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.

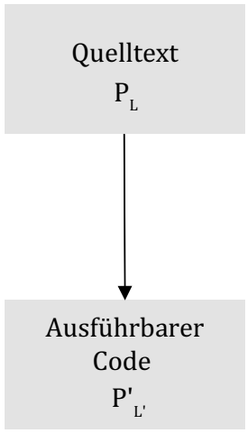

Abbildung 1.1 Funktion eines Compilers

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 werde 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*.

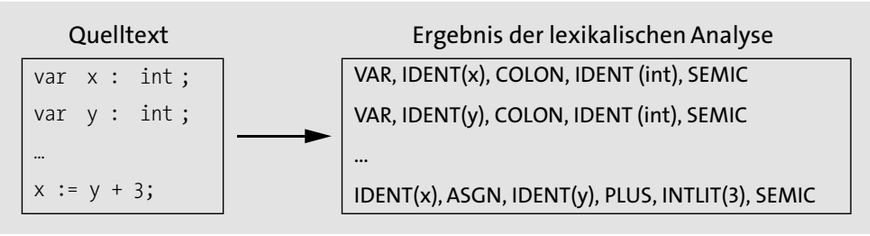

Abbildung 1.3 Lexikalische Analyse (Beispiel)

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.

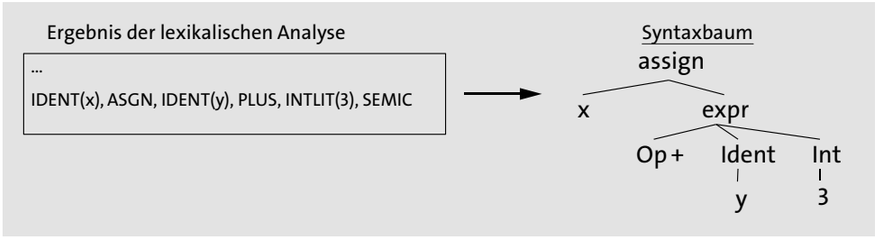

Abbildung 1.4 Ergebnis der Syntaxanalyse (Beispiel)

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 Design Patterns, des *Visitor-Pattern*, durchlaufen.

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.

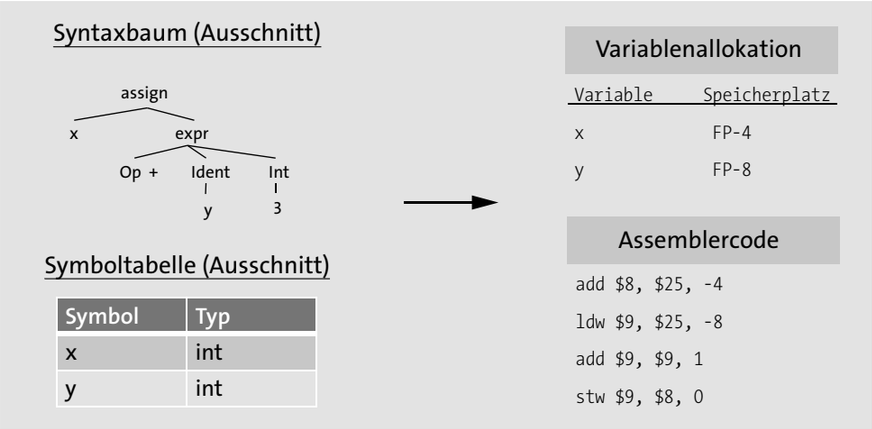

Abbildung 1.5 Backend des Compilers (Beispiel)

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.

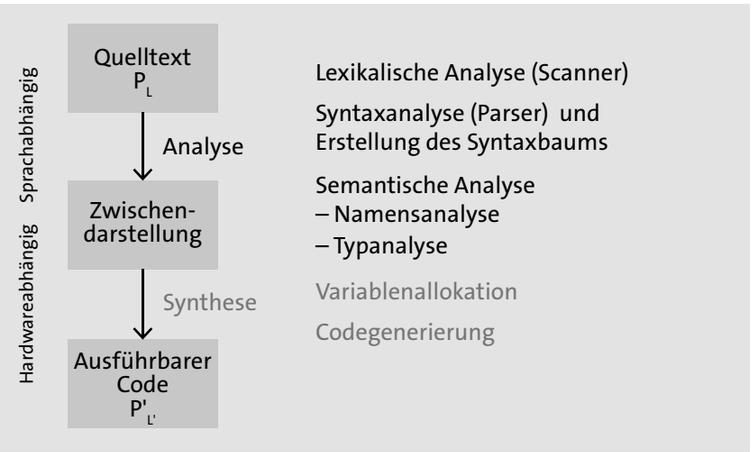

Abbildung 1.6 Phasenmodell eines Compilers

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.