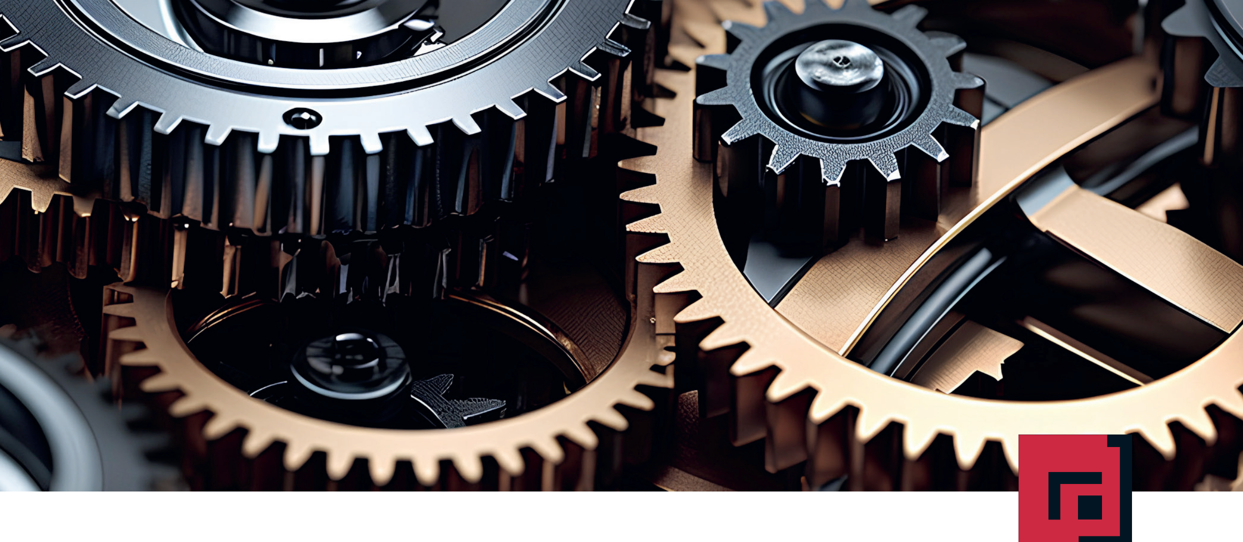

Hanspeter Mössenböck

Compilerbau

Grundlagen und Anwendungen

dpunkt.verlag

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.

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).

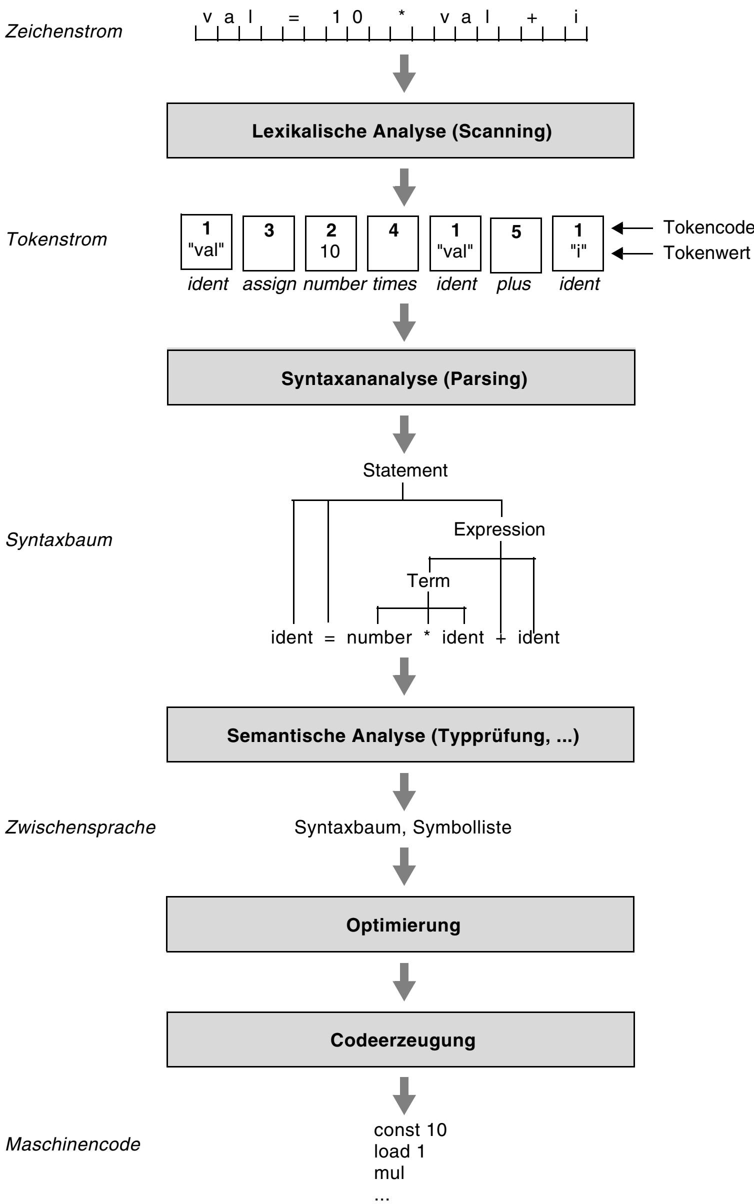

Abb. 1.1 *Dynamische Struktur eines Compilers*

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.

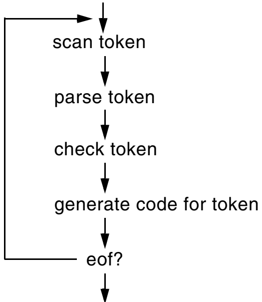

Abb. 1.2 *Einpass-Compiler*

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.

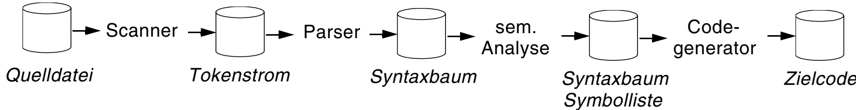

Abb. 1.3 *Mehrpass-Compiler*

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).

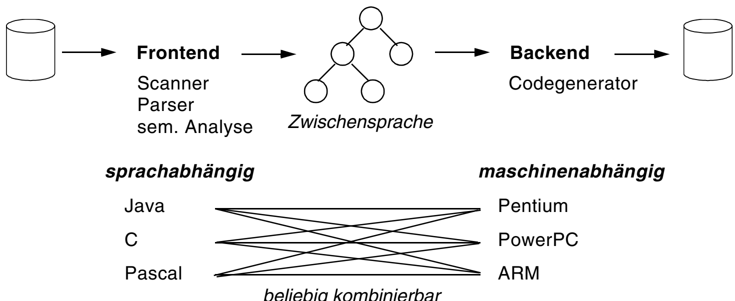

Abb. 1.4 *Zweipass-Compiler mit Zwischensprache*

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).

Scanner Parser ... Codegenerator Lader

Quellprogramm Zielprogramm

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).

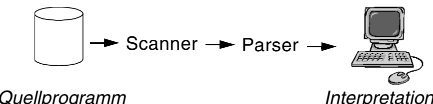

Abb. 1.6 *Interpreter*

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).

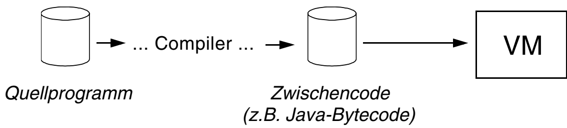

Abb. 1.7 *Interpretation von virtuellem Code*

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.

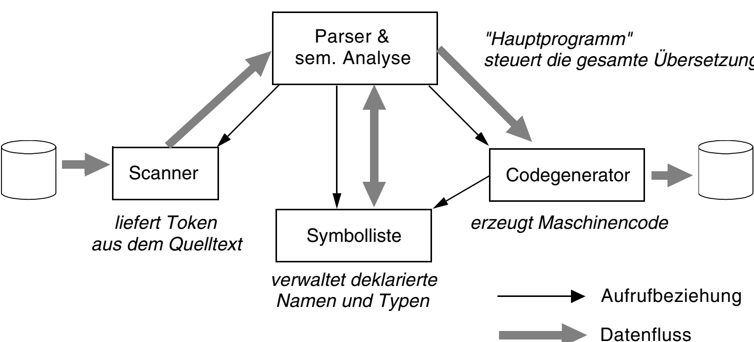

Abb. 1.8 *Statische Struktur eines Compilers*

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
I	trennt Alternativen	a I b I c	a oder b oder c
(...)	gruppiert Alternativen	a (b I c)	ab I ac
[...]	Option	[a] b	ab I b
{...}	Wiederholung (0..unendlich oft)	{a} b	b I ab I aab I aaab I ...

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.

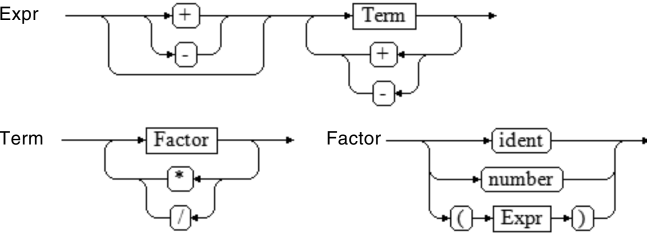

Abb. 1.9 *Grammatik als Syntaxdiagramme*

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,

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

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 α 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 ω_1 und ω_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 (ω_1 ist leer), bei Rechtsrekursion kann es in eine Kette abgeleitet werden, die mit diesem Nonterminalsymbol endet (ω_2 ist leer). Wie aus dem obigen Beispiel ersichtlich ist, kann man damit

Wiederholungen ausdrücken. Zentralrekursion wird hingegen verwendet, um Klammerstrukturen auszudrücken (ω_1 und ω_2 sind nicht leer), wobei es genau so viele Vorkommen von ω_1 gibt wie von ω_2.

Neben der direkten Rekursion, die im obigen Beispiel gezeigt wurde, gibt es auch die indirekte Rekursion. Die folgende vereinfachte Expr-Grammatik

```
Expr   =  Term {"+" Term}.
Term   =  Factor {"*" Factor}.
Factor =  ident | "(" Expr ")".
```

ist indirekt zentralrekursiv (Expr $\Rightarrow^*$ ω_1 Expr ω_2), weil Expr über mehrere Stufen in eine Kette abgeleitet werden kann, die wieder Expr enthält:

Expr $\Rightarrow$ Term $\Rightarrow$ Factor $\Rightarrow$ (Expr)

Beseitigung von Linksrekursion

Bei der Top-down-Syntaxanalyse, die wir uns in Kapitel 3 ansehen, ist Linksrekursion störend und muss daher beseitigt werden. Die linksrekursive Produktion

```
X = b | X a.
```

besteht aus zwei Alternativen. Die erste beginnt mit b und die zweite mit X, aber die terminalen Anfänge von X sind ebenfalls b. Wenn der Parser ein X erkennen möchte und in der Eingabe ein b findet, kann er sich daher nicht zwischen den beiden Alternativen entscheiden, weil beide mit b beginnen.

Glücklicherweise kann man Linksrekursion immer in eine Iteration umwandeln. Das Nonterminalsymbol X kann wie folgt in einen Satz abgeleitet werden:

X $\Rightarrow$ Xa $\Rightarrow$ Xaa $\Rightarrow$ ... $\Rightarrow$ baaa...a

Es ist leicht zu sehen, dass man die linksrekursive Produktion von X in eine iterative EBNF-Produktion umformen kann, die nicht mehr rekursiv ist:

```
X = b {a}.
```

Hier ist ein weiteres Beispiel. Die folgende linksrekursive Produktion

```
Expr = Term | Expr "+" Term.
```

führt zu den Ableitungen

Expr $\Rightarrow$ Expr + Term $\Rightarrow$ Expr + Term + Term $\Rightarrow$ ... $\Rightarrow$ Term + Term + ... + Term

woraus sich folgende iterative EBNF-Produktion ergibt:

```
Expr = Term {"+" Term}.
```

Grammatikklassen nach Chomsky

Der amerikanische Linguist *Noam Chomsky* forschte in den 1950er-Jahren an Grammatiken als Ersetzungssysteme, bestehend aus Regeln $\alpha = \beta$, durch die eine Kette α in eine Kette β abgeleitet werden kann. Je nach Form von α und β unterscheidet er vier Klassen von Grammatiken:

Klasse 0: Unbeschränkte Grammatiken. Hier können α und β beliebige Ketten von Terminal- und Nonterminalsymbolen sein, zum Beispiel:

```
X     =  a X b | Y c Y.
a Y c =  d.
d Y   =  b b.
```

Damit kann man zum Beispiel X wie folgt in einen Satz ableiten:

$X \Rightarrow aXb \Rightarrow aYcYb \Rightarrow dYb \Rightarrow bbb$

Unbeschränkte Grammatiken sind zwar die mächtigste Klasse, weil sie Sprachen nach komplexen Regeln erzeugen können. Allerdings gibt es keinen generellen Algorithmus, um solche Sprachen zu analysieren. Man sagt, dass solche Sprachen durch *Turingmaschinen* erkennbar sind.

Klasse 1: Kontextsensitive Grammatiken. Hier gilt, dass $|\alpha| \leq |\beta|$ sein muss. Die linke Seite kann eine beliebig lange Kette sein, zum Beispiel:

```
a X = a b c.
```

aber sie darf nicht mehr Symbole enthalten als die rechte Seite. Wie man sieht, wird hier der Kontext eines Nonterminalsymbols berücksichtigt: X kann nur dann in b c abgeleitet werden, wenn davor ein a steht. Mit kontextsensitiven Grammatiken können Sprachen erzeugt werden, die durch *linear beschränkte Automaten* (eine Variante von Turingmaschinen) erkennbar sind.

Klasse 2: Kontextfreie Grammatiken. Bei diesen Grammatiken besteht α aus einem einzigen Nonterminalsymbol, während β eine beliebige Kette sein darf, zum Beispiel:

```
X = a b c.
```

Im Prinzip wird auch gefordert, dass β nicht leer sein darf, aber Grammatiken mit Regeln, bei denen β leer ist, können leicht in solche umgeformt werden, bei denen β nicht leer ist. Kontextfreie Grammatiken erzeugen Sprachen, die mit *Kellerautomaten* erkannt werden können (siehe die Kapitel 3 und 8).

Klasse 3: Reguläre Grammatiken. Auch hier besteht α nur aus einem einzigen Nonterminalsymbol, aber β darf nur ein Terminalsymbol sein oder ein Terminalsymbol gefolgt von einem Nonterminalsymbol, zum Beispiel:

```
X = b.
X = b Y.
```

Reguläre Grammatiken erzeugen Sprachen, die durch *endliche Automaten* erkannt werden können (siehe Kapitel 2).

Für den Compilerbau sind lediglich kontextfreie und reguläre Grammatiken relevant. Für sie gibt es effiziente Erkennungsalgorithmen. Kontextfreie Grammatiken werden in der Syntaxanalyse verwendet, reguläre Grammatiken in der lexikalischen Analyse.

1.5 Syntaxbäume

Der Syntaxanalysator hat die Aufgabe, einen Satz einer Sprache gemäß einer Grammatik zu analysieren und seine syntaktische Korrektheit zu überprüfen. Dabei wird ein Syntaxbaum aufgebaut, der die Zerlegung des Satzes in einzelne Satzteile beschreibt.

Für den Aufbau eines Syntaxbaums ist es einfacher, mit reiner BNF (*Backus-Naur-Form*) zu arbeiten, die im Gegensatz zur EBNF keine Klammern zur Gruppierung von Alternativen ((...)), zur Darstellung von Optionalität ([...]) und zur Darstellung von Iterationen ({...}) enthält. Optionalität muss durch mehrere Alternativen ausgedrückt werden und Iteration durch Linksrekursion. Unsere Expr-Grammatik sieht in reiner BNF wie folgt aus:

```
Expr = Sign Term | Expr AddOp Term.
Term = Factor | Term MulOp Factor.
Factor = ident | number | "(" Expr ")".
Sign = "+" | "-" | .
AddOp = "+" | "-".
MulOp = "*" | "/".
```

Für den Eingabesatz 10 + 3 * x wird zum Beispiel der in Abb. 1.11 dargestellte Syntaxbaum aufgebaut:

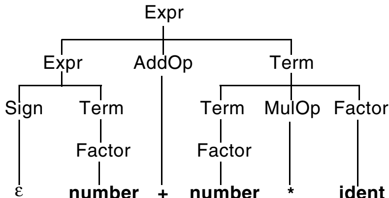

Abb. 1.11 *Konkreter Syntaxbaum für den Satz* 10 + 3 * x

Man sieht, dass sich darin alle Ableitungen vom Startsymbol bis zum Satz widerspiegeln (Expr ⇒ Expr AddOp Term, Term ⇒ Term MulOp Factor usw.). Man nennt diesen Syntaxbaum daher den *konkreten Syntaxbaum* (*parse tree*).

Neben dem konkreten Syntaxbaum gibt es auch den *abstrakten Syntaxbaum*, der die logische Struktur eines Satzes widerspiegelt und wesentlich kompakter ist als der konkrete Syntaxbaum. Seine Blätter stellen die Operanden dar und seine inneren Knoten die Operatoren (Abb. 1.12).

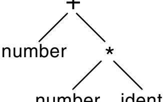

Abb. 1.12 *Abstrakter Syntaxbaum für den Satz* 10 + 3 * x

Der abstrakte Syntaxbaum wird in optimierenden Compilern oft für die interne Darstellung eines Programms verwendet, auf der Optimierungen durchgeführt werden.

Mehrdeutigkeit

Eine Grammatik ist mehrdeutig, wenn man für einen Satz mehrere Syntaxbäume angeben kann. Aus folgender Grammatik (T steht für Term und F für Factor)

```
T = F | T "/" T.
F = id.
```

kann zum Beispiel der Satz id / id / id abgeleitet werden:

$$T \Rightarrow T / T \Rightarrow T / T / T \Rightarrow F / F / F \Rightarrow id / id / id$$

Wie aus Abb. 1.13 ersichtlich, können über diesem Satz zwei unterschiedliche Syntaxbäume aufgebaut werden.

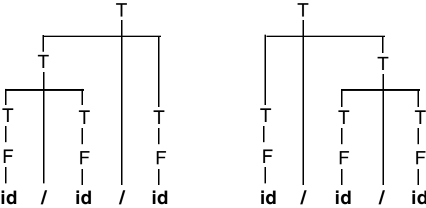

Abb. 1.13 *Unterschiedliche Syntaxbäume über dem Satz* id / id / id

Mehrdeutige Grammatiken sind für die Syntaxanalyse ungeeignet, weil sie unterschiedliche Interpretationen des Satzes zulassen. Der linke Syntaxbaum in Abb. 1.13 besagt zum Beispiel, dass zuerst die ersten beiden Vorkommen von id dividiert werden, während der zweite Syntaxbaum ausdrückt, dass zuerst die letzten beiden Vorkommen von id dividiert werden.

Glücklicherweise ist in diesem Beispiel aber nicht die Sprache mehrdeutig, sondern nur die Grammatik. Wenn man die Grammatik umformt zu

```
T = F | T "/" F.
F = id.
```

können die gleichen Sätze erzeugt werden, aber die Divisionen werden immer von links nach rechts durchgeführt. Die Mehrdeutigkeit wurde also beseitigt.

Es gibt allerdings Sprachen, die inhärent mehrdeutig sind, und man muss nicht lange suchen, um sie zu finden. Die meisten Sprachen der C-Familie (C, C++, C# und Java) enthalten eine inhärente Mehrdeutigkeit, die »*Dangling Else*« genannt wird. Die if-Anweisung kann in diesen Sprachen zwei Formen annehmen:

```
Statement = "if" Condition Statement
          | "if" Condition Statement "else" Statement
          | ... .
```

Wenn man folgende geschachtelte if-Anweisung betrachtet

```
if (a < b) if (b < c) x = c; else x = b;
```

so kann das Schlüsselwort else entweder zum ersten if gehören oder zum zweiten. Es lassen sich also zwei unterschiedliche Syntaxbäume aufbauen (siehe Abb. 1.14), was eine Mehrdeutigkeit darstellt. In diesem Fall ist die Sprache selbst mehrdeutig und nicht nur die Grammatik. Man kann die Grammatik nicht so umformen, dass die Mehrdeutigkeit verschwindet.

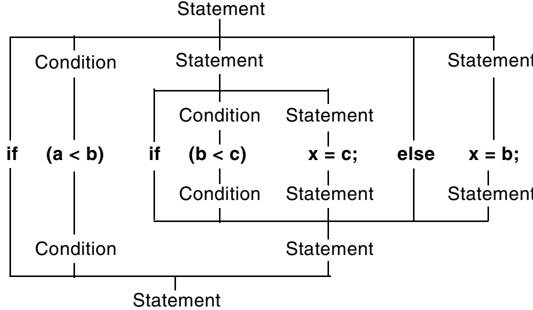

Abb. 1.14 *Mehrdeutigkeit beim »Dangling Else«*

Man löst diese Mehrdeutigkeit so, dass das else dem unmittelbar vorausgehenden if zugeordnet wird. Wenn der Parser die beiden Alternativen der if-Anweisung verfolgt und ein else findet, wird mit der zweiten Alternative fortgefahren, anstatt die erste Alternative zu beenden. Es gilt also der untere Syntaxbaum aus Abb. 1.14. Das ist aber eine reine Konvention, die die Mehrdeutigkeit im Grunde nicht beseitigt.

1.6 MicroJava

Das vorliegende Buch zeigt die Prinzipien des Compilerbaus anhand der Programmiersprache *MicroJava*. Der folgende Abschnitt gibt einen kurzen Überblick über diese Sprache. Eine detailliertere Beschreibung ihrer Syntax und Semantik ist in Anhang A zu finden.

MicroJava ist eine Teilmenge von Java und ist einfach genug, um dafür in einer Lehrveranstaltung einen vollständigen Compiler zu entwickeln, aber auch realistisch genug, um daran die wichtigsten Prinzipien des Compilerbaus zu studieren.

Ein MicroJava-Programm besteht aus einer einzigen Datei, die mit dem Schlüsselwort program beginnt und globale Daten und Methoden enthalten kann. Die Hauptmethode, bei der die Programmausführung beginnt, heißt main(). Es gibt in MicroJava folgende Elemente:

- *Werttypen*: int und char, wobei Zeichen in einem einzigen (ASCII-) Byte gespeichert werden.
- *Referenztypen*: Das sind eindimensionale Arrays wie in Java sowie Klassen mit Feldern, aber ohne Methoden und ohne Vererbung. Variablen dieser Typen enthalten Referenzen.
- *Konstanten*: int-Konstanten (z.B. 3) und char-Konstanten (z.B. 'x'), aber keine String-Konstanten.
- *Variablen*: Diese können lokal zu einer Methode oder global zum gesamten Programm sein.
- *Methoden*: Diese sind global zum Programm; es gibt keine klassenlokalen Methoden.
- *Kommentare*: Zeilenendekommentare, die von "//" bis zum Ende der Zeile gehen.

Das folgende Beispiel zeigt ein MicroJava-Programm mit allen wesentlichen Elementen:

```
program P
    final int size = 10;            // constant declaration

    class Table {                   // class declaration (without methods)
        int[] pos;                  // array declaration (one-dimensional)
        int[] neg;
    }

    Table val;                      // declaration of a global variable

{
    void main()
        int x, i;                   // declaration of local variables
    {
```

```
        //---------- initialize val ------------
        val = new Table;
        val.pos = new int[size];
        val.neg = new int[size];
        i = 0;
        while (i < size) {
            val.pos[i] = 0; val.neg[i] = 0;
            i++;
        }
        //---------- read values ---------
        read(x);
        while (x != 0) {
            if (0 <= x && x < size) {
                val.pos[x]++;
            } else if (-size < x && x < 0) {
                val.neg[-x]++;
            }
            read(x);
        }
    }
}
```

MicroJava-Programme werden in MicroJava-Bytecode übersetzt, der von der MicroJava-VM (siehe Kapitel 6) ausgeführt werden kann.

1.7 Übungsaufgaben

Zu allen Übungsaufgaben in diesem Buch gibt es Musterlösungen, die von [Download] heruntergeladen werden können.

1. *Grammatik von E-Mail-Adressen*. Erstellen Sie eine Grammatik für E-Mail-Adressen. Diese bestehen aus einem Adressteil und einem Domänenteil, die durch "@" getrennt sind. Sowohl der Adressteil als auch der Domänenteil sollen in dieser Grammatik aus einer Liste von Namen (ident) bestehen, die durch Punkte getrennt sind. Während der Adressteil aus nur einem einzigen Namen bestehen darf, muss der Domänenteil zumindest zwei Namen aufweisen, zum Beispiel:

   ```
   max.mustermann@some.company.com
   ```

2. *Grammatik vereinfachter boolescher Ausdrücke*. Erstellen Sie eine Grammatik boolescher Ausdrücke, die als Operanden Namen (ident) sowie die Konstanten true und false haben und als Operatoren &&, || und ! (wobei ! Vorrang vor && und && Vorrang vor || haben soll). Teilausdrücke sollen auch geklammert werden können. Orientieren Sie sich dabei an der Grammatik arithmetischer Ausdrücke aus Abschnitt 1.4. Beispiel:

   ```
   (big || small) && ready || big && ! ready
   ```

3. *Grammatik römischer Zahlen*. Erstellen Sie eine Grammatik der römischen Zahlen von 1 bis 20 (also I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, XVII, XVIII, XIX, XX). Terminalsymbole sind I, V und X.

4. *Terminale Anfänge und Nachfolger (1)*. Geben Sie die terminalen Anfänge und Nachfolger aller Nonterminalsymbole in folgender Grammatik an (Namen, die mit Kleinbuchstaben beginnen, sind Terminalsymbole):

```
Course   = Intro Section {Section} Final.
Intro    = lecture [questions].
Section  = {lecture | questions} (project | test).
Final    = [panic] test.
```

5. *Terminale Anfänge und Nachfolger (2)*. Geben Sie die terminalen Anfänge und Nachfolger aller Nonterminalsymbole in folgender Grammatik an:

```
Message = Header [Data] Status.
Header  = "get" | "put" [number | "final"].
Data    = number {number}.
Status  = "ok" | number.
```

6. *Beseitigung von Linksrekursion (1)*. Formen Sie folgende linksrekursive Grammatik mittels EBNF-Iteration in eine äquivalente nichtrekursive Grammatik um:

```
A = A B | a b.
B = B d | c.
```

7. *Beseitigung von Linksrekursion (2)*. Formen Sie folgende linksrekursive Grammatik mittels EBNF-Iteration in eine äquivalente nichtrekursive Grammatik um:

```
List = List number | .
```

8. *Syntaxbäume (1)*. Gegeben sei folgende BNF-Grammatik für vereinfachte arithmetische Ausdrücke:

```
Expr    = Term | Expr AddOp Term.
Term    = Factor | Term MulOp Factor.
Factor  = number | "-" Factor | "(" Expr ")".
AddOp   = "+" | "-".
MulOp   = "*" | "/".
```

   a) Zeichnen Sie den konkreten und den abstrakten Syntaxbaum für den Ausdruck 3 + 5.

   b) Zeichnen Sie den konkreten und den abstrakten Syntaxbaum für den Ausdruck (7 - 2) * 5 + 1.

9. *Syntaxbäume (2)*. Gegeben sei folgende stark vereinfachte Grammatik von Anweisungen:

```
Statement =
   ident "=" ident "-" ident
|  "{" Statement {";" Statement} "}"
|  "if" "(" ident ">" ident ")" Statement
|  "while" "(" ident ">" ident ")" Statement.
```

   Zeichnen Sie den abstrakten Syntaxbaum des Programms

```
if (a > b) {
    while (a > b) a = a - b;
    b = b - a
}
```

   wobei "if", "while", "=", ">", "-" und ";" als Operatoren aufgefasst werden und ident als Operand. Der Operator ";" verkettet zwei Anweisungen.

10. *Mehrdeutigkeit*. Die Grammatik

```
List = ident | List "," List.
```

   ist mehrdeutig, weil man über gewisse Sätze unterschiedliche Syntaxbäume aufbauen kann.

   a) Zeichnen Sie alle möglichen Syntaxbäume für den Satz ident , ident , ident.

   b) Transformieren Sie die Grammatik so, dass sie die gleiche Sprache erzeugt, aber eindeutig ist.

3 Syntaxanalyse

Nach der lexikalischen Analyse liegt das Quellprogramm als Tokenstrom vor, der nur noch die bedeutungstragenden Symbole enthält; alle Leerzeichen, Tabulatoren, Zeilenenden und Kommentare wurden entfernt.

Die Aufgabe des Syntaxanalysators (des *Parsers*) ist es nun, den Tokenstrom nach der gegebenen Grammatik zu analysieren und (zumindest virtuell) einen Syntaxbaum aufzubauen. Gelingt das, ist das Programm syntaktisch korrekt und kann weiterverarbeitet werden. Gelingt es nicht, liegt ein Syntaxfehler vor, der gemeldet werden muss; anschließend soll die Syntaxanalyse fortgesetzt werden, sodass eventuelle weitere Fehler erkannt werden können.

Wie bei der lexikalischen Analyse beginnen wir mit einem kurzen Theorieteil, in dem wir uns kontextfreie Grammatiken und ihren Erkennungsmechanismus, nämlich Kellerautomaten, ansehen.

3.1 Kontextfreie Grammatiken und Kellerautomaten

In Kapitel 2 haben wir gesehen, dass reguläre Grammatiken keine Zentralrekursion ausdrücken können. Da aber Zentralrekursion in Programmiersprachen häufig vorkommt, brauchen wir zur Beschreibung solcher Sprachen die nächsthöhere Grammatikklasse, nämlich *kontextfreie Grammatiken* (KFG). Eine Grammatik heißt kontextfrei, wenn alle Produktionen folgende Form haben:

$x = \alpha.$

Auf der linken Seite steht ein einzelnes Nonterminalsymbol, die rechte Seite α besteht aus einer Folge von Terminal- und Nonterminalsymbolen; in EBNF können auch Metasymbole wie |, (...), [...] und {...} vorkommen. Ein einfaches Beispiel einer kontextfreien Grammatik ist folgende indirekt zentralrekursive EBNF-Grammatik der arithmetischen Ausdrücke:

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

Kontextfreie Sprachen werden durch *Kellerautomaten* erkannt, die wir uns nun näher ansehen.

Kellerautomaten

Ein Kellerautomat (engl. *Push Down Automaton*, kurz PDA) besteht wie ein endlicher Automat (DFA) aus Zuständen und Zustandsübergängen. Im Gegensatz zu einem endlichen Automaten

- erlaubt er allerdings Zustandsübergänge nicht nur mit Terminalsymbolen, sondern auch mit Nonterminalsymbolen;
- merkt er sich die Zustandsübergänge in einem Keller und kann somit nach der Erkennung eines Nonterminalsymbols ein Stück des Wegs zurückgehen und mit dem erkannten Nonterminalsymbol weiterfahren.

Nehmen wir als Beispiel eine sehr einfache zentralrekursive Grammatik:

```
E = x | "(" E ")".
```

Der Kellerautomat zur Erkennung der Sprache dieser Grammatik sieht in einer ersten Version wie folgt aus (Abb. 3.1):

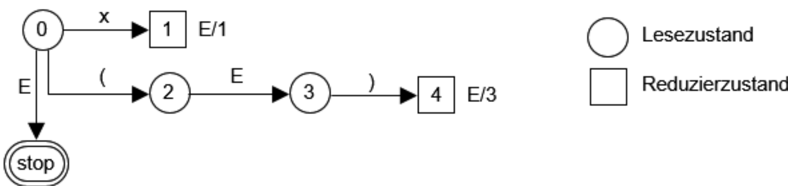

Abb. 3.1 *Vorläufiger Kellerautomat für E = x | "(" E ")".*

Es fällt sofort auf, dass es zwei Arten von Zuständen gibt. In *Lesezuständen* (durch Kreise dargestellt) wird wie beim endlichen Automaten das nächste Eingabesymbol gelesen, und es wird damit in einen Folgezustand übergegangen (z.B. von 0 mit x nach 1). In *Reduzierzuständen* (durch Quadrate dargestellt) wird ein Nonterminalsymbol erkannt. Daher werden dort die zuletzt erkannten Symbole zu diesem Nonterminalsymbol reduziert, wobei der Automat ein Stück seines Wegs zurückgeht und anschließend mit dem erkannten Nonterminalsymbol fortsetzt. Die Reduzierzustände sind mit einer Reduzieraktion beschriftet. In Zustand 4 lautet diese Beschriftung »E/3«, was bedeutet, dass der Automat im Zustand 4 das Nonterminalsymbol E erkannt hat und 3 Kanten zurückgeht (bis in den Zustand 0). Von dort setzt er mit E fort und gelangt in den Stoppzustand. Dies erklärt auch die Bedeutung von Zustandsübergängen mit Nonterminalsymbolen.

Auch im Zustand 2 gibt es einen Übergang mit E, und zwar in den Zustand 3. Bevor dieser Übergang aber stattfinden kann, muss zuerst das Nonterminalsymbol E erkannt werden. Man kann sich das so vorstellen, wie wenn der E-Automat im Zustand 2 rekursiv aufgerufen wird. Nach der Erkennung von E kehrt der rekursiv aufgerufene Automat in den Zustand 2 zurück, und es erfolgt der Übergang mit E (Abb. 3.2).

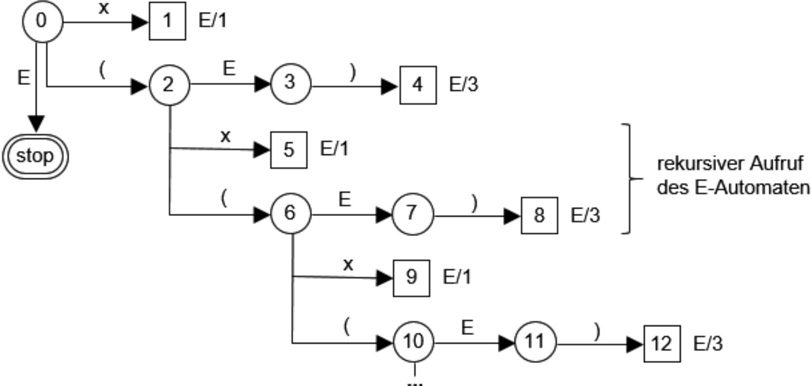

Abb. 3.2 *Rekursiver Aufruf des E-Automaten im Zustand 2*

Das Problem bleibt aber bestehen: Im Zustand 6 findet wieder ein Übergang mit E statt, sodass vorher wieder der E-Automat aufgerufen werden muss, usw. Dies würde zu endloser Rekursion führen.

Wenn wir aber genau hinsehen, stellen wir fest, dass die Zustände 1 und 5 identisch sind und ebenso die Zustände 2 und 6. Wir können also von 2 mit x nach 1 gehen und von 2 mit einer öffnenden Klammer wieder nach 2. Somit ergibt sich folgender endgültige Kellerautomat (Abb. 3.3):

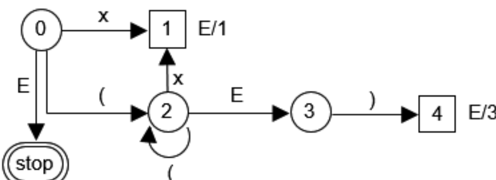

Abb. 3.3 *Kellerautomat für E = x | "(" E ")".*

Bei der Erkennung des Satzes ((x)) durchläuft der Automat folgende Zustände:

Keller	*Aktion*
0	gehe mit **(** in den Zustand 2
0 **2**	gehe mit **(** in den Zustand 2
0 2 **2**	gehe mit **x** in den Zustand 1
0 2 2 **1**	reduziere x zu E und gehe eine Kante zurück
0 2 **2**	gehe mit **E** in den Zustand 3
0 2 2 **3**	gehe mit **)** in den Zustand 4
0 2 2 3 **4**	reduziere (E) zu E und gehe 3 Kanten zurück
0 **2**	gehe mit **E** in den Zustand 3
0 2 **3**	gehe mit **)** in den Zustand 4
0 2 3 **4**	reduziere (E) zu E und gehe 3 Kanten zurück
0	gehe mit **E** in den Stoppzustand
0 **stop**	Satz erkannt

Wie man sieht, merkt sich der Kellerautomat also den Weg, den er gegangen ist, in Form der durchlaufenen Zustände in einem »Keller« und kann daher nach einer Reduktion diesen Weg zurückgehen und mit dem erkannten Nonterminalsymbol fortsetzen. Der Kellerautomat ist also mächtiger als der endliche Automat, weil er seine »Geschichte« kennt und nach einer Reduktion zurückgehen kann. Daher kann er auch im Gegensatz zu einem endlichen Automaten mit Zentralrekursion umgehen.

Einschränkungen kontextfreier Grammatiken

Wie wir gesehen haben, können reguläre Grammatiken nicht mit Zentralrekursion umgehen. Gibt es auch bei kontextfreien Grammatiken ähnliche Einschränkungen? Ja, die gibt es: Kontextfreie Grammatiken können keine Kontextbedingungen ausdrücken.

Eine kontextfreie Grammatik beschreibt lediglich die *Syntax* einer Sprache, aber nicht ihre (statische) *Semantik*. Diese wird durch Kontextbedingungen ausgedrückt, die vom Compiler ebenfalls geprüft werden müssen. Beispiele für solche Kontextbedingungen sind:

- *Jeder Name muss vor seiner Verwendung deklariert worden sein.*
  Die Deklaration eines Namens steht meist viele Zeilen vor seiner Verwendung und gehört somit zu ihrem Kontext. Daher kann die Anweisung

  ```
  x = 3;
  ```

  richtig oder falsch sein, je nachdem, ob x deklariert wurde oder nicht. Mit kontextfreien Grammatiken kann man das nicht ausdrücken.

- *In Ausdrücken müssen die Typen der Operanden übereinstimmen.*
  Die Typen der Operanden werden bei ihrer Deklaration festgelegt und gehören somit zum Kontext des Ausdrucks.

Wie kann man dieses Problem lösen? Eine Möglichkeit wäre, auf die nächsthöhere Grammatikklasse umzusteigen, also auf kontextsensitive Grammatiken. Diese sind aber für den Compilerbau zu kompliziert und können nicht effizient analysiert werden.

Daher lösen wir das Problem, indem wir die Prüfung der Kontextbedingungen auf die nächste Compilerphase (die semantische Analyse) verschieben. Das Programm

```
char x;
...
x = 3
```

wird also vom Parser als syntaktisch korrekt erkannt. Die Inkompatibilität bei der Zuweisung wird erst bei der semantischen Analyse entdeckt und gemeldet.

Kontextbedingungen

Zur Beschreibung der Semantik von Programmiersprachen gibt es zwar formale Notationen (z.B. [Schm86], [GTWW77]), die aber meist aufwendig zu erstellen und schwer zu lesen sind. Wir verwenden daher eine halbformale Notation, bei der für jede Produktion der Grammatik die zu prüfenden Kontextbedingungen in natürlicher Sprache aufgelistet werden.

Hier sind einige Beispiele. Die vollständige Liste der Kontextbedingungen für MicroJava ist Anhang A zu entnehmen.

Statement = Designator "=" Expr ";".

- Designator muss eine Variable, ein Arrayelement oder ein Objektfeld bezeichnen.
- Der Typ von Expr muss mit dem Typ von Designator zuweisungskompatibel sein.

Factor = "new" ident "[" Expr "]".

- ident muss einen Typ bezeichnen.
- Der Typ von Expr muss int sein.

$Designator_0$ = $Designator_1$ "[" Expr "]".

- Der Typ von $Designator_1$ muss ein Array sein.
- Der Typ von Expr muss int sein.

Wie man sieht, beziehen sich die Kontextbedingungen auf die Symbole in den Produktionen (z.B. ident, Expr oder Designator). Kommt ein Symbol mehrfach vor (wie in der dritten Produktion), unterscheiden wir die Vorkommen durch Indizes (z.B. $Designator_1$).

Wir werden später sehen, dass die semantische Analyse keine separate Phase des Compilers ist, sondern in den Parser integriert wird. Schritthaltend mit der syntaktischen Analyse einer Produktion erfolgt auch ihre semantische Analyse. Dabei werden die entsprechenden Kontextbedingungen geprüft und eventuelle Fehler gemeldet.

Vergleich regulärer und kontextfreier Grammatiken

Zusammenfassend wollen wir nochmals reguläre und kontextfreie Grammatiken gegenüberstellen und uns ihr Anwendungsgebiet, ihre Besonderheiten und ihre Einschränkungen bewusst machen (Abb. 3.4).

Reguläre Grammatiken werden bei der lexikalischen Analyse verwendet, kontextfreie Grammatiken bei der Syntaxanalyse. Der Erkennungsmechanismus regulärer Sprachen ist der endliche Automat (DFA), der Erkennungsmechanismus kontextfreier Sprachen ist der Kellerautomat (PDA), der mächtiger ist als der endliche Automat, weil er sich die »Geschichte« der Analyse in einem Keller merkt und somit auch Klammerkonstrukte (Zentralrekursion) verarbeiten kann.

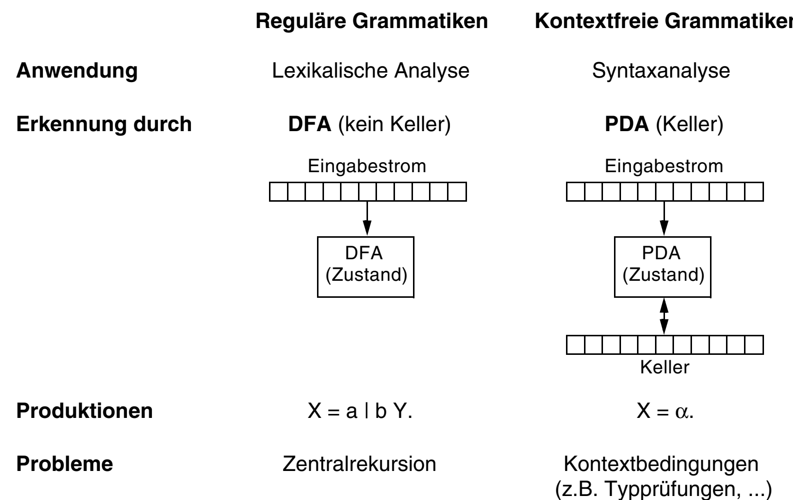

Abb. 3.4 *Reguläre versus kontextfreie Grammatiken*

Die Produktionen regulärer Grammatiken dürfen auf der rechten Seite nur ein Terminalsymbol oder ein Terminalsymbol gefolgt von einem Nonterminalsymbol enthalten (nach einer anderen Definition dürfen sie nur aus einer einzigen nichtrekursiven EBNF-Regel bestehen), während die Produktionen kontextfreier Grammatiken auf der rechten Seite beliebige EBNF-Konstrukte aufweisen können.

Reguläre Grammatiken können keine Klammerstrukturen darstellen und somit nicht mit Zentralrekursion umgehen. Bei kontextfreien Grammatiken ist Zentralrekursion kein Problem; sie können aber keine Kontextbedingungen ausdrücken und somit nicht die Anforderungen an die semantische Korrektheit eines Programms beschreiben.

3.2 Rekursiver Abstieg

Wir kommen nun zur eigentlichen Syntaxanalyse, also zur Implementierung eines Parsers für eine gegebene Grammatik. Gleich vorweg sei gesagt, dass es mehrere Syntaxanalyseverfahren gibt. Wir beschreiben hier das einfachste Verfahren, das gleichzeitig das einzige ist, das man ohne Werkzeuge (d.h. ohne Parsergeneratoren) per Hand implementieren kann. Ein alternatives Verfahren wird in Kapitel 8 beschrieben.

Man nennt das hier beschriebene Verfahren den *rekursiven Abstieg* (engl. *recursive descent*). Es handelt sich um ein Top-down-Verfahren, bei dem der Syntaxbaum für einen gegebenen Eingabesatz von oben nach unten aufgebaut wird. Nehmen wir als Beispiel die Grammatik

```
X = a X c | b b.
```

und den Eingabesatz a b b c. Die Analyse beginnt mit dem Startsymbol der Grammatik (hier X) und dem Eingabesatz a b b c. Dazwischen muss ein Syntaxbaum aufgebaut werden, der das Startsymbol auf den Eingabesatz abbildet (Abb. 3.5).

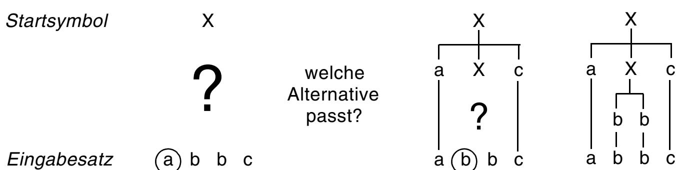

Abb. 3.5 *Arbeitsweise der Top-down-Syntaxanalyse*

Das erste Eingabesymbol ist a. Welche der beiden Alternativen von X passt dazu? Die erste Alternative ist a X c, die zweite b b. Es kann daher lediglich die erste Alternative passen, da nur sie mit einem a beginnt. Wir setzen sie also als nächste Ebene unter dem Startsymbol X ein und können das erste und letzte Symbol des Eingabesatzes mit a X c zur Deckung bringen.

Es verbleiben somit im Syntaxbaum das Nonterminalsymbol X und der noch unabgedeckte Rest des Eingabesatzes (b b). Das erste Symbol dieses Rests ist b, das nur zur zweiten Alternative von X passt. Wir setzen also diese Alternative (b b) als nächste Ebene des Syntaxbaums ein und können damit den Rest des Eingabesatzes abdecken. Der Syntaxbaum ist somit vollständig aufgebaut und der Eingabesatz ist als syntaktisch korrekt erkannt.

Allgemein gilt: Der Parser wählt die passende Alternative eines Nonterminalsymbols immer aufgrund des nächsten Eingabesymbols (des *Vorgriffssymbols*) und der *terminalen Anfänge* dieser Alternativen.

Der Parser als Klasse

Der Parser ist so wie der Scanner eine Klasse mit globalen Feldern und Methoden. Zu jedem Zeitpunkt schaut er ein Token voraus und steuert damit die Analyse. Man nennt dieses Token das *Vorgriffssymbol* (*lookahead token*). Sein Tokencode wird in einem globalen Feld sym gespeichert:

```
private static int sym;       // token code of the lookahead token
```

Ferner merkt sich der Parser die beiden aktuellen Token t und la:

```
private static Token t;       // most recently recognized token
private static Token la;      // lookahead token (not yet recognized)
```

Token ist der Typ der Objekte, die vom Scanner für die einzelnen Terminalsymbole geliefert werden (siehe Kapitel 2). Abb. 3.6 zeigt den Zusammenhang zwischen t, la und sym.

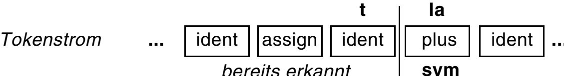

Abb. 3.6 *Aktuelle Token im Tokenstrom*

Der senkrechte Strich deutet die aktuelle Parserposition an. t ist das zuletzt erkannte Token, la ist das Vorgriffstoken, das noch nicht erkannt wurde und sym ist dessen Tokencode. Jedes Mal, wenn ein Token vom Parser erkannt wurde, wird die aktuelle Parserposition durch die Hilfsmethode scan um ein Token weiterbewegt:

```
private static void scan() {
    t = la;
    la = Scanner.next();
    sym = la.kind;
}
```

Am Beginn der Syntaxanalyse wird scan aufgerufen, sodass das erste Token des Eingabestroms in la steht und dessen Tokencode in sym; t ist noch undefiniert.

Wir werden uns nun ansehen, wie man ganz systematisch (fast mechanisch) aus einer gegebenen EBNF-Grammatik einen Parser ableiten und implementieren kann. Wir geben dazu für jedes Konstrukt der Grammatik (d.h. für Terminalsymbole, Nonterminalsymbole, Alternativen, Optionen und Iterationen) ein Muster an, das zeigt, wie man dieses Konstrukt in ein Stück Parsercode umsetzt.

Erkennung von Terminalsymbolen

Wenn auf der rechten Seite einer Produktion ein Terminalsymbol a vorkommt, so lautet die entsprechende Parseraktion check(a).

check ist eine Hilfsmethode, die den Tokencode des zu erkennenden Tokens als Parameter mitbekommt und prüft, ob dieses Token mit dem Vorgriffssymbol sym übereinstimmt. Ist das der Fall, wird weitergelesen, ansonsten wird ein Fehler gemeldet.

```
private static void check (int expected) {
    if (sym == expected) scan();  // token recognized => move ahead
    else error(name[expected] + " expected");
}
```

Als Fehlermeldung wird einfach ausgegeben, welches Token erwartet, aber nicht erkannt wurde. Dazu benutzen wir ein globales Array name, das mit den Namen der einzelnen Token in der Reihenfolge ihrer Tokencodes initialisiert ist:

```
private static String[] name = {"?", "identifier", "number", ..., "+", "-", ...};
```

Die Methode error gibt die Fehlermeldung zusammen mit der entsprechenden Zeilen- und Spaltennummer aus, deren Werte dem Vorgriffstoken la entnommen werden:

```
private static void error (String msg) {
    System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
    System.exit(1);  // for a better solution see later
}
```

Nach der Ausgabe des Fehlers wird die Compilation hier mit System.exit(1) abgebrochen. Natürlich ist das keine gute Lösung, denn wir wollen ja in einem einzigen Durchgang möglichst viele Fehler erkennen. Wie man das macht, sehen wir uns in Abschnitt 3.4 an.

Die Tokencodes, die wir als Parameter von check verwenden, werden wie im Scanner (Kapitel 2) als benannte Konstanten deklariert:

```
static final int none = 0, ident = 1, number = 2, ...;
```

Erkennung von Nonterminalsymbolen

Wenn auf der rechten Seite einer Produktion ein Nonterminalsymbol X vorkommt, so besteht die Parseraktion darin, eine gleichnamige Parsermethode aufzurufen:

```
private static void X() {
    ... actions to parse X ...
}
```

Für jedes Nonterminalsymbol der Grammatik gibt es also eine gleichnamige Parsermethode, die dieses Nonterminalsymbol erkennt.

Das Startsymbol der MicroJava-Grammatik heißt MicroJava und hat ebenfalls eine gleichnamige Parsermethode, die das gesamte MicroJava-Programm analysiert, indem sie weitere Parsermethoden aufruft und Terminalsymbole mit check erkennt. Die Hauptmethode der Klasse Parser lautet daher:

```
public static void parse() {
    scan();          // fill la and sym
    MicroJava();     // parse the whole MicroJava program
    check(eof);      // make sure that there is nothing left after the program
}
```

Erkennung von Sequenzen

Wir können nun diese beiden Muster zusammensetzen und uns ansehen, wie Sequenzen von Terminal- und Nonterminalsymbolen erkannt werden. Für die Produktion:

```
X = a Y c.
```

werden Parsermethoden für X und Y geschrieben. Die Parsermethode für X sieht wie folgt aus:

```
private static void X() {
    // sym holds the first token of X
    check(a);
    Y();
    check(c);
    // sym holds the first token after X
}
```

Man sieht, dass die Muster zur Erkennung von Terminal- und Nonterminalsymbolen mechanisch eingesetzt werden können, um die Sequenz a Y c zu erkennen. Abb. 3.7 zeigt eine Simulation des Ablaufs der Parsermethoden für X und Y.

Abb. 3.7 *Simulation der Parsermethoden für X und Y*

Jeder Aufruf von check konsumiert das nächste Token des Eingabestroms, bis dieser leer ist.

Erkennung von Alternativen

Kommt auf der rechten Seite einer Produktion eine Alternativenkette $\alpha \mid \beta \mid \gamma$ vor (wobei α, β und γ beliebige EBNF-Ausdrücke sein können), so wird ihre Erkennung im Parser wie folgt umgesetzt (Pseudocode):

```
if (sym ∈ First(α)) { ... parse α ... }
else if (sym ∈ First(β)) { ... parse β ... }
else if (sym ∈ First(γ)) { ... parse γ ... }
else error("..."); // find a meaningful error message
```

Es wird der Reihe nach geprüft, ob das Vorgriffssymbol sym zu einer der Alternativen passt (wobei z.B. First(α) die terminalen Anfänge von α sind). Ist das der Fall, wird die entsprechende Alternative analysiert. Wenn keine Alternative passt, wird ein Fehler gemeldet, wobei es wichtig ist, hier eine aussagekräftige Fehlermeldung zu finden. Sehen wir uns dazu ein Beispiel an.

Für folgende Grammatik

```
X = a Y | Y b.
Y = c | d.
```

bestimmen wir zunächst die terminalen Anfänge ihrer Alternativen:

```
First(c)   = {c}
First(d)   = {d}
First(aY)  = {a}
First(Yb)  = First(Y) = {c, d}
```

Wir können nun die Parsermethoden für X und Y wie folgt implementieren:

```
private static void X() {
    if (sym == a) {                       // if sym matches the first alternative
        check(a);
        Y();
    } else id (sym == c || sym == d) {    // if sym matches the second alternative
        Y();
        check(b);
    } else error("invalid start of X");
}

private static void Y() {
    if (sym == c) {                       // if sym matches the first alternative
        check(c);
    } else if (sym == d) {                // if sym matches the second alternative
        check(d);
    } else error("invalid start of Y");
}
```

Als Fehlermeldungen geben wir hier aus, dass das Vorgriffssymbol kein gültiger Anfang von X oder Y ist.

Erkennung von EBNF-Optionen

Kommt auf der rechten Seite einer Produktion ein optionaler EBNF-Ausdruck [α] vor, lautet die entsprechende Parseraktion:

```
if (sym ∈ First(α)) { ... parse α ... } // no error branch!
```

Für die Produktion

```
X = [a b] c.
```

sieht die Parsermethode von X wie folgt aus:

```
private static void X() {
    if (sym == a) {                       // if sym matches start of option, i.e., a
        check(a);
        check(b);
    }
    check(c);
}
```

Lautet die Eingabe a b c, so wird die Option betreten, die a und b analysiert. Anschließend wird c erkannt. Ist die Eingabe hingegen nur c, wird die Option übersprungen und es wird nur c erkannt. Der optionale Teil [a b] darf also fehlen.

Erkennung von EBNF-Iterationen

Wenn auf der rechten Seite einer Produktion eine Iteration {α} vorkommt (wobei α wieder ein beliebiger EBNF-Ausdruck sein kann), so wird das wie folgt in Parsercode umgesetzt:

```
while (sym == First(α)) { ... parse α ... }
```

Betrachten wir als Beispiel folgende Grammatik:

```
X = a {Y b} c.
Y = d | e.
```

Die terminalen Anfänge des wiederholten Teils Y b sind die terminalen Anfänge von Y, also d und e. Die Parsermethode von X lautet daher:

```
private static void X() {
    check(a);
    while (sym == d || sym == e) {          // while sym matches First(Y b)
        Y();
        check(b);
    }
    check(c);
}
```

Falls die terminalen Anfänge einer Iteration sehr zahlreich sind, kann es effizienter sein, die Schleife so lange zu durchlaufen, bis ein terminaler *Nachfolger* der Iteration auftritt, wobei die Schleife aber zur Sicherheit auch mit eof abgebrochen werden sollte:

```
private static void X() {
    check(a);
    while (sym != c && sym != eof) {
        Y();
        check(b);
    }
    check(c);
}
```

Wenn möglich, sollte aber die erste Form der Schleife verwendet werden, da die Schleife sonst erst mit eof verlassen wird, falls der terminale Nachfolger (hier c) in der Eingabe fehlt.

Umgang mit großen First-Mengen

Bei der Erkennung von Alternativen, Optionen und Iterationen muss das Vorgriffssymbol sym mit den terminalen Anfängen dieser Konstrukte verglichen werden. Die Menge dieser terminalen Anfänge kann sehr groß sein.

Als Faustregel gilt: Falls die Menge der terminalen Anfänge mehr als vier Elemente enthält, sollte mit der Klasse BitSet gearbeitet werden. Wenn zum Beispiel die terminalen Anfänge zweier Nonterminalsymbole X und Y wie folgt lauten:

```
First(X) = {a, b, c, d, e}
First(Y) = {f, g, h, i, j}
```

so kann man diese Mengen im Parser folgendermaßen deklarieren:

```
import java.util.BitSet;
...
private static BitSet firstX = new BitSet();
private static BitSet firstY = new BitSet();
```

und sie am Beginn des Parsers entsprechend initialisieren:

```
firstX.set(a); firstX.set(b); firstX.set(c); firstX.set(d); firstX.set(e);
firstY.set(f); firstY.set(g); firstY.set(h); firstY.set(i); firstY.set(j);
```

Die Analyse der Produktion

```
Z = X | Y.
```

kann dann auf folgende Weise umgesetzt werden:

```
private static void Z() {
    if (firstX.get(sym)) X();            // if sym ∈ First(X)
    else if (firstY.get(sym)) Y();       // if sym ∈ First(Y)
    else error("invalid start of Z");
}
```

Enthält eine Menge von terminalen Anfängen hingegen weniger als fünf Elemente, ist es effizienter, diese Elemente direkt abzufragen. Für die Menge

```
First(X) = {a, b, c}
```

ist es besser, die Abfrage wie folgt zu schreiben:

```
if (sym == a || sym == b || sym == c) ...
```

Vermeiden von Mehrfachabfragen

Wir haben im bisherigen Teil dieses Kapitels Muster angegeben, nach denen man eine EBNF-Grammatik systematisch in Parsercode umsetzen kann. Der entstehende Parser ist zwar korrekt, enthält aber Ineffizienzen, weil manche Abfragen mehrfach erfolgen. Das lässt sich optimieren. Für die Produktion

```
X = a | b.
```

würde die unoptimierte Parsermethode lauten:

```
private static void X() {
    if (sym == a) check(a);
    else if (sym == b) check(b);
    else error("invalid start of X");
}
```

Der Aufruf von check(a) prüft, ob sym == a ist und liest in diesem Fall mit scan weiter. Dass sym == a ist, wurde aber schon in der Methode X geprüft und ist daher redundant. Allgemein kann man jedes check(x) durch ein scan ersetzen, wenn vorher überprüft wurde, dass sym == x ist. Somit ergibt sich folgende optimierte Lösung:

```
private static void X() {
    if (sym == a) scan();
    else if (sym == b) scan();
    else error("invalid start of X");
}
```

Auf diese Weise lassen sich viele Mehrfachabfragen vermeiden, wie auch bei folgender Grammatik:

```
X = {a | Y d}.
Y = b | c.
```

Die Parsermethode von X lautet unoptimiert:

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) check(a);
        else if (sym == b || sym == c) { Y(); check(d); }
        else error("a, b or c expected");
    }
}
```

Auch hier kann man check(a) durch scan ersetzen. Man kann aber noch mehr tun: Da die Schleife nur betreten wird, wenn sym a, b oder c ist, ist die Abfrage im ersten else-Zweig überflüssig. Wenn sym nicht a ist, muss es b oder c sein. Man kann daher diese Abfrage eliminieren, und auch der Fehlerzweig entfällt. Die optimierte Lösung lautet:

```
private static void X() {
    while (sym == a || sym == b || sym == c) {
        if (sym == a) scan();
        else /* unconditionally */ { Y(); check(d); } // no error branch
    }
}
```

Aber selbst das ist noch nicht optimal. In dieser Lösung wird zweimal sym == a geprüft, was man durch folgende Implementierung vermeiden kann:

```
private static void X() {
    for (;;) { // endless loop
        if (sym == a) scan();
        else if (sym == b || sym == c) { Y(); check(d); }
        else break;
    }
}
```

Hier wird jede Abfrage nur ein einziges Mal durchgeführt. Man kann dieses Muster immer dann anwenden, wenn eine Iteration über Alternativen vorliegt, also zum Beispiel $\{\alpha \mid \beta \mid \gamma\}$. Man implementiert dann die Iteration durch eine Endlosschleife und fragt darin die einzelnen Alternativen der Reihe nach ab. Wenn keine passt, wird die Schleife verlassen.

Korrekte Ermittlung der terminalen Anfänge

In Abschnitt 1.4 haben wir uns angesehen, wie man die terminalen Anfänge von Nonterminalsymbolen bestimmt. Auch in den Parsermethoden der vorangegangenen Abschnitte haben wir immer wieder die terminalen Anfänge von Alternativen, Optionen und Iterationen ermittelt. Im Prinzip ist das einfach, aber es gibt gewisse Feinheiten, die zu beachten sind.

Wenn man die terminalen Anfänge eines EBNF-Ausdrucks $\alpha_0\alpha_1$ berechnet und α_0 löschbar ist, müssen nicht nur die terminalen Anfänge von α_0 berücksichtigt werden, sondern auch die terminalen Nachfolger von α_0, also die terminalen Anfänge von α_1. In folgender Grammatik

```
X = Y a.
Y = {b} c        // can start with b and c
  | [d]          // can start with d and a(!)
  | e.           // can start with e
```

beginnt die erste Alternative von Y mit dem löschbaren Konstrukt {b}. Die Alternative muss also betreten werden, wenn b oder c vorliegt. Die zweite Alternative von Y ist zur Gänze löschbar. Sie muss betreten werden, wenn d oder ein Nachfolger von [d] vorliegt (hier a als Nachfolger von Y). Die Parsermethode von Y lautet also:

```
private static void Y() {
    if (sym == b || sym == c) {
        while (sym == b) scan();
        check(c);
    } else if (sym == d || sym == a) {
        if (sym == d) scan();
    } else if (sym == e) {
        scan();
    } else error("invalid start of Y");
}
```

Würde die zweite Alternative von Y nicht auch mit a betreten, würde bei Vorliegen von a ein Fehler gemeldet. Da sie aber mit a betreten wird und kein d vorliegt, wird Y verlassen und a wird in der Parsermethode von X erkannt.

Hier ist noch ein weiteres Beispiel. In folgender Grammatik

```
U = V e        // can start with d (i.e., First(V)) and e, since V is deletable
  | f.         // can start with f
V = {d}.
```

ist das Nonterminalsymbol V löschbar. Da die erste Alternative von U mit V beginnt, muss sie betreten werden, wenn ein terminaler Anfang von V vorliegt (hier d) oder ein terminaler Nachfolger von V (hier e). Die Parsermethoden von U und V lauten also:

```
private static void U() {
    if (sym == d || sym == e) {
        V(); check(e);
    } else if (sym == f) {
        scan();
    } else error("invalid start of U");
}

private static void V() {
    while (sym == d) scan();
}
```

Wird die erste Alternative von U mit e betreten, wird zunächst V aufgerufen. Die Methode V kehrt aber zurück, ohne die Schleife auszuführen; anschließend wird e in U erkannt.

Man sieht aus diesen Beispielen, dass bei der Berechnung der terminalen Anfänge eines EBNF-Ausdrucks auch die Löschbarkeit berücksichtigt werden muss.

Der Syntaxbaum im rekursiven Abstieg

Am Beginn dieses Kapitels wurde behauptet, dass der Parser einen Syntaxbaum über dem Tokenstrom aufbaut. Wo ist aber dieser Syntaxbaum? Die oben beschriebenen Parseraktionen bauen keinen auf.

Die Antwort lautet: Beim rekursiven Abstieg wird der Syntaxbaum nur implizit aufgebaut, d.h., er steckt in den Parsermethoden, die gerade aktiv sind, oder anders gesagt, in den Produktionen, an denen gerade gearbeitet wird. Betrachten wir als Beispiel folgende Grammatik:

```
X = a Y d.
Y = b c.
```

Beim Aufruf der Parsermethode von X wird am Teilbaum mit der Wurzel X und den Söhnen a, Y und d gearbeitet (siehe Abb. 3.8a).

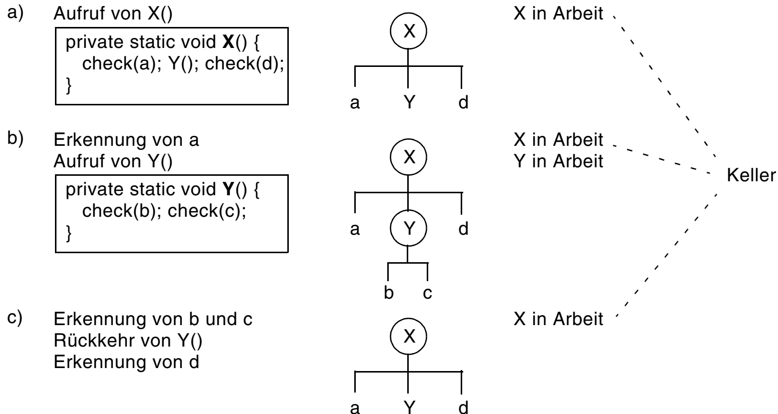

Abb. 3.8 *Impliziter Aufbau eines Syntaxbaums beim rekursiven Abstieg*

Die Parsermethode von X erkennt a und ruft anschließend die Parsermethode von Y auf, wodurch der Syntaxbaum implizit um eine weitere Ebene mit den Symbolen b und c erweitert wird (Abb. 3.8b).

Die Parsermethode von Y erkennt b und c und kehrt dann zu X zurück, wodurch der implizite Teilbaum mit der Wurzel Y wieder abgebaut wird (Abb. 3.8c).

Der Syntaxbaum also wird beim rekursiven Abstieg nicht explizit aufgebaut, sondern existiert nur in »gedachter« Form, nämlich in Form aller Produktionen, an denen der Parser gerade arbeitet.

Auch der Keller des Kellerautomaten liegt beim rekursiven Abstieg nur implizit vor, nämlich in Form der Aufrufkette aller gerade aktiven Parsermethoden. Wenn die Methode X die Methode Y aufruft, wird die Aufrufkette verlängert; wenn Y zurückkehrt, wird sie wieder verkürzt. Man kann sich das so vorstellen, wie wenn der Kellerautomat nach Erkennung von Y an den Anfang von Y zurückgeht und dann mit Y in der Parsermethode von X fortfährt.

3.3 LL(1)-Eigenschaft

Damit eine Grammatik für den rekursiven Abstieg geeignet ist, muss sie die LL(1)-Eigenschaft erfüllen. LL(1) bedeutet, dass die Sätze der Grammatik von **l**inks nach rechts mit **l**inkskanonischen Ableitungen und **1** Vorgriffssymbol erkennbar sind. Diese Erklärung ist allerdings nicht besonders hilfreich. Daher verwenden wir folgende Definition:

- Eine Grammatik ist LL(1), wenn alle ihre Produktionen LL(1) sind.
- Eine Produktion ist LL(1), wenn für alle Alternativen $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ in ihr gilt: $\forall\ i \neq j$: $First(\alpha_i) \cap First(\alpha_j) = \{\}$.

Es muss also gelten, dass die terminalen Anfänge aller Alternativen einer Alternativenkette paarweise disjunkt sind. Mit anderen Worten: Es müssen alle Alternativen einer Alternativenkette mit unterschiedlichen Terminalsymbolen beginnen, sodass sich der Parser aufgrund des Vorgriffssymbols eindeutig für eine der Alternativen entscheiden kann. Ist das nicht der Fall, liegt ein LL(1)-Konflikt vor.

Leider erfüllen viele Grammatiken, die man in Sprachbeschreibungen oder im Internet findet, diese Bedingung nicht. Man kann sie aber meist umformen, sodass sie LL(1) werden und somit für den rekursiven Abstieg geeignet sind.

Beseitigung von LL(1)-Konflikten

LL(1)-Konflikte lassen sich meist durch *Faktorisierung*, also durch Herausheben der gemeinsamen Anfänge von Alternativen, beseitigen. Betrachten wir dazu die Produktion der if-Anweisung in Java:

```
IfStatement  =  "if" "(" Expr ")" Statement
             |  "if" "(" Expr ")" Statement "else" Statement.
```

Die beiden Alternativen dieser Produktion beginnen beide mit if, weshalb die Produktion nicht LL(1) ist. Sie ist auch nicht LL(k) für ein beliebiges k, weil Expr und Statement beliebig lang sein können und man daher im schlimmsten Fall unbeschränkt weit vorausschauen muss, um festzustellen, ob ein else folgt oder nicht.

Wenn man allerdings die gemeinsamen Anfänge der Alternativen heraushebt und die Alternativen erst danach beginnen lässt, erhält man

```
IfStatement = "if" "(" Expr ")" Statement (
                                          | "else" Statement
                                          ).
```

oder in EBNF-Darstellung:

```
IfStatement = "if" "(" Expr ")" Statement ["else" Statement].
```

Die Alternativen sind verschwunden und mit ihnen der LL(1)-Konflikt. Allerdings werden wir später sehen, dass noch immer ein LL(1)-Konflikt vorliegt, um den wir uns auf Seite 61 kümmern werden.

Um LL(1)-Konflikte besser zu erkennen und sie durch Faktorisierung beseitigen zu können, ist es manchmal hilfreich, ein Nonterminalsymbol durch die rechte Seite seiner Produktion zu ersetzen. Betrachten wir dazu folgende Grammatik:

```
Statement   =  Designator "=" Expr ";"
            |  ident "(" [ActualParameters] ")" ";".
Designator  =  ident {"." ident}.
```

Ein Statement kann hier eine Zuweisung oder ein Methodenaufruf mit optionalen Parametern sein. Beide Alternativen beginnen mit ident, was man aber besser sieht, wenn man Designator in Statement einsetzt:

```
Statement   = ident {"." ident} "=" Expr ";"
            | ident "(" [ActualParameters] ")" ";".
```

Nun ist es offensichtlich, dass beide Alternativen mit ident beginnen. Indem man ident heraushebt, kann man den LL(1)-Konflikt beseitigen:

```
Statement = ident  ( {"." ident} "=" Expr ";"
                   | "(" [ActualParameters] ")" ";"
                   ).
```

Die Produktion enthält noch immer eine Alternativenkette, wobei aber die erste Alternative mit "." oder "=" beginnt und die zweite mit "(". Die terminalen Anfänge der Alternativen sind also disjunkt, sodass der LL(1)-Konflikt beseitigt ist. Der Parser kann sich aufgrund des Vorgriffssymbols entscheiden, welche Alternative er betreten soll.

Beseitigung von Linksrekursion

Linksrekursion stellt immer einen LL(1)-Konflikt dar und muss daher beseitigt werden, bevor eine Grammatik für den rekursiven Abstieg geeignet ist. Wir haben allerdings bereits in Kapitel 1 gesehen, dass das immer möglich ist. Schauen wir uns dazu nochmals folgende linksrekursive Grammatik an:

```
IdentList = ident | IdentList "," ident.
```

Sie erzeugt die Sätze:

```
ident
ident "," ident
ident "," ident "," ident
...
```

Es ist leicht zu erkennen, dass dieselben Sätze auch durch folgende EBNF-Grammatik erzeugt werden können:

```
IdentList = ident {"," ident}.
```

Linksrekursion kann auf diese Weise immer durch EBNF-Iteration ersetzt werden.

Versteckte LL(1)-Konflikte in EBNF-Konstrukten

Die LL(1)-Bedingung fordert, dass der Parser immer, wenn er zwischen Alternativen wählen muss, die passende Alternative aufgrund des Vorgriffssymbols ermitteln kann. Neben expliziten Alternativenketten, wie wir sie in den obigen Beispielen betrachtet haben, gibt es aber auch implizite Alternativen, die sich hinter EBNF-Optionen oder EBNF-Iterationen verstecken.

Wenn der Parser vor einer Option [α] β steht, kann er entweder die Option betreten und α erkennen oder die Option überspringen und mit β weitermachen. Es handelt sich hier also um eine implizite Alternativenkette $\alpha\beta \mid \beta$. Daher müssen die terminalen Anfänge von α und β disjunkt sein.

Ähnlich ist es bei Iterationen. Wenn der Parser vor einer Iteration {α} β steht, kann er entweder die Iteration betreten und α erkennen oder die Iteration überspringen und mit β weitermachen (denken Sie daran, dass Iterationen eine null- oder mehrmalige Wiederholung bedeuten). Daher müssen auch hier die terminalen Anfänge von α und β disjunkt sein.

Bei jedem Auftreten einer Option oder Iteration in einer Grammatik muss daher Folgendes überprüft werden:

```
[α] β        First(α) ∩ First(β) muss {} sein
{α} β        First(α) ∩ First(β) muss {} sein
```

Steht eine Option oder Iteration am Ende einer Produktion, müssen ihre terminalen Anfänge disjunkt zu den Nachfolgern der Produktion sein:

```
X = α [β].   First(β) ∩ Follow(X) muss {} sein
X = α {β}.   First(β) ∩ Follow(X) muss {} sein
```

Dies betrifft auch den Sonderfall, dass eine Produktion eine leere Alternative aufweist:

```
X = α | .    First(α) ∩ Follow(X) muss {} sein
```

Glücklicherweise lassen sich solche versteckten LL(1)-Konflikte meist durch Umformung der Grammatik beseitigen. In der Produktion

```
Name = [ident "."] ident.
```

tritt zum Beispiel so ein versteckter LL(1)-Konflikt auf, weil die Option mit ident beginnt und nach der Option wieder ein ident folgt. Wenn der Parser also vor der Option steht und das Vorgriffssymbol ident ist, kann er sich nicht entscheiden, ob er die Option betreten oder sie überspringen soll. Um den LL(1)-Konflikt zu beseitigen, müssen wir die Produktion umformen. Dazu überlegen wir uns, welche Phrasen erzeugt werden können:

```
ident "." ident
ident
```

Die Produktion lässt sich also wie folgt umformen:

```
Name = ident ["." ident].
```

Der ursprüngliche LL(1)-Konflikt ist dadurch verschwunden, aber wir müssen nun überprüfen, ob die Anfänge der Option (also ".") und die Nachfolger von Name disjunkt sind. Dazu müssten wir die Nachfolger von Name berechnen, was aber hier nicht gezeigt wird.

Hier ist ein weiteres Beispiel, das nicht mehr so trivial ist:

```
Program = Declarations ";" Statements.
Declarations = Decl {";" Decl}.
```

Da eine Iteration vorkommt, müssen wir prüfen, ob ihre terminalen Anfänge (also ";") und die Nachfolger von Declarations disjunkt sind, was nicht der Fall ist, weil auf Declarations wieder ein ";" folgt. Man sieht diesen LL(1)-Konflikt besser, wenn man Declarations in Program einsetzt:

```
Program = Decl {";" Decl} ";" Statements.
```

Wenn der Parser vor der Iteration steht und das Vorgriffssymbol ";" ist, kann er sich nicht entscheiden, ob er in die Iteration einsteigen oder sie überspringen soll. Die Produktion von Program lässt sich aber umformen zu

```
Program = Decl ";" {Decl ";"} Statements.
```

Der ursprüngliche LL(1)-Konflikt ist dadurch verschwunden, aber wir müssen noch prüfen, ob $First(Decl) \cap First(Statements) = \{\}$ ist, wofür wir uns die Produktionen von Decl und Statements ansehen müssten, die hier nicht gezeigt wurden.

Die Überprüfung der LL(1)-Bedingungen ist also aufwendig, kann aber durch Werkzeuge (siehe Kapitel 7) unterstützt werden. In der MicroJava-Grammatik (Anhang A) wurden bereits alle LL(1)-Konflikte beseitigt. Wenn man aber eine Grammatik aus einer externen Quelle verwendet, müssen die LL(1)-Bedingungen geprüft und eventuelle LL(1)-Konflikte durch Umformung beseitigt werden, bevor die Grammatik für den rekursiven Abstieg verwendet werden kann.

Dangling Else

In Kapitel 1 haben wir gesehen, dass bei geschachtelten if-Anweisungen wie

```
if (a < b) if (b < c) x = c; else x = b;
```

eine Mehrdeutigkeit entsteht, die man *»Dangling Else«* nennt: Das else kann entweder dem äußeren oder dem inneren if zugeordnet werden. Diese Mehrdeutigkeit wird üblicherweise dadurch gelöst, dass wir das else per Konvention immer dem unmittelbar vorausgehenden if zuordnen.

Das Dangling Else stellt aber auch einen LL(1)-Konflikt dar. Betrachten wir dazu die Grammatik

```
Statement = "if" "(" Expr ")" Statement ["else" Statement]
          | ... .
```

Die Grammatik enthält eine Option am Ende der Produktion. Daher müssen die terminalen Anfänge der Option (also else) und die Nachfolger von Statement disjunkt sein. Wie man sieht, kann aber auf Statement wieder ein else folgen, wodurch ein LL(1)-Konflikt entsteht. Wenn der Parser vor der Option steht und das Vor-

griffssymbol else ist, kann er sich nicht entscheiden, ob er die Option betreten und somit das else dem inneren if zuordnen soll oder ob er die Option überspringen und somit das else dem äußeren if zuordnen soll.

Leider lässt sich dieser LL(1)-Konflikt nicht durch Umformung der Grammatik beseitigen – er ist inhärent. Wie kann man das Problem daher lösen?

Dazu müssen wir uns überlegen, was der Parser macht, wenn ein LL(1)-Konflikt nicht beseitigt wird. Wenn das Vorgriffssymbol zu mehr als einer Alternative passt, wählt der Parser immer die erste passende Alternative. In der Grammatik

```
X = a b c
  | a d.
```

beginnen beide Alternativen mit a. Wenn das Vorgriffssymbol a ist, wählt der Parser immer die erste Alternative; die zweite würde nie betreten. Das ist mit Sicherheit nicht das gewünschte Verhalten, daher muss dieser LL(1)-Konflikt durch Umformung der Grammatik beseitigt werden, was aber einfach ist:

```
X = a (b c | d).
```

Wie sieht das aber beim Dangling Else aus? Wenn der Parser in der Grammatik

```
Statement  =  "if" "(" Expr ")" Statement ["else" Statement]
           |  ... .
```

vor der Option steht und entscheiden muss, ob er mit dem Vorgriffssymbol else die Option betreten oder überspringen soll, wählt er die erste Variante: Er betritt die Option und ordnet somit das else dem inneren if zu. Das ist in diesem Fall genau das gewünschte Verhalten, daher brauchen wir diesen LL(1)-Konflikt nicht zu beseitigen. Der Parser macht implizit das Richtige.

Ein LL(1)-Konflikt ist also lediglich eine Warnung, dass der Parser bei mehreren passenden Alternativen die erste davon wählt. Wenn das wie im Fall des Dangling Else das gewünschte Verhalten ist, kann der LL(1)-Konflikt ignoriert werden, ansonsten muss man ihn beseitigen.

Weitere Anforderungen an eine Grammatik

Neben der LL(1)-Bedingung muss eine Grammatik noch weitere Eigenschaften erfüllen, um für die Syntaxanalyse geeignet zu sein. Diese gelten nicht nur für den rekursiven Abstieg, sondern für alle Syntaxanalyseverfahren.

- **Vollständigkeit**. Für jedes Nonterminalsymbol muss es eine Produktion geben. Die Grammatik

  ```
  X = a Y Z.
  Y = b b.
  ```

  ist zum Beispiel unvollständig, weil es keine Produktion für Z gibt.

- **Terminalisierbarkeit.** Jedes Nonterminalsymbol muss sich direkt oder indirekt in eine Folge von Terminalsymbolen ableiten lassen. Die Grammatik

```
X = a Y | c.
Y = b Y.
```

  ist zum Beispiel nicht terminalisierbar, weil sich Y nicht in eine Folge von Terminalsymbolen ableiten lässt, sondern zu einer endlosen Rekursion führt.

- **Zirkularitätsfreiheit.** Ein Nonterminalsymbol darf nicht in sich selbst ableitbar sein. Es darf also keine Ableitung der Form $X \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow X$ geben. Die Grammatik

```
X = a | Y.
Y = b | X.
```

  ist zum Beispiel zirkulär, weil es eine Ableitung $X \Rightarrow Y \Rightarrow X$ gibt.

3.4 Syntaxfehlerbehandlung

Wenn der Parser einen Syntaxfehler entdeckt, muss er diesen melden. Anschließend sollte er aber mit der Analyse fortsetzen, um eventuelle weitere Syntaxfehler zu entdecken. Allgemein sollte eine gute Syntaxfehlerbehandlung folgende Anforderungen erfüllen:

- Der Parser sollte pro Übersetzung möglichst viele Fehler finden.
- Der Parser darf auch bei groben Fehlern nicht abstürzen.
- Die Fehlerbehandlung sollte die fehlerfreie Analyse nicht bremsen.
- Die Fehlerbehandlung sollte den Parsercode nicht aufblähen.

Diese Anforderungen sind teilweise widersprüchlich. Je besser die Fehlerbehandlung sein soll, desto mehr Aufwand muss im Parser getrieben werden, was wiederum den Parsercode aufbläht und meist auch die Analyse eines fehlerfreien Programms bremst. Es gibt daher für den rekursiven Abstieg verschiedene Verfahren, die diese Anforderungen unterschiedlich gut erfüllen:

- Fehlerbehandlung im »Panic Mode«
- Fehlerbehandlung mit allgemeinen Fangsymbolen
- Fehlerbehandlung mit speziellen Fangsymbolen

Wir sehen uns nun diese Verfahren der Reihe nach an. Die Fehlerbehandlung im Panic Mode ist das einfachste Verfahren, das aber immer nur einen einzigen Fehler entdecken kann. Die anderen Verfahren können mehrere Fehler entdecken, sind aber unterschiedlich aufwendig und erfüllen daher die letzten beiden Anforderungen unterschiedlich gut.

3.4.1 Fehlerbehandlung im Panic Mode

Dies ist das Verfahren, das wir in den bisherigen Beispielen verwendet haben. Wenn der Parser einen Fehler entdeckt, meldet er ihn durch Aufruf der Methode error und bricht dann die Syntaxanalyse ab.

```
private static void error (String msg) {
    System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
    System.exit(1); // terminate parsing
}
```

Dieses Verfahren hat den Vorteil, dass es billig ist und den Parsercode nicht aufbläht, da er außer der Methode error keine weitere Fehlerbehandlung enthält. Wenn das Quellprogramm fehlerfrei ist, wird die Syntaxanalyse in keiner Weise gebremst. Allerdings wird nur der erste Fehler entdeckt. Nach der Korrektur des Fehlers muss das Quellprogramm erneut compiliert werden, um eventuelle weitere Fehler zu entdecken.

Für Sprachen, in denen die Programme kurz sind (z.B. für Kommandosprachen), ist diese Technik durchaus ausreichend. Für Sprachen wie Java oder auch MicroJava, in denen Programme länger sein können, ist dieses Verfahren allerdings nicht zufriedenstellend. Hier sollte eines der anderen Verfahren verwendet werden.

3.4.2 Fehlerbehandlung mit allgemeinen Fangsymbolen

Bei diesem Verfahren findet nach Entdeckung eines Syntaxfehlers ein *Wiederaufsatz* statt, d.h., der Parser und das fehlerhafte Programm müssen so synchronisiert werden, dass weiteranalysiert werden kann und eventuelle weitere Fehler entdeckt werden. Dazu wird eine Menge von *Fangsymbolen* berechnet, mit denen sich der Parser nach einem Syntaxfehler wieder »fangen« und sich somit mit dem Quellprogramm synchronisieren kann. Sehen wir uns dazu ein Beispiel an. Angenommen, wir haben folgende Grammatik

```
X = a Y e f.
Y = b c d.
```

und die Eingabe lautet

```
a b x y e f eof
```

Wie aus Abb. 3.9 ersichtlich, wird X betreten und a erkannt. Anschließend wird Y betreten und b erkannt, bevor festgestellt wird, dass das nächste Symbol x nicht zum erwarteten Symbol c passt. Der Fehler wird gemeldet. Anschließend werden alle Terminalsymbole gesammelt, die in der Grammatik nach der Fehlerstelle folgen können. Diese bilden die Menge der Fangsymbole, mit denen die Analyse an irgendeiner Stelle der Grammatik fortgesetzt werden kann.

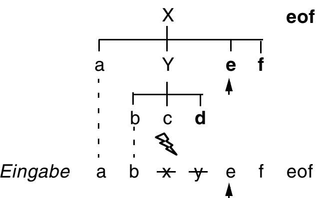

Abb. 3.9 *Sammeln von Fangsymbolen*

Sehen wir uns das Wiederaufsatzverfahren nun im Detail an. Es besteht aus folgenden drei Schritten:

1. Aus der Grammatik wird eine Menge von »Fangsymbolen« berechnet, mit denen die Analyse nach der Fehlerstelle fortgesetzt werden kann. Der Fehler wurde entdeckt, weil das erwartete Symbol c nicht zum Eingabesymbol x passte. Die Fangsymbole sind alle Nachfolger der Symbole, an denen der Parser gerade arbeitet. Der Parser arbeitet an c, dessen Nachfolger d ist. Ferner arbeitet er an Y mit den Nachfolgern e und f und an X mit dem Nachfolger eof. Die Fangsymbole sind also {d, e, f, eof}.
2. Es werden nun alle Symbole der Eingabe überlesen, die keine Fangsymbole sind. Es werden also die Symbole x und y überlesen, bevor mit e ein Fangsymbol auftritt. Der Eingabestrom wurde somit synchronisiert: Mit dem nächsten Symbol e kann irgendwo in der Grammatik fortgesetzt werden.
3. Im letzten Schritt wird der Parser an die Stelle gesteuert, an der das gefundene Fangsymbol e auftritt. Dazu läuft der Parser einfach weiter, bis er diese Stelle erreicht (siehe Pfeil in Abb. 3.9). Die Eingabe und die Grammatik sind dann synchronisiert, und die Analyse kann fortgesetzt werden.

Zur Berechnung der Fangsymbole werden jeder Parsermethode die Nachfolger des entsprechenden Nonterminalsymbols als Parameter mitgegeben, zum Beispiel:

```
private static void X (BitSet sux) {
    ...
}
```

Je nach Kontext, in dem dieses Nonterminalsymbol auftritt, kann die Nachfolgermenge sux unterschiedlich sein, wie in Abb. 3.10 zu sehen ist.

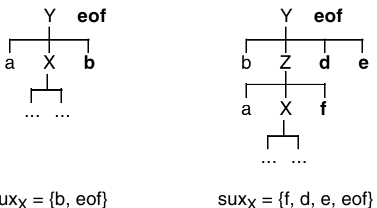

Abb. 3.10 *Nachfolger eines Nonterminalsymbols X in unterschiedlichem Kontext*

Daher muss man die kontextspezifische Nachfolgermenge eines Nonterminalsymbols seiner Parsermethode als Parameter mitgeben. Das Symbol eof ist jedenfalls immer in der Nachfolgermenge enthalten, da es ein Nachfolger des Startsymbols ist. Somit ist sichergestellt, dass spätestens mit eof synchronisiert werden kann.

Für die Fehlerbehandlung mit allgemeinen Fangsymbolen müssen die Muster zur Erkennung von Terminal- und Nonterminalsymbolen, von Alternativen, Optionen und Iterationen angepasst werden, was wir uns nun genauer ansehen.

Erkennung von Terminalsymbolen

Wenn auf der rechten Seite einer Produktion ein Terminalsymbol a auftritt, das von weiteren Elementen α_i (Symbolen, Alternativen, Optionen, Iterationen) gefolgt wird

$$X = \ldots \mathbf{a}\ \alpha_1\ \alpha_2 \ldots \alpha_n.$$

so lautet die Parseraktion (in Pseudocode):

$$\text{check}(\mathbf{a}, \text{First}(\alpha_1) \cup \text{First}(\alpha_2) \cup \ldots \cup \text{First}(\alpha_n) \cup \text{sux}_X);$$

Der Methode check wird also die Nachfolgermenge des Terminalsymbols als zweiter Parameter mitgegeben:

```
private static void check (int expected, BitSet sux) {
    if (sym == expected) scan();
    else error(name[expected] + " expected", sux); // pass sux to the error method
}
```

Der erste Teil der Nachfolgermenge ($\text{First}(\alpha_1) \cup \text{First}(\alpha_2) \cup \ldots \cup \text{First}(\alpha_n)$) kann statisch aus der Grammatik ermittelt werden. Die Nachfolger des Nonterminalsymbols (sux_X) müssen allerdings zur Laufzeit hinzuaddiert werden. Die Parsermethode für die Produktion

```
X = a b c.
```

lautet also (in Pseudocode):

```
private static void X (BitSet sux) {
    check(a, {b, c} ∪ sux);
    check(b, {c} ∪ sux);
    check(c, sux);
}
```

Die Mengen werden als BitSets verwaltet, die am Anfang des Parsers deklariert und initialisiert werden. Die Menge {b, c} wird zum Beispiel wie folgt erstellt:

```
BitSet fs1 = new BitSet();
...
fs1.add(b); fs1.add(c);
```

Der erste Aufruf von check in der Parsermethode von X lautet dann in Java:

```
check(a, ((BitSet)fs1.clone().or(sux));
```

Man beachte, dass eine Kopie von fs1 erstellt werden muss (um fs1 nicht zu zerstören), bevor sux hinzuaddiert werden kann. Das ist natürlich mühsam zu schreiben, bläht den Parsercode auf und bremst auch die Syntaxanalyse.

Erkennung von Nonterminalsymbolen

Die Erkennung von Nonterminalsymbolen verläuft ähnlich wie bei Terminalsymbolen. Wenn auf der rechten Seite einer Produktion ein Nonterminalsymbol Y auftritt (gefolgt von weiteren Elementen α_i)

$$X = \ldots Y\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n.$$

so wird die Parsermethode von Y aufgerufen, wobei wieder die Nachfolger als Parameter mitgegeben werden:

$$Y(First(\alpha_1) \cup First(\alpha_2) \cup \ldots \cup First(\alpha_n) \cup sux_X);$$

Auch hier kann wieder der erste Teil ($First(\alpha_1) \cup First(\alpha_2) \cup \ldots \cup First(\alpha_n)$) statisch aus der Grammatik berechnet werden, während sux_X zur Laufzeit hinzuaddiert werden muss. Die Produktion

```
X = a Y c.   // assume First(Y) = {b}
```

wird zum Beispiel durch folgende Parsermethode erkannt (in Pseudocode):

```
private static void X (BitSet sux) {
    check(a, {b, c) ∪ sux);
    Y({c} ∪ sux);
    check(c, sux);
}
```

Das Startsymbol MicroJava wird durch Aufruf von MicroJava({eof}) erkannt.

Überlesen fehlerhafter Symbole

Beim Auftreten eines Syntaxfehlers wird die Methode error aufgerufen, der neben einer Fehlermeldung auch die Menge der Fangsymbole mitgegeben wird:

```
private static void error (String msg, BitSet sux) {
    System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
    errors++;
    while (!sux.get(sym)) scan();  // while (sym ∉ sux) scan();
    // sym ∈ sux
}
```

Diese Methode überliest mittels scan() so lange Token aus dem Eingabestrom, bis ein Fangsymbol aus der Menge sux auftritt. Da nun mehrere Fehler erkannt werden können, müssen sie mithilfe einer Variablen errors gezählt werden, die am Anfang des Parsers wie folgt deklariert wird:

```
public static int errors = 0; // error counter
```

Diese Variable kann am Ende der Syntaxanalyse abgefragt werden, um auszugeben, wie viele Syntaxfehler erkannt wurden.

Nach der Synchronisation des Eingabestroms in der Methode error enthält das Vorgriffssymbol sym nun ein Fangsymbol, mit dem die Analyse fortgesetzt werden kann.

Synchronisation mit der Grammatik

Nachdem fehlerhafte Symbole überlesen wurden, muss der Parser schließlich an die Stelle der Grammatik gesteuert werden, an der mit dem erkannten Fangsymbol fortgesetzt werden kann. Dazu läuft der Parser einfach weiter (und produziert dabei eventuell Folgefehlermeldungen), bis er an die Stelle kommt, wo das erkannte Fangsymbol erwartet wird. Sehen wir uns dazu ein Beispiel an. Für die Produktion

```
X = a b c.
```

lautet die Parsermethode (in Pseudocode):

```
private static void X (BitSet sux) {
    check(a, {b, c} ∪ sux);
    check(b, {c} ∪ sux);
    check(c, sux);
}
```

Angenommen, die Eingabe lautet nicht a b c, sondern x y c. Der Parser versucht, durch Aufruf von check(a, {b, c} ∪ sux) ein a zu erkennen, aber das nächste Eingabesymbol ist x und nicht a, daher wird ein Fehler gemeldet. Die Fangsymbolmenge lautet {b, c} ∪ sux. Die Methode error überliest nun alle Symbole, die nicht in dieser Menge enthalten sind. x und y werden also überlesen, aber c ist ein Fangsymbol, daher stoppt das Überlesen hier (sym == c).

Der Parser läuft nun einfach weiter und kommt zu check(b, {c} ∪ sux). Die Erkennung von b schlägt ebenfalls fehl, weil sym das Symbol c enthält. Daher wird ein weiterer Fehler gemeldet. Die Fangsymbolmenge lautet nun {c} ∪ sux. Diesmal wird allerdings nichts mehr überlesen, weil das Vorgriffssymbol c in dieser Menge enthalten ist.

Wenn der Parser nun weiterläuft und zu check(c, sux) kommt, wird c erkannt, und die Synchronisation ist somit abgeschlossen. Der Parser kann nun fortfahren und versuchen, weitere Fehler zu entdecken.

Unterdrücken von Folgefehlermeldungen

Wie wir gesehen haben, läuft der Parser nach der Erkennung eines Syntaxfehlers einfach weiter, bis er an die Stelle kommt, an der das erkannte Fangsymbol erwartet wird. Dabei können Folgefehlermeldungen produziert werden, die natürlich unerwünscht sind, weil sie keine echten Fehler sind, sondern eine Folge des ursprünglichen Fehlers.

Folgefehlermeldungen können durch eine einfache Heuristik unterdrückt werden: Fehler werden nur dann gemeldet, wenn seit dem letzten Fehler mindestens drei Symbole korrekt erkannt wurden. Wir führen dazu eine globale Variable für die aktuelle Distanz zum letzten Fehler ein:

```
private static int errDist = 3;
```

Anfangs tun wir so, als ob der letzte Fehler bereits drei Symbole zurückliegt. Immer wenn ein Symbol durch Aufruf von scan korrekt erkannt wurde, wird die Fehlerdistanz um eins erhöht:

```
private static void scan() {
    ...
    errDist++;  // another token correctly parsed
}
```

In der Methode error melden wir einen Fehler nur dann, wenn errDist >= 3 ist. Nach dem Überlesen fehlerhafter Symbole setzen wir errDist wieder auf 0, und die Zählung beginnt von Neuem:

```
private static void error (String msg, BitSet sux) {
    if (errDist >= 3) {
        System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
        errors++;
    }
    while (!sux.get(sym)) scan();
    errDist = 0; // restart counting
}
```

Durch diese einfache Heuristik werden Folgefehlermeldungen unterdrückt. Wenn zwischen zwei Syntaxfehlern allerdings weniger als drei korrekt erkannte Symbole liegen, wird die Meldung des zweiten Fehlers »geschluckt«. Meist liegen Syntaxfehler aber nicht so dicht nacheinander, sodass die Heuristik in der Praxis gut funktioniert.

Erkennung von Alternativen

Auch das Muster zur Erkennung von Alternativen muss für die Fehlerbehandlung mit allgemeinen Fangsymbolen angepasst werden. Um eine Alternativenkette

$X = \alpha \mid \beta \mid \gamma .$

zu parsen, haben wir bisher folgendes Muster verwendet (in Pseudocode):

```
private static void X() {
    if (sym ∈ First(α)) ... parse α ...
    else if (sym ∈ First(β)) ... parse β ...
    else if (sym ∈ First(γ)) ... parse γ ...
    else error("invalid X");
}
```

Wenn sym allerdings aufgrund eines fehlerhaften Quellprogramms zu keiner der Alternativen passt, wird ein Fehler gemeldet, und wir haben dann keine Möglichkeit mehr, nach Überlesen fehlerhafter Symbole in eine der Alternativen einzusteigen. Daher ist es besser, bereits vor der Alternativenkette zu prüfen, ob sym zu einer der Alternativen passt. Wenn nicht, können wir den Fehler bereits hier melden, eventuelle fehlerhafte Symbole überlesen und haben dann immer noch die Möglichkeit, in eine der Alternativen einzusteigen. Das neue Muster lautet also:

```
private static void X (BitSet sux) {
    if (sym ∉ First(α) ∪ First(β) ∪ First(γ))
        error("invalid X", First(α) ∪ First(β) ∪ First(γ) ∪ sux);
    // sym matches one of the alternatives or sux
    if (sym ∈ First(α)) ... parse α ...
    else if (sym ∈ First(β)) ... parse β ...
    else if (sym ∈ First(γ)) ... parse γ ...
    // no error branch
}
```

Die Methode error überliest fehlerhafte Symbole, bis eines aus der angegebenen Menge auftritt. Dieses ist dann entweder ein gültiger Anfang einer der Alternativen oder ein gültiger Nachfolger von X. Die Menge First(α) ∪ First(β) ∪ First(γ) kann wieder statisch aus der Grammatik berechnet werden, während sux zur Laufzeit hinzuaddiert werden muss. Wie man sieht, brauchen wir auch keinen Fehlerzweig mehr, weil ein eventueller Fehler bereits vorher gemeldet wurde.

Erkennung von Optionen

Ähnlich müssen wir auch bei EBNF-Optionen bereits vor der Option prüfen, ob das Vorgriffssymbol zur Option oder zu ihren Nachfolgern passt und anderenfalls einen Fehler melden. Für die Produktion

```
X = [α] β.
```

lautet das neue Muster (in Pseudocode):

```
private static void X (BitSet sux) {
    if (sym ∉ First(α) ∪ First(β)) error("invalid X", First(α) ∪ First(β) ∪ sux);
    if (sym ∈ First(α)) ... parse α ...
    ... parse β ...
}
```

Erkennung von Iterationen

Bei EBNF-Iterationen verwenden wir zur Fehlerbehandlung ein völlig neues Muster mit einer Schleife, die auf jeden Fall betreten wird. In der Schleife prüfen wir, ob das Vorgriffssymbol zur Iteration oder zu ihren Nachfolgern passt. Ist das nicht der Fall, melden wir einen Fehler, bleiben aber in der Schleife, um nach Überlesen eventueller fehlerhaften Symbole mit dem nächsten Durchlauf der Iteration fortsetzen zu können. Für die Produktion

```
X = {α} β.
```

lautet das Muster (in Pseudocode):

```
private static void X (BitSet sux) {
    for (;;) {  // enter loop even if sym ∉ First(α)
        if (sym ∈ First(α)) ... parse α ...                  // correct case 1: process α
        else if (sym ∈ First(β) ∪ sux) break;                // correct case 2: leave loop and process β
        else error("invalid X", First(α) ∪ First(β) ∪ sux);  // error case: synchronize and stay in loop
    }
    ... parse β ...
}
```

Die Methode error meldet den Fehler und überliest dann so lange Symbole, bis ein terminaler Anfang von α auftritt (mit dem die Iteration fortgesetzt werden kann) oder ein terminaler Anfang von β (mit dem die Iteration verlassen wird) oder ein Symbol aus sux (mit dem die Iteration ebenfalls verlassen wird; in diesem Fall würden beim Parsen von β Folgefehler gemeldet, die aber unterdrückt werden).

Beispiel

Abschließend sehen wir uns nun ein Beispiel an, bei dem die Muster für alle EBNF-Konstrukte (Alternativen, Optionen und Iterationen) zum Einsatz kommen. Für die Grammatik

```
X = a Y | b {c d}.
Y = [b] d.
```

sehen die Parsermethoden mit Fehlerbehandlung wie folgt aus (für die Berechnung der Fangsymbolmengen wird aus Lesbarkeitsgründen wieder Pseudocode verwendet):

```
private static void Y (BitSet sux) {
    if (sym != b && sym != d) error("invalid Y", {b, d} ∪ sux);
    if (sym == b) scan();
    check(d, sux);
}
```

```
private static void X (BitSet sux) {
    If (sym != a && sym != b) error("invalid X", {a, b} ∪ sux);
    if (sym == a) {
        scan(); Y(sux);
    } else if (sym == b) {
        scan();
        for (;;) {
            if (sym == c) {
                scan();
                check(d, {c} ∪ sux); // c is also a successor of d
            } else if (sym ∈ sux) {
                break;
            } else {
                error("c expected", {c} ∪ sux);
            }
        }
    }
}
```

Zusammenfassung

Die Fehlerbehandlung mit allgemeinen Fangsymbolen ergibt einen guten und raschen Wiederaufsatz. Sie ist systematisch anwendbar, indem für die Erkennung von Terminalsymbolen, Nonterminalsymbolen, Alternativen, Optionen und Iterationen vertraute Muster verwendet werden.

Der Nachteil dieser Fehlerbehandlungstechnik ist allerdings, dass sie einigermaßen kompliziert ist, den Parsercode aufbläht und die Analyse (auch fehlerfreier) Programme bremst, da laufend Fangsymbolmengen berechnet und den Parsermethoden mitgegeben werden müssen.

3.4.3 Fehlerbehandlung mit speziellen Fangsymbolen

Bei dieser Fehlerbehandlungstechnik erfolgt der Wiederaufsatz nach Syntaxfehlern nur an besonders »sicheren« Stellen, d.h. an Stellen, an denen Schlüsselwörter erwartet werden, die an keiner anderen Stelle der Grammatik vorkommen. Solche Stellen sind zum Beispiel:

- der Beginn einer *Anweisung*, an dem Schlüsselwörter wie if oder while erwartet werden, oder
- der Beginn einer *Deklaration*, an dem Schlüsselwörter wie public, static oder void erwartet werden.

Wenn wir ein if sehen, wissen wir, dass wir am Beginn einer Anweisung sind, weil dieses Symbol sonst nirgends in der Grammatik vorkommt. Wenn wir ein void sehen, wissen wir, dass eine Deklaration beginnt.

Wir berechnen für jede dieser Stellen die Menge der dort erwarteten Symbole als Fangsymbolmengen. Allerdings kann am Beginn einer Anweisung oder einer Deklaration auch das Symbol ident vorkommen, das kein sicheres Fangsymbol ist, weil ident auch an vielen anderen Stellen der Grammatik vorkommen kann. Daher nehmen wir ident aus der Menge der Fangsymbole heraus.

An jeder dieser Synchronisationsstellen fügen wir in den Parser folgenden Code ein (in Pseudocode):

```
if (sym ∉ expectedSymbols) {
    error(...);  // no parameter for successors; no synchronization in error
    while (sym ∉ safeExpectedSymbols ∪ {eof}) scan();  // synchronization
    errDist = 0;
}
```

Wenn an der Synchronisationsstelle keines der dort erwarteten Symbole im Eingabestrom vorliegt, melden wir einen Fehler. Anschließend überlesen wir Symbole des Quellprogramms, bis ein »sicheres« Fangsymbol auftritt. Mit diesem kann dann die Syntaxanalyse fortgesetzt werden. Beim Überlesen stoppen wir auch mit eof, um eine Endlosschleife zu vermeiden.

Diese Technik hat den Vorteil, dass den Parsermethoden und der Methode error keine Fangsymbolmengen mitgegeben werden müssen. Die Fangsymbolmengen an den Synchronisationsstellen können statisch aus der Grammatik berechnet werden und müssen nicht zur Laufzeit gebildet werden, was den Parsercode kompakt hält und zur Effizienz des Parsers beiträgt. Nach Auftreten eines Syntaxfehlers läuft der Parser einfach weiter, bis er zur nächsten Synchronisationsstelle kommt, an der dann der Wiederaufsatz stattfindet.

Sehen wir uns nun zum Beispiel die Synchronisationsstelle am Beginn einer Anweisung an. Die dort erwarteten Symbole (ident, if, while, ...) werden am Beginn des Parsers in einer globalen Variablen firstStat gespeichert:

```
private static BitSet firstStat = new BitSet();
...
firstStat.set(ident);
firstStat.set(if_);
firstStat.set(while_);
...
```

Ferner bilden wir eine »sichere« Fangsymbolmenge syncStat, die identisch zu firstStat ist, aber kein ident enthält, da dies ein unsicheres Fangsymbol wäre; dafür fügen wir eof als Fangsymbol hinzu:

```
private static BitSet syncStat;
...
syncStat = (BitSet) firstStat.clone();
syncStat.clear(ident);
syncStat.add(eof);
```

Die Parsermethode für Statement sieht dann wie folgt aus:

```
private static void Statement() {
    if (! firstStat.get(sym)) { // synchronization
        error("invalid start of statement");
        while (! syncStat.get(sym)) scan();
        errDist = 0;
    }
    // parse Statement
    if (sym == if_) {
        scan();
        check(lpar); Condition(); check(rpar);
        Statement();
        if (sym == else_) { scan(); Statement(); }
    } else if (sym == while_) {
        ...
}
```

Wie man sieht, findet die Fehlerbehandlung samt Synchronisation lediglich am Beginn von Statement statt. Der Rest des Parsercodes bleibt davon unberührt. Es werden dort keine Fangsymbolmengen berechnet und auch nicht an die Parsermethoden oder an check als Parameter übergeben. Der Parsercode wird somit nicht aufgebläht und bleibt effizient.

Auch die Methode error bekommt keine Fangsymbolmenge mit und überliest auch keine fehlerhaften Symbole, da dies an den Synchronisationsstellen (z.B. am Beginn von Statement) geschieht. Die Heuristik mit der Fehlerdistanz wird wie oben beschrieben eingesetzt, um Folgefehlermeldungen zu unterdrücken.

```
private static void error (String msg) {
    if (errDist >= 3) {
        System.out.println("line " + la.line + ", col " + la.col + ": " + msg);
        errors++;
    }
    errDist = 0;
}
```

Sehen wir uns nun den Wiederaufsatz anhand der Analyse des folgenden fehlerhaften MicroJava-Programmstücks an:

```
if a > b , max = a;
```

Der Parser betritt Statement (siehe oben) mit dem Symbol if. Da if ein gültiger Statement-Anfang ist, wird kein Fehler gemeldet und if wird mit scan() erkannt. Beim Aufruf von check(lpar) wird ein Fehler gemeldet, da das nächste Symbol ein ident ist. Die Fehlermeldung lautet "(expected". Der Parser setzt nun einfach fort und ruft Condition() auf, wodurch das Programmstück a > b korrekt erkannt wird. Beim anschließenden Aufruf von check(rpar) wird wieder ein Fehler gemeldet (") expected"). Der Parser setzt wieder fort, ruft Statement() auf und kommt zur Synchronisationsstelle. Das Vorgriffssymbol comma ist kein gültiger Statement-Anfang, weshalb error

aufgerufen wird. Da aber der letzte Fehler weniger als drei Symbole zurückliegt, wird die Fehlermeldung unterdrückt (Folgefehler). Der Parser überliest nun Symbole, bis ein Fangsymbol aus syncStat auftritt, was erst beim Strichpunkt der Fall ist. Ein Strichpunkt ist ein gültiger Statement-Anfang (leere Anweisung), mit dem das Parsen von Statement fortgesetzt werden kann. Der Wiederaufsatz ist somit geglückt, und die fehlenden Klammern rund um a > b wurden als Fehler gemeldet. Das Programmstück , max = a wurde überlesen, und es wird mit dem Strichpunkt fortgesetzt.

Man sieht, dass bei der Fehlerbehandlung mit speziellen Fangsymbolen nur an weniger Stellen synchronisiert wird und unter Umständen Programmstücke vor dem Wiederaufsatz überlesen werden. Dafür ist diese Technik wesentlich einfacher als die Fehlerbehandlung mit allgemeinen Fangsymbolen und führt zu kompakterem und effizienterem Parsercode.

Synchronisation am Beginn einer Iteration

Die oben beschriebene Technik lässt sich allerdings noch verbessern. Wenn eine Synchronisationsstelle am Beginn einer Iteration liegt, kann es zu einem suboptimalen Wiederaufsatz kommen. Sehen wir uns als Beispiel die Produktion von Block an:

```
Block = "{" {Statement} "}".
```

Die Parsermethode dazu sieht wie folgt aus (in Pseudocode):

```
private static void Block() {
    check(lbrace);
    while (sym ∈ First(Statement)) {
        Statement();
    }
    check(rbrace);
}
```

Wenn nun aber nach lbrace kein gültiger Statement-Anfang kommt, wird die Schleife gar nicht betreten und die Synchronisationsstelle in Statement wird nicht erreicht. Dadurch wird der Wiederaufsatz behindert. Man kann dieses Problem beheben, indem man die Parsermethode wie folgt umschreibt:

```
private static void Block() {
    check(lbrace);
    while (sym ∉ {rbrace, eof}) {
        Statement();
    }
    check(rbrace);
}
```

Anstatt zu prüfen, ob sym ein terminaler Anfang von Statement ist, wird geprüft, ob es ein terminaler Nachfolger der Anweisungsfolge ist (oder eof, um eine Endlosschleife zu vermeiden). Dadurch wird die Schleife auch betreten, wenn sym kein terminaler Anfang von Statement ist, und die Synchronisationsstelle in Statement wird erreicht. Dort wird bei einem Fehler bis zum nächsten Statement-Anfang überlesen, und die Analyse kann damit fortfahren. Allerdings müssen wir in syncStat auch das Symbol rbrace aufnehmen, damit das Überlesen auch mit diesem Symbol stoppt. Es ist also ein gewisses »Tuning« der Fehlerbehandlung nötig.

Zusammenfassung

Die Fehlerbehandlung mit speziellen Fangsymbolen greift nur an wenigen Stellen in den Parsercode ein. Es wird nur an Stellen synchronisiert, an denen besonders »sichere« Terminalsymbole erwartet werden, die nirgendwo sonst in der Grammatik vorkommen. Der Rest des Parsers bleibt unverändert. Dadurch bleibt der Parsercode kompakt und effizient.

Allerdings ist diese Technik nicht ganz so systematisch wie die Fehlerbehandlung mit allgemeinen Fangsymbolen und erfordert gewisse Erfahrung und manchmal auch ein wenig »Tuning«, um die richtigen Fangsymbolmengen zu bilden.

Der Parser des MicroJava-Compilers verwendet aus den oben geschilderten Gründen die Fehlerbehandlung mit speziellen Fangsymbolen.

3.5 Übungsaufgaben

1. *Kellerautomat*. Zeichnen Sie den Kellerautomaten für die Grammatik

   ```
   E = x | E "+" x.
   ```

   Geben Sie wie im Beispiel auf Seite 43 an, wie der Eingabesatz x + x analysiert wird und welche Zustände bei jedem Analyseschritt im Keller stehen.

2. *Rekursiver Abstieg (1)*. Schreiben Sie eine Parsermethode für die Produktion

   ```
   Assignment = ident "=" Expr ";" .
   ```

   Verwenden Sie (hier und in den nachfolgenden Aufgaben) die auf Seite 35 deklarierten Tokencodes. Sie brauchen keine Fehlerbehandlung zu implementieren.

3. *Rekursiver Abstieg (2)*. Schreiben Sie Parsermethoden für folgende Produktionen:

   ```
   VarDecl = ("static" | "final") Type ident {"," ident} ";".
   Type    = ident.
   ```

4. *Rekursiver Abstieg (3)*. Schreiben Sie Parsermethoden für folgende Produktionen:

```
MethodCall  = ident "(" [ Parameters ] ")" ";".
Parameters  = Par {"," Par}.
Par         = ["out" | "ref"] ident.
```

5. *Rekursiver Abstieg (4)*. Schreiben Sie Parsermethoden für folgende Produktionen:

```
Signature = ["public" | "private"] ("void" | Type) ident "(" [ Param {"," Param} ] ")" ";".
Type      = ident ["[" "]"].
Param     = Type ident.
```

6. *Rekursiver Abstieg (5)*. Schreiben Sie Parsermethoden für folgende Produktionen:

```
EnumType  = Visibility "enum" ident "{" [IdentList ] "}".
Visibility = [ "public" | "private" ].
IdentList  = ident {"," ident}.
```

7. *LL(1)-Bedingung (1)*. Gegeben sei folgende Grammatik:

```
Decl      = Type Variable ";"
          | Decl Type Variable ";".
Type      = "int"
          | "string"
          | Type "*".
Variable  = ident
          | ident "[" "]".
```

a) Finden Sie alle LL(1)-Konflikte.
b) Formen Sie die Grammatik so um, dass sie LL(1) ist.

8. *LL(1)-Bedingung (2)*. Gegeben sei folgende Grammatik:

```
DeclStatList= DeclStat | DeclStatList DeclStat.
DeclStat     = Declaration | Statement.
Declaration  = Type ident ";".
Type         = "int" | ident.
Statement    = ident "=" Expr ";".
```

a) Finden Sie alle LL(1)-Konflikte.
b) Formen Sie die Grammatik so um, dass sie LL(1) ist.

9. *LL(1)-Bedingung (3)*. Gegeben sei folgende Grammatik (back, contents, cover, index, intro, page und preface sind Terminalsymbole):

```
Book     = cover Heading Body back.
Heading  = [preface] contents.
Body     = {Chapter} Chapter [index].
Chapter  = intro | Chapter page.
```

a) Finden Sie alle LL(1)-Konflikte.
b) Formen Sie die Grammatik so um, dass sie LL(1) ist.

10. *Syntaxfehlerbehandlung mit allgemeinen Fangsymbolen*. Schreiben Sie für folgende Grammatik Parsermethoden und implementieren Sie dabei eine Fehlerbehandlung mit allgemeinen Fangsymbolen (Abschnitt 3.4.2). Sie können die Mengenoperationen in Pseudocode formulieren.

```
A = { b C | d }.
C = [ c ] e.
```

A ist das Startsymbol, dessen Nachfolger eof ist.

11. *Syntaxfehlerbehandlung mit speziellen Fangsymbolen*. Schreiben Sie für folgende Grammatik Parsermethoden und implementieren Sie dabei eine Fehlerbehandlung mit speziellen Fangsymbolen (Abschnitt 3.4.3).

```
Program     = "program" ident {Declaration} "{" {Statement} "}".
Declaration = ("public" | "private") Type ident ";".
Type        = ident ["[" "]"].
```

Da public und private Schlüsselwörter sind, die nur am Anfang einer Deklaration vorkommen, ist der Deklarationsanfang ein guter Synchronisationspunkt. Die Parsermethode für Statement brauchen Sie nicht zu implementieren.

12. *Implementierung des Parsers*. Implementieren Sie den vollständigen Parser von MicroJava gemäß der Grammatik in Anhang A. Implementieren Sie eine Fehlerbehandlung mit speziellen Fangsymbolen und wählen Sie den Beginn von Deklarationen (in Program) sowie den Beginn von Statement als Synchronisationsstellen.

    Laden Sie anschließend von [Download] die Datei TestParser.zip herunter und testen Sie damit Ihren Parser. Es sollten dabei alle in der Beispieldatei BuggyParserInput.mj (in TestParser.zip) enthaltenen Syntaxfehler gemeldet werden.

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

Compilerbau – Grundlagen und Anwendungen

- Compilerbau praxisnah erklärt
- Systematische Einführung mit zahlreichen Übungsaufgaben
- Entwicklung eines Compilers für *MicroJava*
- Mit umfangreichem Zusatzmaterial zum Buch

Das Buch behandelt die praxisrelevanten Grundlagen des Compilerbaus, von der lexikalischen Analyse über die Syntaxanalyse bis zur Semantikverarbeitung und zur Codeerzeugung. Weitere Themen sind die systematische Beschreibung von Übersetzungsprozessen durch attributierte Grammatiken sowie der Einsatz eines Compilergenerators zur automatischen Erzeugung der Kernteile eines Compilers.

Als durchgängiges Beispiel wird ein Compiler für *MicroJava* – eine einfache Java-ähnliche Programmiersprache – entwickelt, der ausführbaren Bytecode – ähnlich dem Java-Bytecode – erzeugt.

Das Buch kann als Begleitliteratur zu einer einführenden Compilerbau-Vorlesung oder zum Selbststudium verwendet werden, um die Arbeitsweise von Compilern zu verstehen und Compiler oder compilerähnliche Werkzeuge zu implementieren, wie sie in der Praxis der Softwareentwicklung häufig vorkommen. Die im Buch behandelten Techniken können immer dann angewendet werden, wenn eine strukturierte Eingabe vorliegt, die durch eine Grammatik beschrieben werden kann.

Die einzelnen Kapitel enthalten über 70 Übungsaufgaben, mit denen das Gelernte vertieft werden kann.

Webseite zum Buch:
http://ssw.jku.at/CompilerBuch

- Musterlösungen zu den Übungsaufgaben
- Folien einer zweistündigen Vorlesung
- Quellcode des *MicroJava*-Compilers
- Weitere Materialien

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.

€ 29,90 (D)

Interesse am E-Book?
www.dpunkt.plus

Gedruckt in Deutschland
Papier aus nachhaltiger Waldwirtschaft
Mineralölfreie Druckfarben

ISBN 978-3-98889-008-5