terminalen Nachfolger von Nonterminalsymbolen kennen, also die Terminalsymbole, die auf ein Nonterminalsymbol in beliebigem Kontext folgen können. Um die terminalen Nachfolger von Expr zu ermitteln, müssen wir uns ansehen, wo Expr auf der rechten Seite einer Produktion vorkommt und welche Terminalsymbole dort folgen können. Expr kommt in der Produktion von Factor vor und wird dort von ")" gefolgt. Da Expr aber das Startsymbol unserer Grammatik ist, wird es auch von einem speziellen Symbol eof (*end of file*) gefolgt, das das Ende des Eingabestroms (d.h. des Ausdrucks) kennzeichnet. Die terminalen Nachfolger von Expr sind also:

Follow(Expr) = **")", eof**

Das Nonterminalsymbol Term kommt an zwei Stellen in der Produktion von Expr vor. Beim ersten Vorkommen folgt eine Iteration ({...}). Wenn diese betreten wird, folgt "+" oder "-". Iterationen können aber auch nullmal ausgeführt werden; in diesem Fall folgen die Nachfolger der Iteration, hier also die terminalen Nachfolger von Expr, die wir bereits kennen:

Follow(Term) = "+", "-", Follow(Expr) = **"+", "-", ")", eof**

Ähnlich ist es beim Nonterminalsymbol Factor, das an zwei Stellen in der Produktion von Term vorkommt. Beim ersten Vorkommen folgt wieder eine Iteration, die mit "*" oder "/" beginnen kann. Wenn die Iteration übersprungen wird, folgen die terminalen Nachfolger von Term, die bereits bekannt sind:

Follow(Factor) = "*", "/", Follow(Term) = **"*", "/", "+", "-", ")", eof**

## Weitere Begriffe der formalen Sprachen

Programmiersprachen sind aus Sicht der Theorie formale Sprachen. Obwohl wir in diesem Buch nur so viel Theorie verwenden, wie wir für den praktischen Compilerbau brauchen, gibt es doch einige Begriffe, die man kennen sollte.

Wir haben bereits erwähnt, dass die Menge der Terminal- und Nonterminalsymbole einer Grammatik das *Alphabet* der Grammatik bildet.

Mit dem Begriff *Kette* bezeichnet man eine endliche Folge von Terminal- oder Nonterminalsymbolen aus einem Alphabet. Ketten werden durch griechische Buchstaben bezeichnet. Beispiele von Ketten aus dem Alphabet unserer Expr-Grammatik sind:

```
α = ident + number
β = - Term + Factor * number
```

Die *leere Kette*, die aus keinem Symbol besteht, bezeichnet man mit ε.

Wenn man in einer Kette α ein Nonterminalsymbol durch die rechte Seite seiner Produktion ersetzt, erhält man eine neue Kette β. Man nennt das eine *direkte Ableitung* und schreibt $\alpha \Rightarrow \beta$. Im folgenden Beispiel wird Factor durch ident ersetzt:

- Term + **Factor** * number ⇒ - Term + **ident** * number

Wenn die Ableitung über mehrere Zwischenstufen erfolgt,

$$\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n \Rightarrow \beta$$

nennt man das eine *indirekte Ableitung* und schreibt $\alpha \Rightarrow^* \beta$. Wird in einer Ableitung $\alpha \Rightarrow \beta$ das linkeste Nonterminalsymbol ersetzt, spricht man von einer *linkskanonischen Ableitung*; wird das rechteste Nonterminalsymbol ersetzt, spricht man von einer *rechtskanonischen Ableitung*.

Die Umkehrung einer Ableitung nennt man *Reduktion*. Findet man in einer Kette β eine Symbolfolge, die der rechten Seite einer Produktion entspricht, und ersetzt man diese Symbolfolge durch das entsprechende Nonterminalsymbol, so hat man β zu einer Kette α reduziert.

Parser arbeiten entweder *top-down*, indem sie aus dem Startsymbol einer Grammatik einen Satz der Sprache ableiten (siehe Abschnitt 3.2), oder *bottom-up*, indem sie einen Satz der Sprache zum Startsymbol reduzieren (siehe Abschnitt 8.1).

Eine aus einem Nonterminalsymbol direkt oder indirekt ableitbare Kette nennt man eine *Phrase* dieses Nonterminalsymbols. Aus Term lassen sich zum Beispiel folgende Ketten ableiten, die somit Term-Phrasen sind:

```
Factor
Factor * Factor
ident * Factor
...
```

Eine aus dem Startsymbol abgeleitete Phrase nennt man eine *Satzform*. Aus Expr lassen sich zum Beispiel folgende Satzformen ableiten:

```
Term + Term - Term
Term + Factor * ident - Term
...
```

Besteht eine Satzform nur aus Terminalsymbolen, so spricht man von einem *Satz* der Grammatik. Sätze unserer Expr-Grammatik sind zum Beispiel:

```
ident * number + ident
number * ( ident + ident )
...
```

Alle aus dem Startsymbol einer Grammatik ableitbaren Sätze bilden die (*formale*) *Sprache* dieser Grammatik. Die Sprache MicroJava ist also die Menge aller gültigen MicroJava-Programme. Meist gibt es unendlich viele solcher Sätze (also unendlich viele MicroJava-Programme).

Ein weiterer Begriff der formalen Sprachen ist die *Löschbarkeit*. Eine Kette $\alpha$ heißt löschbar, wenn sie in die leere Kette abgeleitet werden kann ($\alpha \Rightarrow^* \varepsilon$). In folgender Grammatik

```
X = Y Z.
Y = [b].
Z = c | d | .
```

ist zum Beispiel Y löschbar, weil b optional ist und Y somit in die leere Kette abgeleitet werden kann. Die Produktion von Z besteht aus drei Alternativen, von denen die letzte leer ist; Z kann also ebenfalls in die leere Kette abgeleitet werden und ist daher löschbar. Da Y und Z löschbar sind, kann auch X in die leere Kette abgeleitet werden und ist ebenfalls löschbar.

### Rekursion

Der Begriff der Rekursion (Selbstbezüglichkeit) ist aus der Mathematik bekannt und hat auch bei Grammatiken eine wichtige Bedeutung. Eine Produktion eines Nonterminalsymbols X nennt man rekursiv, wenn X in eine Kette abgeleitet werden kann, die wiederum X enthält ($X \Rightarrow^* \omega_1\ X\ \omega_2$). Die Ketten $\omega_1$ und $\omega_2$ können leer sein, womit sich drei Formen der Rekursion ergeben:

| | | |
|---|---|---|
| **Linksrekursion** | X = b \| X a. | X ⇒ Xa ⇒ Xaa ⇒ Xaaa ⇒ ... ⇒ baaaaa |
| **Rechtsrekursion** | X = b \| a X. | X ⇒ aX ⇒ aaX ⇒ aaaX ⇒ ... ⇒ aaaaab |
| **Zentralrekursion** | X = b \| "(" X ")". | X ⇒ (X) ⇒ ((X)) ⇒ (((X))) ⇒ ... ⇒ (((((b))))) |

Bei Linksrekursion kann ein Nonterminalsymbol in eine Kette abgeleitet werden, die wieder mit diesem Nonterminalsymbol beginnt ($\omega_1$ ist leer), bei Rechtsrekursion kann es in eine Kette abgeleitet werden, die mit diesem Nonterminalsymbol endet ($\omega_2$ ist leer). Wie aus dem obigen Beispiel ersichtlich ist, kann man damit