Kapitel 1
Einleitung

Programs must be written for people to read, and only incidentally for machines to execute.
– Herold Abelson & Gerald Jay Sussmann

1.1 Compiler und Sprache

Was ist ein Compiler?

Das Wort »compilare« ist lateinisch und bedeutet so viel wie »zusammenstellen«, »aufhäufen«.

Sicherlich haben Sie schon mit Compilern gearbeitet: Der Java-Compiler `javac` übersetzt Java-Programme in Java-Bytecode, der auf der Java Virtual Machine (JVM) ausgeführt werden kann, der C-Compiler `gcc` erzeugt Maschinenprogramme, die direkt auf Ihrem Rechner ausgeführt werden können.

> **Definition 1.1: Compiler**
>
> Unter einem Compiler versteht man ein Programm, das einen in einer Programmiersprache geschriebenen Quelltext in ein ausführbares Programm übersetzt.
>
> Compilerbau ist die Informatik-Disziplin, die sich mit dem Entwurf und der Implementierung von Compilern befasst.

Der Compiler ist also das Programm, das Ihren Programmen dazu verhilft, ausgeführt zu werden, wie es in der zweiten Hälfte des Zitats von Abelson und Sussmann aus ihrem Standardwerk *Structure and Interpretation of Computer Programs* heißt.

Abbildung 1.1 veranschaulicht diese Definition und gibt Ihnen den ersten, noch sehr groben Überblick über den Compiler.

P und P' stehen dabei für Programme, die in den Programmiersprachen L und L' abgefasst sind. Der Compiler könnte zum Beispiel Ihr Programm *hello.java*, das in Java geschrieben ist, in *hello.class* übersetzen, das in der Sprache des Java-Bytecodes abgefasst ist.

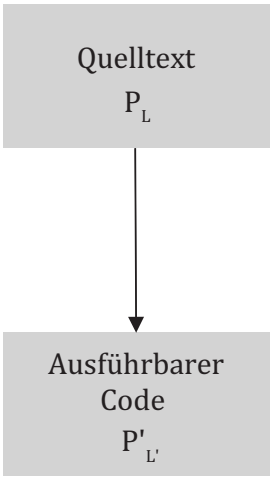

Abbildung 1.1 Funktion eines Compilers

Warum sollten Sie sich mit Compilerbau beschäftigen? Dazu fallen mir gleich mehrere gute Gründe ein:

- Wenn Sie verstehen, wie ein Compiler funktioniert, werden Sie auch ein besserer Programmierer oder eine bessere Programmiererin, weil Sie besser abschätzen können, warum die Programmiersprache, die Sie gerade benutzen, dieses oder jenes Feature hat oder nicht hat.
- Compilerbau ist unzertrennbar mit dem Design von Programmiersprachen verbunden – das Verständnis dafür, was alles getan werden muss, um einen Compiler für eine Sprache zu erstellen, hilft beim Erlernen jeder Programmiersprache.
- In dem Kapitel über die Konzepte von Programmiersprachen werden Sie – sofern Sie mindestens eine Programmiersprache kennen – viele Dinge wiedererkennen. Hier erfahren Sie, wie die Elemente der Programmiersprachen zusammenhängen.
- Wir werden einige Datenstrukturen und Muster benutzen, die zum Lösen vieler anderer Probleme ebenfalls nützlich sind.
- Compilerbau ist ein sehr anschauliches Beispiel dafür, wie Theorie (und davon werden Sie einige in diesem Buch lernen!) die praktische Arbeit enorm erleichtern kann: Lesen Sie die Geschichte des ersten Compilers auf der nächsten Seite!
- Gleichzeitig ist ein großer Teil der Theorie auch für viele andere Gebiete der Informatik relevant. Endliche Automaten beispielsweise werden Ihnen oft begegnen, weil sie ein häufig anwendbares Modell sind.
- Selbst einen Compiler zu schreiben ist ein spannendes Projekt!

Lassen Sie uns aber, bevor wir nun mit dem eigentlichen Inhalt beginnen, einen Blick in die Vergangenheit werfen:

Die ersten Compiler entstanden in den 1950er-Jahren, ohne dass eine Theorie über Compilerbau geschweige denn -werkzeuge existierten. John Backus beschreibt in [Backus J. , 1978], wie bis 1957 fast ausschließlich direkt in Maschinencode programmiert wurde, was aber dazu führte, dass die Programmierung und vor allem die Fehlersuche überproportional viel Zeit benötigten. Backus, damals angestellt bei IBM, startete daher Anfang 1954 das Projekt *FORTRAN*, eine Abkürzung für »FORmula TRANslator«, und die erste Version der gleichnamigen Sprache wurde im Frühjahr 1957 ausgeliefert. Die Zeitspanne von drei Jahren mag Ihnen sehr lang erscheinen, aber bedenken Sie, dass es keinerlei Vorerfahrungen und vor allem keine Algorithmen und Werkzeuge gab!

Der erste Compiler ist nach Meinung vieler der *A-0*-Compiler von Grace Hopper, die später die Programmiersprache COBOL entwickelte [Hopper, 1955].

In diesem Buch werden Sie viel Theorie lernen, vor allem über formale Sprachen, Automaten und Parser-Verfahren. Denken Sie dabei immer daran, dass Sie ohne diese Theorie *niemals* in der Lage wären, einen Compiler in wenigen Wochen zu entwickeln!

Nach FORTRAN und dann 1960 dem ersten *COBOL*-Compiler, ebenfalls entwickelt von Grace Hopper, hat vor allem die Entwicklung von *ALGOL* (»ALGOrithmic Language«) die Evolution von Programmiersprachen deutlich vorangetrieben. Von ALGOL stammt unter anderem *Pascal* ab [Wirth, The programming language pascal, 1971] und von diesem leiten sich direkt oder indirekt die meisten der heute populären Sprachen her.

Nachdem wir einen ersten, sehr kurzen Blick auf Compiler geworfen haben, müssen wir im zweiten Schritt den in der Definition vorkommenden Begriff »Programmiersprache« untersuchen.

Eine Programmiersprache ist eine Sprache, in der »Programme« geschrieben werden. Was aber ist dann eine *Sprache*?

Die Nutzung des Wortes »Sprache« ist hier nicht zufällig – vor allem in den Kapiteln 3 und 4 werden wir die Beziehungen zwischen natürlichen Sprachen und Sprache, wie man sie in der Informatik nutzt, tiefer beleuchten.

So viel sei vorweggenommen: Jede Sprache besteht aus Grundeinheiten, zum Beispiel den Buchstaben und Zeichen (Komma, Punkt, ...) in der deutschen Sprache. Aus diesen Zeichen bilden wir Wörter. Legen wir das lateinische *Alphabet* (plus im Deutschen die Umlaute) als Menge der zulässigen Buchstaben zugrunde, könnten wir unendlich viele *Wörter* bilden, von denen die meisten im Deutschen aber völlig sinnfrei wären. Für die deutsche Sprache enthält der Duden alle gebräuchlichen Wörter. Wir nehmen insbeson-

dere bei gesprochener Sprache eher die Wörter als Einheiten wahr als die Buchstaben, aus denen die Wörter bestehen.

Um dieses Denkmuster zu vervollständigen, fügen wir Wörter zu *Sätzen* zusammen. Dabei könnten wir beliebig Wörter aus dem Duden aneinanderreihen – in den seltensten Fällen ergibt das aber einen korrekten und sinnvollen Satz. Warum?

Es fehlt noch die Bauanleitung für Sätze!

Obwohl der Duden nur endlich viele »zugelassene« Wörter enthält, können Sie daraus unendlich viele korrekte deutsche Sätze formen. Der Mechanismus, der dies bewerkstelligt, sozusagen die Bauanleitung, ist die *Grammatik*.

Diese Bauanleitungen sind meistens relativ kompakt – selbst die für die meisten von uns komplizierte deutsche Grammatik wird im Duden auf nur ca. 70 Seiten beschrieben, während das Verzeichnis der Wörter 1100 Seiten umfasst.

Mit diesen drei Konzepten – »Zeichen«, »Wörter« und »Grammatik« – können wir nun also korrekte Sätze zusammenbauen.

Abbildung 1.2 Von Zeichen zu Wörtern und Sätzen

Das reicht aber leider noch nicht, wie beispielsweise der Satz »Fahrrad spielt flüssiges Buch« zeigt:

- Besteht der Satz aus lauter Buchstaben des Alphabetes? Ja!
- Besteht der Satz nur aus Wörtern, die im Duden stehen? Ja!
- Ist der Satz grammatikalisch korrekt? Ja!

Der Inhalt des Satzes ist aber völlig sinnfrei. Hm.

Neben den in Abbildung 1.2 gezeigten formalen Konzepten, die die *Syntax* beschreiben, gehört zu einer Sprache auch die Bedeutung, die *Semantik*. In jeder Sprache (sowohl in den natürlichen Sprachen als auch in den Programmiersprachen) gibt es neben den syntaktischen Regeln daher auch semantische Regeln, die die Bedeutung der Sätze festlegen.

Alle Begriffe, die wir bis hierhin anhand einer natürlichen Sprache eingeführt haben, finden sich auch so in Programmiersprachen: In Kapitel 3 werden wir uns ansehen, wie

wir – jetzt für eine Programmiersprache und nicht für eine natürliche Sprache – aus Zeichen Wörter bilden können.

In den folgenden Kapiteln 4 und 5 starten wir in die nächste Stufe: Wir erstellen eine Grammatik und bauen damit aus den Wörtern Sätze. Die Sätze sind in einer Programmiersprache verfasste Quelltexte, und wir können dann feststellen, ob die Programme syntaktisch korrekt sind.

Kapitel 6 ist überschrieben mit »Semantische Analyse«. Darunter versteht man im Compilerbau aber »nur« die Prüfung gewisser Regeln, die über die reine Syntax hinausgehen, zum Beispiel »Variablen müssen deklariert werden, bevor sie verwendet werden« oder »ganze Zahlen und Zeichenketten kann man nicht miteinander multiplizieren«. Mitnichten ist der Compiler aber in der Lage, den »Inhalt«, also die Bedeutung, eines Programms zu verstehen oder gar mit der Intention des Programmierers abzugleichen.

Eine formale Beschreibung der Semantik von Programmiersprachen geht deutlich über den Compilerbau hinaus und ist eine andere, mit dem Compilerbau verwandte Informatikdisziplin.

1.2 Aufbau dieses Buches

Wie ist dieses Buch aufgebaut?

Im nächsten Kapitel werden wir uns mit den verschiedenen Arten von Programmiersprachen, ihren Charakteristika und wichtigsten Konzepten befassen. Dieses Kapitel legt die für dieses Buch notwendigen Grundlagen, indem

1. die in Programmiersprachen üblichen Begriffe eingeführt und erklärt werden und
2. die auf uns als Compilerbauer zukommenden Aufgaben beim Bau eines Compilers für diese Programmiersprachen diskutiert werden.

Leider (oder zum Glück) gibt es unglaublich viele verschiedene Programmiersprachen mit noch viel mehr unterschiedlichen Sprachkonstrukten – ein Buch, das alle behandelt, wäre niemals komplett. Daher müssen wir uns auf die wesentlichen Konzepte beschränken. Wir versuchen Ihnen aber immer Hinweise auf weiterführende Literatur zu geben.

Gegen Ende des zweiten Kapitels führen wir die Programmiersprache SPL ein, anhand derer wir den Bau eines Compilers erklären. Wir werden Ihnen dabei immer wieder Teile des C- und des Java-Codes des Compilers zeigen, sodass Sie diesen erweitern können, um Ihren eigenen SPL-Compiler zu erstellen. Daher lohnt es sich, die Sprachbeschreibung genau zu lesen, sodass die Entscheidungen, wie wir den SPL-Compiler entwickeln,

einleuchtend sind. Der fertige Compiler wird ca. 2000 Zeilen Code umfassen. So viel Code werden wir hier nicht abdrucken, aber es wird genügend Code geben, damit Sie verstehen, wie man einen Compiler entwickelt.

SPL steht für *Simple Programming Language*, und mit dem Design von SPL wurden mehrere Ziele verfolgt:

- SPL ist mächtig genug, um damit etwas Sinnvolles tun zu können.
- SPL ist an existierende Sprachen angelehnt, zum Beispiel C, sodass das Erlernen der Sprache nur geringen Aufwand erfordert.
- SPL soll möglichst keine Verdopplungen enthalten, das heißt keine Sprachelemente, die identische Funktionalität haben. In modernen Programmiersprachen gibt es solche Situationen häufig, weil es die Sprache für den Benutzer angenehm handhabbar macht – denken Sie zum Beispiel an die verschiedenen Schleifen in Java: While, Do-While, For und Foreach. Im Endeffekt sind diese alle auf die While-Schleife zurückzuführen. Wir werden in den verschiedenen Kapiteln jeweils am Ende auch immer erläutern, wie man Elemente »größerer« Programmiersprachen als SPL kompilieren kann.
- Bei den Datenstrukturen gilt Ähnliches: Wesentlich ist der Unterschied zwischen primitiven und zusammengesetzten Datentypen – dazu führen wir in SPL Felder (*Arrays*) ein. Auch hier werden wir detailliert erklären, wie andere Datentypen – zum Beispiel Verbünde (*Records / Structs*) oder Verbünde mit Varianten (*Unions*) – übersetzt werden können. Die semantische Analyse ist für SPL natürlich deutlich einfacher als bei »echten« Programmiersprachen. Sie erfordert aber noch so viele Aktivitäten, die bei jedem Compiler anfallen, dass die Typanalyse wahrscheinlich in der praktischen Umsetzung am schwierigsten ist. Gleichzeitig können wir mit Feldern auch die Tätigkeiten zur Bereitstellung von Speicherplatz zur Laufzeit beschreiben.

Zu Beginn des dritten Kapitels greifen wir die Diskussion über Sprachen wieder auf, indem wir eine Beschreibungssprache für die Grundeinheiten von Programmiersprachen vorstellen, also das, was bei den natürlichen Sprachen die Buchstaben, Zeichen bzw. Wörter sind. Unser Compiler soll in der Lage sein, aus der Eingabe des Quelltextes eines SPL-Programms

- zu prüfen, dass diese nur die zugelassenen Einheiten wie Schlüsselwörter, Sonderzeichen, Zahlen und Namen enthält und
- diese als Datenstruktur zurückzuliefern.

Abbildung 1.3 zeigt als Beispiel links einen kleinen Ausschnitt aus einem Quelltext eines SPL-Programms und rechts die Liste der erkannten Grundeinheiten. Letztere nennen wir *Token*.

Wegen der Analogie zum Duden als Lexikon der zugelassenen Wörter nennt man diese Phase des Compilers *lexikalische Analyse*.

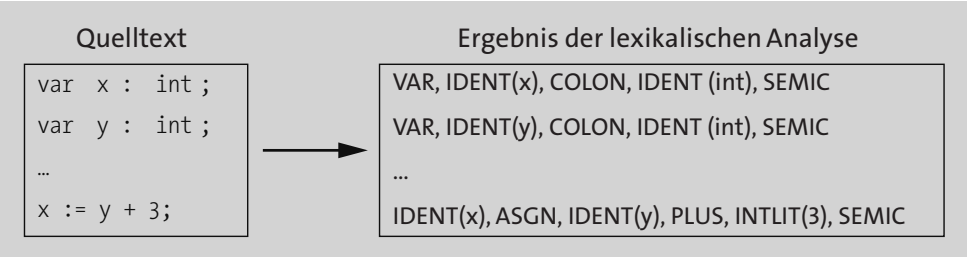

Abbildung 1.3 Lexikalische Analyse (Beispiel)

Das Programm, das den Quelltext einliest und diese Analyse durchführt, ist ein *Scanner*. Wir werden vorstellen, wie man einen Scanner mithilfe eines Werkzeugs – eines Scannergenerators – automatisch erzeugen kann, wie der Scanner funktioniert und wie der Scannergenerator aus einer Beschreibungssprache, den *regulären Ausdrücken*, den Scanner erstellt. Dazu benötigen wir etwas Theorie aus dem Gebiet der *endlichen Automaten*.

Wir werden sehen, wie man aus einem regulären Ausdruck einen endlichen Automaten generieren kann: Der Scannergenerator übersetzt die Liste der regulären Ausdrücke in einen endlichen Automaten. Dieser Scanner liest die Eingabe, also den Quelltext eines SPL-Programms, und erstellt daraus eine Liste von Tokens.

Es gibt eine ganze Reihe von Scannergeneratoren – wir werden *Flex* (für C) und *JFlex* (Java) detailliert erklären und Beispiele zeigen.

Im nächsten Abschnitt oder, wie man im Compilerbau richtiger sagt, in der nächsten *Phase* der Syntaxanalyse wenden wir eine Grammatik auf diese Liste der Tokens an, um zu beurteilen, ob das Eingabeprogramm den Regeln der Grammatik genügt, also syntaktisch richtig ist. Sollte dies der Fall sein, erwarten wir wieder eine Datenstruktur, die das Quellprogramm in seinen syntaktischen Einheiten darstellt. Diese Datenstruktur wird als *abstrakter Syntaxbaum* (engl. *Abstract Syntax Tree*, abgekürzt mit AST) bezeichnet und ist in Abbildung 1.4 für das oben angegebene Beispiel exemplarisch dargestellt. Das Programm, das diese beiden Aufgaben übernimmt, wird *Parser* genannt.

Der abstrakte Syntaxbaum ist die Grundlage für alle weiteren Phasen des Compilers.

Für die Implementierung der Syntaxanalyse gibt es mehrere Verfahren. Neben der direkten Programmierung eines Parsers werden wir auch hier die automatische Erstellung eines Parsers mit einem Werkzeug, einem sogenannten *Parsergenerator*, untersuchen.

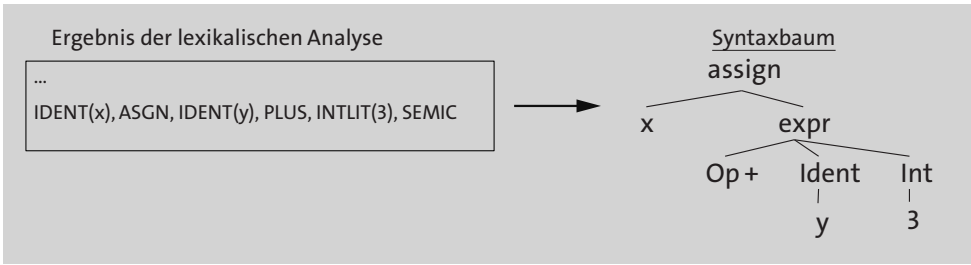

Abbildung 1.4 Ergebnis der Syntaxanalyse (Beispiel)

Die Eingabe für den Parsergenerator ist eine *Grammatik*, die bei Programmiersprachen analog zu den Grammatiken in natürlichen Sprachen zu betrachten ist. Mit einer Grammatik lassen sich alle Wörter einer Sprache erzeugen. Der Parsergenerator erzeugt aus dieser Grammatik einen *Kellerautomaten*, der einen Algorithmus darstellt, mit dem entschieden werden kann, ob der Quelltext syntaktisch richtig ist oder Fehler enthält. In diesem Kapitel steckt die meiste, bereits erwähnte Theorie. Die große Leistung der 1960er-Jahre war es, effiziente Verfahren für die Syntaxanalyse zu finden.

Im Gegensatz zu der lexikalischen Analyse mit dem endlichen Automaten gibt es für die Syntaxanalyse mehrere unterschiedliche Parserstrategien. Die einfachsten Parser sind sicherlich die *Top-Down-Parser*, bei denen man die Regeln der Grammatik der Programmiersprache so lange anwendet, bis man den Text des Eingabeprogramms komplett gelesen hat. Dazu werden wir zunächst ein Beispiel eines rekursiven Abstiegsparsers betrachten, bei dem man durch rekursive Aufrufe von Funktionen, die die Regeln der Grammatik darstellen, das Programm erkennt. Anstatt diese Parser selbst zu programmieren, kann man sie auch generieren lassen. Dazu werden wir uns *ANTLR* ansehen, einen sehr populären Parsergenerator, der eine Variante der sogenannten LL-Grammatiken verwendet.

Wenn es Top-Down-Parser gibt, dann sicherlich auch *Bottom-Up-Parser*, bei denen man umgekehrt vorgeht, also versucht, aus der Eingabe quasi rückwärts die Grammatikregeln anzuwenden. Es zeigt sich – das werden wir aber nur an Beispielen belegen und nicht formal beweisen –, dass dieses Verfahren, *LR-Parser* genannt, mächtiger ist als das LL-Verfahren.

Mit vielen Beispielen motivieren wir, wie man den LR-Parser erzeugt und wie er Schritt für Schritt eine Eingabe verarbeitet.

Insgesamt werden wir vier LR-Verfahren besprechen:

- LR(0), bei dem der Parser nicht in der Eingabe vorausschauen kann,
- SLR(1), ein besseres Verfahren, bei dem der Parser ein Zeichen vorausschauen kann,

- LR(1), eine sehr mächtige, aber kompliziertere Methode mit Vorausschau, und
- LALR(1), das bei den Parsergeneratoren gebräuchlichste Verfahren, das kompakte Parser erzeugt.

Kapitel 5 schließt an die eigentliche Syntaxanalyse an und sorgt für die Erstellung des abstrakten Syntaxbaums. Hierzu nutzen wir ein Verfahren, an die Regeln der Grammatik Aktionen anzuhängen, die den Baum berechnen. Dieses Verfahren wird als *attributierte Grammatik* bezeichnet.

Die Erkennung des Programms ist damit abgeschlossen und es schließt sich in Kapitel 6 die schon erwähnte semantische Analyse an, die wiederum in zwei Teilphasen geteilt ist. In beiden Phasen können wir den abstrakten Syntaxbaum mithilfe eines bekannten objekt-orientierten Design Patterns, des *Visitor-Pattern*, durchlaufen. Seit Java 21 steht mit den Type Patterns in switch-Anweisungen eine weitere Möglichkeit zur Verfügung, um diese Aufgabe zu erledigen.

In der *Namensanalyse* werden sämtliche im Quelltext vorkommenden Namen (Prozedurnamen, Typnamen, Parameternamen, Variablennamen, Klassennamen usw.) erfasst und in einer weiteren Datenstruktur, der *Symboltabelle*, abgelegt. Dabei speichern wir nicht nur die Namen ab, sondern auch weitere Informationen, zum Beispiel den Typ der Variablen.

Für alle weiteren Phasen benötigen wir sowohl den abstrakten Syntaxbaum als auch die Information in der Symboltabelle. Wir bezeichnen daher beide zusammen als Zwischendarstellung des Programms.

Im zweiten Teil der semantischen Analyse, der *Typanalyse*, müssen wir uns eingehend mit der Definition der Typregeln der Sprache auseinandersetzen. In Abschnitt 2.2.8 haben wir bereits die Konzepte hinter den Datentypen kennengelernt. In der Beschreibung der Sprache SPL in Abschnitt 2.3 haben wir konkret definiert, wie in SPL mit Datentypen umgegangen wird. Ein Beispiel für eine der Regeln, die in SPL gelten, ist beispielsweise: »Die Typen der Argumente eines Prozeduraufrufs müssen mit den Typen der formalen Parameter in der Deklaration der Prozedur übereinstimmen.« Zu den Aufgaben des Compilers gehört es, entweder diese Regeln direkt zu überprüfen oder, falls das noch nicht möglich ist, Code zu erzeugen, der diese Prüfung während der Ausführung des Programms durchführt.

Für die semantische Analyse gibt es zwar auch Werkzeuge, aber diese haben leider nicht die Abdeckung der Anforderungen oder die Verbreitung wie die Werkzeuge für die lexikalische und die Syntaxanalyse, sodass das Mittel der Wahl die Entwicklung von selbst geschriebenem Code ist.

Mit der semantischen Analyse ist das sogenannte *Frontend* des Compilers fertig. Dieser Begriff zielt darauf ab, dass die Aktivitäten des Compilers nur von der Programmier-

sprache abhängen, aber nicht von der Art der Zielmaschine, für die wir den ausführbaren Code erstellen.

Dementsprechend nennt man die folgenden Phasen *Backend*. Das Backend ist sehr wohl von der Zielmaschine abhängig. Dieser Art der Zweiteilung erlaubt es uns, für eine Programmiersprache einen Compiler für mehrere Zielmaschinen zu entwickeln, weil man das Frontend nur einmal implementieren und lediglich das Backend mehrfach erstellen muss.

In Analogie zur Analyse im Frontend sprechen wir beim Backend von der *Synthese*.

Die erste Phase des Backends ist die *Variablenallokation*, in der wir festlegen, wie die Datentypen der Programmiersprache auf der Zielmaschine, sozusagen in Bytes, abgebildet werden (siehe Kapitel 7). Auch hierfür gehen wir wieder durch den abstrakten Syntaxbaum und nutzen die Einträge in der Symboltabelle, um für jede Prozedur festzulegen, wie die Aufrufparameter der Prozedur und ihre lokalen Variablen im Speicher abgelegt werden sollen.

Hier werden wir – obwohl dies für SPL nicht relevant ist – einen Ausflug zu anderen Programmiersprachen machen, bei denen Daten dynamisch angelegt werden können. Dabei entdecken wir die Probleme, die bei unvorsichtiger Programmierung auftauchen können, und besprechen in dem Abschnitt über *Garbage Collection* kurz, wie zum Beispiel Java seinen Hauptspeicher immer wieder aufräumt.

Schlussendlich kommen wir in Kapitel 8 zur *Codegenerierung*, bei der alle bisher erfassten Informationen (abstrakter Syntaxbaum, Symboltabelle und Variablenallokation) verwendet werden, um Assembler-Code auszugeben, der in ausführbaren Maschinencode umwandelbar ist. Dazu stellen wir eine einfache RISC-Maschine (eine Hardware mit einfachem Befehlssatz) vor, um zu erläutern, wie man die verschiedenen Anweisungen von SPL (und anderer Programmiersprachen) übersetzen kann. Für jede Art von Anweisung kann man ein Schema angeben, wie die Übersetzung erfolgen kann.

Diese beiden Phasen des Backends sind in Abbildung 1.5 dargestellt als Fortsetzung der vorigen Abbildungen:

Die Aufteilung des Compilers in die verschiedenen Phasen ist für das Verständnis eines Compilers vorteilhaft, weil wir uns immer nur auf ein Problem konzentrieren müssen: Was wollen wir in der aktuellen Phase erreichen und wie können wir dieses Ziel erreichen?

Die Phasen sind eine inhaltliche Aufteilung der Aufgaben eines Compilers und erzeugen jeweils Zwischenergebnisse, wie in Abbildung 1.3, Abbildung 1.4 und Abbildung 1.5 gezeigt, die dann in den jeweils folgenden Phasen weiterverwendet werden.

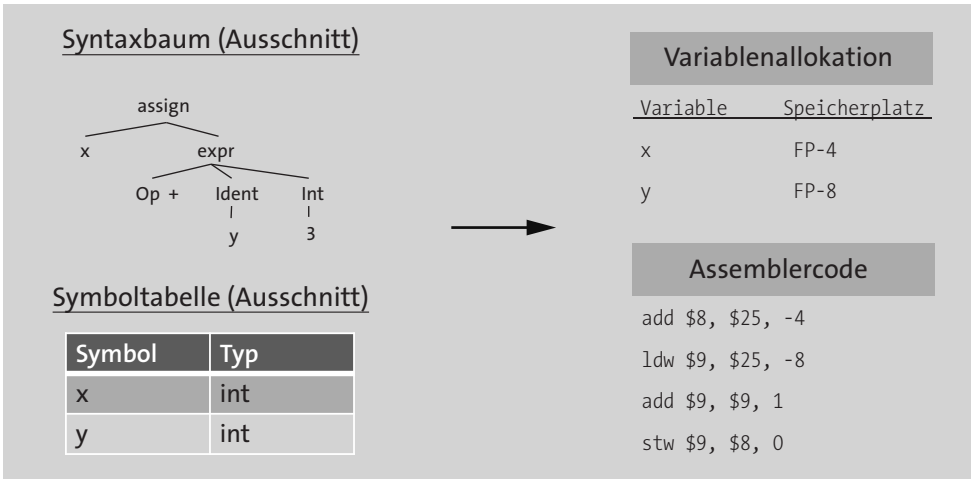

Abbildung 1.5 Backend des Compilers (Beispiel)

In der Implementierung werden Compiler oft so entworfen, dass mehrere Phasen zu einem *Durchgang* (engl. *pass*) zusammengefasst werden. In einem Durchgang durchläuft der Compiler das Quellprogramm beziehungsweise eine Darstellung des Quellprogramms (zum Beispiel den abstrakten Syntaxbaum) nur einmal. Oft werden beispielsweise die lexikalische und die Syntaxanalyse in einem Durchgang durchgeführt.

Die Phasen des Compilers werden wir in diesem Buch jeweils in einem eigenen Kapitel besprechen, sodass Sie immer gut nachvollziehen können, wo wir gerade stehen.

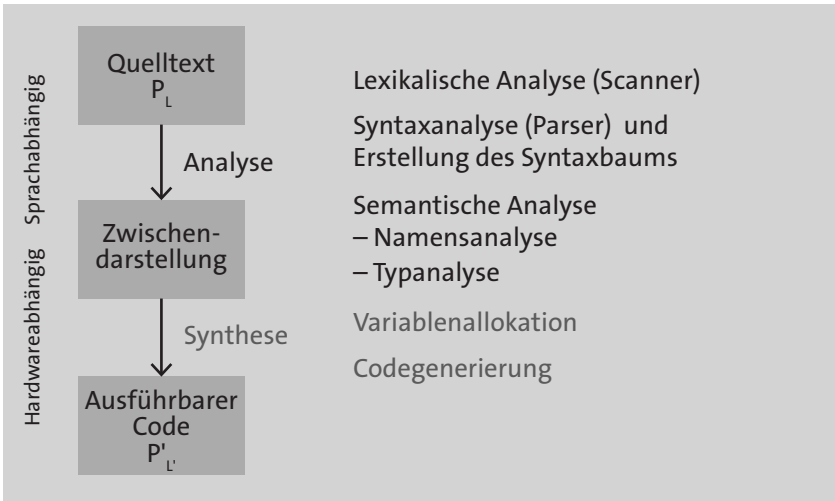

Abbildung 1.6 Phasenmodell eines Compilers

Wir orientieren uns dabei an der »klassischen Darstellung« der Phasen – siehe zum Beispiel [Aho, Sethi & Ullman, 1986] – und Abbildung 1.6 fasst die Phasen nochmals zusammen. Wir werden diese Abbildung immer wieder nutzen, um zu zeigen, wo wir uns gerade befinden.

Ein Kapitel bleibt noch übrig: Kapitel 9 beschäftigt sich mit der Optimierung des generierten Codes. Dabei ist das Wort »Optimierung« eher im übertragenen Sinne zu verstehen, weil es eigentlich darum geht, den erzeugten Code zu verbessern. Diese Verbesserung kann eine Verringerung der Laufzeit oder des Platzbedarfes oder ein geringerer Speicherverbrauch des erzeugten Programms sein.

Code-Optimierung ist ein Thema, mit dem sich ein zweiter Band füllen ließe, da es sehr viele Techniken und Ansätze dazu gibt. Die Intention hier ist, Ihnen einen Überblick zu verschaffen und einzelne Verfahren beispielhaft tiefergehender zu erklären.

Optimierungen waren schon im allerersten FORTRAN-Compiler enthalten, weil John Backus annahm, dass die bis dahin unbekannten höheren Programmiersprachen nur akzeptiert werden würden, wenn die kompilierten Programme ähnlich schnell liefen wie die von Hand geschriebenen Maschinenprogramme. Tatsächlich wurde das Ziel erreicht – umso erstaunlicher, wenn man bedenkt, dass die ersten Veröffentlichungen zur Optimierung erst einige Jahre später erschienen sind.

Um Code-Optimierungen durchzuführen, hat es sich als vorteilhaft herausgestellt, nicht auf dem abstrakten Syntaxbaum aufzusetzen, weil dieser weit entfernt von Maschinensprache ist. Gleichzeitig ist Maschinensprache (oder Assembler-Code) natürlich, wie oben erläutert, spezifisch pro Zielmaschine und sehr kleinteilig, sodass wir auf einer *Zwischensprache* aufsetzen werden.

Basierend auf dieser Darstellung können wir eine erste Aussage über den Ablauf des Programms treffen, indem wir es in die Einheiten, *Basisblöcke* genannt, zerlegen.

Innerhalb dieser Einheiten können wir *lokal optimieren*, zum Beispiel indem gemeinsame Teilausdrücke, also Berechnungen, die mehrfach vorkommen, nur einmal berechnet und zwischengespeichert werden.

Wir werden feststellen, dass man einige von diesen Optimierungen auch über mehrere der Einheiten hinweg anwenden kann, wenn man gewisse Informationen darüber besitzt, was das Programm »dazwischen« tut.

Diese Informationen gewinnen wir aus der *Datenflussanalyse*, in der wir nachvollziehen, wie das Programm die Werte seiner Variablen verändert.

Die Arbeiten hierzu von Allen und Kildall stammen aus der Zeit Ende der 1960er- bzw. Anfang der 1970er-Jahre und ihre Verfahren sind heute in gängigen Compilern eingebaut.

Bei der Datenflussanalyse gibt es eine gemeinsame Grundlage und Formeln, mit denen man Mengen von Informationen zur jeweils aktuellen Fragestellung der Optimierungstechnik für den Beginn eines Basisblocks und für das Ende berechnet. Die Einzelheiten der Berechnungen sind jedoch pro Fragestellung leicht unterschiedlich. Hier werden wir ein Beispiel für das Problem *der erreichenden Definitionen* durchrechnen.

Die meiste Zeit »verbringt« ein Programm üblicherweise in Schleifen (oder rekursiven Funktionen), und daher sind auch kleinste Optimierungen in Schleifen ein großer Hebel, um Programme substanziell zu beschleunigen.

Die klassischen Verfahren sind hier das Aufrollen von Schleifen bei bekannten Ober- und Untergrenzen, das Zusammenfügen von gleichartigen Schleifen, das Herauslösen von Code (zum Beispiel zur Berechnung von Ausdrücken), der nicht von den Schleifenvariablen abhängt, und die Transformation von Induktionsvariablen.

Es verbleiben noch einige, trotzdem sinnvolle Optimierungen, die sich zum einen mit Prozeduren und Prozeduraufrufen beschäftigen, wie die Ersetzung von bestimmten Rekursionen durch Schleifen (*Eliminierung von Endrekursion*), das Ersetzen des Prozeduraufrufs durch den Schleifenrumpf selbst (*Inlining*) und die Vereinfachung des Codes für Prozeduren, die selbst keine weiteren Prozeduraufrufe enthalten (*Leaf Procedures*). Letztere lässt sich durch wenig Code direkt in Ihren Compiler einbauen!

Als letzte Optimierung werden wir betrachten, wie man durch geschicktes »Abzählen« der abstrakten Syntaxbäume (mithilfe der sogenannten *Ershov-Zahl*) für Ausdrücke mit möglichst wenigen Prozessor-Registern auskommt. (Prozessor-Register sind spezielle, sehr schnelle Speicherplätze.) Da das Programm durch die Verwendung der Register deutlich schneller wird, die Anzahl der Register aber meist klein ist, kann man mit dieser Optimierung noch einmal Zeit einsparen. Auch diese Optimierung kann man mit wenig Code in den SPL-Compiler einbauen.

Kapitel 5
Abstrakter Syntaxbaum

Je abstrakter die Kunst wird, desto mehr wird sie Kunst.
– Robert Musil

Ist ein abstrakter Baum Kunst? Was ist denn ein abstrakter Syntaxbaum überhaupt?

5.1 Einleitung

Im vorigen Kapitel haben wir erläutert, wie wir mit den verschiedenen Parser-Verfahren entscheiden können, ob das Eingabewort zur Sprache gehört oder nicht. Reicht uns das für die restlichen Phasen des Compilers? Wie Sie vermutlich leicht erraten können, lautet die Antwort: »Nein!«

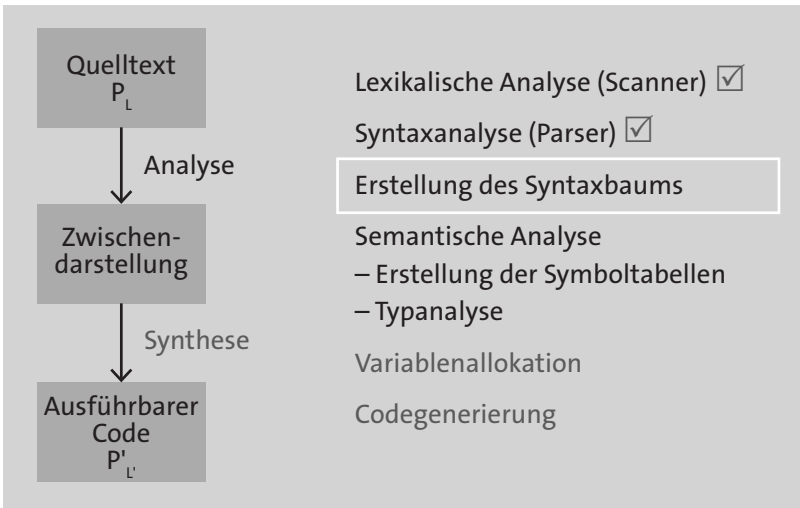

Abbildung 5.1 Überblick über die Phasen des Compilers

In dieser Phase bauen wir auf der Syntaxanalyse auf und erzeugen eine Darstellung des Syntaxbaums, die alle für die folgenden Phasen notwendigen Informationen enthält.

Das Verfahren, die Analyse und die Synthese durch die Syntax der Programmiersprache zu steuern, heißt *syntaxgesteuerte Übersetzung* – alle Berechnungen, die wir dafür brauchen, hängen wir an die Nichtterminalsymbole der kontextfreien Grammatik der Programmiersprache.

Welche Informationen benötigen denn die späteren Phasen? Alle? Sehen wir uns zu einem SPL-Prozedurrumpf mit zwei Zuweisungen den Syntaxbaum an:

```
{
  x:=0;
  y:=x+1;
}
```

Listing 5.1 SPL-Fragment

Der Syntaxbaum könnte – abhängig natürlich von der von Ihnen implementierten Grammatik – wie folgt aussehen:

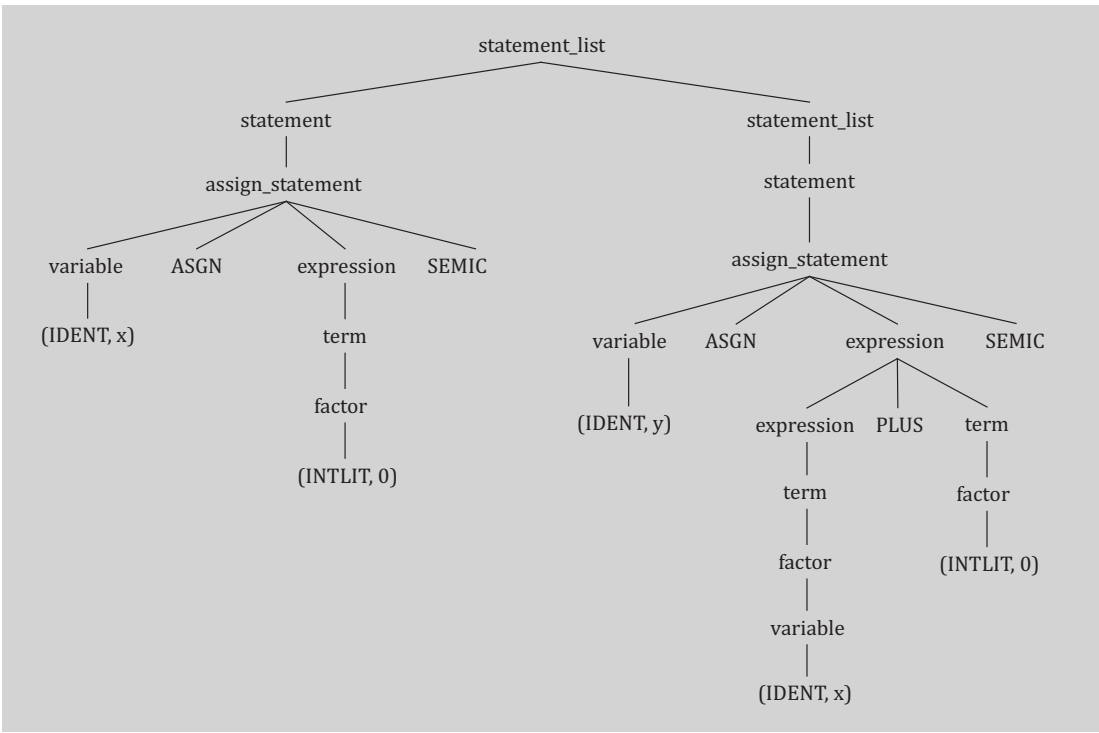

Abbildung 5.2 Syntaxbaum zu einem SPL-Fragment

Wie Sie sehen, ist der Syntaxbaum für so ein kleines Stück Code sehr groß. Wie erkennen aber auch, dass zum Beispiel in dem linken Teilbaum unterhalb des Knotens *assign_statement* die beiden Blätter *ASGN* und *SEMIC* redundant sind: Der Parser hat aufgrund des Tokens *ASGN* erkannt, dass es sich um ein *assign_statement* handelt; und aufgrund des Tokens *SEMIC* wurde das Ende der Anwendung korrekt erkannt. Warum bewahren wir diese beide Tokens auf? Sie sind nur für die Syntax notwendig – für die weiteren Phasen sind sie ohne Belang!

Ebenso ist die 1:1-Ableitung zum Beispiel im rechten Teilbaum unterhalb von *expression* unnötig – Präzedenzen und Assoziativitäten sollte der Parser richtig erkennen und abbilden; für die späteren Phasen sind die Zwischenebenen wie *term* und *factor* irrelevant.

Zusammengefasst: Der Syntaxbaum enthält viel »syntaktischen Zierrat«, der nach der Syntaxanalyse nicht mehr benötigt wird. Aus diesem Grund reden wir auch von einem *abstrakten Syntaxbaum* (*Abstract Syntax Tree*, *AST*). Wir lassen die Terminal- und Nichtterminalsymbole, die wir nicht mehr brauchen, einfach weg!

Donald Knuth ist die Erkenntnis zu verdanken, dass die Erzeugung eines abstrakten Syntaxbaums direkt während des Parsens erfolgen kann. In [Knuth D. E., 1968] führt er den Begriff der *attributierten Grammatik* ein. Die Nutzung von attributierten Grammatiken ist ein Standardverfahren im Compilerbau.

Das Ziel dieser Phase ist es also, während des Parsens einen abstrakten Syntaxbaum zu erzeugen, auf dem wir in den folgenden Kapiteln weiterarbeiten. Der Parser liefert dann also nicht nur die Aussage »Quelltext syntaktisch richtig/falsch«, sondern gibt auch den kompletten abstrakten Syntaxbaum zurück.

5.2 Attributierte Grammatiken

Bevor wir erklären, wie der Parser den abstrakten Syntaxbaum erzeugt, benötigen wir noch etwas Theorie.

Betrachten wir nochmals die Grammatik für arithmetische Ausdrücke, die wir im vorangegangenen Kapitel 4 verwendet haben.

Beispiel 5.1

Expr → *Expr* + *Term* | *Term*

Term → *Term* * *Factor* | *Factor*

Factor → *(Expr)* | *IntLiteral*

Der Syntaxbaum für den Ausdruck 2 + 3 × 4 sieht wie folgt aus:

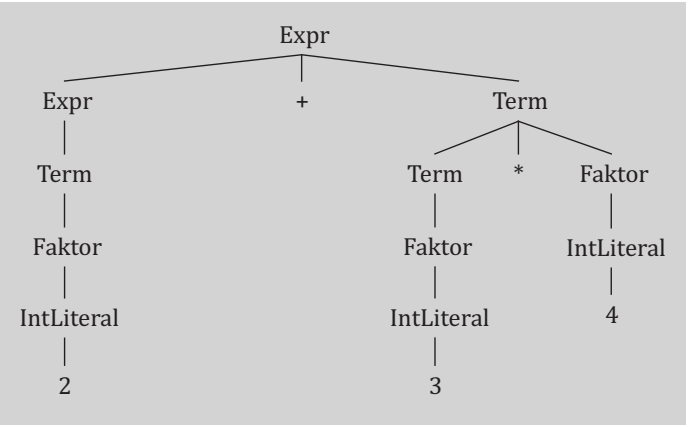

Abbildung 5.3 Syntaxbaum für den Ausdruck »2+3*4«

Schon in der Einleitung hatten wir festgestellt, dass die Unterscheidung zwischen Ausdruck (*Expr)*, *Term* und *Faktor* im Parser notwendig ist, um die Assoziativitäten und Präzedenzen richtig abbilden zu können. Da dies aber jetzt durch die Baumstruktur gegeben ist, können wir diese drei Ebenen durch den einheitlichen Begriff *Expr* ersetzen:

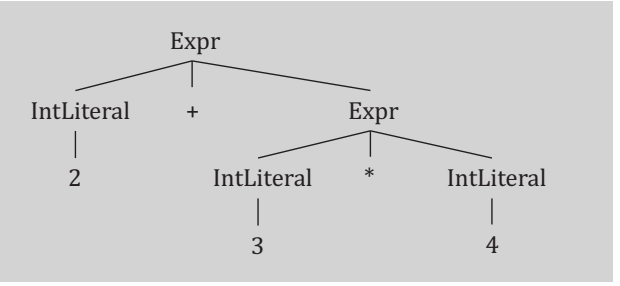

Abbildung 5.4 AST für den Ausdruck »2+3*4«

Man könnte nun diesen Baum benutzen, um den Wert des Ausdrucks zu berechnen, indem man den Wert als Attribut an jeden Nichtterminalknoten einfach dazuschreibt (siehe Abbildung 5.5).

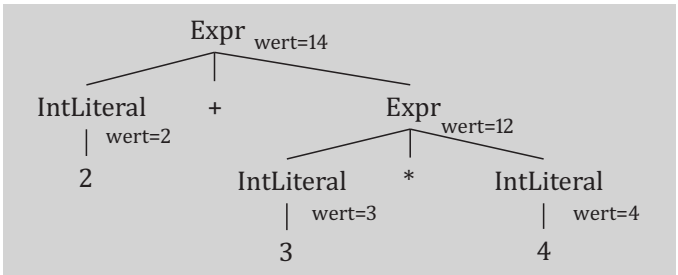

Abbildung 5.5 Attributierter AST

Wie haben wir die Werte ausgerechnet? Nun, wenn wir an den Blättern beginnen, ist klar, dass die Werte der *IntLiterale* »nach oben« gegeben werden müssen. Die *Expr*-Knoten berechnen nun aus den Werten ihrer Nachfolger ihren eigenen Wert.

Eine Information (in dem Beispiel oben: *wert*), die wir an die Knoten anhängen, nennen wir *Attribute*. Jeder Knoten kann mehrere Attribute besitzen. Die Idee der attributierten Grammatiken ist, die Menge der Attribute an die Terminale und Nichtterminale zu hängen und die Regeln für die Berechnung der Attribute zusammen mit den entsprechenden Produktionen der Grammatik anzugeben.

In dem Beispiel oben werden die Werte von unten nach oben berechnet – wir sprechen dann von *synthetisierten Attributen*. Die Information wird synthetisiert, also erzeugt, aus den Attributen der Nachfolger des Knotens im Baum.

Umgekehrt können auch Informationen von oben nach unten weitergeleitet werden – beispielsweise um Kontextinformationen weiterzugeben. In diesem Fall spricht man von *ererbten Attributen*. Attribute der Nachfolger des Knotens »erben« also Attribute des Knotens selbst.

Donald Knuth beschreibt in [Knuth D. E., 1990] sehr detailliert, dass die Idee zu ererbten Attributen eigentlich von Peter Wegner stammt, der Knuth fragte: »Why can't attributes be defined from the top down as well as from the bottom up?« Knuth schreibt: »A shocking idea!« [Knuth D. E., 1990]. Offensichtlich war die Idee trotzdem gut ...

Definition 5.1: Attributierte Grammatik

Eine *attributierte Grammatik* [Knuth D. E., 1968] ist eine erweiterte Grammatik *G = (N, T, P, S)* mit folgenden Erweiterungen:

1. Jedem Symbol $\alpha \in N \cup T$ ist eine endliche Menge von Attributen $A(\alpha)$ zugeordnet. Wir unterscheiden zwischen synthetisierten Attributen $A_s(\alpha)$ und ererbten Attributen $A_i(\alpha)$, die disjunkt sind. Weiterhin gilt: $A_i(S) = \emptyset$ und $A_i(t) = \emptyset$ für jedes Terminalsymbol t.

2. Zu jeder Produktion $X \to \beta_1 \ldots \beta_m$ $(m \geq 0)$ werden die Berechnungsvorschriften für die
   - ... synthetisierten Attribute aus $A_s(X)$ angegeben als Regeln $y_j := f(a_1, \ldots, a_k)$, wobei $0 \leq k \leq m$ und jedes a_i ein Attribut eines der Terminal- oder Nichtterminalsymbole $\beta_1, \ldots, \beta_m$ ist und $\{y_1, \ldots, y_r\} = A_s(X)$.
     Das heißt, jedem synthetisierten Attribut des Nichtterminalsymbols auf der linken Seite wird eine Berechnungsvorschrift als Funktion *f* auf einer Teilmenge der Attribute der Terminal- oder Nichtterminalsymbole auf der rechten Seite der Produktion zugeordnet.
   - ... ererbten Attribute angegeben als Regeln $z_j = g(b_1, \ldots, b_p)$, wobei die z_j Attribute aus $A_i(\beta_1) \cup \ldots \cup A_i(\beta_m)$ und die b_i Attribute aus $A(X)$ sind.
     Die Werte der ererbten Attribute der Nichtterminalsymbole auf der rechten Seite der Regel werden also berechnet als Funktion der Attribute des Nichtterminals auf der linken Seite der Produktion.

Eine attributierte Grammatik ist also eine Grammatik, bei der jedes Terminal- oder Nichtterminalsymbol Attribute haben kann. Diese Attribute sind entweder synthetisiert oder ererbt. Dabei fordern wir, dass das Startsymbol (das ja bei einer erweiterten Grammatik nur auf der linken Seite vorkommen kann) keine Attribute erbt (von wem auch?) und dass Terminalsymbole keine synthetisierten Attribute besitzen. Auch die letzte Forderung ist einleuchtend, weil die Terminalsymbole keine Nachfolger im Baum haben. Die Attributwerte der Terminalsymbole werden von Scanner bereitgestellt – so wie wir dies in Abschnitt 3.5 bereits gesehen haben.

Die Berechnungsvorschriften werden an die Produktionen angehängt. Dabei werden die Werte der synthetisierten Attribute des Nichtterminalsymbols auf der linken Seite der Produktion aus den Werten der Attribute der Symbole der rechten Seite berechnet.

Bei ererbten Attributen ist es genau umgekehrt: Die Werte der ererbten Attribute der Symbole auf der rechten Seite werden aus den Attributen des Nichtterminals auf der linken Seite berechnet.

Das Attribut *a* eines Symbols *X* wird als *X.a* geschrieben.

Natürlich könnte es vorkommen, dass ein Attribut *v* von einem anderen Attribut *w* abhängt. In diesem Fall muss *w* natürlich vor *v* berechnet werden. Es stellt sich dann die Frage, in welcher Reihenfolge die Knoten des Syntaxbaums (und seine Berechnungsregeln) ausgewertet müssen, sodass die Abhängigkeiten berücksichtigt werden. Eine Reihenfolge, die dies leistet, heißt *topologische Sortierung*. Sollte eine wechselsei-

tige Abhängigkeit bestehen (hängt zum Beispiel *v* von *w* ab und umgekehrt), dann gibt es natürlich keine solche Reihenfolge.

Die Parsergeneratoren machen jedoch keine Abhängigkeitsanalyse, sondern schränken die Verwendung der Attribute ein, wie wir im nächsten Abschnitt sehen werden.

Beispiel 5.2

Erweitern wir die Grammatik aus dem vorangegangenen Beispiel 5.1 um ein neues Startsymbol:

Ausdruck → *Expr*
Expr → *Expr* + *Term* | *Term*
Term → *Term* * *Factor* | *Factor*
Factor → *(Expr)* | *IntLiteral*

Um das Ergebnis der Auswertung des Ausdrucks zu berechnen, erstellen wir für jedes Nichtterminal ein Attribut *wert:*

$A_s(Ausdruck) = A_s(Expr) = A_s(Term) = A_s(Factor) = A_s(IntLiteral) = \{wert\}$

Ererbte Attribute werden nicht benötigt.

In den Produktionen treten gleiche Symbole wie *Expr* und *Term* teilweise doppelt auf. Zur Unterscheidung nummerieren wir in diesen Fällen diese Symbole von links nach rechts durch. Die Produktionen mit ihren Berechnungsregeln sind dann:

Ausdruck → *Expr* { *Ausdruck.wert* = *Expr.wert* }
Expr → *Expr* + *Term* { $Expr_1.wert = Expr_2.wert + Term.wert$ }
Expr → *Term* { *Expr.wert* = *Term.wert* }
Term → *Term* * *Factor* { $Term_1.wert = Term_2.wert * Factor.wert$ }
Term → *Factor* { *Term.wert* = *Factor.wert* }
Factor → *(* *Expr* *)* { *Factor.wert* = *Expr.wert* }

IntLiteral erhält den Wert durch die lexikalische Analyse.

Lassen Sie uns diese Erkenntnisse nun mit den Ausführungen zu abstrakten Syntaxbäumen in Abschnitt 5.1 zusammenführen.

Wir gehen aus von einer kontextfreien Grammatik für unsere Programmiersprache SPL und müssen beim Parsen eines Eingabeprogramms nicht nur entscheiden, ob dieses zur Sprache gehört (siehe Kapitel 4), sondern auch den abstrakten Syntaxbaum erstellen.

Dafür sind die attributierten Grammatiken bestens geeignet: Die Nichtterminalsymbole erhalten ein synthetisiertes Attribut, das dem abstrakten Syntaxbaum des entsprechenden Teilbaums entspricht. Der Startknoten enthält dann schlussendlich den gesam-

ten abstrakten Syntaxbaum. Die Berechnungsfunktionen sind die »Konstruktoren« der Knoten des abstrakten Syntaxbaums.

Betrachten wir dazu zunächst noch mal das Beispiel in Abbildung 5.2 und daraus den linken Teilbaum, der der Zuweisung `x:=0` entspricht:

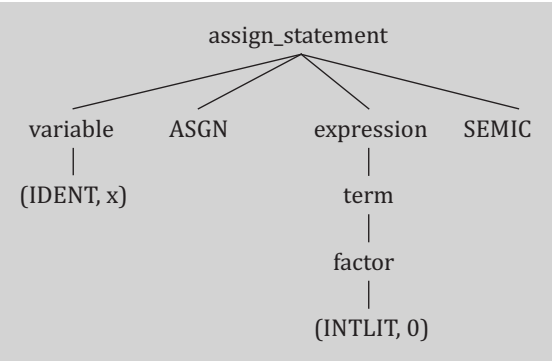

Abbildung 5.6 Teilbaum des Syntaxbaums

Wie schon besprochen, sind die Terminalzeichen ASGN und SEMIC nur syntaktisch notwendig, übrig bleiben nur die Variable und der Ausdruck. Somit lautet die attributierte Produktion:

assign_statement → variable ASGN expression SEMIC

```
{ assign_statement.ast = "Assign_statement(" +
                         variable.ast +
                         "," +
                         expression.ast +
                         ")"
                         }
```

Die Produktionen für *variable* und *expression* müssen Sie entsprechend definieren.

Die Zuweisung `x:=0` wird also als die folgende Zeichenkette repräsentiert: *Assign_statement(Variable(x),IntLiteral(0))*

Anmerkung: Die Darstellung eines abstrakten Syntaxbaums als Zeichenkette eignet sich zwar gut zur textuellen Darstellung – zur Weiterverarbeitung ist sie jedoch eher »unhandlich«. Wir werden im nächsten Unterkapitel sehen, welche Datenstrukturen gut geeignet sind, um die Informationen zu speichern.

Wie aber geschieht die Berechnung der Attribute im Parser?

Bei Top-Down-Parsern wird ja anhand des Lookahead-Tokens die anzuwendende Produktion $X \to Y_1 \ldots Y_k$ ausgewählt (dabei sei X das oberste Symbol auf dem Stack). Die ererbten Attribute der Y_i werden zugewiesen, und wenn die rechte Seite der Produktion »abgearbeitet« ist, werden die Berechnungsfunktionen angewandt, um die synthetisierten Attribute von X zu berechnen.

Hierbei könnte natürlich eines der Y_i wiederum ein Nichtterminalsymbol sein, auf das eine Produktion mit zugeordneten Berechnungsfunktionen angewendet wird. Daher muss sich der Parser die Berechnungsvorschriften ebenfalls auf dem Stack merken, um nach vollständigem Erkennen der rechten Seite die Berechnungsvorschriften anzuwenden. Die synthetisierten Attribute werden also berechnet, wenn die Regel zuoberst auf dem Stack liegt.

Bottom-Up-Parser arbeiten ja – wie der Name schon sagt – von unten nach oben. Ein Handle, das der rechten Seite einer Produktion entspricht, wird erkannt, und dann wird in der Parsertabelle nachgesehen, nach welcher Regel reduziert werden soll. Wenn aber die Regel erst dann feststeht, wie können dann ererbte Attribute berechnet werden?

In den meisten Fällen wird der Stack als Array definiert (so bei Bison und CUP), sodass nicht nur der Zugriff auf das oberste Element möglich ist, sondern auch auf die Elemente darunter. Weiß man, an welcher Stelle im Stack der Wert steht, der für die Berechnung des ererbten Attributs notwendig ist, so kann der Parser, wie eben erklärt, auf diesen zugreifen. Je nach Grammatik ist aber ein zusätzlicher Trick notwendig.

Beispiel 5.3 [Aho, Sethi & Ullman, 1986]

$S \to aAC \quad \{C.i := A.s\}$
$S \to bABC \quad \{C.i := A.s\}$
$C \to c \quad \{C.s := f(C.i)\}$

Bei dieser Grammatik kann der Parser nicht entscheiden, an welcher Position auf dem Stack das Attribut *A.s* steht: Bei Anwendung der ersten Produktion steht das Attribut an zweitoberster Stelle, bei Anwendung der zweiten Produktion steht B »dazwischen«. Hier kann man ein *Markierungssymbol* M in die zweite Produktion einfügen, sodass diese lautet:

$S \to bABMC \; \{M.i := A.S; C.i := M.s\}$ und

$M \to \varepsilon \{M.s := M.i\}$

M kopiert also das Attribut »von A zu C«, und der richtige Wert steht immer an zweitoberster Stelle im Stack.

Man unterscheidet im Allgemeinen S-attributierte Grammatiken und L-attributierte Grammatiken:

Bei einer *S-attributierten Grammatik* kommen nur synthetisierte Attribute vor, was ja sehr gut zu Bottom-Up-Parsern wie Bison und CUP passt. Bei einer *L-attributierten Grammatik* können ererbte und synthetisierte Attribute vorkommen. Dabei muss man beachten, dass der Parser natürlich wissen muss, welchen Wert die Attribute haben, wenn er eine Regel wie $X \to Y_1 \ldots Y_n$ anwendet: Hat Y_i ein Attribut a_i, so kann a_i nur von den ererbten Attributen von X abhängen (diese sind ja zu Beginn der Auswertung schon bekannt) oder von den Attributen (egal ob ererbt oder synthetisiert) von $Y_1, \ldots, Y_{i-1}$. Oder anders gesagt: a_i kann nicht von Attributen von $Y_{i+1}, \ldots, Y_n$ abhängen.

Daher sind L-attributierte Grammatiken gut für Top-Down-Parser geeignet.

Implementiert man einen Parser mit rekursivem Abstieg, so sind die ererbten Attribute Eingabeparameter der Funktionen und synthetisierte Attribute sind Rückgabewerte. Das Startsymbol der Grammatik hat dann ein Attribut, dessen Wert der komplette abstrakte Syntaxbaum des Programms ist.

Eine *Regelmitte-Aktion* wie in $X \to v\{A\}w$ soll bedeuten, dass die Aktion A nach dem Lesen von v und noch vor der Bearbeitung von w ausgeführt werden soll.

Ein Bottom-Up-Parser weiß ja erst nach dem Lesen des Handles vw, dass die Ableitungsregel $X \to vw$ angewandt werden soll. Das heißt, es ist unmöglich, schon vor dem Lesen von w zu wissen, dass die Aktion A auszuführen ist.

Man kann aber eine Regelmitte-Aktion in eine *Regelende-Aktion* umwandeln, indem man ein neues Nichtterminalsymbol Y einführt, sodass die ursprüngliche Regel $X \to vw$ ersetzt wird durch $X \to vYw$ und eine neue Regel $Y \to \varepsilon\{A\}$ eingeführt wird.

Man erkennt leicht, dass diese geänderte Grammatik dieselbe Sprache erzeugt.

Das folgende Code-Stück aus dem von CUP generierten Parser zeigt, wie die Aktion der Regel

$term \to term_{lhs}\ STAR\ factor_{rhs}$

$\{term.result = BinaryExpression(MUL, term_{lhs}.result, factor.result_{rhs})\}$

auf den Stack zugreift:

```
Expression lhs =(Expression)((java_cup.runtime.Symbol)
          CUP$Parser$stack.elementAt(CUP$Parser$top-2)).value;
Expression rhs = (Expression)((java_cup.runtime.Symbol)
          CUP$Parser$stack.peek()).value;
RESULT = new BinaryExpression(new Position(opleft, opright),
             BinaryExpression.Operator.MUL, lhs, rhs);
```

Listing 5.2 Ausschnitt aus dem generierten CUP-Parser

Um kenntlich zu machen, welches Symbol der rechten Seite der Produktion gemeint ist, haben wir die Namen `lhs` und `rhs` verwendet. Bei Auswertung der Aktion werden zunächst die Ergebnisse (`.value`) der beiden Teilergebnisse `lhs` und `rhs` vom Stack von den Positionen `top-2` und `top` (`=peek()`) genommen und gecastet (dazu später mehr).

5.3 Erzeugung des AST für SPL

Um zu verstehen, wie wir jetzt unsere Grammatik für SPL so attribuieren, dass sie einen abstrakten Syntaxbaum erzeugt, müssen wir erst noch drei Punkte klären:

1. Wie soll der abstrakte Syntaxbaum aussehen? Erstellen wir zum Beispiel einen Knotentyp für *Expr, Term* und *Factor* oder nutzen wir nur einen Knotentyp »binärer Ausdruck« mit dem Operator als Unterknoten?
2. In welcher Datenstruktur soll der abstrakte Syntaxbaum abgebildet werden?
3. Wie spezifiziert man die Attribute und Berechnungsfunktionen in Bison, CUP und ANTLR?

Beginnen wir mit Frage 1: Es zeigt sich, dass zur vollständigen Klärung Wissen über die Verwendung des abstrakten Syntaxbaums in den späteren Phasen notwendig ist. So werden wir zum Beispiel später alle arithmetischen Operatoren gleich behandeln, da die Präzedenzregeln durch den Parser bereits gewährleistet sind. Es bietet sich daher an, *einen* Knotentyp »binärer Ausdruck« (*binaryExpression*) zu nutzen.

Zu Frage 2 für Java: Es bietet sich natürlich an, den abstrakten Syntaxbaum objektorientiert zu implementieren: Jeder Knoten im Baum ist ein Objekt, und »gleichartige« Knoten, also solche, die das gleiche Sprachkonstrukt implementieren, gehören zur gleichen Klasse. In dem Buch von Andrew Appel [Appel, 2002] wird in Ausschnitten dieser Ansatz beschrieben.

Wir hatten schon erwähnt, dass wir in den späteren Phasen den abstrakten Syntaxbaum mehrmals durchlaufen werden, um zum Beispiel die Verwendung von Datentypen zu überprüfen. Da wir bei einem solchen Durchlauf nicht vorher wissen können, zu welcher Klasse der Knoten gehört, definieren wir eine Oberklasse `Node`:

```
public abstract class Node implements Visitable {
    public final Position position;
    Node(Position position) {
       this.position=position;
    }
...
}
```

Listing 5.3 Die Klasse »Node« im abstrakten Syntaxbaum

Das Interface `Visitable` werden wir im nächsten Kapitel genauer erklären; `Position` ist eine Hilfsklasse, die die Zeilen- und die Spaltennummern des Symbols im Quelltext bezeichnet.

Anmerkung: Die `ComplexSymbolFactory` von CUP bietet die Klasse `Location` an, die die gleiche Funktionalität hat [Ananian, Flannery, Wang, Appel & Petter, 2020]. Da die einfachere `SymbolFactory` für unsere Zwecke ausreichend ist, benötigen wir die komplexere Variante nicht.

Zweckmäßigerweise sollten Sie noch Methoden vorsehen, mit denen Sie den abstrakten Syntaxbaum formatiert ausgeben können.

Die nächste wichtige Klasse, die wir benötigen, ist `Statement`. Für jede Art von Anweisung (Zuweisung, If-Then-Else, While-Schleifen etc.) legen wir eine Unterklasse von `Statement` an. Die Member der Klassen ergeben sich aus der Syntax: Die Klasse `WhileStatement` muss natürlich ein Member für die Bedingung und eines für den Schleifenrumpf enthalten. Listing 5.4 zeigt die Klasse `Statement` und exemplarisch `AssignStatement`, bei der wir in `target` die Variable und unter `value` den Ausdruck speichern.

Beispiel 5.4

```
public abstract class Statement extends Node {
    public Statement(Position position) {
        super(position);
    }
}
package compiler.ast;
public class AssignStatement extends Statement {
    public final Variable target;
    public final Expression value;
    public AssignStatement(Position position, Variable target,
                           Expression value) {
        super(position);
        this.target = target;
        this.value = value;
    }
}
```

Listing 5.4 Die Klassen »Statement« und »AssignStatement« im AST

Neben den Anweisungen benötigen wir natürlich auch Knoten für Ausdrücke, die aus den IntLiteralen, Variablen und binären Ausdrücken bestehen:

```
package compiler.ast;
public abstract class Expression extends Node {
    public Type dataType = null;
    public Expression(Position position) {
        super(position);
    }
}
...
package compiler.ast;
public class BinaryExpression extends Expression {
    public enum Operator {
        ADD, SUB, MUL, DIV, EQU, NEQ, LST, LSE, GRT, GRE;

        public boolean isArithmetic() {
            return List.of(ADD, SUB, MUL, DIV).contains(this);
        }
        public boolean isComparison() {
            return !this.isArithmetic();
        }
    public final Operator operator;
    public final Expression leftOperand;
    public final Expression rightOperand;
    public BinaryExpression(Position pos, Operator operator,
                            Expression leftOperand, Expression rightOperand)
         { ... }
}
```

Listing 5.5 Die Klassen »Expression« und »BinaryExpression« im AST

Wie Sie sehen, ist `Expression` eine abstrakte Klasse und `BinaryExpression` erweitert diese.

Neben `BinaryExpression` benötigen wir auch noch Erweiterungen von `Expression` für Integer-Literale und Variablenzugriffe. Um einen L-Value auch im AST von einem R-Value zu unterscheiden, kapseln wir Variablen in einem Ausdruck in einer Klasse `VariableExpression`, die dann `Expression` erweitert. Erstellen Sie diese Klassen nach dem obigen Muster!

Vielleicht fragen Sie sich, wie das unäre Minus wie zum Beispiel in `-(x+1)` abgebildet wird. Da das unäre Minus der einzige einstellige Operator ist, kann man der Einfachheit halber einen binären Ausdruck mit Operator `SUB` und einem linken Operanden `0` verwenden. In dem obigen Beispiel würde dies `0-(x+1)` entsprechen.

Was fehlt noch? Leider noch eine ganze Menge: Typ-, Variablen- und Prozedurdeklarationen sowie die Klassen für die ihnen zugeordneten syntaktischen Elemente, wie Parameterdeklaration etc.

Für viele der rekursiven Produktionen, wie zum Beispiel *statement_list*, lassen sich die Knoten als Listen darstellen. Als Beispiel zeigen wir hier die Klasse `ProcedureDeclaration`:

```
package compiler.ast;
public class ProcedureDeclaration extends GlobalDeclaration {
    public final List<ParameterDeclaration> parameters;
    public final List<VariableDeclaration> variables;
    public final List<Statement> body;

    public ProcedureDeclaration(Position position,
                                Identifier name,
                                List<ParameterDeclaration> parameters,
                                List<VariableDeclaration> variables,
                                List<Statement> body) {
        super(position, name);
        this.parameters = parameters;
        this.variables = variables;
        this.body = body;
    }
}
```

Listing 5.6 Die Klasse »ProcedureDeclaration« im abstrakten Syntaxbaum

`GlobalDeclaration` ist dabei wieder eine abstrakte Klasse, die durch `ProcedureDeclaration` und `TypeDeclaration` erweitert wird.

Die abstrakten Klassen dienen zum einen der »Kategorisierung« der Knoten, erhöhen aber auch die Typsicherheit, indem wir – wo es geht – immer diese Klassen verwenden statt der obersten Klasse `Node`.

Die in Abschnitt 3.5 gezeigte Klasse `Symbol` enthält die Attribute der Terminalsymbole.

In den folgenden Abschnitten werden wir diese Klassen um Attribute für die späteren Phasen erweitern.

Die Frage nach der Datenstruktur ist also für Java beantwortet: Die Attribute eines Symbols werden in Klassen zusammengefasst. Was aber ist mit Frage 3: »Wie spezifizieren wir Attribute und Berechnungsfunktionen in den Parsergeneratoren?«

Zunächst müssen wir in dem Abschnitt der Symbollisten für die Terminalsymbole und die Nichtterminalsymbole die Klassen angeben. Für Terminalsymbole, die nicht im abstrakten Syntaxbaum vorkommen werden (wie `ASGN`, `SEMIC`, ...), ist dies nicht notwendig, wohl aber für `Identifier` und `IntLiteral`:

```
terminal String     IDENT;       // Bezeichner
terminal Integer    INTLIT;      // Integer-Literale
```

Für Nichtterminalsymbole werden die oben definierten Klassen als Typen angegeben:

```
non terminal Statement statement, assign_statement;
non terminal Expression expr, term, factor;
```

Der von CUP generierte Parser ordnet allen Terminal- und allen Nichtterminalsymbolen Objekte der Klasse `java_cup.runtime.Symbol` zu: Der Scanner generiert für jedes erkannte Token ein Objekt dieser Klasse, und CUP gibt Objekte dieser Klassen zurück.

`Symbol` hat folgende Attribute:

```
public int sym;          // eine Nummer, die das Symbol identifiziert
public int parse_state;  // Zustand des Parsers
public int left;         // Zeilennummer
public int right;        // Spaltennummer
public Object value;     // der »Wert«
```

Da die Werte im Member `value` alle den Typ `Object` haben, werden sie im generierten Java-Code auf den richtigen Typ gecastet. Aufgrund der Typdeklarationen in den Symbollisten ist klar, welcher Cast notwendig ist.

In Listing 5.2 hatten wir dieses Verfahren schon gesehen:

```
Expression lhs =(Expression)((java_cup.runtime.Symbol)
          CUP$Parser$stack.elementAt(CUP$Parser$top-2)).value;
```

Die Berechnungsvorschriften bzw. -funktionen werden *semantische Aktionen* genannt (nicht zu verwechseln mit den Aktionen des Shift-Reduce-Algorithmus!) und werden bei CUP wie folgt spezifiziert:

- Jedes Symbol auf der rechten Seite einer Produktion kann mit einem Label versehen werden; Symbol und Label werden durch einen Doppelpunkt getrennt.

  Beispiel: `expressions:lhs`

  Das Nichtterminalsymbol auf der linken Seite wird implizit als `RESULT` bezeichnet.

Die Labels dienen dazu, die Attribute der Symbole zu benennen, sodass man sie in den Aktionen ansprechen kann.

- Jedes Symbol `x` hat Attribute `xleft` und `xright`, die die Zeilen- und die Spaltennummer angeben. Diese werden wir benutzen, um das Position-Objekt zu füllen.
- Die Aktionen sind Java-Code, der in `{:` und `:}` geklammert wird. Wir haben zwar oben davon gesprochen, dass die Aktionen die Attributwerte berechnen, tatsächlich können Sie aber beliebige Java-Anweisungen nutzen.

  Beispiel: `{: RESULT=new BinaryExpression(...); :}`
- Die Ausführung der Aktionen erfolgt, wie in Abschnitt 5.2 beschrieben, wenn der Parser eine Reduktion mit der Produktion durchführt, an der die Aktion hängt.

Hier ist jetzt der Parser mit Attributen für unser Mini-SPL:

```
import java_cup.runtime.*;
import compiler.utils.CompilerError;
import compiler.ast.*;
import compiler.symboltable.Identifier;
import java.util.ArrayList;
parser code {:
public void syntax_error(Symbol currentToken) {
  throw CompilerError.SyntaxError(new Position(currentToken.left,
                                               currentToken.right));
  }
:}
terminal  ASGN, SEMIC;
terminal  PLUS, MINUS, STAR, SLASH;
terminal  String IDENT;
terminal  Integer INTLIT;
non terminal Program         program;
non terminal List<Statement> statement_list;
non terminal Statement       statement, assign_statement;
non terminal Expression      expression, term, factor;
program::= statement_list:stmts {:
                          RESULT=new Program(new Position(stmtsleft,
                          stmtsright), stmts);
                          :};  /* Startproduktion */
statement_list::=  /* epsilon */ {: RESULT = new ArrayList<Statement>(); :}
          | statement:hd statement_list:tl {: tl.add(0, hd); RESULT = tl; :};
statement::=  assign_statement:as SEMIC {: RESULT =as; :}
          | error SEMIC;
assign_statement::= IDENT:id ASGN:asgn expression:exp {:
```

```
              RESULT=new AssignStatement(new Position(asgnleft, asgnright),
                     new Identifier(id), exp);
                                                          :};
expression::= expression:lhs PLUS:op term:rhs {:
              RESULT=new BinaryExpression(new Position(opleft, opright),
                          BinaryExpression.Operator.ADD, lhs, rhs);
                                                :}
                    | expression:lhs MINUS:op term:rhs {:
              RESULT=new BinaryExpression(new Position(opleft, opright),
                          BinaryExpression.Operator.SUB, lhs, rhs); :}
                    | term:t {: RESULT= t;
                                                           :};
term::= term:lhs STAR:op factor:rhs {:
              RESULT=new BinaryExpression(new Position(opleft, opright),
                          BinaryExpression.Operator.MUL, lhs, rhs);
                                                            :}
                     | term:lhs SLASH:op factor:rhs {:
              RESULT=new BinaryExpression(new  Position(opleft, opright),
                          BinaryExpression.Operator.DIV, lhs, rhs); :}
                     | factor:fac {: RESULT= fac;
                                                            :};
factor::= IDENT:id {:
              RESULT=new VariableExpression(new Position(idleft, idright),
                     new Identifier(id)); :}
                  | INTLIT:wert {:
              RESULT=new IntLiteral(new Position(wertleft, wertright),
                           wert); :};
```

Listing 5.7 Mini-SPL-Parser mit Attributen

Übersetzen Sie diese Datei wie in Kapitel 4 beschrieben. Ergänzen Sie in den Klassen im Package `compiler.ast` noch `toString()`-Methoden, um den abstrakten Syntaxbaum auch textuell ausgeben zu können.

In der Klasse `Main` müssen Sie den Aufruf des Parsers wie folgt ergänzen:

```
Parser parser = new Parser(scanner, symbolFactory);
Program prog = (Program) parser.parse().value;
System.out.println(prog);
```

Die Klasse `Identifier` ist ein Wrapper um `String`, die wir aus Gründen der Effizienz einführen. Die Definition von `Identifier` finden Sie in Abschnitt 6.2.1.

In C stehen leider keine Klassen zur Verfügung, daher müssen wir `structs` und `unions` verwenden, um den abstrakten Syntaxbaum abzubilden. Dabei sehen wir vor, dass die verschiedenen `unions` durch ein Tag unterscheidbar sind. Als Beispiel zeigen wir die Definition der Datenstruktur für Ausdrücke:

```
typedef enum {
    OP_ADD,
    OP_SUB,
    OP_MUL,
    OP_DIV
} binary_operator;
typedef enum {
    TAG_BINEXP,
    TAG_INTLIT,
    TAG_VAR
} tag_exp;
typedef struct expression {
    int line;
    tag_exp tag;
    union {
        struct {
            binary_operator operator;
            struct expression *leftOperand, *rightOperand;
        } binaryExpression;
        struct {
            int value;
        } intLiteral;
        struct {
            struct variable *name;
        } variableExp;
    } u;
} Expression;
typedef struct statement_list {
    bool                  isEmpty;
    Statement            *head;
    struct statement_list *tail;
} StatementList;
typedef StatementList Program;
```

Listing 5.8 Ausschnitt aus der Definition des abstrakten Syntaxbaums in C

Zusätzlich sehen wir noch »Pseudo-Konstruktoren« vor, die einen Knoten erzeugen und die von dem von Bison generierten Parser in den Aktionen aufgerufen werden (analog zu den Aufrufen der Konstruktoren der Java-Klassen in CUP):

```
Expression *newExpression(int line, tag_exp tag) {
    Expression *node = allocate(sizeof(Expression));
    node->line = line;
    node->tag = tag;
    return node;
}
```

Listing 5.9 Pseudo-Konstruktor für Expression-Knoten im abstrakten Syntaxbaum

Anhand des Beispiel-Parsers für Mini-SPL wird deutlich, wie man diese verwendet.

Natürlich müssen Sie noch weitere Strukturen und Konstruktoren definieren, die Anweisungen und alle anderen SPL-Sprachkonstrukte und deren entsprechende »Unterklassen« abbilden.

```
%{
/*
 * parser.y -- Mini SPL parser specification
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utils.h"
#include "symboltable/identifier.h"
#include "ast.h"
#include "scanner.h"
#include "parser.h"

void yyerror(Program**, char *);
%}
%parse-param {Program** program}
%union {
  NoVal         noVal;
  IntVal        intVal;
  StringVal     stringVal;
  Expression    *expression;
  Variable      *variable;
  Statement     *statement;
  StatementList *statementList;
```

```
}
%token <noVal>         ASGN SEMIC
%token <noVal>         PLUS MINUS STAR SLASH
%token <stringVal>     IDENT
%token <intVal>        INTLIT
%type <expression>     expression term factor
%type <variable>       variable
%type <statement>      statement assign_statement
%type <statementList>  statement_list
%type <statementList>  program
%start program;
%%
program:        statement_list {*program=$1; };
statement_list: /* epsilon */ {$$=emptyStatementList();}
           | statement statement_list {$$=newStatementList($1, $2);};
statement:        assign_statement {$$=$1;} ;
assign_statement: IDENT ASGN expression SEMIC
                { $$=newAssignStatement($1.line,
                  newVariable($1.line, newIdentifier($1.val)), $3);};
expression:       term {$$ = $1;}
                | expression PLUS term
                  { $$=newBinaryExpression($2.line, OP_ADD, $1, $3);}
                | expression MINUS term
                  { $$=newBinaryExpression($2.line, OP_SUB, $1, $3);};
term:             factor {$$ = $1;}
                | term STAR factor
                  { $$=newBinaryExpression($2.line, OP_MUL, $1, $3);}
                | term SLASH factor
                  { $$=newBinaryExpression($2.line, OP_DIV, $1, $3);};
factor:           variable {$$ = newVariableExpression($1->line, $1);}
                | INTLIT {$$ = newIntLiteral($1.line, $1.val);};
variable:         IDENT
                  { $$=newVariable($1.line, newIdentifier($1.val));};
%%
void yyerror(Program** program, char *msg) {
  error("%s in line %d", msg, yylval.noVal.line);
}
```

Listing 5.10 Bison-Datei für Mini-SPL mit abstraktem Syntaxbaum

Ähnlich wie wir das schon bei CUP gesehen haben, müssen wir auch bei Bison dem Parsergenerator mitteilen, welche Typen für die verschiedenen Nichtterminalsymbole ver-

wendet werden. In Kapitel 4 hatten wir erkannt, dass hinter `%union` die Varianten des Rückgabetyps der Terminalsymbole verwendet werden. Jetzt erweitern wir die `union` um die Typen der Attribute der Nichtterminalsymbole. Letztere verknüpfen wir mit dem Nichtterminalsymbol durch:

```
%type <Typ> Non-Terminal
```

Mit `%parse-param` können wir dem Parsergenerator mitteilen, dass wir beim Aufruf von `yyparse()` weitere Argumente übergeben wollen. Da wir von `yyparse` als Rückgabewert einen Zeiger auf den abstrakten Syntaxbaum zurückerhalten wollen, schreiben wir:

```
%parse-param {Program** program}
```

Die Aktionen werden als C-Code in Mengenklammern eingeschlossen angegeben. Dabei ist `$$` der Wert des Nichtterminalsymbols auf der linken Seite der Produktion (analog zu `RESULT` in CUP). Die Attribute der Symbole auf der rechten Seite der Produktion werden nicht mit Labeln versehen wie bei CUP, sondern einfach beginnend mit `$1` durchnummeriert. Dabei ist zu beachten, dass natürlich auch die Terminalsymbole mitgezählt werden.

Aktionen, die das Ergebnis des einzigen Symbols auf der rechten Seite einer Produktion lediglich zur linken Seite kopieren, also `$$=$1`, kann man weglassen, da diese als Default eingesetzt werden. Um zu vermeiden, dass dies aber nicht die Aktion ist, die Sie nutzen wollten, sollten Sie immer die Zuweisung an `$$` explizit hinschreiben.

Wie werden die Aktionen von Bison umgesetzt? Zunächst erstellt Bison aus der `%union`-Deklaration den Typ `YYSTYPE`:

```
typedef union YYSTYPE
{
#line 22 "parser.y"
  NoVal         noVal;
  IntVal        intVal;
  StringVal     stringVal;
  Expression    *expression;
  Variable      *variable;
  Statement     *statement;
  StatementList *statementList;
} YYSTYPE;
```

Listing 5.11 Von Bison generierter YYSTYPE-Typ

Danach wird eine Variable `yyval` vom Typ `YYSTYPE` deklariert

```
YYSTYPE yyval;
```

und entsprechend in die jeweilige Aktion eingesetzt:

```
#line 58 "parser.y"
    {(yyval.expression) = newBinaryExpression((yyvsp[(2) - (3)].noVal).line,
     OP_ADD, (yyvsp[(1) - (3)].expression), (yyvsp[(3) - (3)].expression));;}
```

Dies ist die übersetzte Aktion für die Regel *expression → expression PLUS term*.

`yyvsp` ist ein Zeiger auf `YYSTYPE` und ist der Stack des Parsers, der die semantischen Werte enthält (`v` steht für *Value*, `s` für *Stack*). Die Nummerierung der Symbole (`$1`, `$2`, `$3`) entspricht den jeweils ersten Elementen beim Zugriff auf den Stack.

Für eine Mini-SPL-Datei mit dem Inhalt

```
a:=3+4;
x:=1*3;
```

erzeugt das obige Programm die Ausgabe:

```
Program(
  AssignStatement(
    Variable(a),
    BinaryExpression(
      ADD,
      IntLiteral(3),
      IntLiteral(4))),
  AssignStatement(
    Variable(x),
    BinaryExpression(
      MUL,
      IntLiteral(1),
      IntLiteral(3))))
```

Listing 5.12 Ausgabe eines abstrakten Syntaxbaums

Bleibt noch zu klären, wie wir Aktionen für den dritten Parsergenerator ANTLR definieren. Die Antwort ist: Gar nicht!

In ANTLR ist es unüblich, den abstrakten Syntaxbaum mittels einer attributierten Grammatik zu implementieren. Vielmehr erzeugt ANTLR automatisch einen Ableitungsbaum sowie Java-Klassen, die einen sogenannten *Visitor* implementieren. Dabei han-

delt es sich um ein Entwurfsmuster, das wir später auch in dem mit CUP implementierten Compiler benötigen und das in Abschnitt 6.2.2 genauer besprochen wird.

Die von ANTLR erzeugten Visitor-Klassen können Sie benutzen, um jeden Knoten des konkreten Syntaxbaums (*Concrete Syntax Tree, CST*) zu »besuchen« und Berechnungen auf diesen Knoten auszuführen. Diese Berechnungen erzeugen hier den abstrakten Syntaxbaum (siehe Listing 5.14).

Man kann nun debattieren, ob man mit dem *CST* oder dem *AST* weiterarbeiten sollte. Wegen der Weiterverwendung des abstrakten Syntaxbaums in den folgenden Kapiteln nutzen wir den Mechanismus des Visitors, um den AST mit den gleichen Java-Klassen wie für CUP zu erzeugen.

Da unsere Grammatik Alternativen für einige Nichtterminalsymbole enthält (siehe zum Beispiel *expression* unten), müssen wir diese eindeutig kennzeichnen, sodass wir diese beim Durchlauf durch den CST eindeutig identifizieren können. Dazu erlaubt es ANTLR, am Ende jeder Alternative nach dem Zeichen # ein Label anzugeben (siehe Listing 5.13).

```
grammar spl;
program:            statement_list           # stmtsLbl
                    ;
statement_list:  statement+                  # stmtLbl
                    ;
statement:  empty_statement                  # emptyLbl
                    | assign_statement       # asmtmLbl
                    ;
empty_statement: ';' ;
assign_statement:  IDENT ':=' expression ';'  # assignLbl
                    ;
expression:           term '+' expression    # plusLbl
                    | term '-'  expression   # minusLbl
                    | term                   # termLbl
                    ;
term:                 factor                 # factorLbl
                    | factor '*' term        # mulLbl
                    | factor '/' term        # divLbl
                    ;
factor:               IDENT                  # identLbl
                    | INTLIT                 # intLitLbl
                    ;
```

```
INTLIT: ('0' .. '9') + ;
IDENT: ('a' .. 'z' | 'A' .. 'Z') ('a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '_')*;
WHITESPACE: [ \t\r\n] -> skip;
```

Listing 5.13 ANTLR-Grammatik für Mini-SPL mit Labels

Erwähnt werden soll noch, dass es einen zweiten Mechanismus gibt, um den CST zu durchlaufen: ANTLR kann wahlweise auch einen *Listener* erzeugen. Visitors sind etwas leichter umzusetzen, sodass wir mit dem Befehl

`java -jar antlr-4.8-complete.jar -no-listener -visitor -package compiler.parser spl.g4`

neben dem Parser ein Interface `SPLVisitor` und eine Klasse `SPLBaseVisitor` erzeugen lassen können.

Der `SPLBaseVisitor` implementiert das `SPLVisitor`-Interface und beinhaltet Basisdefinitionen für jede Alternative, die den Namen `visit<Label>` tragen.

Diese Methoden haben einen Parameter `ctx`, der von der ANTLR-Klasse `ParserRuleContext` ableitet. Diese Klasse enthält alle Informationen des CST, zum Beispiel die Anzahl der Knoten auf der nächsten Ebene, deren Kontexte etc. Die Methoden geben den konstruierten Knoten des abstrakten Syntaxbaums zurück.

Unsere Aufgabe ist es nun, einen `AstSplVisitor` zu schreiben, der `SPLBaseVisitor` erweitert, indem die `visit`-Methoden so umgeschrieben werden, dass sie – analog zu den CUP-Aktionen – die Konstruktoren der AST-Klassen aufrufen und »nach oben« zurückgeben.

Listing 5.14 zeigt einen kleinen Ausschnitt aus dem Visitor zu der Grammatik aus Listing 5.13:

```
package compiler.visitor;
import compiler.ast.*;
import compiler.spl.*;
import compiler.utils.*;
import compiler.symboltable.Identifier;
import org.antlr.v4.runtime.tree.ParseTree;

public class AstSplVisitor extends splBaseVisitor<Node> {
  // Startregel: program -> statement_list
  @Override
  public Program visitStmtsLbl(splParser.StmtsLblContext ctx) {
    var pos=new Position(ctx.getStart().getLine(),
                         ctx.getStart().getCharPositionInLine());
```

```
    return new Program(pos, (StatementList)visit(ctx.statement_list()));
  }

  // statement_list -> statement+
  @Override
  public StatementList visitStmtLbl(splParser.StmtLblContext ctx) {
    var pos=new Position(ctx.getStart().getLine(),
                         ctx.getStart().getCharPositionInLine());
    StatementList stmts = new StatementList(pos);
    for (int i = 0; i < ctx.getChildCount(); i++) {
      // Get the i-th child node of `parent`.
      ParseTree child = ctx.getChild(i);
      Node stmt = visit(child);
      stmts.add((Statement) stmt);
    }
     return stmts;
  }
  ...
// assign_statement -> IDENT ':=' expression ';'
  @Override
  public Statement visitAssignLbl(splParser.AssignLblContext ctx) {
    var pos=new Position(ctx.getStart().getLine(),
      ctx.getStart().getCharPositionInLine());
    var target = new Identifier(ctx.IDENT().getText());
    Expression exp  = (Expression) visit(ctx.expression());
    return new AssignStatement(pos, target, exp);
  }
```

Listing 5.14 Ausschnitt aus »AstSplVisitor«

Wie Sie sehen, enthält der Kontext `ctx` auch Informationen über Zeilen- und Spaltennummern, die wir nutzen, um – wie bei CUP – das `Position`-Objekt zu füllen.

Damit können wir nun den Schluss unseres Hauptprogramms abwandeln, sodass es den gleichen abstrakten Syntaxbaum wie der CUP-Parser ausgibt:

```
...
ParseTree ast = parser.program();
AstSplVisitor splv = new AstSplVisitor();
Program prog = (Program) splv.visit(ast);
System.out.println(prog);
```

Listing 5.15 Änderungen am Hauptprogramm des ANTLR-Parsers

5.4 Zusammenfassung

Wir haben zunächst die Notwendigkeit einer kompakteren und doch so weit vollständigen Darstellung des Quellprogramms erläutert. Die von Donald Knuth entdeckten attributierten Grammatiken erlauben eine syntaxgesteuerte Übersetzung, bei der das Parsen eines Quellprogramms gleichzeitig die Erzeugung des abstrakten Syntaxbaums steuert. Der abstrakte Syntaxbaum ist die von uns gesuchte kompakte Darstellung des Quellprogramms, mit der wir in den nächsten Phasen weiterarbeiten werden.

Das (in Teilen) angegebene Skelett für den abstrakten Syntaxbaum von SPL erlaubt – sowohl in Java als auch in C – eine leichte Handhabung des abstrakten Syntaxbaums.

Wir haben nun das Frontend des Compilers abgeschlossen: Nach der lexikalischen Analyse (Kapitel 3) und der Syntaxanalyse in Kapitel 4 haben wir einen abstrakten Syntaxbaum erzeugt, der alle Informationen enthält, die wir benötigen, um die nächsten Phasen zu beginnen.

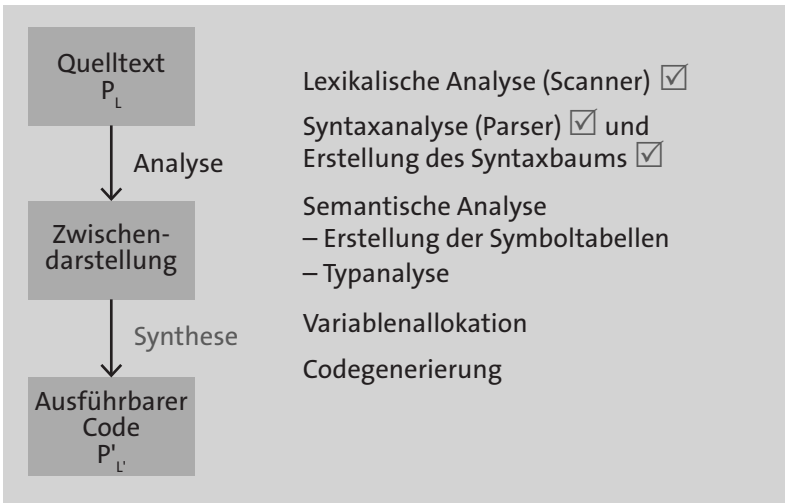

Abbildung 5.7 Stand nach der Erzeugung des abstrakten Syntaxbaums

Nach diesen Vorbereitungen werden wir im nächsten Kapitel die Analysephase mit der semantischen Analyse abschließen. Dafür stehen uns leider keine so weit verbreiteten und umfassenden Werkzeuge wie Bison, CUP oder ANTLR zur Verfügung. Das Mittel der Wahl ist daher der selbst zu entwickelnde C- oder Java-Code.

5.5 Übungen

5.5.1 Erweiterungen

1. Aufgabe: Unsere bisherige Grammatik kann kein Minuszeichen als Vorzeichen behandeln. Ergänzen Sie in der Grammatik eine Produktion für die Behandlung des unären Minus, und definieren Sie eine entsprechende Aktion.
2. Aufgabe: Als Nächstes sollten Sie versuchen, Typdeklarationen umzusetzen. Dazu müssen Sie die Grammatik erweitern und auch neue Klassen erstellen.
3. Aufgabe: Das obige Beispiel kennt noch keine Variablendeklarationen. Erweitern Sie die Grammatik um Variablendeklarationen. Dazu müssen Sie neue Klassen im abstrakten Syntaxbaum erstellen.
4. Aufgabe: Ergänzen Sie die weiteren Anweisungsarten: If-Then-Else, While-Schleife und zusammengesetzte Anweisung.
5. Aufgabe: Ergänzen Sie die Grammatik um Prozedurdeklarationen und Prozeduraufrufe.

5.5.2 ANTLR

6. Aufgabe: Programmieren Sie den Visitor für die SPL-Grammatik in ANTLR.