Compiler Framework für die virtuelle Maschine von Java

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Tobias W. Röthlisberger

Mai 1999

Betreuer der Arbeit:
Dr. Markus Lumpe

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Weitere Informationen zu dieser Diplomarbeit, insbesondere über die verwendeten Tools, die Ausführung von Beispielen und eine *online* Version der Dokumentation, findet man auf:
`http://www.iam.unibe.ch/~scg`

Die Adresse des Autors:

Tobias Röthlisberger
Zwinglistrasse 16
CH-3007 Bern

oder

Software Composition Group
Institut für Informatik und angewandte Mathematik - Universität Bern
Neubrückstrasse 10
CH-3012 Bern
Email: `roethlis@iam.unibe.ch`
WWW: `http://www.iam.unibe.ch/~roethlis`

Zusammenfassung

Ein Compiler übersetzt ein Programm einer Quellsprache in eine Zielsprache. Um Programme in einer beliebigen Umgebung ausführen zu können, muss eine plattformunabhängige Zielsprache gesucht werden. Die Plattformunabhängigkeit wird dadurch erreicht, dass die Programme nicht auf einer konkreten, sondern auf einer virtuellen Maschine ausgeführt werden, die als Schnittstelle zwischen compiliertem Code und Plattform dient. Die virtuelle Maschine von Java („JVM“) verarbeitet nicht nur Java, sondern alle Programme, die in einem genau spezifizierten Format dargestellt werden.

Beim Compilerbau trennt eine Zwischensprache die Analyse des Quellprogrammes von der Synthese zum Zielprogramm; man kann die beiden Teile unabhängig voneinander behandeln. Durch die Verwendung einer Zwischensprache muss nicht jede Quellsprache einzeln in eine bestimmte Zielsprache übersetzt werden, sondern die verschiedenen Analyse-Teile, die Compilerfrontends der Quellsprachen, arbeiten mit einem einzigen Synthese-Teil, dem Compilerbackend für die Zielsprache zusammen.

In dieser Diplomarbeit wird ein Compilerframework aufgebaut, das dank der Zielmaschine JVM plattformunabhängig ist, und eine Zwischensprache verwendet, die die Eigenschaften verschiedenartigster Sprachen umfasst. Mittels einer Analyse von Programmiersprachen des imperativen, des objektorientierten, des funktionalen und des logischen Paradigmas werden die Anforderungen an diese generelle Zwischensprache aufgestellt. Für jedes Paradigma wird die Grammatik einer Beispielsprache definiert und ein Parser konstruiert. Diese Parser übersetzen Programme ihrer Quellsprachen in die generelle Zwischensprache. Das Compilerbackend dieser Arbeit generiert aus der generellen Zwischensprache JVM-Code. Der Compiler wird mit Hilfe eines Frameworks konstruiert, indem die Zwischensprache und die Codegenerierung in Klassen mit möglichst genereller Funktionalität aufgeteilt werden. Dadurch können sowohl die untersuchten Sprachen, als auch in dieser Arbeit nicht behandelte Programmiersprachen nach JVM Code compiliert werden.

Persönliches

Nach einigen Jahren an der Universität Bern und mehreren Monaten in der Software Composition Group (SCG) möchte ich mich an dieser Stelle laut und deutlich bei allen bedanken, die mir auf meinen Wegen geholfen und mein Umfeld (der Student lebt nicht nur vom Testat allein) mitgestaltet haben:

"VIELEN DANK!"

- Dr. **Markus Lumpe** hat mich bei dieser Diplomarbeit betreut, mit Inputs und Ideen versorgt und sein langjähriges Compilerbauwissen mit mir geteilt. Ohne seine Hilfe bei Hindernissen und seine Unterstützung beim Codecrawling und Gruppendiskussionen hätte diese Arbeit kein Ende gefunden.
- Prof. Dr. **Oscar Nierstrasz** hat sich immer für meine Arbeit interessiert und mich zu realisierbaren Teilzielen angehalten. Als Leiter der SCG bietet er ein konstruktives, hochstehendes und internationales Umfeld, in dem das Studieren Spass macht.
- Besonderer Dank geht an die hervorragenden Reviewer **Matthias Rieger** und **Sander Tichelaar**. Sie lasen die verschiedenen Entwürfe meiner Arbeit mit der nötigen Distanz durch und regten mich zu mehr Bildchen, kürzeren Sätzen und neuen Kapiteln an.
- **Franz Achermann** führte mich, kurz vor der Mittagspause, in *JavaCC* und *JUnit* ein. **Jean-Guy Schneider** war immer mit Tips und Tricks zu Handen und lieh mir gratis seine Bücher. **Serge Demeyer** hielt die besten Vorlesungen an der Universität Bern (IMHO), er möge den Studenten noch lange erhalten bleiben. **Stephane Ducasse** begeisterte mich für Smalltalk; leider war die Zeit zur Ausführung all seiner Vorschläge zu knapp. **Isabelle Huber** half mir beim Ausfüllen von Formularen, Einhalten von Anmeldefristen und liess sich mein Testatbuch nicht stehlen.
- **Michele Lanza** erreichte seine Tagesziele schon am frühen Vormittag und telefonierte anschliessend mit sämtlichen Handyträgern in Italien. **Daniel Frey** wusste meine Unix-Tips sehr zu schätzen und schickte mich als Dank zu einer billigeren Telefongesellschaft. **Fredi Frank** lieferte Einführungen und Erklärungen der Innerschweizer Politik im allgemeinen und der Nidwaldner im speziellen. **Georges Golomingi** verbrachte seine Wochenenden mit mir und seiner Diplomarbeit, anstatt sie mit seiner Familie zu geniessen. **Roger Blum** war der grosse Abwesende. **Daniel Kühni** und **Michael Held** haben sich bereits von der SCG verabschiedet und imposante Visitenkarten hinterlassen.
- Verschiedene Mitglieder der **Familie Günter** unterhielten mich mit Spiel & Spass, boten mir Jobs an und versuchten, mich im Studium zu überrunden.
- **Peter G.** (vom Zürisee) animierte mich zum Musizieren und zum Pflanzen; die drei Schwarzbärte aus Worb (**Willi, Chris, Mik**) führen beide Themen weiter.
- Herzlichen Dank an **Mama** und **Papa**. Meine Eltern haben mich beim Studium in jeglicher Hinsicht unterstützt, obwohl ich ihrer Ansicht nach hätte Matrose werden sollen. Mein Bruder **Adrian** erfreute mich mit Frühlingsblumen und 5-Stern-Pasteten, meine Schwester **Bettina** mit Tanzdarbietungen und wechselnder Haarfarbe.

♡ **Barbara Vogt** ist der Sonnenschein in meinem Leben. Ihr Strahlen, ihr Lachen, ihre Farben und ihre Suppen erwarteten mich zu Hause; ihre Diskussionen über I-Mäxchen lenkten meine Gedanken vollends von dieser Arbeit weg.

Inhaltsverzeichnis

Kapitel 1

Einleitung

> Eine Idee ist nichts anderes als der Begriff von einer Vollkommenheit, die sich in der Erfahrung noch nicht vorfindet.
>
> *Immanuel Kant, Über Pädagogik*

1.1 Idee und Ziel der Arbeit

In der vorliegenden Diplomarbeit geht es um die Übersetzung verschiedenartigster Programmiersprachen in den Code der virtuellen Maschine von Java („Java Virtual Machine“, „JVM“) [Lin97]. Um nicht jede Programmiersprache einzeln zu übersetzen, wird eine generelle Zwischensprache beschrieben, die die Eigenschaften der verschiedenen Sprachen umfasst und die in einer für die JVM verständlichen Form ausgedrückt werden kann. Die Konstruktion einer Zwischensprache und deren Übersetzung in eine Zielsprache bilden das „Backend“ eines Compilers. Dieser Compiler wird mit Hilfe eines Frameworks konstruiert, indem die Zwischensprache und die Codegenerierung in Klassen mit möglichst genereller Funktionalität aufgeteilt werden. Dadurch können sowohl die untersuchten Sprachen, als auch Erweiterungen davon oder in dieser Arbeit nicht behandelte Programmiersprachen nach JVM Code compiliert werden.

Die Motivation für diese Idee ist die Plattformunabhängigkeit der JVM. Dadurch wird Code jeder Programmiersprache, die sich auf die generelle Zwischensprache übertragen lässt, auf jedem Computer mit einer JVM lauffähig.

1.2 Problemstellung

Es ist für jeden Informatiker motivierend, wenn er seine Ideen und die daraus entstehenden Produkte, seien dies Programme oder Programmiersprachen, jedem Interessierten zeigen und in jeder Umgebung verwenden kann. Mit der Verbreitung des weltweiten Internets hat sich Java zu einem globalen Standard der Interoperabilität entwickelt; compilierte Javaprogramme sind netzwerk- und plattformunabhängig. Nun ist Java zwar eine einfache, objektorientierte, nebenläufige und portable Programmiersprache, doch sie eignet sich nicht für alle Problemstellungen. Ausserdem will man sich nicht auf eine einzige Sprache festlegen. Daher wäre eine Kombination der Plattformunabhängigkeit mit Sprachunabhängigkeit wünschenswert. Die Plattformunabhängigkeit erreicht Java dadurch, dass die Programme nicht auf einer konkreten, sondern auf einer virtuellen Maschine ausgeführt werden, die als Schnittstelle zwischen compiliertem Code und Plattform dient. Die Virtuelle Maschine von Java wurde zwar zusammen

mit der objektorientierten Sprache Java bei Sun Microsystems entwickelt, verarbeitet aber nicht nur Java Programme, sondern alle, die in einem genau spezifizierten Format dargestellt werden können. Mit der JVM kommt man folglich dem Ideal der Plattformunabhängigkeit und Sprachunabhängigkeit recht nahe, zudem ist sie frei erhältlich und deshalb fast allgegenwärtig.

Es existieren verschiedene Compiler für Nicht-Java-Sprachen, die Code für die JVM produzieren, deshalb muss ein weiterer Compileransatz begründet werden: Ein grundlegendes Konzept im Compilerbau ist die Verwendung einer Zwischensprache (Abbildung 1.1). Dadurch wird das Compilerfrontend vom Backend getrennt; man kann die beiden Teile unabhängig voneinander behandeln.

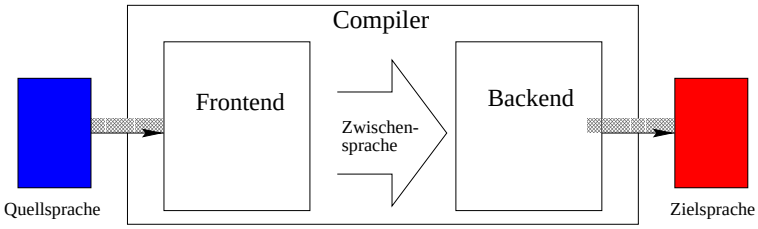

Abbildung 1.1: Compiler mit Zwischensprache

Für eine bestimmte Quellsprache muss ein Frontend vorhanden sein, das mit verschiedenen Backends für Zielsprachen zusammenarbeitet. Oder man muss ein Backend für eine Zielmaschine konstruieren, das von den Frontends für verschiedene Quellsprachen angesprochen wird.

Beim vorliegenden Ansatz hat man sich für die JVM als Zielmaschine entschieden. Nun sollen aber nicht verschiedene Frontends für verschiedene Quellsprachen passend zu diesem einen Backend aufgestellt werden, sondern man legt bewusst keine Quellsprache(n) fest und konzentriert sich auf eine möglichst allgemeine Zwischensprache. Diese soll die Eigenschaften verschiedenartigster Programmiersprachen umfassen, also in einem weiten Sinne sprachunabhängig sein. Um diese Forderung zu testen und die Eigenschaften von Programmiersprachen zu analysieren, wird exemplarisch je ein Frontend für eine imperative, eine objektorientierte, eine funktionale und eine logische Sprache entworfen. Die paradigmaspezifischen Frontends, die Zwischensprache und das Compilerbackend sollten wiederverwendbar und offen für Experimente mit neuen Sprachen sein. Angestrebt wird ein Compilerframework, das dank der Zielmaschine JVM plattformunabhängig ist, und alle Programmiersprachen, die sich im Rahmen der gängigen Paradigmen bewegen, verstehen kann.

1.3 Teilziele der Arbeit

Aus der Problemstellung „Compilerframework für die JVM“ folgen die nachstehenden Teilziele, die die einzelnen Schritte der Arbeit vorgeben:

1. Analyse von Programmiersprachen.
2. Definition einer Zwischensprache, die generell verwendbar ist.
3. Aufstellen der Anforderungen, die die JVM an einen Compiler stellt.
4. Konstruktion eines Compilerbackends für die JVM.

5. Die Summe der Teilziele ergibt ein Compilerframework mit genereller Zwischensprache für verschiedene Paradigmen und Bytecode-Backend. Die Zusammenarbeit der einzelnen Komponenten und der Bearbeitungsfluss im Compilerframework sind in Abbildung 1.2 dargestellt.

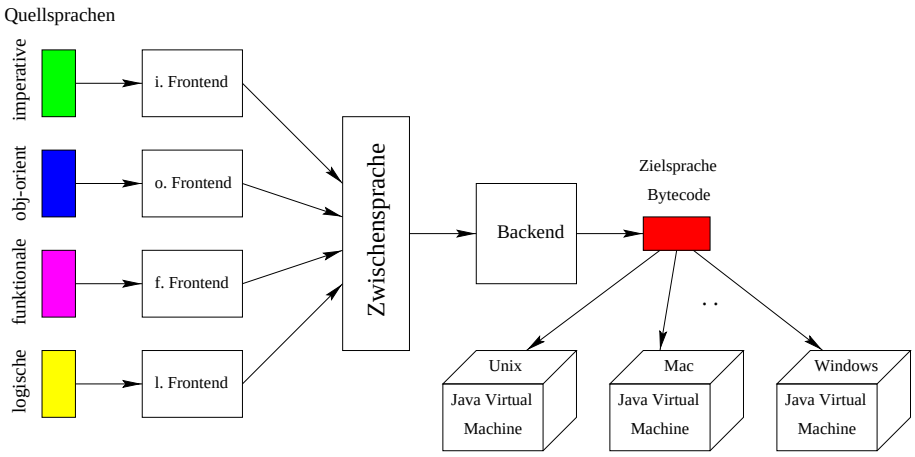

Abbildung 1.2: Compilerframework mit genereller Zwischensprache für verschiedene Paradigmen und Bytecode-Backend.

1.4 Erwartete Resultate

Von der Bearbeitung der Teilziele erwartet man die folgenden Resultate und Beantwortung folgender Fragen. Sie werden in Kapitel 9 rückblickend beantwortet.

- Es soll eine repräsentative Auswahl an Programmiersprachen getroffen werden, damit die Zwischensprache möglichst allgemein definiert werden kann. Wie sieht die Schnittmenge der untersuchten Programmiersprachen aus? Lassen sie sich in einer generellen Zwischensprache vereinigen oder müssen gewisse Elemente ausgeschlossen werden?

- Die zu definierende Zwischensprache ist Teil des Compilerbackends, das erstellt werden soll. Welche Form hat die Zwischensprache?

- Aus einem Zwischenprogramm soll Code für die JVM generiert werden. Ist die Wahl der JVM als Zielmaschine berechtigt? Wie wirkt sich die JVM als Zielmaschine auf den Bau eines Compilerbackends aus? Welche Vorteile und Nachteile ergeben sich daraus?

- Es soll gezeigt werden, wie der allgemeine Ansatz für ein offenes Framework zum Abbilden beliebiger Sprachen auf die Zwischensprache aussieht. Ist das Framework flexibel und wiederverwendbar?

- Ist die Problemstellung dieser Arbeit in die Praxis überführbar? Wo müssen Abstriche gemacht werden, was sind die Stärken des Ansatzes?

1.5 Vorgehen

Die vorliegende Diplomarbeit gliedert die Behandlung der zugrundeliegenden Problemstellung, die dafür nötige Theorie und die Beantwortung der Fragen in die folgenden Kapitel:

Kapitel 1 stellt die **Problemstellung** dieser Arbeit vor. Die erwarteten Resultate, die Teilziele und der Aufbau der Arbeit werden formuliert.

Kapitel 2 geht nach der Definition und einer Klassifizierung verschiedener **Programmiersprachen** genauer auf imperative (2.3), objektorientierte (2.4), funktionale (2.5) und logische (2.6) Konzepte ein. Eine Grammatik einer Beispielsprache wird für jedes Paradigma aufgestellt.

Kapitel 3 vergleicht die paradigmabezogenen Darstellungen und konstruiert eine **generelle Zwischensprache**, worauf die Paradigmasprachen abgebildet werden.

Kapitel 4 beschreibt die Zielmaschine des Compilers, die **virtuelle Maschine von Java**. Es werden vor allem die Grundlagen und die Punkte berücksichtigt, die für die Codegenerierung relevant sind.

Kapitel 5 stellt das Design und die Implementation eines Tools zur Analyse von für die JVM compilierten Programmen vor: **ClassFile-Analyzer**.

Kapitel 6 führt in die Grundlagen und Begriffe des **Compilerbaus** ein. Das Modell von Abbildung 1.1 wird in einzelne Phasen unterteilt und Möglichkeiten zur Optimierung aufgezeigt.

Kapitel 7 beschreibt das Design und die Implementation des Backends. Die **Codegenerierung** für die JVM wird detailliert betrachtet. Es wird ein Parser/Builder und ein ABCompiler entwickelt.

Kapitel 8 : Der praktische Teil der Arbeit wird an einem **Fallbeispiel** exemplarisch durchlaufen. Anhand eines Beispielprogrammes wird der Weg durch das Framework (Abbildung 1.2) Schritt für Schritt erläutert.

Kapitel 9 listet die erreichten Ziele auf. Es wird diskutiert, wie die Resultate dieser Arbeit erreicht wurden, welche offenen Probleme bestehen und wie weiterführende Arbeiten im behandelten Gebiet aussehen könnten.

Anhang A zeigt für jede Paradigmasprache die vollständige Grammatik und die abstrakte Syntax. Zusätzlich wird für die imperative Paradigmasprache die Implementation des Parsers aufgelistet. Auch die Grammatik der generellen Zwischensprache und von Gentle findet man im Anhang. Weiter die Zwischenprogramm- und die `class` File-Darstellung des Fallbeispiels.

Kapitel 2

Programmiersprachen

> Die Grammatik ist die Experimentalphysik der Sprachen.
>
> *Rivarol, Maximen & Gedanken*

Um eine möglichst allgemein verwendbare Zwischensprache zu generieren, müssen die verschiedenartigsten Programmiersprachen analysiert und ihr Einfluss auf eine solche Zwischensprache untersucht werden, siehe Abbildung 1.2. Dieses Kapitel liefert eine Definition von Programmiersprache und eine Klassifizierung einer Reihe von Sprachen nach Paradigmen. Dabei fallen zwei Gruppen auf, in die sich die meisten Programmiersprachen einteilen lassen: prozedurale und deklarative Sprachen. Die charakteristischen Eigenschaften von je zwei Vertretern werden bei der Konstruktion der generellen Zwischensprache berücksichtigt.

Nach den generellen Betrachtungen stellt jeder Abschnitt ein Paradigma und seine grundlegenden Konzepte vor, definiert eine Beispielsprache, die die elementaren Konstrukte ihres Paradigmas beinhaltet, sowie möglichst einfach und konfliktfrei ist, und leitet daraus eine abstrakte Syntax ab.

Eine Einteilung ist immer relativ, deshalb werden in dieser Arbeit Sprachen der bekanntesten Paradigmen analysiert. Die Geschichte der Programmiersprachen entwickelte sich von Maschinencode über Assemblersprachen zu höheren Sprachen, die zuerst prozedural waren. Das imperative Paradigma definiert die meisten grundlegenden Begriffe (Abschnitt 2.3). Darauf baut das objektorientierte Paradigma auf (Abschnitt 2.4). Die deklarativen Sprachen werden in funktionale, welche auf dem λ-Kalkül basieren (Abschnitt 2.5), und logische, welche auf der Hornlogik basieren (Abschnitt 2.6), unterteilt. Diese vier Zwischensprachen werden im Kapitel 3 analysiert und auf die generelle Zwischensprache abgebildet.

2.1 Generelle Definitionen

Die generellen Definitionen stützen sich, soweit nicht anders angegeben, auf die Schlagwörter im „Lexikon Informatik“ [Lex97], die „Encyclopedia of Computer Science“ [Ral92] und die einleitenden Kapitel von „Programming Language Essentials“ [Bal94].

2.1.1 Programmiersprachen

Eine **Programmiersprache** dient der Mitteilung von Aufgabenstellungen an Rechenanlagen. Sie sollte den Programmierer in der Handhabung und Reduktion der Komplexität der Aufgabe unterstützen, ohne die Ausdrucksweise einzuschränken. Deshalb verlangt eine Hochsprache

vom Benutzer keinerlei Wissen über den Maschinencode, bedient sich einer Notation, die einer natürlichen Sprache ähnlich ist und ist maschinenunabhängig.

Programmiersprachen vereinfachen eine Reihe von Problemen, da diese nicht (mehr) in Maschinencode formuliert werden müssen, aber sie stellen uns auch vor einige neue Probleme: Da Computer immer noch nur die Maschinensprache verstehen, müssen alle Programme, die in höheren Sprachen geschrieben sind, in diese Maschinensprache übersetzt werden. Dieser Vorgang wird im allgemeinen von Compilern durchgeführt.[1]

Ein weiteres Problem ist die Spezifikation der Sprache. Im Minimum muss definiert sein, welche Menge von Symbolen in gültigen Programmen verwendet werden kann, was die Menge der gültigen Programme ist und was die Bedeutung jedes gültigen Programms ist.

1. Die Definition der erlaubten Symbole ist relativ einfach: man kann das Vokabular auflisten, was der **lexikalische Struktur** der Sprache entspricht. Verschiedene Programmiersprachen sind sich in der lexikalischen Struktur ähnlich. Die vorkommenden Token oder lexikalische Einheiten fallen meist unter eine der Kategorien *Bezeichner, Operatoren, Literale, Schlüsselwörter, Separatoren, Kommentare* oder *Layout*. Beim **Parsen** eines eingegebenen Programmes werden die Token erkannt und strukturiert.

2. Die **Syntax** beschreibt die Erscheinung und Struktur der wohlgeformten Sätze einer Sprache, was den Programmen einer Programmiersprache entspricht. Im allgemeinen ist es üblich, die formal richtigen Programme durch eine Grammatik zu definieren, deren Regeln die gültigen Beziehungen zwischen den lexikalischen Elementen der Sprache angeben.

3. Die **Semantik** betrachtet die Sätze, interpretiert deren Komponenten und legt so die Bedeutung einer Sprache fest. Je nach Art der Interpretation unterscheidet man zwischen **interpretativer** (*operational*) Semantik, die alle maschineninternen Zustände während der Programmausführung beschreibt, **funktionaler** (*denotational*) Semantik, die die Abhängigkeit der Ausgabe von den Eingabedaten abbildet, und **axiomatischer** Semantik, die nur die Bedeutung eines Programms beschreibt und auf Zustände verzichtet. Die Beschreibung erfolgt in einem Modell oder Gegenstandsraum. So wird für die operationale Semantik eine ideale Maschine definiert, welche die Sprache interpretiert, indem die Bedeutung eines Programmes als Modifikation der Zustände beschrieben wird. Dies kann zum Beispiel mit einer SECD-Maschine (Stack, Environment, Control, Dump) geschehen [Thi94]. Für funktionale Semantik kann man eine Abbildung definieren, die einen Satz in einer allgemein verständlichen Sprache zu jedem Programm assoziiert. Diese Sprache kann zum Beispiel der Lambda Kalkül sein, womit dann ein äquivalentes Programm, also eines, das die gleiche Funktionalität definiert, formuliert wird.

   Einen anderen Ansatz wählen die Autoren von „The Theory of Parsing, Translation and Compiling“ [Aho72], die die Frage nach der Bedeutung ignorieren. Sie beschäftigen sich mit dem Bau eines effizienten Gerätes, das ein Quellprogramm in ein Zielprogramm überführt. Sie geben sich mit der Ausgabe dieses Compilers zufrieden und interpretieren die Bedeutung der Programme nicht. Die Beschreibung dessen was geschieht, wenn die Syntax abgearbeitet wird, wird in den untersuchten Sprachdefinitionen meist nur rudimentär angedeutet und ist kaum in Benutzerhandbüchern zu finden.

[1] Ein Interpreter dagegen ist ein Programm, das Befehle einer Programmiersprache lesen, analysieren und unmittelbar ausführen kann. Die Maschinenprogrammerfassung wird dabei nicht aufbewahrt. In dieser Arbeit werden Interpreter nicht behandelt.

Kompakte Definition von Programmiersprache

Eine Sprache ist charakterisiert durch die Menge der möglichen Wörter, aus denen nach den Regeln der Grammatik die formal richtigen Sätze, also Programme, gebildet werden. Die Interpretation der Programmiersprache ordnet den Wörtern und Sätzen der Sprache ihre Bedeutung im Gegenstandsraum zu, über den mit der Sprache Aussagen gemacht werden.

2.1.2 Paradigmen

Ein bewährter Weg, Programmiersprachen zu klassifizieren, ist anhand der Paradigmen, die sie verkörpern [Bal94]. Im Griechischen hatte das Wort 'paradigma' die Bedeutung von 'Muster' oder 'Beispiel'. Und da das Lehren häufig durch Vorzeigen von Beispielen geschieht, änderte die Bedeutung zu 'bestimmtes Beispiel aus der Lehre'.

In der Informatik ist ein **Paradigma** eine kohärente Menge von Methoden, die einen Problembereich effektiv behandeln. Ein Paradigma kann üblicherweise durch ein einfach zu formulierendes Prinzip charakterisiert werden. Natürlich ist eine solche Definition vereinfachend, aber genau darum lassen sich Paradigmen gut als Oberbegriffe verwenden, die in ihrer Bedeutung alle nötigen Konzepte und Techniken einschliessen. Für den Problembereich der Programmiersprachen wurde eine Menge von Paradigmen entwickelt. Die bekanntesten davon werden in dieser Arbeit untersucht: Imperative, Objektorientierte, Funktionale und logische Programmiersprachen.

Eine nicht-vollständige Aufzählung von Programmiersprachen in Tabelle 2.1 soll einen Überblick und somit eine Auswahlmenge für die kommenden Sprachanalysen bieten. Eine breit angelegte Aufzählung von Sprachen findet sich in „ Programming Languages: History and Fundamentals“ [Sam69].

Die häufigen Mehrfachzuweisungen deuten auf die Schwierigkeit hin, Programmiersprachen durch Paradigmen zu klassifizieren. Es lässt sich vor allem bei neueren Sprachen oder Versionen ein Trend zu objektorientierten Elementen und teilweise funktionalen Eigenschaften ausmachen. Diese Mehrdeutigkeiten könnten sich positiv auf eine allgemeine (Zwischen-)sprache auswirken.

2.1.3 Polymorphismus

Bei der Diskussion des Typensystems von Programmiersprachen fällt oft der Begriff „Polymorphismus“, der hier basierend auf dem Standardwerk „On Understanding Types, Data Abstraction and Polymorphism“ von Cardelli und Wegner eingeführt wird [Car85].

In **monomorphen** Sprachen hat jeder Wert und jede Variable ganau einen bestimmten Typ, im Gegensatz zu **polymorphen** Sprachen, wo einige Werte oder Variablen mehr als einen Typ haben dürfen. **Polymorphismus** wird die Fähigkeit von Variablen genannt, verschiedene Formen anzunehmen. Cardelli und Wegner klassifizieren vier Arten von Polymorphismus, diese lassen sich in zwei Kategorien unterteilen. **Universal** polymorphe Funktionen arbeiten mit einer beliebigen Anzahl von Typen, die alle eine gemeinsame Struktur aufweisen. Die Implementation führt für jeden Argumenttyp den genau gleichen Code aus. **Ad hoc** polymorphe Funktionen arbeiten mit einer endlichen Menge von verschiedenen und möglicherweise nicht verbundenen Typen und führen für jeden Argumenttyp anderen Code aus:

- **Universal parametric**: Eine Funktion hat einen (impliziten oder expliziten) Typparameter, welcher den Typ der Funktion bei jeder Verwendung neu bestimmt. Ein Objekt kann in verschiedenen Typumgebungen uniform verwendet werden.

	Sprachparadigmen						
Programmiersprache	Imperativ	Objektorientiert	Funktional	Logisch	Skript	Note	Referenzen
Ada	√	√				OO: Ada95	[Goo83, Bar96]
Algol 60	√						[Nau60, Dij62]
C	√						[Ker78]
C++	√	√				hybrid	[Ell95]
CLOS		√	√				[Pae93]
Cobol	√						[USG65]
Eiffel		√					[Mey88, Mey92]
Fortran	√						[Bac57]
Gentle	√		√			a)	[Wai89, Sch97]
Gofer			√			pure	[Jon91]
Haskell			√			pure	[Hud89, Hud92]
Java		√					[Gos96]
Latex					√		[Kop88]
Lisp	√	b)	√			non-pure	[Ber66]
ML	√		√			non-pure	[Mil90]
Modula		√					[Car92]
Oberon		√					[Rei91]
Pascal	√						[Wir75]
Perl		√			√	OO: V5.0	[Wal90]
PL/I	√						[IBM66]
Prolog				√			[Sic96]
Python		√			√		[Ros96, Lut96]
Scheme		√	√			non-pure	[Dyb87]
Self		√					[Hol91]
Simula		√					[Bir73]
Smalltalk		√	√			non-pure	[Gol89]
TCL					√		[Wel97]
Unix Shell					√		[Bou78]

a b

[a] Functional Compiler Description Language plus Patternmatching.
[b] Neuere Lisp sind objektorientiert

Tabelle 2.1: Klassifikation von Programmiersprachen nach Paradigmen.

- **Universal inclusion**: Ein Objekt kann zu beliebig vielen Klassen gehören, die möglicherweise subtyped sind. Dadurch kann eine Variable auf Objekte verschiedener Klassen verweisen, die konkrete Klasse wird dynamisch gebunden.
- **Ad hoc overloading**: Der gleiche Variablenname wird zur Bezeichnung verschiedener Funktionen benutzt, im Kontext wird entschieden, welche Instanz des überladenen Namens ausgeführt wird.
- **Ad hoc coercion**: Argumente werden (statisch oder dynamisch) zu dem Typ konvertiert, den die Funktion erwartet. Sonst resultiert ein Typfehler.

Wie schon der Namenswahl anzumerken ist, betrachten Cardelli und Wegner die ad hoc Formen nicht als wahren Polymorphismus. Bei der Untersuchung der Programmiersprachpradigmen wird sich zeigen, wie Polymorphismus interpretiert wird.

2.2 Das Compilerkonstruktionssystem Gentle

Bei der Analyse der verschiedenen Programmiersprachen und der Entwicklung einer generell verwendbaren abstrakten Syntax, wird das Compilerkonstruktionssystem **Gentle** verwendet. Da die restlichen Abschnitte des Kapitels mit Gentlecode illustriert sind, wird Gentle an dieser Stelle etwas näher vorgestellt.

Gentle wurde am GMD Lab in Karlsruhe von Friedrich Wilhelm Schröer entwickelt und in „Three Compiler Specifications“ [Wai89] zum ersten Mal beschrieben. Die folgenden Angaben stammen aus dem Handbuch für Gentle, „The GENTLE Compiler Construction System“ [Sch97], das sich auch online auf der Website von GMD First befindet. Dort ist ebenfalls eine für nichtkommerzielle Zwecke frei erhältliche Version des Gentle Compilerkonstruktionssystems verfügbar:
`http://www.first.gmd.de/gentle/distribution.html`

2.2.1 Prinzipien

Gentle ist eine deklarative Programmiersprache in der Tradition von logischen Sprachen wie Prolog [Sic96]. Programme werden als Regeln geschrieben, die entweder als Grammatikregeln, als logische Anweisungen oder als Prozeduren interpretiert werden können. Diese Regeln werden mit Pattern Matching angewendet; Gentle ist auch eine funktionale Sprache.

Gentle basiert auf den Prinzipien *rekursive Definition* und *strukturelle Induktion*: Die Definition von Input und internen Datenstrukturen geschieht durch Auflistung von Alternativen (strukturbildende Operationen), die angeben, wie die zu definierenden Elemente aus gegebenen Bestandteilen aufgebaut sind. Die Regeln für mögliche Alternativen folgen rekursiv der Struktur der Elemente, indem sie deren Bestandteile abarbeiten. Dabei ist das Prinzip der *Lokalität* wichtig, womit man komplexe Interaktionen vermeiden und das System verstehen kann, indem man kleine, isolierte Teile des Systems versteht. Somit leitet Gentle zu *datenorientierter Methodik* an, das heisst, die Struktur der Daten wird durch die Struktur der Algorithmen gespiegelt.

2.2.2 Anwendung

Gentle deckt in einer einheitlichen Notation das ganze Spektrum des Compilerbaus ab, von der Analyse über die Transformation bis zur Synthese. Gentle bietet ein einheitliches, integriertes Framework für die folgenden Aufgaben an: Spracherkennung, Definition von abstrakten Syntaxbäumen, Baumtraversierung, Source-to-Source Übersetzung und Codegenerierung.

Eine Spezifikation in Gentle wird automatisch in portablen und effizienten Lex, Yacc und C-Code übersetzt. Der Benutzer muss nicht auf diesem Level arbeiten und kann die Implementationsdetails Gentle überlassen.Trotzdem können bei Bedarf in C geschriebene zusätzliche Teile eines Compilers in Gentle eingebunden werden. Es handelt sich daher bei Gentle um eine offene Sprache.

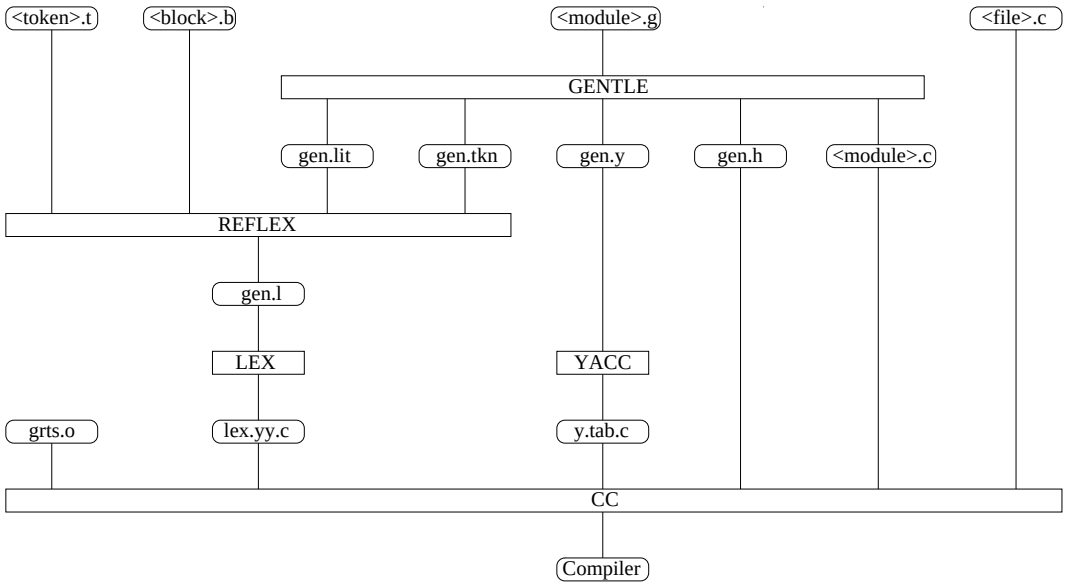

Abbildung 2.1: Files, Programme und deren Zusammenarbeit bei der Compilergenerierung. Abbildung nach „The GENTLE Compiler Construction System, Getting Started“ [Sch97].

Praktisch setzt sich ein Projekt, das mit Gentle spezifiziert wird, wie in Abbildung 2.1 zusammen. Der Gentle Compiler wird für jedes Gentle Modul oder File **<module>.g** aufgerufen und übersetzt diese in C Files. Zusätzlich werden folgende Files für die Grammatikspezifikation generiert: **gen.lit** enthält Lex Spezifikationen für terminale Symbole. **gen.tkn** enthält eine Liste der Token. **gen.h** ist ein C Header, der die Datentypen für Attribute und Code der Token einführt. **gen.y** ist eine Yacc Spezifikation der Ausgangsgrammatik [Joh75]. Nun wird mit dem Programm Reflex eine Lex Spezifikation produziert [Les75]. Dazu werden die Files **<token>.t** **<block>.b** verwendet, die die Beschreibung der Token enthalten. Lex und Yacc werden zur Generation des Scanners und des Parsers benutzt. Schliesslich übersetzt der C Compiler die generierten und die vom Benutzer geschriebenen C Module in ein ausführbares Programm.

2.2.3 Syntax und Semantik von Gentle

Im Anhang A.1 auf Seite 111 ist die vollständige Syntax von Gentle dargestellt. Im folgenden Abschnitt werden ein paar der wichtigsten Spezifikationen aufgezeigt. Dabei gelten folgende Konventionen:

- Die Syntax wird in Backus Naur Form dargestellt.
- Terminale Symbole werden in Anführungszeichen eingeschlossen.
- **Ident** bezeichnet eine Folge von Buchstaben oder Zahlen, die mit einem Buchstaben beginnen muss.

- Per Definition beginnen Variablen mit Grossbuchstaben und Funktoren mit Kleinbuchstaben.
- Kommentare beginnen mit **„- -“** und enden am Zeilenende.

Eine Gentle Spezifikation ist eine Liste von Deklarationen. Eine Deklaration führt einen Typ, ein Prädikat, eine Kontextvariable oder eine Tabelle ein.

```
Declaration = TypeDecl | PredicateDecl | VariableDecl | TableDecl .
```

Eine Typen-Deklaration definiert einen Typ und eine Menge von Werten dieses Typs. Dabei werden der Typname und alle möglichen Alternativen festgelegt. Diese Werte werden Terme genannt und durch Anwendung eines Funktors auf Argumente konstruiert. Ein Funktor besteht aus einer Konstanten, gefolgt von keinem oder mehreren Argumenten, die wiederum Terme sind. Die Argumente können mit einem vorangestellten Bezeichner dokumentiert werden. Falls eine Alternative aus einem Funktor mit Argumenten besteht, werden die Typen dieser Argumente aufgeführt.

Eine Typen-Deklaration ohne Funktoren führt einen abstrakten Typ ein. Die Werte von abstrakten Typen werden nicht in Gentlefiles spezifiziert sondern üblicherweise in Files der Art „token.t“.

```
TypeDecl = "'type'" Ident [ ["="]  { TermSpec } ] .
TermSpec = IdentLC [ "(" { ParamSpec } ")" ] .
ParamSpec = [ Ident ":" ] Ident .
```

Eine Prädikat-Deklaration definiert Nonterminals, Token, Aktionen und Bedingungen. Dabei wird ein Prädikat mit Ein- und Ausgabeparametern (getrennt durch einen Pfeil) unter Angabe deren Typen eingeführt und einer Kategorie zugewiesen. Die Regeln werden mit den Eingabewerten getestet (Pattern Matching) und der Body der ersten passenden Regel ausgeführt. Falls das Resultat eines Members falsch ist (= „fail“), lässt sich die dazugehörende Regel nicht anwenden und die folgenden Regeln werden bearbeitet. Falls die Resultate aller Members richtig sind, werden die Ausdrücke dieser Regel evaluiert, was die Ausgabewerte liefert.

```
PredicateDeclaration = Category Ident Signature Rules .
Category = "'nonterm'" | "'token'" | "'action'"
       | "'condition'" | "'choice'" | "'sweep'" .
Signature =  [ "(" { ParamSpec }  [ "->" { ParamSpec }  ] ")" ] .
Rules = { "'rule'" Head [":"] Body ["."] } .
Head = Ident [ "(" { Pattern } [ "->" [ { Expression} ] ")" ] .
```

Je nach Kontext in dem Terme auftreten, werden sie als **expression** oder **pattern** bezeichnet. Mit Expressions werden Terme aus gegebenen Bestandteilen konstruiert, wogegen ein Term in seine Komponenten aufgeteilt werden kann, indem sein Wert an einem Pattern gematched wird.

Kontextvariablen und -Tabellen halten Werte eines bestimmten Typs fest und werden in Kontextabfragen abgefragt.

Positionsangabe

Neben vordefinierten Prädikaten wie **print** zur Ausgabe von Werten und Relationen bietet Gentle auch ein Konstrukt (**@**) an, das die Koordinaten im Quelltext angibt. Diese Werte sind vom vordefinierten Typ **POS**.

Die Positionsangabe kann nach einem Nonterminal (Abbildung 2.2, 2.Regel) oder Token (Abbildung 2.2, 1.Regel) stehen und definiert ihren Ausgabewert als die Koordinate des Quelltexts des vorangegangenen Symbols. Sie kann auch am Anfang einer Grammatikregel stehen und definiert dann die Koordinate des Quelltexts, die vom Body dieser Regel erkannt wird. Diese Koordinate ist diejenige des Tokens oder des Nonterminals ganz links im Body. Falls der Body leer ist, wird der aktuelle Koordinatenwert verwendet.

```
'rule' Statement(-> stmt(P, E, S)) :
       "IF" @(->P) Expression(-> E)  "THEN" Statement(-> S)

'rule' Statement(-> stmt(P, E, S)) :
       "IF" Expression(-> E)  @(->P) "THEN" Statement(-> S)
```

Abbildung 2.2: Positionsangaben in Gentle Regeln.

Im obigen Beispiel wird in der ersten Regel „P“ als Koordinate von „IF“ und in der zweiten Regel als Koordinate der „Expression“ definiert. In beiden Fällen wird „P“ an den Funktor „stmt“ weitergegeben, dessen erstes Argument vom Typ „POS“ sein muss.

Ein Compiler, der die lexikalische Analyse, das Parsing und die sematische Analyse in einem Durchgang durchführt, kann eine Fehlerausgabe an den Benutzer mit der laufenden Position des Lexers ergänzen, da diese meist eine sinnvolle Annäherung an die wirkliche Position des Fehlers im Quellprogramm ist.

Dagegen hat ein Compiler, der abstrakte Syntaxbäume als Zwischendarstellung verwendet und pro Durchgang nur eine Phase behandelt, einen Nachteil bei der Konstruktion von semantischen Fehlernachrichten. Die Position des Lexers kann nicht mehr verwendet werden, da dieser bereits am Ende des Files angekommen ist. So muss die Position im Quellprogramm für jeden Knoten des AST gespeichert werden, für den Fall, dass ein solcher Knoten einen semantischen Fehler enthält. Daher werden die Datenstrukturen der abstrakten Syntax um Positionsfelder ergänzt. Es können alle Strukturen eine Positionsangabe haben, oder nur die Blätter des AST, dabei wird die Position der Knoten von den Positionen ihrer Subbäume abgeleitet.

2.2.4 Motivation für die Verwendung von Gentle

Gentle ist für das Erstellen eines Prototyps sehr geeignet, da das System die lexikalische Analyse, den konkreten Parser und die Übersetzung in lauffähigen C-Code automatisch vornimmt. Mit der Angabe einer Grammatik ist das Frontend eines Compilers rasch konstruiert.

Weiter erlaubt es Gentle, mit einfachen Mitteln eine Spezifikation einer Grammatik zu definieren. Es reicht, eine vorgegebene Syntax einer Sprache in Typen, Symbole und Regeln zu übersetzen, dadurch wird ein Quellprogramm in seine syntaktischen Einzelheiten geparst. Falls keine Fehler vorkommen erhält man als Ausgabe Zwischencode, der aus den Typen und Funktoren zusammengesetzt ist, oder aufgrund von semantischen Aktionen wird Code produziert. Man kann nun die Typen und Funktoren als abstrakte Syntax betrachten, die zur vorgegebenen Grammatik und deren konkreten Syntax gehört. Neben den abstrakten Typen und den Funktoren sind die folgenden Begriffe wichtig und werden mit der Definition von Gentle verwendet: Token, Terminal, Nonterminal.

Die paradigmaspezifischen Frontends der nächsten Kapitel werden mit Gentle implementiert. Die resultierende Darstellung mit Typen und Funktoren entspricht der abstrakten Syntax der Zwischensprachen.

2.3 Imperative Programmiersprachen

Imperative Programmiersprachen sind von den hier untersuchten Sprachen der Maschinensprache am ähnlichsten. So muss ein imperatives Programm explizit Initialisierung, Speicherzuweisung und Speicherbereinigung vornehmen. Diese Sprachen sind nahe der Hardware (von-Neumann-Rechner) konzipiert, ihr wichtigstes Element ist die Änderung der Daten im Speicher durch Zuweisungen.

2.3.1 Prinzipien

Charakteristisch für imperative Sprachen ist die Formulierung eines Problems als eine Serie von diskreten Anweisungen oder Schritten, die das Programm bilden. Die Ausführung erfolgt sequentiell von Schritt zu Schritt, also in einem singulären Kontrollfluss, der angibt, *Wie* ein Problem gelöst wird, im Gegensatz zu nichtprozeduralen oder deklarativen Sprachen, die angeben, *Was* gelöst wird. Bei imperativen Sprachen liegt das Gewicht auf der Operationsweise, das *Wie* überdeckt das *Was*.

Imperative Sprachen entsprechen dem grundlegenden Modell der Datenverarbeitung, der Turing-Maschine, mit je einem Band für die Eingabe, die Ausgabe und die Systemzustände, also einem Keller-Automaten. Die dabei verwendeten Informationsträger, die Daten, müssen mit einer Datendeklaration spezifiziert werden. Im Gegensatz zu Maschinen- und Assemblersprachen greifen imperative Sprachen nicht direkt auf Daten im Speicher zu, sondern durch Variablen, welche Zahlen und Werte von vordefinierten Typen repräsentieren. Die Manipulation der Daten wird vollständig von statusverändernden Befehlen, den Anweisungen, kontrolliert. Daten können mit Typkonstruktoren in abgeleiteten Datentypen gruppiert werden.

In der „Encyclopedia of Computer Science“ [Ral92] werden generelle höherstufige Programmiersprachen wie C [Ker78], FORTRAN [Bac57], PASCAL [Wir75] und PL/I [IBM66] auch *prozedurale* Sprachen genannt, da dort der Gebrauch von Prozeduren zentral ist. Schon früh in der Entwicklung der Programmiersprache wurde erkannt, dass Programme an verschiedenen Stellen den gleichen Prozess ausführen. Durch Anwenden des Abstraktionsprinzips (*Abstraction Principle* [Sch94]) kann der erkannte Prozess benannt und als Prozedur oder Unterprogramm definiert werden. Diese enthalten den Code eines Prozesses an einer einzigen Stelle im Programm und können beliebig oft aufgerufen werden.

2.3.2 Konzepte

Nun folgt die Beschreibung einiger Konzepte, die in den meisten imperativen Sprachen zur Realisierung der Prinzipien verwendet werden:

- Sämtliche Deklarationen und Anweisungen müssen als **Programm** gruppiert werden, das effektiv ausgeführt wird. Imperative Sprachen verschachteln den Kontrollfluss strikt: Es ist jederzeit genau eine Routine aktiv, die von genau einer anderen Routine aufgerufen wurde und die im Moment angehalten ist. Diese Beziehungen unter den Routinen werden im Systemstack gespeichert, der lokale Daten und Linkinformationen für den Routinenaufruf enthält. Eine Routine hat nur solange aktuelle Daten, wie sie aktiv im Stack ist. Sie kann jedoch auch mit globalen Werten arbeiten.

- Eine Reihe von Anweisungen, die prinzipiell zusammengehören und vom restlichen Programm abstrahiert werden können, werden sinnvollerweise als **Unterprogramm** (routine, subprogram) eingekapselt und am ursprünglichen Ort mit einem **Aufruf** (routine invocation, procedure call) ersetzt. Ein Aufruf identifiziert das Unterprogramm mit seinem Namen, übergibt die aktuellen Parameter, leitet den Kontrollfluss zum Anfang

des Unterprogrammes, wartet bis dieses beendet ist und führt danach den Kontrollfluss zurück zu der Anweisung nach diesem Aufruf. **Rekursive Aufrufe** erlauben Unterprogrammen, sich selbst aufzurufen. Man unterscheidet zwei Arten von Unterprogrammen, nämlich *Funktionen*, die einen Wert berechnen, der mit einer speziellen Anweisung (*return*) an den Aufruf zurückgegeben wird. Dieser muss als Operand in einem Ausdruck stehen, der den Rückgabewert verwendet. Unterprogramme ohne Rückgabewert heissen *Prozeduren*, sie verändern einen äusseren Zustand. *Operatoren* sind eine andere Notationsart von Funktionen. **Libraries** oder eingebaute Funktionen werden von den meisten Sprachen angeboten und können via einen Referenznamen wie Unterprogramme aufgerufen werden. Sie sind bereits übersetzt und werden vom Linker/Loader beim Compilieren automatisch an das Benutzerprogramm gebunden.

- Daten werden spezifiert, indem jede Dateneinheit eine Darstellung und eine Anzahl von Eigenschaften, nämlich ihren Typ, und einen bezeichnenden Namen zugewiesen erhält. Diese Deklaration verbindet die Dateneinheit in der Maschine mit der Struktur und dem Namen auf der Programmebene. Die Dateneinheit kann durch Maschinenbefehle verändert werden und wird im folgenden **Variable** genannt. Eine Initialisierung der Variablen bei der Deklaration ist in den meisten Sprachen möglich. Uninitalisierte Variablen haben einen zufälligen oder unbestimmten Wert, der nicht interpretierbar ist. **Konstanten** können nach ihrer Initialisierung nicht mehr verändert werden, der Compiler verbietet jegliche Modifikationen. Sie werden mit dem Programm übersetzt und im Programmspeicher als Textsegment ohne Schreibrechte gespeichert.

- Ein **Datentyp** legt eine Menge von Werten und eine Menge von darauf zugeschnittenen Operationen fest. Imperative Sprachen enthalten meist die vordefinierten Standardtypen *Real*, *Integer* und *Character*, die man zum Beispiel in Pascal [Wir75] als Teilbereichstypen noch genauer spezifizieren kann (*short, long, signed, unsigned*). Mit Typkonstruktoren können aus diesen atomaren Typen strukurierte oder zusammengesetzte Typen gebildet werden. So sind *Records*, die Komponenten aus unterschiedlichen Datentypen gruppieren, *Arrays*, bei denen mit einem Index auf Elemente vom gleichen Typ zugegriffen wird, und aufzählende Typen wie *Boolean* konstruierbar.

- **Strong Typing**, die explizite Typendeklaration der Variablen muss vor deren Verwendung geschehen und lässt auch lokale Variablen zu, die einen beschränkten Gültigkeitsbereich haben. Ein **Typ** ist der Wertebereich einer Menge von Operationen.

- Ein Programm ändert den internen Maschinenstatus, das heisst die Variablenwerte, und den externen Status, der in Input/Output-Geräten gespeichert ist. Typische Konstrukte dazu sind imperative **Anweisungen** (statement) wie Wertzuweisung, Laufanweisung, bedingte Anweisung, deklarative Vereinbarungen und Ein-/Ausgabeanweisung. Dies sind die elementaren Sätze einer Sprache, sie entsprechen jeweils einem Programmschritt.

- Die Verwendung von **Blockstrukturen** (compound) erlaubt die Unterteilung eines Programmes in Blöcke, die auch verschachtelt werden dürfen. Ein Block von Anweisungen kann entweder mit Schlüsselwortpaaren wie „Begin - End“, „If - Then - Endif“ oder geschweiften Klammern „{ }“ als Einheit angegeben werden. Eine Blockstruktur definiert den Gültigkeitsbereich von allen Bezeichnern, die innerhalb deklariert werden. Diese Variablen sind nur in diesem Block gültig und können, nachdem der Kontrollfluss den Block verlassen hat, nicht mehr verwendet werden.

- Das wichtigste Element bei imperativen Sprachen ist die **Zuweisung** (assignment). Diese Anweisung evaluiert den Ausdruck rechts des Zuweisungssymbols, indem die angegebenen

Operationen ausgeführt werden. Danach wird der alte Wert der Variable auf der linken Seite des Symbols mit dem erhaltenen Resultat überschrieben. Diese Variable muss ein einfacher Wert sein, sie kann nicht eine zusammengesetzte Form wie in einer mathematischen Gleichung annehmen. Die Zuweisung ist die einzige Anweisung, die den Speicher verändert und dadurch eine neue Ausgangslage für den nächsten Schritt vorgibt.

- **Output**-Anweisungen verändern den externen Status mit Hilfe des internen Status und **Input**-Anweisungen benutzen den externen Status zur Veränderung des internen Status. Beide Arten müssen mit der wachsenden Anzahl von Geräten standhalten und müssen je nach Anwendung und verwendeten Daten konvertiert werden. Die meisten Sprachen bieten dafür in Libraries vordefinierte überladene Funktionen an.

- Eine **Selektion** oder bedingte Anweisung wird verwendet, wenn nicht einfach die textuell nächste Anweisung ausgeführt werden soll. Auf den einfachen Vergleichsoperationen der Maschinensprache aufbauend, bieten imperative Sprachen Anweisungen an, die Tests und Entscheidungen als syntaktische Konstrukte formulieren. Dabei wird grundsätzlich von einer Testbedingung ausgegangen, die als Vergleich mit zwei möglichen Resultaten (wahr oder falsch) formuliert wird. Je nach Resultat wird eine der zwei alternativen Aktionen ausgeführt und die andere ignoriert. Beide Aktionswege dürfen beliebig lang oder leer sein. Eine andere Form der Selektion als das eben beschriebene *if then else* ist die *switch/case*-Anweisung. Hier wird aufgrund einer Integerzahl oder eines Aufzählungswertes aus mehreren Codeteilen einer ausgewählt.

- **Schleifen** (loops) sind Anweisungen, mit denen wiederholte Operationen spezifiziert und kontrolliert werden. Üblicherweise kann der Beginn und das Ende einer Schleife, die Anzahl der Wiederholungen und ein Zähler angegeben werden. Darauf werden die zur *FOR-Schleife* gehörenden Anweisungen solange wiederholt, bis eine endliche, vorberechnete Anzahl abgearbeitet ist. Die Zählervariable darf innerhalb der Schleife verwendet werden. Bei der *WHILE-Schleife* wird der Schleifenkörper solange durchlaufen, wie ein bestimmtes Kriterium erfüllt ist, aber unabhängig von der Anzahl Schleifen. Die *REPEAT-Schleife* führt ihre Anweisungen mindestens einmal aus und prüft anschliessend ihre Abbruchbedingung. Solange diese nicht erfüllt ist, wird wiederholt.

- Ein **Ausdruck** (expression) besteht aus einer Kombination von Termen und Operatoren, die meist nach eingeschränkten Regeln der Algebra konstruiert werden. Dabei ist besonders wichtig, dass jede arithmetische Operation explizit angegeben wird. Ein implizite Schreibweise wie die mathematische Multiplikation ist nicht erlaubt. Auch müssen die Ausdrücke linear dargestellt werden, die Reihenfolge ihrer Auswertung folgt aus der Definition der Bindung zwischen den einzelnen Elementen. Den Operatoren werden Prioritäten zugewiesenen. Dabei haben Operatoren mit höherer Priorität eine stärke Bindung und Vorrang vor Operatoren mit tieferer Priorität. Die Reihenfolge der Auswertung von Operatoren der gleichen Priorität wird durch ihre Assoziativität bestimmt.

Nachdem das Paradigma imperativer Sprachen umrissen ist, wird in den folgenden Abschnitten eine Beispielsprache aufgestellt, die über ein Frontend und die generelle Zwischensprache mit dem Backend nach Bytecode übersetzt werden kann.

2.3.3 Grammatik einer imperativen Beispielsprache

Ausgehend von den Prinzipien und Konzepten des imperativen Paradigmas und der Beispielsprache MINILAX in „Three Compiler Specifications“ [Wai89] wird nun eine imperative Beispielsprache vorgestellt. Dieser Abschnitt definiert die Syntax dieser Sprache als kontextfreie

Grammatik in erweiterter Backus-Naur-Form[2], der vollständige Parser für diese Sprache ist als Gentlecode im Anhang A.2 zu finden.

Notation

Diese Grammatik ist eine Menge von hierarchisch aufgebauten Produktionen, die beschreiben, wie syntaktisch korrekte Programme geformt werden. Jede Produktionsregel definiert ein Nonterminal, beginnend mit seinem Namen, gefolgt vom Definitionszeichen „::=“. Rechts davon folgen eine oder mehrere Alternativen, wie dieses Nonterminal konstruiert werden kann. Diese Alternativen werden durch einen senkrechten Strich „|“ voneinander getrennt. Nonterminals werden in spitze Klammern „<“, „>“ eingeschlossen, Terminale stehen ohne Klammern, Schlüsselwörter werden „GROSS“ geschrieben. Doppelte eckige Klammern „[[ ]]“ umfassen optionale Konstrukte. Alles was in doppelt geschweiften Klammern „{{ }}“ steht, kann null oder beliebig oft wiederholt werden. Kommentare beginnen mit zwei Querstrichen „−−“ und sind bis zum Ende der Linie gültig.

Terminale und Schlüsselwörter sind die lexikalischen Zeichen, die die Sprachen bilden. Die terminalen Elemente dieser Grammatik sind ganze oder reelle Zahlen, „Integer“ respektive „Double“, und Bezeichner „Ident“.

Syntax

Das Startsymbol dieser Grammatik ist <Program>, eine syntaktisch korrekte Eingabe besteht aus genau einem Programm. Nach dem Programmbezeichner folgt der Deklarationsteil, der aus beliebig vielen Deklarationen bestehen kann, danach der Anweisungsteil, dieser darf eine beliebige Anzahl von Anweisungen enthalten. Die Bestandteile des Programmes werden durch Schlüsselwörter abgetrennt. Das Programmende wird mit einem Punkt angegeben.

```
<Program> ::= PROGRAM Ident ; DECLARE <Declaration> BEGIN <Statement> END.
```

Deklarationssyntax

Die Spezifikation der Dateneinheiten geschieht im Deklarationsteil eines Programms oder eines Unterprogramms unter Angabe des Variablennamens, gefolgt von einem Doppelpunkt „:“ und dem Datentyp. Dies alles muss für jede Variable aufgelistet werden, Mehrfachdeklarationen sind nicht möglich. Konstanten könnten mit einem zusätzlichen Schlüsselwort analog deklariert werden.

Im Deklarationsteil werden auch Unterprogramme aufgeführt. Nach dem Bezeichner stehen in Klammern die formalen Parameter, die zwei Arten von Werten erwarten. Erstens Werteparameter, die nur zur Berechnung im Unterprogramm verwendet werden und deren Wert im aufrufenden Programm nicht ändert (call-by-value - Übergabe). Zweitens Variablen, mit „VAR“ gekennzeichnet, die nach Beenden des Unterprogrammes die Rückgabewerte enthalten, also tatsächlich verändert werden (call-by-reference - Übergabe). Der Anweisungsblock des Unterprogramms wird, wie der des Hauptprogramms, zwischen „BEGIN“ und „END“ angegeben.

```
<Declaration> ::= <Declaration> ; <Declaration>       -- several declarations
                | Ident : <Type>                      -- variable declaration
                | PROCEDURE Ident [[ ( <FormalParam> ) ]] ;
                  DECLARE <Declaration>
                  BEGIN <Statement> END               -- procedure declaration
```

[2] **Backus-Naur-Form**, eine Notationsform für Ersetzungsregeln kontextfreier Grammatiken zur Definition der Syntax von Programmiersprachen. Entspricht im Aufbau kontextfreien Chomsky-Grammatiken. [Lex97]

```
<FormalParam> ::= <FormalParam> ; <FormalParam>       -- formal parameters
                | VAR Ident : <Type>                  -- variable parameter
                | Ident : <Type>                      -- value parameter
```

Als Datentypen können die Standardtypen „INTEGER“, „REAL“ und „CHARACTER“ in der einfachen Form ohne Bereichsangabe verwendet werden. Der zusammengesetzte Datentyp „Array“ und die booleschen Wahrheitswerte werden den Standardtypen gleichgesetzt.

```
<Type> ::= INTEGER
         | REAL
         | CHARACTER
         | BOOLEAN
         | ARRAY [ <Expression> .. <Expression> ] OF <Type>
```

Anweisungssyntax

Die Anweisungen, die im Haupt- oder in Unterprogrammen vorkommen können, sind:

Die **Zuweisung** (assignment) setzt sich aus einer Variable, gefolgt vom Zuweisungssymbols „:=“ und einem auszuwertenden Ausdruck zusammen. Im Hinblick auf eine generelle Syntax ist die Verwendung von beliebigen Ausdrücken anstelle der Variable erlaubt.

Ein **Block** (compound) ist die kleinste Gruppierungseinheit der Grammatik und erlaubt der Syntax, mehrere Anweisungen an der Stelle einer einzelnen Anweisung zu verwenden.

Der **Aufruf** von Unterprogrammen erfordert die Angabe des Programmbezeichners und der erforderlichen aktuellen Parameter. Mit dieser syntaktischen Form können Funktionen, Prozeduren und eingebaute Libraries sowohl normal als auch rekursiv aufgerufen werden.

Input/Output: Diese Grammatik unterstützt die zwei grundlegenden Operationen zum Lesen von Input und Schreiben von Output. Die als Parameter verwendeten Werte werden mit beliebigen Ausdrücken gebildet.

Die Grammatik beschränkt sich auf die einfache **Selektion**. Die Bedingung ist ein beliebiger Ausdruck, die erste Alternative eine einfache oder Blockanweisung. Die zweite Alternative kann leer sein oder eine beliebige Anweisung enthalten. Die Selektion wird mit „ENDIF“ abgeschlossen.

In der **WHILE-** und der **REPEAT-Schleife** steht jeweils ein Ausdruck als Bedingung und eine Anweisung als Schleifenkörper. Bei der **FOR-Schleife** wird die Laufvariable, deren Anfangs-, Schlusswert und Schrittweite zur Berechnung der Bedingung angegeben. Es können beliebige Ausdrücke dafür verwendet werden, die durch die Schlüsselwörter abgetrennt werden.

```
<Statement> ::= <Statement> ; <Statement>             -- several statements
              | <Expression> := <Expression>          -- assignment
              | { <Statement> }                       -- compound statement
              | Ident ( <Expression> )                -- procedure call
              | READ ( <Variable> )                   -- read-in
              | WRITE ( <Expression> )                -- write-out
              | IF <Expression> THEN <Statement>
                [[ ELSE <Statement> ]] ENDIF          -- selection
              | WHILE <Expression> DO <Statement>     -- while loop
              | REPEAT <Statement>
                UNTIL <Expression>                    -- repeat loop
              | FOR <Variable> := <Expression>
                TO <Expression> STEP <Expression>
                DO <Statement>                        -- for loop
```

Ausdruckssyntax

In dieser imperativen Sprache können alle gängigen mathematischen Operationen verwendet werden: arithmetische, relative und logische. Der syntaktische Aufbau ist immer derselbe. Ein

beliebiger Ausdruck bildet den ersten Operanden, gefolgt von einer Operation und dem zweiten Operanden. Die logische Verneinung, die positive und negative Vorzeichenfunktion dagegen bestehen nur aus dem Operator, gefolgt von einem beliebigen Ausdruck, dem Operanden. Dabei sind bereits deklarierte Variablen, ganze und reelle Zahlen mögliche Formen der Operanden. Ausdrücke können auch beliebig geklammert werden.

```
<Expression> ::= <Expression> <Relop> <Expression>    -- relative operations
               | <Expression <Op> <Expression>        -- arithmetic, logic
               | NOT <Expression>                     -- logical not
               | - <Expression>                       -- unary negation
               | + <Expression>                       -- unary positive
               | ( <Expression> )
               | <Variable>                           -- variables
               | Integer                              -- integer numbers
               | Double                               -- real numbers

<Op> ::=  AND | OR | + | - | * | / | MOD | DIV        -- operators

<Relop> ::= == | != | < | <= | > | >=                  -- relative operators
```

Eine Variable ist entweder ein einfacher Bezeichner oder ein Array, das sich aus dem Arraynamen, einer Variable, und dem Index, einem geklammerten Ausdruck, zusammensetzt.

```
<Variable> ::= Ident                                  -- variable
             | <Variable> [ <Expression> ]            -- array
```

Die Darstellung dieser Grammatik verzichtet der Anschaulichkeit halber auf einige Regeln, die implementierte Grammatik ist komplexer. So besteht <Expression> aus Nonterminals, die die Bindung zwischen den verschiedenen Ausdrücken bestimmen, und einer Regel für jeden Operator. Generell wird in der implementierten Version eine Liste, also mehrere Elemente eines Nonterminals, mit einem eigenen Nonterminalnamen bezeichnet und durch zwei dazugehörende Regeln repräsentiert.

2.3.4 Darstellung der Grammatik in der Implementation

Wie stellt man eine Grammatik im Compiler dar? Der einfachen Manipulierbarkeit wegen benutzen wir strukturierte Bäume [App97]. Ein strukturierter Baum hat für jede Produktionsregel einen Knoten mit sovielen Ästen, wie Symbole auf der rechten Seite der Regel vorkommen. Terminale Symbole sind die Blätter des Baumes. Die Grammatik wird folgendermassen in die Baumstruktur transformiert: Jedes grammatikalische Symbol entspricht einer abstrakten Klasse in der Datenstruktur. Jede Regel entspricht einem Konstruktor in dieser Klasse. Deshalb wird die abstrakte Klasse durch eine konkrete für jede Grammatikregel erweitert. Die Komponenten der rechten Regelseiten werden als Parameter an den Konstruktor übergeben.

Praktisch geht man wie folgt vor: Für die oben vorgestellte Grammatik wird ein Parser in Gentle geschrieben. Dieser liefert eine äquivalente Darstellung mit denselben Nontermials und Terminals wie in der Syntaxbeschreibung. Zusätzlich wird jede Konstruktionsalternative mit einem Funktor bezeichnet, der zum abstrakten Typ ihrer Produktion gehört. Man erhält so eine Reihe von abstrakten Typen und den dazugehörigen Konstruktoren. Die Typen werden als abstrakte Klassen, die Konstruktoren als erweiterte konkrete Klassen in Java angegeben, siehe Kapitel 7.

Zur Veranschaulichung dieses Überganges betrachte man ein Beispiel aus der Grammatik in der Gentlecodedarstellung, nämlich die Anweisungen für den Aufruf und die WHILE-Schleife. Zwei Regeln beschreiben die Alternativen zur Konstruktion des Nonterminal <statement>.

Nach dem Schlüsselwort „nonterm“ folgt der Name der Produktion „Statement“ und die Zuweisung an den dazugehörigen abstrakten Typ „STATEMENT“. Eine Regel beginnt mit dem Schlüsselwort „rule“, ihrem Namen, dem gleichen Nonterminal wie in der Grammatikbeschreibung, danach ein Funktor des zugehörigen abstrakten Typs „call/while“, der die Konstruktion dieser Regel beschreibt. Der Doppelpunkt entspricht dem Definitionszeichen der Grammatik in EBNF. Nun folgen die Schlüsselwörter, in Hochkommatas gesetzt, die Terminals und die Nonterminals in der gleichen Reihenfolge wie in der Grammatikregel. Jedes Terminal überweist sich als Parameter an den Funktor. Die Nonterminals rufen die Regeln gleichen Namens auf und übergeben deren Resultate an den Funktor.

```
'nonterm' Statement( ->  STATEMENT )

   'rule' Statement( ->  call( Id, Exprs ) ) :
          Ident( -> Id ) "(" ExprList( ->  Exprs ) ")"

   'rule' Statement( ->  while( Cond, Stmt ) ) :
          "WHILE" Expression( ->  Cond ) "DO" Statement( -> Stmt )
```

Die abstrakte Syntax im Gentlecode zusammengefasst besteht aus dem Typ „STATEMENT“ und der Angabe seiner Alternativen. Diese bestehen aus ihren Namen, den Funktoren „call“ und „while“, und den Typen ihrer Parameter.

```
'type' STATEMENT = call( IDENT, EXPRLIST ),
                   while( EXPRESSION, STATEMENT ).
```

Dies dient nun als Vorlage für Klassen in Java. Der abstrakte Typ „STATEMENT“ wird mit der abstrakten Klasse „Statement“ abgebildet. Die Funktoren des abstrakten Typs werden zu konkreten Klassen, die die abstrakte Klasse erweitern. Die Parameter des Klassenkonstruktors entsprechen denjenigen des Funktors. Nach diesem Muster wird die abstrakte Syntax in Javacode übersetzt. Die Klassen sind noch rudimentär. Diese Templates werden in Kapitel 7 mit Code für die Codegenerierung ergänzt.

```
public abstract class Statement{ }

public class CallStmt extends Statement {

  Ident id;
  ExprList exprs;
  public CallStmt(Ident i, ExprList e) {id = i; exprs = e;}
}

public class WhileStmt extends Statement {

  Expression cond;
  Statement stmt;
  public WhileStmt(Expression e, Statement s) {cond = e; stmt = s}
}
```

2.3.5 Abstrakte Syntax der imperativen Sprache

Abbildung 2.3 listet die abstrakten Typen vollständig auf. Ebenso ihre Funktoren, die sich aus dem Gentlecode ergeben haben. Typen ohne Funktoren sind Token, also terminale Symbole.

2.3.6 Diskussion

Es folgt die Diskussion einiger ausgewählter Probleme zum Entwurf, insbesondere zum Übergang von der darstellenden Grammatik zur Implementationsgrammatik. Zuerst jedoch eine Anmerkung zur Syntax der imperativen Beispielsprache:

```
'type' DECLARATION = declare( IDENT, DEFINITION ) .

'type' DECLLIST = decllist( DECLARATION, DECLLIST ), nil .

'type' DEFINITION = variable( TYPE ), valueparam( TYPE ), varparam( TYPE ),
       proc( DECLLIST, DECLLIST, STMTSEQ ) .

'type' TYPE = integer, real, char, boolean, none,
       array( EXPRESSION, EXPRESSION, TYPE ) .

'type' STMTSEQ = stmt( STATEMENT ), seq( STATEMENT, STMTSEQ ) .

'type' STATEMENT = assignment( EXPRESSION, EXPRESSION ),
       ifthenelse( EXPRESSION, STATEMENT, STATEMENT ),
       while( EXPRESSION, STATEMENT ), repeat( STATEMENT, EXPRESSION ),
       for( VARIABLE, EXPRESSION, EXPRESSION, EXPRESSION, STATEMENT ),
       compound( STMTSEQ ), call( IDENT, EXPRLIST ), read( VARIABLE ),
       write( EXPRESSION ), nil .

'type' EXPRLIST = exprlist( EXPRESSION, EXPRLIST ), nil .

'type' EXPRESSION = relative( EXPRESSION, RELOP, EXPRESSION ),
       and( EXPRESSION, EXPRESSION ), or( EXPRESSION, EXPRESSION ),
       not( EXPRESSION ), mult( EXPRESSION, EXPRESSION ),
       div( EXPRESSION, EXPRESSION ), intdiv( EXPRESSION, EXPRESSION ),
       mod( EXPRESSION, EXPRESSION ), plus( EXPRESSION, EXPRESSION ),
       minus( EXPRESSION, EXPRESSION ), neg( EXPRESSION ), num( INT ),
       double( DOUBLE ), var( VARIABLE ), true, false .

'type' LOGOP = and, or, not .

'type' RELOP = eq, ne, lt, le, gt, ge .

'type' VARIABLE = id( IDENT ), array( VARIABLE, EXPRESSION ) .

'type' IDENT .

'type' DOUBLE .
```

Abbildung 2.3: Abstrakte Syntax der imperativen Grammatik

Mehrere Anweisungen werden mit einem Strichpunkt getrennt, die letzte Anweisung erfordert jedoch kein Strichpunkt. Ebenso darf kein Strichpunkt vor „END“ und vor „}“ stehen. Mit dieser Regelung, die von MINILAX [Wai89] übernommen wurde, erinnert die Grammatik an die imperative Sprache Pascal und erscheint etwas inkonsequent, da nicht alle Anweisungen gleich geschrieben werden. Diese syntaktische Eigenheit hat jedoch keinen Einfluss auf die abstrakte Syntax und wird deshalb beibehalten.

Auflösen von mehrdeutigen Grammatiken

Der Parser, der aus der Übersetzung der imperativen Beispielgrammatik in Gentle entstanden ist, ist vom Typ LALR(1): „L“ steht für die Verarbeitung des Inputs von links nach rechts, „R“ für die Ableitung des am meisten rechts stehenden Zeichens, und die Zahl in der Klammer für die Anzahl Symbole die vorausschauend in die Parsingentscheidung miteinbezogen werden (look-ahead). Die Arbeitsweise und Eigenschaften von LR-Parsern können „Compilers:

Principles, Techniques and Tools“ [Aho86] entnommen werden.

Da die vorliegende Grammatik mehrdeutig ist, und mehrdeutige Grammatiken nicht LR analysiert werden können, müssen die mehrdeutigen Produktionen in eindeutige Produktionen umformuliert werden.

Die Spezifikation von Ausdrücken mit der Produktion <Expression>, die mehrere Alternativen anbietet, ist mehrdeutig und sagt nichts über Assoziativität und Priorität aus. Durch Einfügen von zusätzlichen Nonterminals und Regeln für die Übergänge kann daraus eine eindeutige Grammatik erzeugt werden, die immer noch die gleiche Sprache generiert. Dabei wird der Addition eine tiefere Priorität als der Multiplikation zugewiesen, beide Operatoren werden linksassoziativ gemacht.

```
-- ambiguous rule for expressions

<Expression> ::= <Expression> + <Expression>
               | <Expression> * <Expression>
               | ( <Expression> )
               | <Variable>
```

```
-- unambiguous rules for expressions

<Expression> ::= <Expression> + <Expr1>
               | <Expr1>

<Expr1> ::= <Expr1> * <Expr2>
          | <Expr2>

<Expr2> ::= ( <Expression> )
          | <Variable>
```

Die Regel für die Selektion ist mehrdeutig und erzeugt beim Parsen der folgenden Situation einen shift/reduce-Konflikt : Nach Erkennen des auf „THEN“ folgenden <Statement> liest der Parser ein „ELSE“ ein. Gemäss der ersten Regel sollte dieses „ELSE“ auf den Stack gekellert und weitere Token eingelesen werden, um die Regel zu vervollständigen. Die zweite Regel verlangt aber nach einer Reduktion (reduce) der bereits eingelesenen Symbole durch <Statement>. (In diesem Fall würde „ELSE“ zu einer übergeordneten Selektion gehören.) Der LALR(1)-Parser YACC [Joh75] entscheidet diesen Konflikt zugunsten der ersten Regel, da „ELSE“ aus der Sicht des Programmierers mit „THEN“ assoziiert ist und in der Grammatik ein „ELSE“ ohne den vorausgehenden Teil keine Produktion ist. „ELSE“ wird gekellert und ein <Statement> erwartet, um eine vollständige Selektion zu reduzieren.

```
-- dangling else
<Statement> ::= IF <Expression> THEN <Statement> ELSE <Statement>
              | IF <Expression> THEN <Statement>
              | <OtherStatements>

-- unambiguous selection
<Statement> ::= IF <Expression> THEN <Statement> ELSE <Statement> ENDIF
              | IF <Expression> THEN <Statement> ENDIF
              | <OtherStatements>
```

Um einen shift/reduce-Konflikt beim LR-Parsing zu vermeiden, kann die Selektion mit einem Schlüsselwort („ENDIF“) abgeschlossen werden. Die so modifizierte Grammatik konnte konfliktlos geparst werden.

2.4 Objektorientierte Programmiersprachen

Bei der Programmentwicklung mit dem Wasserfallmodell wird von der Spezifikation über Design zur Implementation top-down vorgegangen. Dies ist grundsätzlich ein funktionenbasierter Ansatz, der eine hierarchische Systemstruktur als Grundlage der Implementation und den Nachteil hat, dass bei Änderungen das gesamte Modell überarbeitet werden muss. Somit ist das Modell teuer und unflexibel und die Wiederverwendung von Code schwierig.

Objektorientiertheit dagegen ist nicht nur ein Sprachstil sondern eine Designmethode, deren Prinzipien bei jeder beliebigen Programmiersprache angewendet werden können. Es geht darum, den Programmstatus in Datenobjekten, nicht Funktionen, einzuschliessen, auf die man nur mit definierten Operationen zugreiffen kann. Die wichtigsten Prinzipien des objektorientierten Paradigmas sind: Datenkapselung, Polymorphismus und dynamisches Binden.

Die Kontrollstrukturen der meisten objektorientierten Sprachen sind laut Bal und Grune [Bal94] ähnlich denjenigen der imperativen Sprachen, wogegen die Datenstrukturen spezifisch objektorientiert sind. Ein Beispiel dafür ist C++ [Ell95], das aus der imperativen Sprache C entstanden ist, eine Ausnahme davon SMALLTALK [Gol89], das Daten als aktive Agenten, die untereinander Nachrichten verschicken, sieht und dafür spezielle Strukturen verwendet. Dieses Kapitel reflektiert den ersten Ansatz, deshalb wird nur auf die objektorientierten Eigenheiten und Datenstrukturen eingegangen und die Kontrollstrukturen aus dem Kapitel über imperative Sprachen werden übernommen.

2.4.1 Prinzipien

Die folgenden Definitionen und Prinzipien des objektorientierten Paradigmas stützen sich auf die Referenzbücher der objektorientierten Sprachen JAVA [Gos96], EIFFEL [Mey88], SMALLTALK [Gol89] und C++ [Ell95], sind aus dem „Lexikon Informatik“ [Lex97] entnommen oder an das Kapitel über objektorientierte Sprachen in „Programming Language Essentials“ [Bal94] angelehnt. Die Punkte für Objekte, Klassen und Vererbung machen zusammen die **Datenkapselung** (data encapsulation, information hiding) aus, bei der Daten nur über Zugriffsmethoden zugänglich sind.

- Eine **Klasse** liefert die statische Beschreibung einer Objektkategorie, also einer Menge von Objekten gleicher Struktur und gleicher Semantik. Sie besteht aus internen Variablen der Klassenobjekte und den Operationen, die auf die Klassenobjekte anwendbar sind. Auf eine Klasse kann nur durch definierte Operationen zugegriffen werden, die Datenstrukturen sind gekapselt. Dadurch sind Änderungen innerhalb einer Klasse jederzeit möglich und zeigen keine Nebeneffekte im restlichen Programm, solange die Schnittstelle nicht verändert wird. Klassen sind die Datentypen der Klassenobjekte und können durch Vererbung zueinander in Verbindung gesetzt werden.

- Ein **Objekt** ist eine Instanz oder Ausprägung einer Klasse. Es hat eigenen, neuen Speicherplatz für die Elemente, die von der Klasse definiert sind. Objekte können nur mit den auf ihnen definierten Operationen manipuliert werden, nicht direkt wie normale Variablen. Diese Operationen dienen als Schnittstelle für den Benutzer des Objektes, die Daten selbst sind für ihn unsichtbar und ihre Darstellung nicht bekannt. Die Kommunikation zwischen Objekten erfolgt durch den Austausch von Nachrichten (messages), die den imperativen Prozeduraufrufen entsprechen, jedoch dynamisch gebunden werden.

- Bei der **Vererbung** werden verschiedene Klassen hierarchisch geordnet. Die generellere Klasse vererbt ihre Operationen an die spezifischeren Klassen der unteren Hierarchiestufen. Diese Unterklassen können die ererbten Operationen überschreiben oder eigene hin-

zufügen. Falls in einer Sprache Klassen nur von genau einer Oberklasse erben können, ist dies eine **singuläre**, falls sie von mehreren Oberklassen erben dürfen, eine **mehrfache** Vererbung. Die Vererbungshierarchie kann ganz zusammenhängend sein, wie diejenige von JAVA oder SMALLTALK, oder nach Belieben in kleine Gruppen unterteilt wie in C++.

Mehrfachvererbung erfordert eine Konfliktauflösungsstrategie, da es zu Widersprüchen und Benennungskonflikten kommen kann: eine mehrdeutige Methode erzeugt eine Fehlermeldung, wird standardmässig als Methode der ersten oder letzten Oberklasse in der Vererbungshierarchie interpretiert, als spezieller Code compiliert, oder ganz verboten und zurückgewiesen. In diesem Fall kann durch Umbennen der Methoden eine eindeutige Situation erreicht werden. [Car94]

- **Polymorphismus** (Vielgestaltigkeit) bedeutet in der objektorientierten Programmierung die Fähigkeit eines Elements, dynamisch auf Instanzen verschiedener Klassen zu verweisen. Vererbung kontrolliert die Anwendung von Polymorphismus durch Beschränkung der kompatiblen Typen. Die Substitionsregel nach Meyer [Mey88] lautet:

  > Gegeben seien ein Objekt x mit Typ A und ein Objekt y mit Typ B. Falls B eine Unterklasse von A ist, kann y überall dort verwendet werden, wo x oder Typ A erwartet wird.

  Anwendungen der Definitionen von Cardelli und Wegner [Car85] in objektorientierten Sprachen: Gemeinsame Fähigkeiten von Klassen werden an eine Oberklasse übertragen, Unterklassen erben die Eigenschaften der Oberklasse (universal inclusion), ein Objekt kann zu mehreren Klassen gehören und wird einheitlich verwendet (universal parametric), syntaktisch gleiche Nachrichten an Objekte verschiedener Klassen rufen semantisch ähnliche Operationen auf (ad hoc overloading) und semantisch nötige Typkonversionen können vom Benutzer wegegelassen werden (ad hoc coercion). Nicht alle objektorientierten Sprachen unterstützen alle vier Formen von Polymorphismus.

- Beim **Dynamischen Binden** (dynamic binding) wird zur Laufzeit entschieden, welcher Code einer Operation ausgeführt wird, im Gegensatz zum statischen Binden, wo dies mit Hilfe des Typensystems zur Compilierzeit geschieht. Da durch die dynamische Bindung an ein Objekt polymorphe Operationen möglich sind, trägt dies erheblich zur Flexibilität objektorientierter Sprachen bei. Der Compiler weiss im voraus nicht, welche Operation angewandt wird, aber er kann immer noch kontrollieren, ob der Aufruf legal ist und eine solche Operation existiert. Statische Typprüfung kann gleichzeitig mit dynamischem Binden verwendet werden. Es gibt auch hybride Ansätze, wie in C++, das standardmässig statisches Binden verwendet und dynamisches Binden nur bei explizit deklarierten Funktionen (*virtual functions*) verwendet, um bei der Ausführung Zeit zu sparen.

Nach Abadi und Cardelli ist eine objektorientierte Programmiersprache **klassenbasiert** (*class-based*), wenn Objekte zu Klassen gehören müssen, die durch Vererbung verändert werden können. Eine Sprache ist **objektbasiert**(*object-based*), wenn sie Objekte als einzige Sprachelemente unterstützt [Aba96].

Weiter unterscheidet man zwischen **hybriden** objektorientierten Sprachen, die Objekte und Werte von Builtin-Typen verschieden behandeln und objektorientiertes Programmieren erlauben und **reinen** objektorientierten Sprachen, die alle Elemente als Objekte betrachten und den objektorientierten Stil verlangen. Einige Beispiele: C++ ist hybrid, JAVA ist fast rein (nur einfache Typen sind keine Objekte), SMALLTALK ist rein objektorientiert.

2.4.2 Konzepte

Konzepte und Methoden, wie die Prinzipien objektorientierter Sprachen angewendet werden:

- Versteckte Daten werden sicher in der Klasse „versiegelt“ und sind nur mit dafür definierten Methoden zugreifbar. Dieser Mechanismus der **Datenkapselung** bietet einerseits Schutz vor unberechtigtem, willkürlichem Zugriff, der zu einem inkonsistenten Zustand führen kann. Andererseits wird eine minimale Schnittstelle gewährleistet, was einfach zum Testen, Anwenden und Dokumentieren ist. Die Sichtbarkeit der Daten wird (in C++) mit folgenden Schlüsselwörtern angegeben: *private, public, protected*.

- **Konstruktoren** initialisieren Objekte einer Klasse und weisen den benötigten Speicherplatz zu, **Destruktoren** entfernen Objekte und deallozieren den Speicherplatz. Es können durch Überladen mehrere Konstruktoren für eine einzige Klasse definiert werden. Java kennt keine expliziten Destruktoren, die Deallokation wird mit automatischer *Garbage Collection* durchgeführt. Mit einem *Finalizer* kann ein Objekt schon vorher seine Resourcen freigeben.

- **Abstrakte Methoden** bestehen nur aus der Definition ihrer Signatur und besitzen keine Implementation. Eine Klasse, die mindestens eine abstrakte Methode enthält, wird zur **abstrakten Klasse**, die nicht instantiiert werden kann. Eine Unterklasse einer abstrakten Klasse muss jede der abstrakten Methoden der Oberklasse überschreiben und eine Implementation dafür liefern, damit sie nicht selbst abstrakt ist und instantiiert werden kann.

- Eine **Schnittstelle** (interface) umfasst abstrakte Methoden und Konstanten. Die Deklaration einer Schnittstelle erzeugt einen neuen Datentyp. Eine Klasse kann von mehreren Schnittstellen erben, muss dabei für jede der abstrakten Methoden eine Implementation liefern. Java verwendet dieses Konzept und kompensiert mit der Vererbung von Methodendeklarationen von verschiedenen Schnittstellen die (fehlende) Mehrfachvererbung.

- Die Methoden eines Typs werden entweder neu deklariert oder wiederverwendet. **Methodenwiederverwendung**: von einer Oberklasse geerbte Methoden werden überschrieben (override), dabei wird der Name und die Signatur beibehalten, während ihre Funktionsweise neu definiert wird. Falls mehrere Methoden mit den selben Namen aber verschiedenen Signaturen in einer Klasse vorkommen (ob geerbt oder deklariert ist unwichtig), sind sie überladen (ad hoc overloading polymorphism).

- Die meisten objektorientierten Sprachen verwenden Variablen, die Referenzen auf Objekte enthalten; die Objekte enthalten die Daten. **Klassenvariablen** sind global verwendbar in einer Klasse, aber es existiert nur eine Kopie davon in der Klasse, nicht jedes Objekt hat eine eigene Instanz. Sie werden über die Klasse, nicht die Objekte zugegriffen. Analog dazu verhält es sich mit **Klassenmethoden**. In Java und C++ werden Klassenvariablen und -methoden durch das Schlüsselwort *static* deklariert.

- Eiffel unterstützt **Programming by contract**: Die Zusammenarbeit zwischen Objekten wird durch einen Vertrag formal beschrieben, ohne explizite Angabe ihrer Implementation. Dabei können Klassen und Operationen *Preconditions* und *Postconditions* enthalten. Diese geben die Bedingungen an, die erfüllt sein müssen, bevor eine Operation ausgeführt wird, respektive nach der Ausführung erfüllt sind. Bei der Vererbung gilt, dass die Preconditions der abgeleiteten Klasse mindestens so streng wie die der Oberklasse sein müssen. Programming by contract unterstützt die Fehlersuche und Dokumentation und erleichtert die Wiederverwendung von Codeteilen. [Mey88]

2.4.3 Grammatik einer objektorientierten Beispielsprache

Die objektorientierte Beispielsprache baut auf der imperativen Beispielsprache auf und fügt ihr Elemente aus CooL[3] und Java hinzu. Daraus ergibt sich eine hybride Sprache, die Variablen wie eine imperative Sprache behandelt aber auch die Anforderung der Objektorientiertheit erfüllt.

Dieser Abschnitt stellt die Syntax der objektorientierten Beispielsprache als kontextfreie Grammatik in erweiterter Backus-Naur-Form vor, dabei wird die gleiche Notation wie im Abschnitt 4 über die imperative Beispielsprache verwendet.

Syntax

Terminale und Schlüsselwörter sind die lexikalischen Zeichen dieser Sprache: Bezeichner werden mit „Ident“ notiert und setzen sich aus einer Folge von Zahlen, Gross- und Kleinbuchstaben zusammen, die mit einem Buchstaben beginnen muss. Schlüsselwörter werden nicht als Bezeichner akzeptiert. Weitere terminale Elemente sind ganze oder reelle Zahlen, „Integer“ respektive „Double“, eine Reihe von Zeichen in Anführungszeichen, „TString“, und ein Buchstabe in Hochkommatas, „Character“.

Das Startsymbol dieser Grammatik ist <Class>, mit der ersten Regel kann es beliebig oft reproduziert werden. Eine syntaktisch korrekte Eingabe besteht aus null oder mehr Klassen.

Eine Klasse beginnt mit dem Schlüsselwort „CLASS“, danach kommt der Klassenbezeichner und die optionale Angabe einer Oberklasse; Mehrfachvererbung wird nicht unterstützt. Danach folgen in geschweiften Klammern die Variablendeklarationen und die Methoden der Klasse. Die erste Methode muss der Konstruktor sein, ein impliziter Defaultkonstruktor oder mehrere Konstruktoren werden vom Parser nicht akzeptiert. Für die restlichen Methoden gelten keine Einschränkungen.

```
<Class> ::= {{ <Class> }}                          -- several classes
          | CLASS Ident [[ EXTENDS Ident ]] {
            <InstVarDecl> <Method> }              -- class declaration
```

Der Zugriff auf Methoden einer Klasse wird durch die Schlüsselwörter von <Scope> angegeben. Falls es sich um eine abstrakte Methode handelt, wird „ABSTRACT“ vorangestellt. Nach den Sichtbarkeitsangaben folgt der Rückgabetyp, der nur beim Konstruktor weggelassen wird, da dieser implizit den Typ seiner Klasse als Rückgabewert annimmt. Der Konstruktor verwendet als Bezeichner den Klassennamen; alle andern Methoden einen beliebigen Namen. Danach folgt in Klammern die optionale Deklaration der formalen Parameter: für jedes Element der Liste wird ein Bezeichner und dessen Typ angegeben. Die Methodendeklaration schliesst mit der Deklaration der lokalen Variablen und einem Anweisungsblock, der mit geschweiften Klammern zusammengefasst ist.

```
<Method> ::= <Method> {{ <Method> }}               -- several methods
           | [[ ABSTRACT ]] [[ <Scope> ]]
             [[ <Type> ]] Ident
             ( [[ <FormalParam> ]] )  {
             <InstVarDecl> <Statement> }           -- classmethod decl

<FormalParam> ::= <FormalParam> {{, <FormalParam> }}  -- formal parameters
                | Ident : <Type>                      -- value parameter
```

[3] CooL ist eine objektorientierte Sprache die von Siemens-Nixdorf mit Gentle entwickelt wurde. Weitere Hinweise, das Referenzmanual und der Quellcode finden sich auf: `http://www.first.gmd.de/gentle/examples.html`

```
<Scope> ::= PUBLIC
          | PRIVATE
          | PROTECTED
          | PRIVATE PROTECTED
```

Die Deklaration der Dateneinheiten muss strikt von den restlichen Anweisungen getrennt zu Beginn einer Klasse oder Methode ausgeführt werden. Mehrfachdeklarationen sind nicht erlaubt. Variablen werden gleich wie in der imperativen Grammatik deklariert. Konstanten werden mit „CONST“ gekennzeichnet, danach wird ihr Typ und der Bezeichner angegeben, letzterem wird ein Ausdruck zugewiesen.

```
<InstVarDecl> ::= {{ <InstVarDecl> }}                 -- several declarations
                | VAR Ident : <Type> ;                -- variable declaration
                | CONST <Type> Ident = <Expression> ; -- constant declaration
```

Die Anweisungen für die Zuweisung, den Block, die Selektion und die Schleifen werden aus der imperativen in die objektorientierte Beispielsprache übernommen. Input und Output werden mit Aufrufen von Methoden, die zu Input/Output-Objekten gehören, ersetzt. Jede Anweisung wird mit einem Strichpunkt „;“ abgeschlossen. Es werden im folgenden Abschnitt nur die geänderten oder neuen Anweisungen behandelt:

Ein **Aufruf** einer Methode setzt sich aus dem Namen der Methode und den aktuellen Parametern zusammen. Die Grammatik beschränkt sich auf einfache Methodennamen, zusammengesetzte könnten eingeführt werden.

Die Anweisung für die **Rückgabe**, die die aktuelle Methode beendet, besteht aus dem Schlüsselwort „RETURN“ gefolgt von einem beliebigen Ausdruck, der einen Wert vom Rückgabetyp an die aufrufende Methode zurückgibt.

Die **Objektdeklaration** beginnt mit der Angabe des Typs, also der Spezifikation der Klasse, und dem Objektbezeichner, dem das neue Objekt zugewiesen wird. Danach folgt das Schlüsselwort „NEW“, wiederum die Angabe des (gleichen) Typs und eine optionale Liste von aktuellen Parametern in Klammern. „NEW“ funktioniert wie ein unärer Operator und kreiert eine dynamische Instanz der spezifizierten Klasse. Diese Grammatik lässt Objektdeklarationen nur in dieser Form mit gleichzeitiger Initialisierung des Objektbezeichners zu, in JAVA kann dies auch voneinander getrennt werden.

Neben der einfachen Selektion bietet diese Grammatik eine mehrfache **Selektion** an. Nach dem Schlüsslwort „SWITCH“ folgt ein Ausdruck, der vom Typ „BOOL“, „CHARACTER“ oder „INTEGER“ sein muss. Damit wird der Wert berechnet, der einen der Fälle zur Bearbeitung auswählt. Ein Fall besteht aus dem Schlüsselwort „CASE“, einer Reihe von Labels, die mit einem Doppelpunkt abgeschlossen werden, und einer Anweisung. Die Labels müssen konstante Ausdrücke vom gleichen Typ wie der Auswahlausdruck sein. Falls das Resultat des Auswahlausdruckes mit keinem der Labels übereinstimmt, wird die Standardanweisung ausgeführt, die am Schluss der Mehrfachauswahl stehen muss und mit „DEFAULT“ bezeichnet ist.

```
<Statement> ::= {{ <Statement> }}                     -- several statements
              | <Expression> = <Expression> ;         -- assignment
              | { <Statement> } ;                     -- compound statement
              | Ident [[ ( [[ <ExprList> ]] ) ]] ;    -- call statement
              | IF <Expression> THEN <Statement>
                [[ ELSE <Statement> ]] ENDIF ;        -- selection
              | WHILE <Expression> DO <Statement> ;   -- while loop
              | REPEAT <Statement>
                UNTIL <Expression> ;                  -- repeat loop
              | FOR <Variable> := <Expression>
                TO <Expression> STEP <Expression>
```

```
                DO <Statement> ;                        -- for loop
              | RETURN <Expression> ;                   -- return value
              | <Type> <Variable> =
                NEW <Type> ( [[ <ExprList> ]] ) ;       -- create new object
              | SWITCH ( <Expression> ) {
                {{ CASE <Label> : <Statement> }}
                [[ DEFAULT : <Statement> ]] } ;          -- switch selection

<Label> ::= <Label> {{, <Label> }}
          | <Expression> [[ .. <Expression> ]]
```

Die Operatoren, die Operanden und die Ausdrücke, die man daraus bilden kann, werden von der imperativen Grammatik übernommen. Ergänzt werden sie mit dem konstanten Ausdruck für Strings und dem Objektzugriff oder Methodenaufruf. Dieser geschieht mit einem Punkt, der den/die Objektnamen und die Objektmethode trennt. Danach folgen in Klammern optional die aktuellen Parameter als Ausdrucksliste.

Gültige Variablen sind einfache Bezeichner oder Arrays, die mit einer Deklaration spezifiziert wurden.

```
<ExprList> ::= <Expression> {{, <Expression>}}        -- several expressions

<Expression> ::= String                               -- string
               | Character                            -- character
               | <Expression> . Ident
                 ( [[ <ExprList> ]] )                 -- method call

<Variable> ::= Ident                                  -- variable
             | <Variable> [ <Expression> ]            -- array
```

Zusätzlich zu den aus der imperativen Grammatik bekannten Standardtypen „INTEGER“, „REAL“ und „CHARACTER“ und dem zusammengesetzten Typ „ARRAY“ sind in dieser Grammatik noch einige Typen vorhanden, deren Verwendung sich an JAVA orientiert: „VOID“ ist eigentlich kein Typ, sondern ein Schlüsselwort, das anstelle eines Rückgabetyps angibt, dass eine Funktion keinen Rückgabewert liefert. In JAVA ist „BOOLEAN“ kein Aufzähl- sondern ein Basistyp, dies wirkt sich nicht auf die Syntax aus. „STRING“ bezeichnet eine Reihe von Werten vom Typ „CHARACTER“ . Mit einem Bezeichner als Typ wird eine Klasse (Objekt-Typ) referenziert.

```
<Type> ::= VOID                                       -- void
         | INTEGER                                    -- integer
         | REAL                                       -- real
         | CHAR                                       -- character
         | BOOLEAN                                    -- boolean
         | STRING                                     -- string
         | Ident                                      -- object type
         | ARRAY [ <Expression> .. <Expression> ]
           OF <Type>                                  -- array
```

Diese Darstellung der Grammatik verzichtet auf einige Regeln der implementierten Grammatik, die für das Verständnis nicht nötig sind. Die Implementation unterteilt die meisten Nonterminals, um verschiedene Prioritäten und Bindungen zu erhalten. Damit können auch Entscheidungskonflikte umgangen werden. Die vollständige Grammatik findet sich im Anhang A.3.1.

2.4.4 Abstrakte Syntax

Der komplette Gentlecode des Parsers für die objektorientierte Beispielsprache ist nicht im Anhang zu finden, da er im Prinzip gleich wie der Parser für die imperative Sprache aufgebaut

ist. Die dazugehörigen Typen und Funktoren können wiederum als abstrakte Syntax für diese Grammatik verwendet und in Javaklassen übersetzt werden. Anhang A.4.1 auf Seite 120 zeigt alle abstrakten Typen und deren Funktoren, also die abstrakte Syntax der objektorientierten Beispielsprache.

2.4.5 Diskussion

- Die vorgestellte Sprache beinhaltet die objektorientierten Prinzipien. Die meisten objektorientierten Sprachen verwenden zusätzliche Anweisungen, Schlüsselwörter und komplexere Deklarationen. Mit dem vorgestellten Modell sollte ein Hinzufügen von zusätzlichen Elementen analog vor sich gehen.

- Die vorgestellte objektorientierte Beispielsprache unterscheidet zwischen der Deklaration von neuen Objekten im Implementationsteil als Anweisung und Deklarationen von allen anderen Typen im Deklarationsteil. Dies ist eine hybride Sichtweise der Typen, die sich daraus ergibt, dass diese Sprache aus einer imperativen Sprache, die Deklarationsteile verwendet, und der Bedingung der Objektorientiertheit, dass Klassen als Typen verwendet werden, zusammengesetzt ist. Somit referenzieren Variablen Objekte und enthalten Daten, je nach Deklarationsart. Ob eine generelle Behandlung von Datentypen diesem Ansatz vorzuziehen wäre, wird sich bei der Übertragung auf die JVM zeigen.

- Bei der Kreation eines Objektes muss semantisch getestet werden, ob der Typnamen gleich dem Klassenname hinter dem Gleichheitszeichen ist. Dies ist für den Parser nicht möglich, da er nur ein Zeichen vorausschauen kann.

- Die Grammatik lässt auch leere Anweisungen zu. Da jede Anweisung abgeschlossen sein muss, besteht eine leere Anweisung nur aus einem Strichpunkt „;“. Somit ist es gültig, mehrere Strichpunkte hintereinander zu setzen.

- Die vorgestellte Sprache verzichtet auf Schnittstellen und begnügt sich mit einfacher Vererbung.

- Diese Beispielsprache wurde wie folgt aufgebaut: die Syntaxbeschreibung von CooL lieferte die objektorientierten Konstrukte, die der imperativen Beispielsprache hinzugefügt wurden. Notation und Schlüsselwörter wurden an Java angelehnt.

- Ein einfacher Ansatz nimmt Methoden grundsätzlich als „PUBLIC“ an, Instanzvariablen als „PRIVATE“. Der Aufruf einer Objektmethode kann nur einer Variable zugewiesen werden.

- Der Gentlecode für den Parser der objektorientierten Beispielsprache kompiliert ohne Konflikte.

2.5 Funktionale Programmiersprachen

Reine funktionale Sprachen gehören wie logische Sprachen in die Gruppe der deklarativen Sprachen, die sich von der zu Grunde liegenden Computerarchitektur abheben und Mechanismen zur Problemspezifikation anbieten. Sie basieren auf der Anwendung von Funktionen und werden deshalb auch applikative Sprachen genannt. Programme und die durch sie manipulierten Werte sind Funktionen und damit als mathematische Objekte formal untersuchbar. Dies erleichtert die Umsetzung von Problemen der realen Welt in ein Modell oder Programm.

Als Vorlage für dieses Kapitel dienen „The Implementation of Functional Programming Languages“ [Pey87], die Abschnitte über funktionale Programmierung im „Lexikon Informatik“ [Lex97] und in „Encyclopedia of Computer Science“ [Ral92] und das Kapitel über funktionale Sprachen in „Programming Language Essentials“ [Bal94].

Funktionale Sprachen sind LISP, ISWIM, Backus's Formal Functional Programming, SCHEME, Hope, Miranda, ML, HASKELL und GOFER.

2.5.1 Prinzipien

Alle funktionalen Sprachen basieren auf der theoretischen Grundlage der funktionalen Programmierung, dem Lambda Kalkül. Der Bereich von Datenobjekten schliesst alles Berechenbare ein, auch die verwendeten Funktionen selbst. Dieser Abschnitt führt die Prinzipien des funktionalen Paradigmas auf:

- **Funktionen** bilden das Basiskonzept der funktionalen Sprachen, da alle Sprachelemente als Funktionen betrachtet werden. Sie haben den gleichen Status wie alle andern Werte (first-class values), können in Ausdrücken auftreten, als Argumente übergeben oder in Datenstrukturen eingefügt werden. Ein funktionales Programm wird topdown aufgebaut, indem eine generelle Funktion in mehrere Funktionen aufgeteilt wird, die Teilprobleme lösen; diese spezielleren Funktionen werden bei Bedarf weiter unterteilt.

  Eine Funktion berechnet aus ihren Eingabewerten ein Resultat. Da weder globale Variablen noch Wertzuweisungen verwendet werden können, erfolgt die Berechnung nur aufgrund der Eingabeparameter und verändert nichts ausser der Ausgabe. Dabei überträgt die Funktion (wie eine mathematische Funktion) Werte eines Bereichs auf Werte eines anderen Wertebereichs. Diese Wertebereiche werden mit der Typendeklaration festgelegt.

- **Funktionsaufrufe** (applications) sind die einzige Kontrollstruktur reiner funktionaler Sprachen, sie können zu Ausdrücken zusammengesetzt werden. Bedingte Ausdrücke werden anstelle von bedingten Anweisungen verwendet, anstelle von Zuweisungen werden Parameter an Argumente gebunden, anstelle von expliziten Sequenzen oder Schleifen benutzt man Pattern mit verschachtelten Aufrufen. Wichtig für die Problemlösung ist, dass eine Funktion mehrere Argumente und ein Resultat mehrere Komponenten haben kann.

- **Referenzielle Transparenz**: Diese Eigenschaft erlaubt weder globale Variablen noch Wertzuweisungen, dies verhindert Seiteneffekte und indirekte Wechselwirkungen. Funktionsaufrufe mit den gleichen Argumenten liefern bei referentieller Transparenz jedesmal die gleichen Resulate. Es ist kein impliziter Programmstatus vorhanden, Informationen können jedoch explizit durch Argumente und Resultate übergeben werden. **Pure funktionale Sprachen** wie HASKELL haben referentielle Transparenz und keine imperativen Eigenschaften, **nicht-pure funktionale Sprachen** wie ML erlauben zum Beispiel Zuweisungen, werden aber vom puren Teil der Sprache dominiert.

Die Eigenschaft, dass die Semantik von vielen funktionalen Programmiersprachen keine unnötigen Sequenzialisierungen benutzt, ist für parallele Prozesse begrüssenswert. Angewendet werden funktionale Sprachen unter anderem als Metasprache für die Definition von anderen Sprachen oder zum schnellen Erstellen von Prototypen. Vorteilhaft ist die implizite Speicherverwaltung, bei der eingebaute Operationen Zuweisung und Löschung von Speicher automatisch vornehmen, der Benutzer muss sich nicht um Speicherangelegenheiten kümmern.

2.5.2 Konzepte

In diesem Abschnitt werden vor allem Konzepte und Techniken beschrieben, die auf Funktionen basieren oder die sich von den Konzepten der anderen Paradigmen unterscheiden:

- Die **Typsysteme** funktionaler Sprachen reichen von schwachem oder keinem Typing (Lisp) über dynamisches Typing bis zu implizitem strengen Typing (Haskell). Bei letzterem werden die Typen statisch mit einem Typchecker bestimmt oder geprüft. Zur Kontrolle, ob ein Programm keine Typfehler hat, bestimmt der Typchecker die Typen aller Ausdrücke. Dadurch wird die Angabe von Typdeklarationen fakultativ. Implizit wird aber für jede Funktion eine Typdeklaration erstellt. Dabei werden neben den vordefinierten auch benutzerdefinierte Typen und Typvariablen verwendet. **Typvariablen** stellen einen beliebigen Typ dar, innerhalb einer Deklaration bezeichnet die gleiche Variable immer den gleichen Typ.

  Unabhängig von der Art des Typsystems können funktionale Sprachen **polymorph** sein: Funktionen mit Typvariablen in ihrer Deklaration führen für alle Typen den gleichen Code aus, sind also *universal parametric*. Typklassen führen einen neuen Typ und Funktionen darauf ein, die von jeder Typinstanz implementiert werden müssen, dadurch wird jedesmal der Funktionsname überladen (*ad hoc overloading*). Funktionen von Typklassen können nicht mit beliebigen Typen ausgeführt werden, sondern nur mit verwandten Typen (*universal inclusion* oder *subtyping*) [Car85, Jon91].

- **Lazy Evaluation**: Es gibt zwei Arten von Ausdrucksauswertung. Einerseits können die auszuwertenden Ausdrücke von innen her auf ihre einfachste Form reduziert werden, mit sogenannter *applicative order reduction* (Lisp, Scheme). Andererseits kann man von aussen her auswerten und Teilausdrücke erst berechnen (*normal order reduction*), wenn das Resultat benötigt wird. Die zweite Art wird auch lazy Evaluation genannt und wegen folgender Vorteile in den meisten funktionalen Sprachen verwendet: Funktionen können auch evaluiert werden, wenn sie ungültige oder unendliche Argumente enthalten (so lange diese nicht gebraucht werden). Es lassen sich unendliche Datenstrukturen konstruieren, zum Beispiel **lazy lists**. Dies sind Listen, die prinzipiell nicht begrenzt sind, ihre Elemente werden erst bei Bedarf generiert. Eine Sprache mit lazy Evaluation hat eine nicht strikte Semantik.

- **Patternmatching**: Eine Funktion besteht aus einer oder mehreren Gleichungen, die textuell nacheinander, mit dem gleichen Funktionsnamen und der gleichen Anzahl Argumente eingegeben werden müssen. Die Argumente werden auch **Pattern** genannt. Die Gleichungen können auch **guards** enthalten, die Bedingungen aufstellen, welche die Funktionsargumente zu erfüllen haben.

- **Rekursive** Funktionen rufen sich selbst auf. Der potentielle Rekursionsabbruch muss vor dem Rekursionsaufruf stehen. Mehrfache rekursive Aufrufe belasten den Programmstack stark. Falls der Rekursionsaufruf als textuell letzte Anweisung steht, wird die Funktion **endrekursiv** genannt und kann eventuell vom Compiler optimiert werden. Rekursive

Funktionen werden anstelle von Iterationen verwendet, wobei jeder Aufruf einem Iterationsschritt entspricht.

- **Higher-order** Funktionen werden durch Komposition von bestehenden Funktionen konstruiert. Sie akzeptieren Funktionen als Argumente, was durch die Betrachtung von Funktionen als first-class Werte ermöglicht wird.

- **Currying** oder teilweise Parametrisierung von higher-order Funktionen ist die Vorgabe von einigen Argumenten einer Funktion, woraus eine speziellere Funktion resultiert. Alle Funktionen mit mehreren Argumenten dürfen parametrisiert werden, deshalb wird die Typendeklaration mit Typenfunktionen ausgedrückt: $\alpha \rightarrow (\alpha \rightarrow \alpha)$

- **Listen** sind eine wichtige Datenstruktur in funktionalen Sprachen. Eine Liste besteht aus der leeren Liste „[ ]“ und beliebig vielen Elementen vom gleichen Typ. Im Gegensatz zu imperativen Sprachen, wo eine Liste aus mit Zeiger verketteten dynamischen Speicherplätzen besteht, erlauben funktionale Sprachen keinen Zugriff auf den Speicher und die Zeiger einer Liste. Diese Abstraktion eliminiert potentielle Fehler und überlässt die Speicherverwaltung dem System (automatic garbage collection). Die Darstellung von Listendefinitionen kann durch **List Comprehension** vereinfacht werden: dabei können wie bei einer mathematischen Mengennotation Bedingungen für die Elemente einer Liste angegeben werden.

2.5.3 Grammatik einer funktionalen Beispielsprache

Dieser Abschnitt stellt eine vereinfachte Grammatik der funktionalen Sprache GOFER vor [Jon91]. GOFER ist ein Dialekt oder eine Weiterführung von HASKELL [Hud92]. Von den vorgestellten Konzepten unterstützt die Sprache lazy evaluation, ein polymorphes Typsystem mit benutzerdefiniertem Überladen, benutzerdefinierte Datentypen, higher-order Funktionen, Patternmatching und List Comprehension.

Notation

Die Darstellung erfolgt analog zu den anderen Kapiteln in EBNF. Weil die Grammatik fast die ganze Funktionalität von GOFER umfasst ist sie sehr umfangreich und beinhaltet einige komplexe Konstrukte. Die gesamte Grammatik ist im Anhang A.3.2 zu finden, an dieser Stelle wird exemplarisch die Umsetzung einiger Konzepte in die Syntax erklärt.

Die Grammatik verwendet die folgenden terminalen Symbole: „Varid“ steht für Funktionen und Variablen und beginnt mit einem Kleinbuchstaben, dem eine beliebige Folge von Buchstaben, Zahlen und den Zeichen „'“ und „_“ folgt. Benutzerdefinierte Funktionen werden mit „Conid“ bezeichnet und müssen mit einem Grossbuchstaben beginnen. Ansonsten setzen sie sich wie Varid zusammen. „Varop“ bezeichnet Operatoren, die aus den folgenden Symbolen gebildet werden dürfen:

```
! # $ % & * + . / < = > ? @ \ ^ | -
```

„Conop“ bezeichnet benutzerdefinierte Operationen, die aus den selben Symbolen wie Varop bestehen dürfen, aber mit einem Doppelpunkt „:“ beginnen müssen. Funktionsnamen, die zwischen zwei Apostroph stehen, werden in der Grammatik als Operatoren betrachtet „`‘Varid‘`“, Operatoren in runden Klammern als Funktionen „`(Varop)`“. Weiter treten ganze Zahlen „Integer“, reelle Zahlen „Float“, einfache Buchstaben „Char“ und Zeichenketten „String“ auf.

Syntax

GOFER kann man auf zwei verschiedene Arten benutzen, deshalb hat die Grammatik zwei Startsymbole. Das Symbol <Interpreter> beschreibt die Syntax von Ausdrücken, die interaktiv in den Gofer Interpreter eingegeben werden können. Es handelt sich dabei um die selben Ausdrücke, wie sie in Programmen verwendet werden, die mit <Module> starten und aus einer Reihe von Definitionen bestehen. Diese Programme werden als Datei in GOFER geladen.

```
<Interpreter> ::= <Expr> [[ <Where> ]]                    -- top-level expression
<Module>      ::= { <Topdeclaration> }                    -- top-level module
```

Ein Programm besteht aus mehreren Deklarationen, von denen jede mit Strichpunkt beendet wird. Wertedeklarationen stehen zwischen geschweiften Klammern. Ein benutzerdefinierter Datentyp beginnt mit dem Schlüsselwort „DATA“, führt einen neuen Namen ein, allfällige Argumente und Konstruktoren, die beschreiben, wie Elemente des Typs konstruiert werden. Dabei wird mit einer Liste von Konstruktoren (getrennt durch „|“)angegeben, wie Typen mit einem neuen Infixoperator oder neuen Funktionsbezeichner zusammenhängen. Ein Synonym für einen bereits bekannten Typ wird mit der „TYPE“-Deklaration eingeführt. Vorrang und Gruppierung der Operatoren können in jedem Programm nach Belieben deklariert werden: Die Schlüsselwörter „INFIXL“, „INFIXR“ und „INFIX“ geben die Assoziativität und die nachfolgende Zahl die Priorität der Operatoren in der Operatorenliste an. Eine Typklasse („CLASS“) besteht aus einer Kopfzeile, in der der Name und die Parameter eingeführt werden. Optional kann der Bereich der verwendbaren Typen mit <Context> eingeschränkt werden. Im <Where>-Teil werden abstrakte Funktionen und Standarddeklarationen aufgeführt. Diese müssen von den Typinstanzen („INSTANCE“), die analog aufgebaut sind, konkret implementiert werden.

```
<Topdeclaration> ::= {{ <Topdeclaration> ; }}              -- several declarations
                   | { <Declaration> }                     -- value declarations
                   | DATA Conid <Varlist> = <Constructor>  -- userdefined datatype
                   | TYPE Conid <Varlist> = <Type>         -- synonym type
                   | INFIXL [[ Integer ]] <Oplist>         -- fixity left
                   | INFIXR [[ Integer ]] <Oplist>         -- fixity right
                   | INFIX  [[ Integer ]] <Oplist>         -- fixity nonassociative
                   | CLASS [[ <Context> => ]]
                     Conid {{ <Type> }} [[ <Where> ]]      -- type class
                   | INSTANCE [[ <Context> => ]]
                     Conid {{ <Type> }} [[ <Where> ]]      -- type instance

<Constructor> ::= <Constructor> [[ | <Constructor> ]]      -- several constructors
                | <Type> Conop <Type>                      -- infix constructor
                | Conid {{ <Type> }}                       -- named types
```

Die folgenden Deklarationen treten in normalen Deklarationsblöcken und im <Where>-Teil der Typklassen und -instanzen auf. Mit der Typdeklaration können die Typen der Variablen, die in Funktionen und Pattern auftreten, angegeben werden. Der optionale Kontext schränkt den Bereich der Typen ein. Eine Funktion kann in verschiedenen Formen auftreten: als konstanter Bezeichner, als Infixoperator, als Funktion mit Argumenten oder als geklammerte Funktion. Dem Funktionskopf wird die rechte Seite der Funktionsgleichung zugewiesen, die aus einem einfachen Ausdruck oder mehreren Bedingungen besteht; diese „guards“ werden durch „|“ getrennt. Optional folgen lokale Deklarationen im <Where>-Teil. Die Bindung von Pattern ist syntaktisch analog aufgebaut. Damit werden hauptsächlich Variablen definiert.

```
<Declaration> ::= {{ <Declaration> ; }}                    -- multiple declarations
                | <Varidlist> ::
```

```
                  [[ <Context> => ]] <Type>                  -- type declaration
                | <Fun> <Rhs> [[ <Where> ]]                   -- function binding
                | <Pattern> <Rhs> [[ <Where> ]]               -- pattern binding

<Fun> ::= <Qvarid>                                           -- constant
        | ( <Fun> )                                          -- parenthesised fun
        | <Fun> <Pattern>                                    -- fun with argument
        | <Pattern> <Qvarop> <Pattern>                       -- infix operator

<Rhs> ::= {{ | <Expr> = <Expr> }}                            -- guarded rhs
        | = <Expr>                                           -- simple righthand side

<Where> ::= WHERE { <Declaration> }                          -- local definitions
```

Die eingebauten Basistypen werden mit *Boolean*, *Integer*, *Float* oder *Character* bezeichnet. Optional kann ein Synonym zu einem vordefinierten Typ angegeben werden. Der Typ einer Funktion besteht aus den Typen ihrer Argumente und dem Typ des Resultats, jeweils getrennt durch einen Pfeil. Listen können von jedem beliebigen Typ gebildet werden, indem der Typ in eckige Klammern gesetzt wird. Ein Tupel fasst eine fixe Anzahl beliebiger Typen in runden Klammern zusammen. Typenvariablen sind Variablen, die in Typendeklarationen verwendet werden. Ein Spezialfall ist der Typ *unit*, der nur ein Element „()“ hat, das zu allen Typen gehört und bei theoretischen Betrachtungen funktionaler Sprachen interessant ist.

```
<Type> ::= Conid {{ <Type> }}                                -- predefined datatype
         | <Type>  ->  <Type>                                -- type of function
         | [ <Type> ]                                        -- list type
         | ( <Type> {{ , <Type> }} )                         -- tuple type
         | Varid                                             -- type variable
         | ()                                                -- unit
```

Ein typisch funktionaler Ausdruck ist die Definition von anonymen oder Lambdafunktionen, die mit einem dem griechischen Zeichen „λ“ ähnlichen Schrägstrich „\“ beginnen, gefolgt von einer beliebigen Anzahl Pattern, von denen jedes für einen Parameter steht, der im Ausdruck rechts vom Pfeil zur Auswertung des Funktionsresultates verwendet wird. „LET“ führt lokale Deklarationen für die nach „IN“ folgenden Ausdrücke ein. Anhand des booleschen Ausdrucks nach „IF“ wird die Selektion zwischen zwei alternativen Ausdrücken entschieden, die beide vom gleichen Typ sein müssen, der auch Typ des ganzen Ausdrucks ist. Beliebige Listenausdrücke stehen in eckigen Klammern „[ ]“. Explizite Typendeklaration erreicht man durch Hinzufügen von „::“, einem optionalen Kontext und einer Typangabe hinter dem zu deklarierenden Ausdruck. Die restlichen Ausdrücke folgen dem Aufbau der anderen Paradigmen und werden hier nicht näher erläutert.

Zwei Formen zur Definition von Listen: Die List Comprehension besteht aus einem Ausdruck, der die Elemente manipuliert, und verschiedenen zusammengesetzten Ausdrücken, die die Elemente generieren, definieren oder selektieren. Für eine arithmetische Sequenz muss der Startwert als erster Ausdruck angegeben werden. Der zweite Ausdruck, der die Schrittweite der Sequenz bestimmt, und der dritte Ausdruck, der letzte Wert in der Liste, sind optional. Falls beide weggelassen werden, wird eine unendliche Liste mit einfacher Schrittweite produziert.

```
<Expr> ::= \ <Pattern> {{ <Pattern> }} -> <Expr>             -- lambda expression
         | LET { <Declaration> } IN <Expr>                   -- local definition
         | IF <Expr> THEN <Expr> ELSE <Expr>                 -- conditional
         | [ <List> ]                                        -- list expressions
         | :: <Expr> :: [[ <Context> => ]] <Type>            -- typed expression

<List> ::= <Expr> {{, <Expr> }}                              -- (enumerated) list
         | <Expr> "|" <Qualifier>                            -- list comprehension
```

```
        | <Expr> [[ , <Expr> ]] .. [[ <Expr> ]]          -- arithmetic sequence

<Qualifier> ::= <Qualifier> {{ , <Qualifier> }}         -- multiple qualifiers
              | <Pattern> <- <Expr>                     -- generator
              | <Pattern> = <Expr>                      -- local definition
              | <Expr>                                  -- boolean guard
```

Beim Start des GOFER Interpreters wird jedesmal eine Datei namens „Standard Prelude“ eingelesen, die die gängigen Operatoren, Datentypen und Funktionen definiert, welche während der ganzen Session zur Verfügung stehen. Diese Elemente entsprechen in etwa den eingebauten Sprachkonstrukten anderer Paradigmen und werden mit der Vorlage der vorgestellten Grammatik definiert. Deshalb finden sich in dieser Beispielgrammatik keine expliziten Operatoren oder Funktionen, sondern nur die Basis einer funktionalen Sprache.

2.5.4 Abstrakte Syntax

Die Übersetzung der vorgestellten Beispielgrammatik nach Gentlecode verläuft analog zu derjenigen im Kapitel über imperative Sprachen beschriebenen (2.3.4). Einige zusätzliche Syntaxelemente wurden der Grammatik (A.3.2) hinzugefügt, die Motivation dazu ist in der Diskussion zu finden. Die abstrakte Syntax der funktionalen Beispielsprache resultiert ebenfalls aus den Typen und Funktoren ihres Parsers. Sie ist im Anhang A.4.2 zu finden.

2.5.5 Diskussion

Dieser Abschnitt liefert eine Begründung der Unterschiede zwischen der funktionalen Beispielgrammatik und ihrer Vorlage, der funktionalen Sprache GOFER. Anschliessend folgt eine Diskussion einiger Probleme und Konflikte, die bei der Implementation der vorgestellten Grammatik aufgetreten sind. Schliesslich werden offene Fragen und Anmerkungen für die kommenden Kapitel formuliert, die funktionale Sprachen betreffen.

Unterschiede zwischen der Beispielgrammatik und Gofer

Die Grammatikbeschreibung von GOFER unterscheidet zwischen Typklassen-, Typinstanzen- und normalen Wertedeklarationen. In der Beispielgrammatik wurden die drei Arten unter einem generellen Nonterminal <Declaration> zusammengefasst, dessen Regeln von allen Arten abgeleitet werden können. Es ist dadurch möglich, Code als richtig zu parsen, der in GOFER falsch ist. Praktisch wird jedoch angenommen, dass nur semantisch korrekte Eingaben eingegeben und geparst werden. Nicht-korrekte Ableitungen werden somit nicht betrachtet.

Die Schreibweise einiger Elemente der Beispielgrammatik musste gegenüber der Grammatik von GOFER verändert werden, um Konflikte zu vermeiden: Zur Unterscheidung von verschiedenen Arten von Nonterminals wird ein Kontext geschaffen, der die Zugehörigkeit der Elemente eindeutig bestimmt. So kann zum Beispiel erst in einem grösseren Zusammenhang entschieden werden, ob die syntaktisch identische Regel, bestehend aus einem Conid und einer Typliste, zu <Predicate>, <Type> oder <Constructor> gehört. Durch syntaktisch redundante Zeichen, etwa verschiedene Klammern, wird die Zugehörigkeit bereits bei LALR(1) klar und der reduce/reduce-Konflikt aufgelöst. Das gleiche Prinzip wird auch zur Unterscheidung von normalen und alternativen Guards angewendet, hier mit verschiedenen Pfeilen.

Die Patternregel „$< Qvarid > + Integer$“ wird von GOFER als Funktion interpretiert, das heisst, als Anwendung des Operatorsymbols „+“ behandelt. Deshalb wurde diese Regel nicht in den Gentlecode übertragen.

Parsingprobleme und Lösungen

In Abbildung 2.4 sind die Regeln für Operatoranwendung bei Ausdrücken und Pattern der vorgestellten Syntax mehrdeutig. Sie produzieren einen shift/reduce-Konflikt, da nicht klar ist, ob zuerst der linke oder der rechte Operand abgeleitet werden soll. Durch Ersetzen des einen Operanden mit einem hierarchisch tieferen Nonterminal wird eine Bindung geschaffen und der Konflikt wie in Abbildung 2.5 umgangen. Prinzipiell können mit den Regeln für die Deklaration von Operatoren beliebige Operatoren definiert werden. Aus praktischen Gründen beschränkt sich die Beispielgrammatik auf eine Gruppe von Operatoren, die alle gleich behandelt werden. Da die üblichen Operatoren linksassoziativ sind, wird der rechte Operand ersetzt [Aho86].

```
'nonterm' Pattern(-> PATTERN)                    -- pattern

   'rule' Pattern(-> application(Left, Op, Right)):    -- operator application
          Pattern(-> Left) Qconop(-> Op) Pattern(-> Right)
   'rule' Pattern(-> patlist(Pat, PatList)):           -- function application
          Pattern(-> Pat) Pattern(-> PatList)
   'rule' Pattern(-> patsecl(Left, Op)):               -- left section
          "(" Pattern(-> Left) Qconop(-> Op) ")"
   'rule' Pattern(-> patsecr(Op, Right)):              -- right section
          "(" Qconop(-> Op) Pattern(-> Right) ")"
   'rule' Pattern(-> var(V)):                          -- variable
          Varid(-> V)
```

Abbildung 2.4: Mehrdeutige Regelaufstellung für Pattern in der funktionalen Grammatik.

Allgemein gilt: Damit zwischen der Operatoranwendung von Ausdrücken und der Section-Schreibweise kein shift/reduce-Konflikt entsteht, müssen die entsprechenden Operanden in der gleichen Hierarchiestufe sein. So werden zum Beispiel in Abbildung 2.5 die Section-Regeln (5. und 6. Regel) an die Application-Regel (1. Regel) angepasst.

Die Betrachtung einer Sequenz von Pattern als Patternliste erzeugt mehrere shift/reduce Konflikte: Da der Parser bei einer Liste von atomaren Pattern nichts über ihre Länge weiss, shiftet er jeweils zum nächsten Element und beginnt erst mit der Reduktion, wenn er beim letzten Pattern angelangt ist. Diese Sichtweise ist jedoch zu generell. Konkret handelt es sich hier um eine Funktion, das erste Pattern bezeichnet ihren Namen, und eine Reihe von Argumenten, nämlich die restlichen Pattern. Werden diese Argumente in Klammern angegeben, tritt nur noch ein shift/reduce-Konflikt mit der weiterführenden Regel auf. Mittels eines Schlüsselwortes vor einer Funktion würde auch dieser Konflikt gelöst. Da es sich dabei um das *dangling else* Schema handelt und die Beispielgrammatik nicht zu stark von Gofer abweichen soll, wird nur die Klammerung der Funktionsargumente neu eingeführt und der Konflikt stehen gelassen.

In einer Bindung der Form `<Qvarid> = <Expr>` kann <Qvarid> sowohl eine Funktion als auch ein Pattern sein, dadurch wird ein reduce/reduce-Konflikt erzeugt. Gofer behandelt zwar Pattern dieser Form als Funktionen, wird nun diese Regel unter <Pattern> gestrichen, so können einfache konstante Pattern nicht mehr auftreten, was jedoch im allgemeinen unbedingt nötig ist. Da <Qvarid> als Funktionsname nirgends sonst auftreten kann, wird die Regel zur Produktion von <Qvarid> unter <Function> gestrichen und so der Konflikt umgangen.

Die direkte Umsetzung der Qualifier in Gentlecode ergibt reduce/reduce-Konflikte. Weil <Qvarid>, <Literal> und <Gcon> eine gemeinsame Teilmenge von <Expr> und <Pattern> bilden, kann der Parser beim Erkennen eines dieser Elemente nicht entscheiden, ob er es zu einem Ausdruck oder zu einem Pattern reduzieren soll. Ein Lösungsansatz ist die Unterscheidung

```
'nonterm' Pattern(-> PATTERN)                    -- pattern

   'rule' Pattern(-> application(Left, Op, Right)):    -- operator application
          Pattern(-> Left) Qconop(-> Op) AppPat(-> Right)
   'rule' Pattern(-> X) : AppPat(-> X)

'nonterm' AppPat(-> PATTERN)                     -- application pattern

   'rule' AppPat(-> apppat(Pat, PatList)):             -- function application
          AtomicPat(-> Pat) "(" Patlist(-> PatList) ")"
   'rule' AppPat(-> X) : AtomicPat(-> X)

'nonterm' AtomicPat(-> PATTERN)                  -- atomic pattern

   'rule' AtomicPat(-> patsecl(Left, Op)):             -- l. section, hierarchy like application
          "(" Pattern(-> Left) Qconop(-> Op) ")"
   'rule' AtomicPat(-> patsecr(Op, Right)):            -- right section
          "(" Qconop(-> Op) AppPat(-> Right) ")"
   'rule' AtomicPat(-> var(V)):                        -- variable
          Varid(-> V)
```

Abbildung 2.5: Eindeutige Aufteilung der Grammatikregeln für Pattern.

aller möglichen Fälle oder Faktorisierung: in einem Guard werden nur boolesche Ausdrücke zugelassen, diese Teilmenge wird unter einem neuen Nonterminal zusammengefasst. Zur Lösung der Konflikte wird aber der Ansatz verwendet, den M. P. Jones bei der Implementation des GOFER Parsers einführt [Jon94]. Die Pattern werden durch Ausdrücke ersetzt, da <Expr> prinzipiell mächtiger ist als <Pattern>, die überzähligen Elemente werden in der semantischen Analyse aussortiert. Dadurch lassen sich aber die folgenden Pattern nicht als Teil von <Qualifier> verwenden: wildcard, irrefutable und as-Pattern.

```
-- original Gofer rules
<Qualifier> ::= <Expr>
              | <Pattern> <- <Expr>
              | <Pattern> = <Expr>
```

```
-- less general, but conflict free
<Qualifier> ::= <Expr>
              | <Expr> <- <Expr>
              | <Expr> = <Expr>
```

Sollen die drei Deklarationen für Vorrang und Gruppierung („INFIXL“„INFIXR“, „INFIX“) mit einem einzigen Funktor zusammengefasst werden, der drei Werte für Assoziativität (left, right, none) aufnimmt? Um diese Schlüsselwörter als Werte eines Typs einzuführen, muss ein neuer Typ definiert werden („FIXITY“), der für jedes Wort einen Funktor hat. Diese Funktoren werden im generalisierten Funktor für Vorrang und Bindung verwendet. Bei diesem Ansatz sind in der abstrakten Syntax 4 Funktoren und 2 Typen beteiligt, gegenüber 3 Funktoren und einem Typ, wenn jede der Regeln in einen eigenen Funktor übertragen wird. Die Variante ohne zusätzlichen Typ wird im Hinblick auf eine möglichst minimale generelle Zwischensprache bevorzugt.

```
-- introduces new type FIXITY
'type' TOPDECL = fixity(FIXITY, INT, OPLIST).

'type' FIXITY = left, right, infix.
```

```
-- straight forward
'type' TOPDECL = infixleft(INT, OPLIST),
                 infixright(INT, OPLIST),
                 nonassoc(INT, OPLIST).
```

Um einen übertragenen Bezeichner eindeutig als normalen oder benutzerdefinierten Operatoren zu erkennen, werden benutzerdefinierte in doppelte Apostrophs gesetzt, sonstige in einfache. Damit wird auch ein Konflikt beim Parsen umgangen.

Unterschiedliche Versionen einer Regel wurden oft auf verschiedene Funktoren abgebildet. Falls es sich um semantisch ähnliche Regeln handelt, wird einfacher ein genereller Funktor verwendet. Dadurch enthält die abstrakte Syntax eine kleinere Anzahl von Funktoren, die im nächsten Arbeitsschritt auf die generelle Zwischensprache abgebildet werden müssen.

2.6 Logische Programmiersprachen

In die Gruppe der deklarativen Sprachen, die spezifizieren, was das zu lösende Problem ist, und nicht wie es gelöst werden soll, gehören ausser den funktionalen Sprachen auch die Programmiersprachen des logischen Paradigmas. Ein logisches Programm besteht aus Fakten und Regeln eines Problems und überlässt die Lösung dem System. Die Programmiersprache verwendet ihre logischen Kontrollstrukturen zur Deduktion der Antworten. Logische Programmierung basiert auf der mathematischen Logik, insbesondere der prozeduralen Interpretation von Hornklauseln.

Als Vorlage für dieses Kapitel dienen das „SICStus Prolog User's Manual“ [Sic96], die Kapitel über logische Sprachen in „Programming Languages: An Interpreter Based Approach“ [Kam90] und „Programming Language Essentials“ [Bal94] und die Abschnitte über logische Programmierung im „Lexikon Informatik“ [Lex97].

2.6.1 Prinzipien

Die meisten logischen Sprachen basieren auf der **Hornlogik**, einer Untermenge der Prädikatenlogik 1. Stufe, die in ihrem Kalkül nur die Quantifizierung von Variablen, aber nicht von Prädikaten erlaubt. Mit SLD-Resolution, die automatisierbar und auf Hornklauseln beschränkt ist, kombiniert mit Unifikation, sind Systeme für logische Sprachen möglich, die für positive Antworten korrekt und vollständig sind. Die Restriktion auf die Hornlogik macht Sinn, weil damit Probleme spezifiert und automatisch ausgeführt werden können.

Eine **Hornklausel** oder Formel hat die generelle Form

$$G_0 \;\; if \;\; G_1, G_2, \ldots G_n.$$

wobei die Bedingungen $G_1, G_2, \ldots G_n$ (**Body**) erfüllt sein müssen, um auf G_0 (**Head**) zu schliessen. Die Kommata im Body stehen für logische Konjunktionen. „if“ ist nicht ausschliessend, es kann weitere Klauseln geben, die Head spezifizieren. G_0 bis G_n werden mit **Goal** bezeichnet. Jedes Goal besteht aus einem Prädikatnamen und beliebig vielen Argumenten. Falls keine Bedingungen angegeben werden, wird ein **Fact** ausgedrückt, der nur aus einem Schluss besteht.

Einem System können Fragen (**Queries**) über die eingegebenen Hornklauseln gestellt werden, indem Goals zum Beweisen eingegeben werden. Das System beantwortet sie durch Pattern Matching der Facts und Regeln, Unifikation von Variablen mit Termen und Backtracking, wenn ein Goal nicht erfüllt werden kann.

2.6.2 Konzepte

Die praktische Umsetzung der Prinzipien des logischen Paradigmas wird durch die folgenden Konzepte realisiert:

- Die **Ausführung der Hornklauseln** nimmt das System automatisch vor. In welcher Reihenfolge der Beweis von alternativen Klauseln für ein Goal und der Beweis von Goals in einem Body versucht wird, ist im reinen Modell undeterminiert, die Reihenfolge ist

dem Programmierer nicht bekannt. Ein Problem kann als UND/ODER-Baum dargestellt werden und die Suchstrategien als Baumtraversierungen. Kann kein Beweis eines Query gefunden werden, ist dies keine Widerlegung, sondern bedeutet nur, dass es mit den vorhandenen Regeln und Fakten nicht abgeleitet werden kann. Die Ausführung der Hornklauseln entspricht in etwa einem Prozeduraufruf, jedoch ist die Argumentenübergabe nicht spezifiziert und Backtracking wird automatisch ausgeführt.

- **Resolution** ist ein algorithmisches Semi-Entschiedungsverfahren zur Prüfung, ob ein Schluss der Prädikatenlogik gültig ist. Der Schluss ist genau dann gültig, wenn das Verfahren nach endlich vielen Schritten abbricht. Die Resolution ist ein Satz der Prädikatenlogik 1. Stufe und Grundlage des darauf beruhenden maschinellen Beweisverfahrens. Die Resolutionsmethode ist vollständig, da ein existierender Beweis eines Query immer gefunden wird. Praktische Beschreibung der Resolution: falls der Head einer Klausel auf ein Goal passt, kann dieses Goal durch den Body der Klausel ersetzt werden.

- **Pattern Matching** dient dazu, festzustellen, ob zwei Muster/Pattern einander entsprechen. Dies ist dann der Fall, wenn sie identisch sind oder wenn es durch Variablenersetzung möglich wäre, sie identisch zu machen. Im Verlauf des Pattern Matching werden die Variablen an die sie ersetzenden Ausdrücke gebunden.

- **Unifikation** ist die Bindung von logischen Variablen an einen Wert oder eine andere Variable bei der Resolution und resultiert in einer Substitution. Dabei gelten die folgenden Regeln: Ein konstanter Wert kann nur mit sich selbst unifiziert werden. Eine ungebundene Variable kann beliebig gebunden werden. Ein strukturierter Term kann nur mit einem anderen Term unifiziert werden, der die gleiche Signatur hat und dessen Argumente rekursiv unifiziert werden können.

- Hornklauseln verwenden **logische Variablen**, die weder deklariert noch initialisiert werden. In einer Klausel bezeichnet eine logische Variable immer das gleiche unspezifizierte Objekt (ähnlich wie eine Variable in einer mathematischen Gleichung). Variablen in verschiedenen Klauseln sind voneinander unabhängig, der lexikalische Bereich einer logischen Variablen beschränkt sich auf eine einzelne Klausel. Durch Pattern Matching wird eine logische Variable genau einmal an einen Wert gebunden, ausser es ergibt sich beim Backtracking ein neuer Wert.

  In Klauseln sind logische Variablen universell quantifiziert, in Queries sind sie existentiell quantifiziert. Logische Variablen sind sowohl Input- wie auch Outputparameter, ihre Übergabe erfolgt als call-by-value, respektive call-by-result, ohne vorherige Spezifikation. Hornklauseln sind demnach eher Relationen als Funktionen, da sie eine Beziehung zwischen verschiedenen Argumenten aufstellen und ihre Argumente nicht als In/Output markiert sind.

- **Datenstrukturen** oder zusammengesetzte Terme bestehen aus einem Funktorbezeichner und beliebig vielen Komponenten. Sie werden wie Variablen oder Konstanten verwendet und unifiziert. Eine der wichtigsten Strukturen in logischen Sprachen ist die Liste, die aus der leeren Liste und beliebig vielen Elementen besteht.

Die bekannteste logische Sprache ist Prolog, von der viele Dialekte existieren. Weitere logische Sprachen existieren vor allem in Forschungsinstituten und sind kaum allgemein bekannt.

Die logische Sprache PROLOG [Sic96] ist eine beschränkte Implementation des Hornkalküls, angereichert mit vielfältigen primitiven Prädikaten für Arithmetik, Ein- und Ausgabe, Filebehandlung und so weiter. Prolog ist keine reine logische Sprache, da sie folgende Restriktionen

aufweist: Evaluation von Aufrufen in Queries und Bodies von Klauseln geschieht strikt von links nach rechts. Das Backtracking nach alternativen Lösungen wird mit Tiefensuche gemacht, die Klauseln werden in der Reihenfolge abgearbeitet, in der sie im Programm auftreten. Es werden spezielle Prädikate zur Kontrolle des Backtracking und zur dynamischen Manipulation von Klauseln definiert. Die Möglichkeit, vollständige Prolog-Systeme zu implementieren, wird aus Effizienzgründen geopfert, der Programmierer kann die Arbeitsweise des Systems in seine Programme miteinbeziehen. Für effiziente Programme müssen effiziente Formulierungen, die imperativen Algorithmen ähnlich sind, verwendet werden.

2.6.3 Grammatik einer logischen Beispielsprache

Als Grammatik einer logischen Beispielsprache wird eine Untermenge der Syntax von SICStus Prolog verwendet. Die vollständige Syntax findet sich im Kapitel „Full Syntax“ des User Manual [Sic96]. Auf die eingebauten Prädikate wird verzichtet, ebenso auf den vollständigen Zeichensatz in den Token. Prinzipiell werden alle Konzepte des logischen Paradigmas unterstützt. Die Grammatik, die im folgenden detailliert vorgestellt wird, ist zusammenhängend im Anhang A.3.3 aufgeführt.

Terminale

Die Grammatik wird in EBNF dargestellt und verwendet sechs terminale Symbole: „Word“ beginnt mit einem Kleinbuchstaben, dem beliebige Buchstaben, Zahlen und „_“ folgen dürfen und wird zur Bezeichnung von Funktionsnamen, Prädikaten oder Konstanten verwendet. „Var“ setzt sich aus den gleichen Bestandteilen zusammen, muss aber mit einem Grossbuchstaben oder „_“ beginnen und steht für Variablen. Falls eine Variable nur einmal in einer Klausel auftaucht, braucht sie keinen Namen und kann mit einem einfachen „_“ als anonyme Variable geschrieben werden. „Integer“ setzt sich aus Zahlen zusammen, die ganze Zahlen ergeben. „Symbol“ besteht aus einer beliebigen Kombination folgender Symbole und wird zur Bildung von Operatoren verwendet:

```
+ - * / \ ^ < > = ` ~ : . ? @ # & $
```

„Char“ ist ein beliebiger Ausdruck zwischen einfachen Anführungszeichen, „String“ zwischen doppelten Anführungszeichen. Word, Symbol und String sind die konstanten elementaren Elemente dieser Grammatik, sie werden unter dem Nonterminal <Atom> zusammengefasst. Darunter fällt auch der Cut „!“, der zur Unterbrechung von Backtracking eingesetzt wird.

Syntax

Ein Programm der logischen Beispielgrammatik beginnt mit dem Startsymbol <Program> und besteht aus beliebig vielen Sätzen, die alle mit einem Punkt abgeschlossen werden. Ein Satz kann mit einem Modulnamen bezeichnet werden, indem der Name, ein atomarer Ausdruck, mit Doppelpunkt vor den Satz gestellt wird. Eine Klausel besteht aus einem Term, gefolgt von „:-“ und ihrem Body. Ein Fact besteht nur aus dem Headterm. Eine Directive, also eine Anweisung an das System, Goals auszuführen, besteht aus einem Body, dem eines der Schlüsselwörter „?-“ oder „:-“ vorangestellt wird. Die erste Directive bildet ein Query, die zweite ein Kommando. Ein Kommando wird wie ein Query behandelt, aber die gebundenen Variablen werden nicht angezeigt und der Benutzer kann kein Backtracking verlangen.

```
<Program> ::= {{ <Sentence> }}                    -- several expressions

<Sentence> ::= <Module> : <Sentence>              -- named sentence
```

```
          | <Term> [[ :- <Body> ]] .          -- clause
          | ?- <Body> .                       -- query
          | :- <Body> .                       -- command
```

Mehrere Bodies werden durch Kommata abgetrennt. Wie ein ganzer Satz kann auch nur ein Teil davon, nämlich ein Body, explizit sichtbar gemacht werden in einem Modul, in dem er nicht definiert wurde, mit dem Modulnamen gefolgt von Doppelpunkt „:“. Zwei alternative Bodies werden durch Strichpunkt getrennt, dies ist eine disjunktive Deklaration einer Klausel. Eine Selektion setzt sich aus einem Term, der Bedingung, gefolgt von „− >“ und einem Body, dem then-Teil, zusammen. Optional gefolgt von einem weiteren Body, dem else-Teil, abgetrennt durch Strichpunkt. Falls einem Term das Schlüsselwort „\+“ vorangestellt wird, handelt es sich um ein nichtbeweisbares Goal, sonst um ein Goal, das beweisen werden muss.

```
<Body> ::= <Body> {{ , <Body> }}          -- several bodies
         | <Module> : <Body>              -- named body
         | <Body> ; <Body>                -- disjunction
         | <Term> -> <Body> [[ ; <Body> ]]  -- if then else
         | [[ \+ ]] <Term>                -- [not provable] goal
         | <Term>                         -- goal
```

Ein Term, der eine Funktion oder ein Prädikat angibt, setzt sich aus einem Atom als Bezeichner und mindesten einem Term als Argument in runden Klammern zusammen. Eine arithmetische, relative oder logische Infixfunktion besteht aus zwei Termen, den Operanden, zwischen denen ein Operator steht. Infixoperatoren sind eine gebräuchliche Schreibweise für binäre Funktionen. Weiter kann ein Term eine Konstante, also eine Zahl oder ein Atom, ein Terminal, also eine Liste oder ein String, oder eine Variable sein.

```
<Term> ::= <Atom> ( <Term> {{ , <Term> }} )   -- compound term
         | <Term> <Op> <Term>                 -- operation
         | <Terminal>                         -- terminal
         | <Constant>                         -- constant
         | Var                                -- variable

<Constant> ::= Integer                        -- integer
             | Float                          -- float
             | <Atom>                         -- atom

<Terminal> ::= <List>                         -- list
             | String                         -- string
```

Diese Darstellung der Grammatik verzichtet auf einige Regeln der implementierten Grammatik, die für das Verständnis nicht nötig sind. Die Implementation unterteilt die meisten Nonterminals, um verschiedene Prioritäten und Bindungen zu erhalten. Damit können auch Entscheidungskonflikte umgangen werden. Die vollständige Grammatik der logischen Beispielsprache ist im Anhang A.3.3 zu betrachten.

2.6.4 Abstrakte Syntax

Die vorgestellte Grammatik einer logischen Beispielsprache wird in Gentle als Frontend implementiert. Die dafür benötigten Typen und ihre Funktoren stellen eine abstrakte Syntax dieser Grammatik dar (A.4.3). Analog zum Abschnitt 2.3.4 werden die Typen auf Java Klassen abgebildet und die Funktoren auf Klassenmethoden. Dies führt zu einer Zwischensprache, die als Template dient, aber noch keinen Code erzeugt.

2.6.5 Diskussion

Zum Schluss dieses Abschnitts über das logische Paradigma werden die Unterschiede zwischen der Implementation der vorgestellen logischen Beispielgrammatik und SICStus Prolog [Sic96] aufgezählt und einige Parsingprobleme diskutiert.

Unterschiede zwischen der logischen Beispielsprache und Prolog

Die Syntaxdarstellung von Prolog hat sowohl für Klauseln als auch für Directives und Grammatikregeln je eine eigene Regel zur Benennung des Nonterminals mit einem atomaren Bezeichner <Module>. Die konkrete Syntax der Beispielsprache A.3.3 fasst diese Benennungsregeln in einer einzigen Regel zusammen, mit der jeder beliebige Satz, also Klausel, Directive oder Grammatikregel, bezeichnet werden kann. In der Benennung von Body besteht kein Unterschied.

Grammatikregeln zu Deklaration benutzerdefinierter Grammatiken sind eine zusätzliche Eigenschaft von Prolog. Sie können in normale Klauseln umformuliert werden, indem die Argumente, die in der Syntax der Grammatikregeln implizit enthalten sind, explizit gemacht werden. Da Grammatikregeln nicht zur Basissprache gehören, werden sie nicht in die Beispielsprache übernommen.

Weitere grammatikalische Elemente von Prolog, die nicht in die implementierte Grammatik übernommen wurden: die vielfältigen Terme und Subterme wurden auf die Operationen beschränkt, die auch in den anderen Paradigmen verwendet werden. Auch von den Token und Zeichen wird nur eine Teilmenge übernommen. Es gibt keine Kommentare.

Parsingprobleme

Bei der Umsetzung der Syntax der Beispielsprache in Gentlecode traten einige Parsingprobleme auf, die wie folgt gelöst wurden:

Um die verschiedenen Formen von Bodies konfliktfrei zu parsen, muss ein zusätzliches Nonterminal <Abody> zur Unterteilung eingeführt werden. Dies führt zu Regeln mit höherer und niedrigerer Priorität. Auch die Regeln der Terme werden durch neudefinierte Nonterminals (<Termlist>, <Term1>, <Term2>) hierarchisch unterteilt. Daraus ergeben sich vier Prioritätsstufen und Linksassoziativität der Operatoren. Die in einem Programm verwendeten Operatoren haben alle die gleiche Priorität, solange keine explizite Deklaration Vorrang und Bindung zuweist.

Bei der Disjunktion von zwei Goals entsteht ein *dangling else* Konflikt, der mit Klammern als begrenzende Schlüsselwörter eliminiert werden kann. Diese Regel kann aber auch als zwei Klauseln geschrieben und somit weggelassen werden:

```
-- s/r conflict
B :- B1 ; B2

-- no conflict
B :- ( B1 ; B2 )
```

```
-- disjunction eliminated
B :- B1.
B :- B2.
```

Ein weiterer shift/reduce-Konflikt der Klasse *dangling else* tritt in der implementierten Grammatik auf und wird belassen, da die Lösung des Parsers, nämlich shiften, in Ordnung ist. Natürlich betrifft der Konflikt die „if_then_else“ Regel, dabei wird die Rolle von „else“ vom Strichpunkt übernommen. Der Konflikt könnte durch ein Schlüsselwort für „Endif“ umgangen werden.

```
Abody ::= Term -> Body [[ ; Body ]]
```

```
-- if then else
```

Ein weiterer Konflikt nach dem *dangling else* Schema wird durch die Regel zur Benennung von Bodies ausgelöst. Weil alle nach dem Doppelpunkt folgenden Bodies an <Module> gebunden werden (siehe Abbildung 2.6, 5. Regel), kann der Parser zwischen einem Komma in dieser Liste und einem Komma in einer Bodyliste nicht mehr unterscheiden. Er handelt dann, wie wenn er ein „else“ vor sich hätte und shiftet. Dieser Konflikt kann umgangen werden, indem die Bindung des Punktes, der hinter jeder Art von Satz steht, gegenüber dem Komma erhöht wird. Um das zu erreichen wird der Punkt so tief wie möglich in der Grammatikstruktur gesetzt, nämlich in der 4. Regel in Abbildung 2.7 hinter dem letzten Body einer Bodyliste.

```
'nonterm' Sentence(-> SENTENCE)                    -- some sentences
   'rule' Sentence(-> nonunit(Head, Bs)):          -- non-unit clause
          Term(-> Head) ":-" Bodies(-> Bs) "."
   'rule' Sentence(-> query(Bs)):                  -- query
          "?-" Bodies(-> Bs)  "."

'nonterm' Bodies(-> BODYLIST)                      -- several bodies
   'rule' Bodies(-> bodies(B, Bs)):                -- conjunction
          Body(-> B) "," Bodies(-> Bs)
   'rule' Bodies(-> body(B)):                      -- one body
          Body(-> B)

'nonterm' Body(-> BODY)                            -- some bodies
   'rule' Body(-> named(M, Bs)):                   -- named body
          Module(-> M) ":" Bodies(-> Bs)
```

Abbildung 2.6: Konfliktsituation bei Body-Regeln. Der Punkt ist in der zweiten Regel.

Nach dem Abschnitt über Module im „Prolog User's Manual“ [Sic96] hat ein Body (oder Goal) höchstens einen expliziten Modulbezeichner. Somit erledigt sich dieser Konflikt von selbst. Sonst müsste man benannte Bodies mit syntaktischer Redundanz kennzeichnen, zum Beispiel klammern.

```
'nonterm' Sentence(-> SENTENCE)                    -- some sentences
   'rule' Sentence(-> nonunit(Head, Bs)):          -- non-unit clause
          Term(-> Head) ":-" Bodies(-> Bs)
   'rule' Sentence(-> query(Bs)):                  -- query
          "?-" Bodies(-> Bs)

'nonterm' Bodies(-> BODYLIST)                      -- several bodies
   'rule' Bodies(-> bodies(B, Bs)):                -- conjunction
          Body(-> B) "," Bodies(-> Bs)
   'rule' Bodies(-> body(B)):                      -- one body
          Body(-> B) "."

'nonterm' Body(-> BODY)                            -- some bodies
   'rule' Body(-> named(M, B)):                    -- named body
          Module(-> M) ":" Body(-> B)
```

Abbildung 2.7: Konfliktfreie Situation, der Punkt ist nach Regel 4 verschoben worden.

Anwendung der Obermenge

Ein <Terminal> in einem <Body> kann direkt abgeleitet werden oder über

$$< Body > \rightarrow < Term > \rightarrow < Terminal > .$$

Weil damit zwei mögliche Wege existieren, um ein terminales Element in einem Body darzustellen, ergibt sich ein r/r-Konflikt. Zur Lösung dieses Konflikts bietet die implementierte Grammatik nur den zweiten Weg über <Term> an. In der Menge, der auf <Term> folgenden Elemente ist ja auch <Terminal>, also ist <Terminal> eine Untermenge von <Term>. Da die Regeln für benutzerdefinierte Grammatiken aus der Beispielsprache gestrichen wurden, und <Terminal> in Body nur in diesem Zusammenhang vorkommt, fällt der Fall weg und wird hier nur der Anschaulichkeit halber erklärt.

Typdeklaration in der logischen Beispielsprache

Die generelle Zwischensprache GZS (Kapitel 3) verlangt die Angabe von Typen bei Deklarationen, aber das Konzept der logischen Variablen in logischen Sprachen beinhaltet keine explizite Typendeklaration dieser Variablen.

Für die Abbildung der logischen Beispielsprache auf die GZS sollten die Variablen typisiert werden. Eine Lösung ist die Angabe von „`any`“, einem Spezialtyp, der logische Variablen bezeichnet.

Logische Variablen haben neben der fehlenden oder beliebigen Typdeklaration auch die Eigenschaft, dass eine logische Variable mit gleichem Namen in verschiedenen Klauseln nicht den gleichen Wert bezeichnet. Eine Zwischenschicht oder ein Interpreter wird diese Mehrdeutigkeiten vor der Codegenerierung lösen müssen. Dies führt für die vorliegende Arbeit aber zu weit; bei der Abbildung auf die GZS beschränkt man sich auf die Angabe des Spezialtyps „`any`“.

Kapitel 3

Generelle Zwischensprache

> Languages die, too, like individuals. They may be embalmed and preserved for posterity, changeless and static, life-like in appearance but unendowed with the breath of life. While they live, however, they change.
>
> *Mario Pei, The Story of Language*

Dieses Kapitel erklärt die Position und die Bedeutung einer generellen Zwischensprache im Zusammenhang dieser Arbeit (vgl. Abbildung 1.2). Es wird gezeigt, wie eine generelle Zwischensprache „GZS“ konstruiert werden kann. Zuerst wird das prinzipielle Vorgehen beschrieben, anschliessend werden die Zwischensprachen, die aus Kapitel 2 resultieren, auf die GZS abgebildet. Die vollständige Auflistung der GZS ist im Anhang A.5 zu finden. Ziel dieses Kapitels ist es, dass ein Frontend eine beliebige Eingabesprache in der vorgestellten Form der GZS liefert.

3.1 Generelle Zwischensprache im Compilerframework

Eine Zwischensprache verbindet das Frontend (6.1) eines Compilers mit dem Backend (6.2), wie in den Abbildungen 1.1 und 6.1 dargestellt. Die generelle Zwischensprache, die in diesem Kapitel konstruiert wird, soll eine beliebige Quellsprache mit dem Backend von Kapitel 6 verbinden. Um eine möglichst sprachunabhängige Zwischensprache zu erhalten, die nicht auf eine Quellsprache und ihr Frontend fixiert ist, soll diese die Eigenschaften verschiedenster Paradigmasprachen umfassen. Sie setzt sich folglich aus Konstrukten zusammen, die aus der Analyse der Beispielsprachen (imperative 2.3, objektorientierte 2.4, funktionale 2.5, logische 2.6) motiviert sind.

Die Frontends der Paradigmasprachen liefern Zwischenprogramme in abstrakter Syntax, die von der jeweiligen Grammatik geprägt ist, wie zum Beispiel die abstrakte Syntax der imperativen Beispielsprache (Abbildung 2.3); dies wird in Abbildung 3.1 mit gestrichelten Pfeilen dargestellt. In diesem Kapitel werden die vier Frontends so umgeformt, dass die vormalige, paradigmaspezifische abstrakte Syntax auf die generelle abstrakte Syntax abgebildet wird. Das heisst, nach diesem „Mapping“ produzieren die Frontends immer abstrakte Syntax im Format der generellen Zwischensprache; in Abbildung 3.1 durchgezogene Pfeile. Das „Mapping“ ist keine zusätzliche Schicht, die zwischen Frontend und Zwischensprache geschoben wird, sondern eine Umformung der semantischen Aktionen in jedem Frontend. Die Grammatiken der Sprachen werden dabei nicht verändert. Ein Frontend für eine hier nicht behandelte Programmiersprache, das mit dem Backend aus Kapitel 7 zusammenarbeiten soll, muss direkt GZS produzieren.

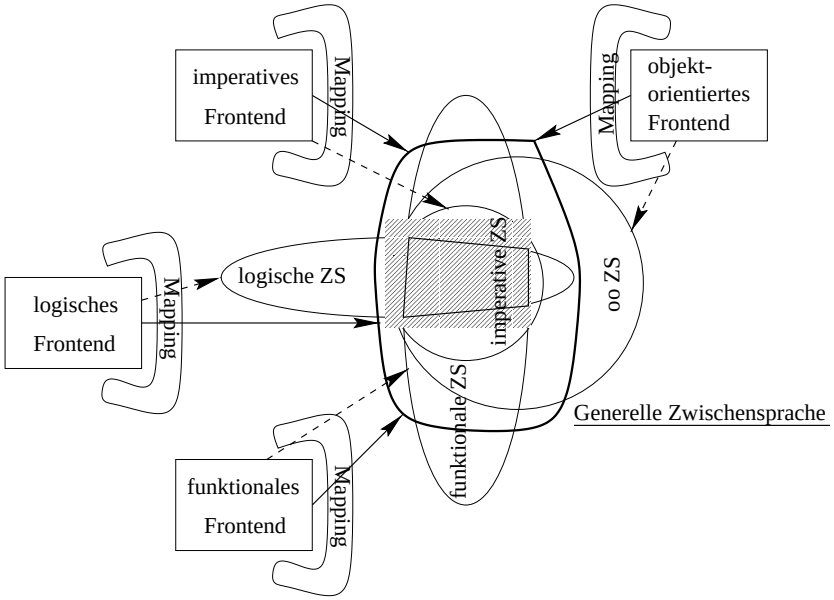

Abbildung 3.1: Mapping der Paradigmasprachen auf GZS .

Die Konstruktion der GZS und die Umformungen der Paradigma-Zwischensprachen finden auf Ebene der abstrakten Syntax statt, da einerseits nur die Bedeutung der syntaktischen Konstrukte interessant ist, die in den abstrakten Typen und Funktoren festgehalten wird, und andererseits die abstrakte Syntax eine geläufige Form der Zwischensprachdarstellung im Compilerbau ist.

Andere Formen von Zwischensprachen sind linearer Pseudocode mit symbolischen Zielmaschinenadressen (3 Adress Code) oder Postfixnotation [Aho86].

3.2 Konstruktion der generellen Zwischensprache

Nach der Einordnung der GZS im Compilerframework und der Motivation für abstrakte Syntax als Zwischensprache wird in diesem Abschnitt die Konstruktion der GZS beschrieben. Die GZS setzt sich aus Elementen der Zwischensprachen der Paradigmasprachen (Kapitel 2) zusammen. Die Auswahl von so wenig Konstrukten wie nötig, die möglichst generell sein sollen, findet nach folgenden Prinzipien statt:

- Jede abstrakte Syntax der entsprechenden Paradigmasprache wird analysiert und ihre gemeinsamen Eigenschaften zusammengefasst. In Abbildung 3.1 ist diese Schnittmenge schraffiert, sie bildet den unproblematischen Kern der GZS . Dazu gehören Bezeichner und Zahlen, diese terminalen Elemente (3.3.7) kommen in jeder untersuchten Sprache vor. Operationen gibt es ebenfalls in allen Sprachen, so gehören auch Ausdrücke (*expressions*) (3.3.6) zum unproblematischen Kern der GZS. Die bedingte Anweisung „`if_then_else`“ ist ebenfalls in jeder Beispielsprache zu finden.

- Da der gemeinsame Kern der GZS nicht als Sprachdefinition ausreicht, gehören der GZS auch Konstrukte an, die nicht in allen Sprachen vorkommen. Bevor ein Element neu in die GZS aufgenommen wird, versucht man, es auf bereits vorhandene abzubilden. Bei der Abbildung unterscheidet man die folgenden Formen:

  - Falls die untersuchten Programmiersprachen verschiedene Namen für die gleiche Bedeutung verwenden, kann dafür ein genereller Name eingeführt werden. Ein Beispiel für diese Namensabstraktion findet sich in Abschnitt 3.3.6, wo der logische Typ „`Term`“, der Ausdrücke bezeichnet, auf den generellen Typ „`Expression`“ abgebildet wird.
  - Falls eine Sprache ein Konstrukt mit sehr genereller Bedeutung enthält, muss dieses Konstrukt aufgeteilt werden (*decomposition*). Dies ist beim funktionalen „`Expression`“-Typ der Fall, der einerseits Ausdrücke, andererseits auch Anweisungen und Deklarationen umfasst; deshalb wird er auf die generellen Typen „`Statement`“ und „`Expression`“ abgebildet (Abbildung 3.9).
  - Spezielle Funktoren, die eine bestimmte Regel in einer Grammatik beschreiben, werden zu generellen Funktoren erweitert, auf die mehrere Konstrukte abgebildet werden können. Diese Generalisierung wird exemplarisch bei der Bildung des generellen Funktors für Schleifen „`loop()`“ dargestellt.

- Spezial- und Problemfälle, die sich weder umbennen, aufteilen noch generalisieren lassen, müssen von Fall zu Fall behandelt werden. Probleme ergeben sich vor allem bei Deklarationen, die sehr sprachabhängige Bedeutungen haben. Auch der Aufbau eines Programmes ist unterschiedlich: ein objektorientiertes Programm kann aus mehreren Klassen bestehen, ein funktionales aus mehreren Topdeklarationen, ein logisches aus einer Reihe von Sentences und ein imperatives aus genau einem Programm.

Abbildung 3.2 ist eine grafische Darstellung der abstrakten Typen, die aus der Anwendung der obigen Prinzipien resultieren. Sie bilden die Grundlage der GZS. Die Pfeile zeigen auf diejenigen Typen, aus denen der abstrakte Typ, von dem sie ausgehen, bestehen kann. Die Abbildung ist also nicht als Transitionsdiagramm oder endlicher Automat zu lesen, sondern als BNF-Diagramm. Terminale Elemente sind doppelt umrandet. Diese grafische Darstellung der Grammatik der GZS soll den Zusammenhang und die Rollen der Typen verdeutlichen. Der Startpunkt für gültige Zwischenprogramme ist `Declarationlist`.

3.2.1 Problem der logischen Beispielsprache

Logische Programmiersprachen sind deklarativ (siehe auch Abschnitt 2.6), das heisst, die für prozedurale Programme erforderlichen Angaben für das Vorgehen beim Ausführen entfallen. In Prolog [Sic96] findet die Steuerung der Abarbeitung im Interpreter statt. Ein eingegebenes Programm funktioniert nur mit einem impliziten Hintergrundmechanismus, einem Resolutionsverfahren, welches mittels Pattern Matching und Unifikation die Gültigkeit der Abfragen aus den Regeln berechnet. Folglich ist ein Programm der logischen Beispielsprache ohne Interpreter unvollständig; es kann nicht auf eine prozedurale (Zwischen-)Sprache abgebildet werden.

Eine potentielle Problemlösung ist ein Interpreter für die logische Beispielsprache. Die Klauseln werden ihm als funktionale Datentypen übergeben, die aus Termen bestehen und ein boolsches Resultat liefern. Der Code des Interpreters und ein logisches Programm werden zusammen als prozedurales Programm betrachtet, dessen Abbildung auf die GZS keine Schwierigkeiten bieten sollte. Das Verfahren soll die gleichen Restriktionen wie der Prolog-Interpreter verwenden, also Abarbeitung der Abfragen in ihrer lexikalischen Reihenfolge, von links nach rechts und Backtracking mit Tiefensuche. Die Ausgabe des Interpreters ist ein boolesches Resultat für jede Abfrage, plus eine Angabe der Variablenbindungen für jedes Query.

Die vorgeschlagene Lösung wird in dieser Arbeit nicht realisiert, man nimmt aber für die Abbildung der logischen Beispielsprache auf die GZS an, dass eine Lösung dieses Problems vorhanden ist.

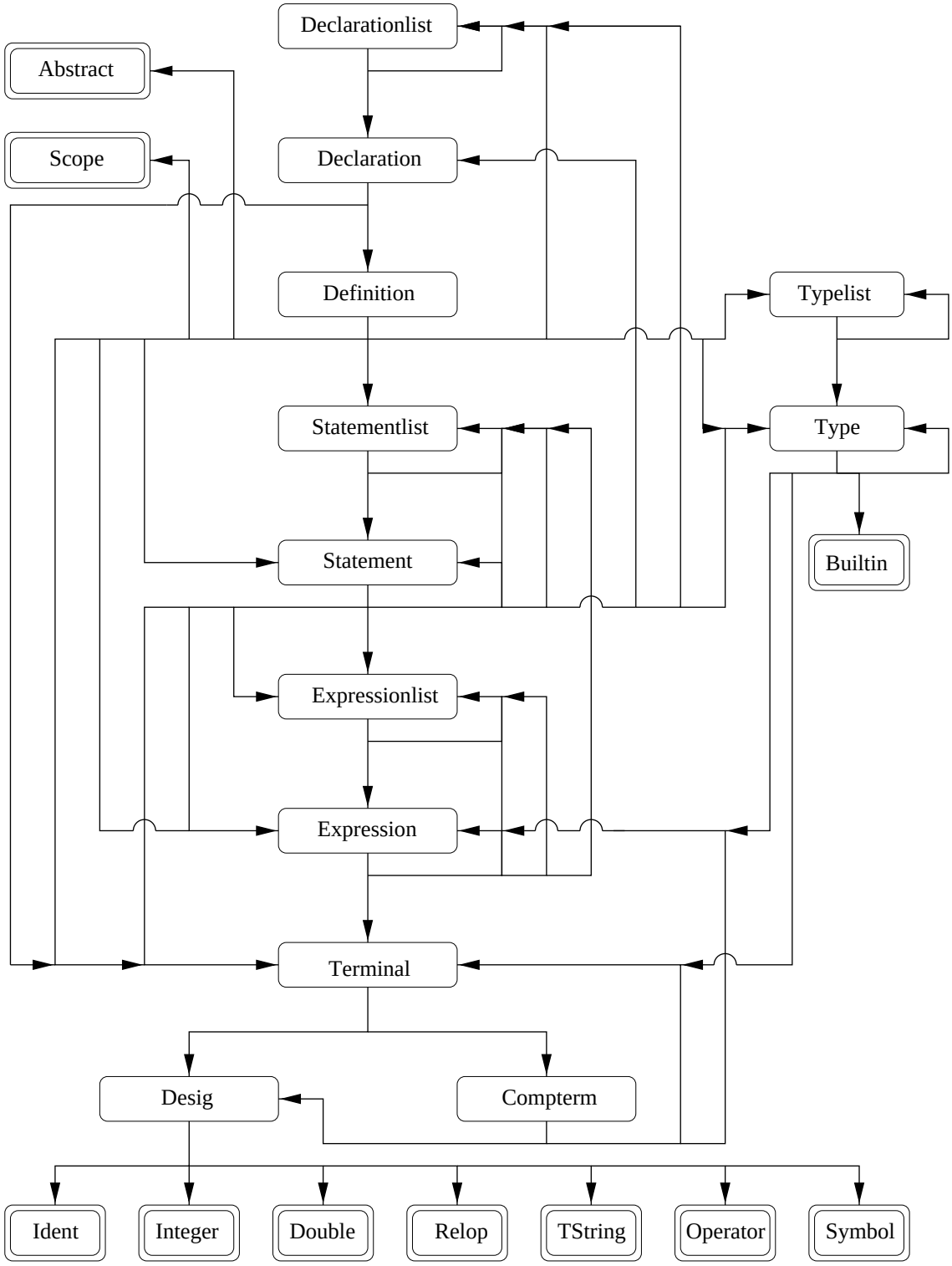

Abbildung 3.2: BNF-Diagramm der GZS-Typen. Die doppelt umrandeten Typen sind Terminale, die einfach umrandeten Nonterminale.

Somit sind Klauseln (überschriebene) Methoden, die durch Queries aufgerufen werden, jeder Parameter kann Input oder Output sein. Der Aufruf der richtigen Methode erfolgt durch den Interpreter, von oben nach unten, mit Backtracking! Ein Query resultiert mit einer Variablenbindung und einem Wahrheitswert. Ein Command erzeugt nur ein boolsches Resultat. Die Deklaration von Variablen geschieht implizit bei ihrer Verwendung.

3.3 Abbildungen auf die generelle Zwischensprache

Im folgenden Abschnitt werden zuerst die abstrakten Typen der Beispielsprachen auf die abstrakten Typen der GZS abgebildet, anschliessend werden für jeden abstrakten Typ die generellen Funktoren aufgestellt. Die Figuren der Unterabschnitte ergeben zusammen die GZS, die zusammenhängend im Anhang A.5 dargestellt ist.

3.3.1 Abbildung der abstraktenTypen

Tabelle 3.1 zeigt die Übertragung der abstrakten Typen der Beispielsprachen auf die abstrakten Typen der GZS. Die Tabelle ist so zu lesen, dass die speziellen abstrakten Typen einer Zeile auf den generellen abstrakten Typ der ersten Spalte abgebildet werden. Dabei werden die Typen in „normaler“ Schreibweise durch Umbenennung oder eine einfache Abbildung übertragen. Die Typen, die in den jeweiligen Sprachen neu eingeführt werden, sind mit „Kapitälchen“ bezeichnet, die „*kursiven*“ Typen sind Problemfälle. Leere Felder geben an, dass dieser Typ in der jeweiligen Sprache nicht benötigt wird.

GZS	Sprachparadigmen			
Abstrakte Typen	**Imperative**	**Objektorient.**	**Funktionale**	**Logische**
Declarationlist	Decllist	Decllist	Top-/decllist	*Sentences*
Declaration	Declaration	Declaration	Top-/decl	*Sentence*
Definition	Definition	Definition	Definition	*Clause, Directive*
Statementlist	Stmtseq	Stmtseq	Exprlist	Bodylist
Statement	Statement	Statement	*Expr*, Rhs, Guard	Body
Expressionlist	Exprlist	Exprlist	Exprlist, Patlist	Termlist
Expression	Expression	Expression	Expr, Pattern, Function	Term
Typelist			Typelist	
Type	Type	Type	Type	
Builtin	Builtin	Builtin		
Terminal	Terminal	Terminal	Terminal	Terminal
Desig	Variable, Relop	Variable, Relop, Logop	Identifier, Literal, Operator	Constant, Module, Operator, Atom
Compterm	Compterm	Compterm	Oplist, Varlist	List
Abstract	Abstract	Abstract	Abstract	Abstract
Scope	Scope	Scope	Scope	Scope

Tabelle 3.1: Abbildung der abstrakten Typen

Um für alle Fälle von Fehlererzeugung gewappnet zu sein, wird eine Positionsangabe für jede Struktur geführt (vgl Abschnitt 2.2.3 im Gentle-Teil). Daher kommt der abstrakte Typ „`Pos`“ in allen Paradigmasprachen und in der generellen Zwischensprache vor. Er ist aber kein

Element der Sprachen und tritt nicht in den EBNF-Grammatiken auf, sondern stammt aus der Parserphase und gibt die Position von Sprachelementen an, die bei der Codeerzeugung und der Fehlerausgabe verwendet wird.

3.3.2 Abbildung der Deklarationen

In Abbildung 3.3 ist `Declaration` der generelle abstrakte Typ für die Struktur der verwendeten Objekte und ihre Deklarationen. Deklarationen können sowohl die oberste Stufe einer Sprache, ein Programm oder eine Klasse, Methoden oder Funktionen und einfache Dateneinheiten bestimmen (vgl. Abstraktionsprinzip [Sch94]).

```
'type' DECLARATIONLIST  = decllist( DECLARATION, DECLARATIONLIST ), nil.
'type' DECLARATION      = declare( POS, TERMINAL, DEFINITION ),
                          decls( DECLARATIONLIST ).
```

Abbildung 3.3: Die abstrakten Typen für Deklarationen und -listen in der GZS.

Deklarationen treten in allen untersuchten Sprachen auf, teilweise auf verschiedenen Ebenen. So werden die Funktoren in Abbildung 3.3 zur Deklaration von Klassen, Objekte, Methoden, Funktionen, Parameter, Variablen und Konstanten verwendet. Dabei sind mit dem Funktor „`declare()`“ folgende Elemente festzuhalten: Die Position „`Pos`“ im Quellcode, wo die Deklaration stattfindet. Der Bezeichner „`Terminal`“ des deklarierten Elementes, um es im folgenden zu benennen. Und schliesslich die Art der Deklaration, zur spezifischen Behandlung. Diese ist vom Typ „`Definition`“, der in der funktionalen und logischen Beispielsprache neu zu definieren ist.

Eine Sequenz von Deklarationen wird mit dem generellen Typ „`Declarationlist`“ bezeichnet (Abbildung 3.3). Der Funktor „`decllist()`“ gibt eine Deklaration, gefolgt von einer Deklarationsliste an. Da Deklarationslisten endlich sind, ist der letzte Wert in einer solchen Folge leer, also „`nil`“. Die Angabe der Positionen erfolgt in den einzelnen Deklarationen selbst, ist daher in „`decllist()`“ nicht erforderlich. Deklarationslisten kommen in der imperativen, der objektorientierten und der funktionalen Beispielsprache vor und werden durch Umbenennen der Funktoren und der abstrakten Typen auf die generelle Zwischensprache abgebildet.

Deklarationen der funktionalen Syntax

Dieser Unterabschnitt behandelt die Abbildung der funktionalen Deklarationslisten „`TopDeclList`“ und „`Decllist`“ auf „`Declarationlist`“ der GZS.

„`declare()`“ wird in der funktionalen Beispielgrammatik neu definiert. Die vormaligen Funktoren von „`Topdecl`“ werden dem Typ „`Definition`“ unterstellt. Um die Deklarationen zu bezeichnen, ist ein Wert „`Terminal`“ an „`declare()`“ zu übergeben. Dafür ist „`Simpletype`“ in der Daten- und der Typdeklaration in „`Conid`“ und „`Varlist`“ zu trennen. Für die Typklassen und -instanzendeklaration wird das Prädikat „`Pred`“ in „`Conid`“ und eine Typliste aufgeteilt. Nun steht mit „`Conid`“ ein „`Terminal`“ zur Verfügung, um diese Deklarationen zu benennen.

Vorausgesetzt, dass pro Fixity-Deklaration nur ein Operator festgelegt wird, kann eine allfällige Operatorenliste durch eine Reihe von mehreren Fixities ersetzt werden. Zum Bezeichnen einer Fixity-Deklaration wird ihr Operator verwendet, der vom Typ „`Terminal`“ ist. Zur Angabe der Priorität des Operators ist ein Integer ausreichend, „`Digit`“ wird nicht mehr verwendet. Da der abstrakte Typ „`Oplist`“ nur in diesem Zusammenhang verwendet wird, kann

er eliminiert werden.[1]

Auch die Deklaration von Typen und Memberfunktionen kann man auf einen Bezeichner eingeschränken, wenn man Variablenlisten durch mehrere Deklarationen ersetzt. Die deklarierte Variable wird als „`Terminal`“ in „`declare()`“ verwendet und bezeichnet die deklarierte Definition.

Eine Reihe von Topdeklarationen, Funktionen- und Patternbindungen treten ohne Bezeichner auf und werden als anonyme Deklarationen weiterverarbeitet. Da aber Topdeklarationen nie aufgerufen werden, ist das Fehlen eines Bezeichners kein Verlust oder Einschränkung. Bindungen können „`nil`“ mit dem Funktionsnamen überschreiben und erhalten so einen Bezeichner. Eine Sequenz von Deklarationen ohne Bezeichner soll als Standardprogramm betrachtet und ausgeführt werden. Dieser Speziallfall wird mit dem Funktor „`decls()`“ abgebildet.

Deklarationen der logischen Syntax

Ein Programm der logischen Beispielsprache besteht in der Regel aus einer Reihe von Sentences und umfasst alle Deklarationen und Definitionen von Klauseln und Direktiven, entspricht also dem Typ „`Declaration(list)`“ (3.3). Ein mit einem Modulnamen bezeichneter Satz wird mit dem Namen als terminalem Bezeichner auf „`declare()`“ abgebildet, unbezeichnete Sätze ohne Bezeichner.

Der generelle abstrakte Typ „`Definition`“ (3.4) umfasst den Funktor „`binding()`“, der zur Definition von Klauseln verwendet wird, wobei der Head als Ausdruck, der Body einer Regel als Anweisungen und ein Fakt als *true* übergeben werden. Der Funktor „`resolve()`“ gibt die Anweisungen der Direktiven und die Angabe, ob die Variablenbindung ausgegeben werden soll, an den Interpreter weiter.

3.3.3 Abbildung der Definitionen

Der abstrakte Typ „`Definition`“ vereinigt die Funktoren, die die verschiedenen Arten von Deklarationen beschreiben, siehe Abbildung 3.4:

```
'type' DEFINITION = variable( TYPE, EXPRESSION ),
                    datatypedef( DECLARATIONLIST, TYPELIST ),
                    class( DECLARATIONLIST, TYPE, DECLARATIONLIST ),
                    method( ABSTRACT, SCOPE, TYPE, DECLARATIONLIST,
                            DECLARATIONLIST, STATEMENTLIST ),
                    binding( EXPRESSION, STATEMENT, DECLARATIONLIST ),
                    infixleft( TERMINAL ), infixright( TERMINAL ), nonassoc( TERMINAL ),
                    resolve( STATEMENTLIST, EXPRESSION ).
```

Abbildung 3.4: Der GZS Typ `Definition` und seine Funktoren.

Der Datentyp „`Type`“ einer Variable, einer Konstante oder eines Parameters wird mit „`variable()`“ festgehalten. Wird die Dateneinheit bei der Deklaration initialisiert, wird ihr ein Ausdruck „`Expression`“ zugewiesen. Die explizite Deklaration des Typs einer Funktionenbindung kann, sofern sie ohne Angabe des Kontexts erfolgt, auf „`variable()`“ abgebildet werden. Dadurch erübrigen sich die Funktoren „`typedecl()`“ und „`memfun()`“ der funktionalen Syntax. Eine Methodendefinition „`method()`“ beginnt mit der Angabe, ob die Methode

[1] Die Verwendung der drei Funktoren „`infixleft()`“, „`infixright()`“ und „`nonassoc()`“ im Zusammenhang mit den Fixity-Deklarationen wurde bereits im Kapitel über funktionale Sprachen diskutiert (2.5.5).

abstrakt oder konkret ist („`Abstract`“), gefolgt vom Sichtbarkeitsbereich „`Scope`“, dem Rückgabewert „`Type`“ und einer Liste von formalen Parametern „`Declarationlist`“. Der Body einer Methode besteht aus einer Liste von lokalen Variablen und Anweisungen. Eine Klassendefinition „`class()`“ besteht aus einer Liste von lokalen Parametern „`Declarationlist`“, einem Vorgänger in der Klassenhierarchie „`Type`“ und dem Body der Klasse, einer Liste von Methoden „`Declarationlist`“.

Die Definition eines neuen Datentypen „`datatypedef()`“ besteht, nebst seinem Namen, aus einer Liste von Typvariablen „`Terminal`“ und einer Liste von Konstruktoren „`Typelist`“. Dieser Funktor wird für Datentypdefinitionen und Datentypsynonyme der funktionalen Beispielsprache verwendet.

Definition von Klassen in der funktionalen Sprache

Funktionale Typklassen sind nicht gleich den objektorientierten und generellen Klassen, da der Overloading Ansatz Gruppierung von verwandten Funktionen erlaubt [Jon91]. Eine Typklasse besteht aus einem Bezeichner, einem optionalen Kontext (Supertyp) einer Liste von Typvariablen und einer Liste von Deklarationen, ihrem Body. Der Body besteht aus Memberfunktionen, die ein abstraktes Interface für die Typinstanzen liefern, und Defaultdefinitionen, die von den Typinstanzen überschrieben werden dürfen. Da eine Typklasse abstrakte Methoden hat, wird sie als abstrakte Klasse betrachtet. Eine Typinstanz dagegen ist eine konkrete Klasse, da sie die konkreten Implementationen der Memberfunktionen liefert.

Die Memberfunktionen der Typklassen werden auf abstrakte Methoden und die Funktionsbindungen der Typinstanzen auf konkrete Implementationen dieser Methoden abgebildet. Beim Parsen ist nicht bekannt, ob eine Typdeklaration oder eine Funktionsbindung in einer Typklasse/-instanz oder ausserhalb stattfindet. Also werden Typendeklarationen ohne Angabe ihres Abstraktheitsgrades auf „`variable()`“ abgebildet, Bindungen sind immer konkret. Deshalb müssen in einer Phase vor der Generierung, in welcher der Kontext von Methoden bekannt ist, aus den Informationen der Funktoren korrekte Klassen geformt und die momentan offenen Parameter besetzt werden.

Restriktion: mehrfacher Kontext und Mehrfachvererbung werden von der GZS nicht unterstützt, Quellcode muss entsprechend umgeformt werden. Da die Kontexthierarchie in der funktionalen Beispielsprache unzusammenhängender Vererbung entspricht, wird dieser Fall weggelassen.

Abbildung imperativer Prozeduren auf Methoden

Prozeduren der imperativen Beispielsprache sind eine Teilmenge der Methoden: Die Listen der Deklarationen und der Anweisungen sind sowohl in Prozeduren als auch in Methoden die gleichen, aber Prozeduren sind immer *private* und *static*. Der Typ eines Rückgabewerts wird vom Funktor „`method()`“ (3.4) als Parameter verlangt, ist aber bei imperativen Prozeduren in der Liste der formalen Parameter eingebunden. Daher müssen diese Parameter beim Parsing getrennt werden. Die oberste Deklaration in der imperativen Syntax wird auf „`class()`“ abgebildet.

Abbilden von funktionalen Bindungen auf Methoden

Die Funktionenbindung in der funktionalen Beispielsprache entspricht der Deklaration einer benannten Methode. Die optionale Reihe von Definitionen wird als lokale Deklaration vor den Anweisungen ausgeführt, die den Rückgabewert bestimmen. Die einfache rechte Seite der Funktionenbindung wird als return-Anweisung, die mehrfache, bewachte (*guarded*) rechte Seite

als Verschachtelung von if-Anweisungen abgebildet. Falls die linke Seite einer Funktionsbindung aus einem Namen und Argumenten (Pattern) besteht, kann sie als Methode betrachtet werden. Falls aber der linke Teil einer Bindung aus einem Funktionsausdruck besteht, also in Infix, Section- oder Argumentennotation dargestellt ist, kann der Name und die Parameter der Funktion nicht einfach abgespalten und an „`method()`“ übergeben werden. Weiter kommt bei dieser Abbildung erschwerend dazu, dass Funktionen häufig durch mehrere Bindungen definiert werden, die alle die gleiche Signatur aufweisen, aber nicht als überschriebene Funktionen betrachtet werden können, da die Gültigkeit einer (Teil-)Funktion nicht vom Kontext sondern vom „Pattern Matching“ der aktuellen auf die formalen Parameter abhängt. Wie können nun mehrere Bindungen eines Funktionsnamens in einer Methode zusammengefasst werden? Eine mögliche Lösung ist die Extraktion der fehlenden Informationen zur Bildung einer einzigen Methode aus mehreren Bindungen mit dem selben Namen. Anstatt „`method()`“ umzuändern, werden Bindungen mit dem Funktor „`binding()`“ an die generelle abstrakte Syntax übergeben und in einer Zwischenphase zu einer gültigen Methode geformt. Diese Methode muss ein „Pattern Matching“ auf ihre Teilfunktionen machen, die passende ausführen und deren Wert zurückgeben.

Zusammenfassend lässt sich sagen, dass Bindungen von der Bedeutung her Methoden entsprechen, im Zusammenhang eines ganzen Eingabeprogramms jedoch nur ein Teil einer Methode sind und deshalb als Bindungen weitergegeben und mit „Pattern Matching“ zu einer gültigen Methode ergänzt werden sollen.

Zusammenfassung der Abbildung der Deklarationen und der Definitionen

Mit den abstrakten Typen „`Declaration(list)`“ (Abbildung 3.3) und „`Definition`“ (Abbildung 3.4) wird die Typenstruktur der imperativen und der objektorientierten Beispielsprache auf die GZS übertragen. Deshalb werden auch die Typen zur Angabe der möglichen Werte der Abstraktheit „`Abstract`“ und der Sichtbarkeit „`Scope`“, die aus der objektorientierten abstrakten Syntax stammen, in die GZS übernommen. Durch die beschriebenen Abbildungen werden in der funktionalen Syntax „`Topdeclaration(list)`“ und in der logischen „`Sentence(s)`“, „`Clause`“ und „`Directive`“ nicht mehr benötigt.

3.3.4 Abbildung der Datentypen

Die funktionale Beispielsprache hat das umfassendste Typensystem der hier untersuchten Paradigmasprachen. Deshalb werden die meisten ihrer Funktoren in die GZS übernommen (Abbildung 3.5). Die imperative und die objektorientierte Beispielsprache verwenden nur eingebaute Typen, die logische Sprache verwendet kein Typsystem. Da die logische Beispielsprache ungetyped ist, wird in allen Deklarationen anstelle einer Typangabe „`nil`“ für einen beliebigen Datentyp eingesetzt.

Reduktion der Funktoren

„`datatype()`“ steht für neu definierte Datentypen und umfasst Funktoren, die aus einem Datentypnamen, gefolgt von optionalen Typvariablen bestehen. Unter der Restriktion, dass nur einfache Typvariablen, also weder Listen- noch Tupletypen, verwendet werden, kann im funktionalen Parser eine Variablenliste anstelle der AtomicTypListe substituiert werden. Objektnamen als Datentypen entsprechen variablen Typen, also wird „`object()`“ auf „`datatype()`“ abgebildet, wobei der Objektname als einfacher Bezeichner vom Typ „`Terminal`“ und ohne Typvariablen übergeben wird.

```
'type' TYPELIST = typelist( TYPE, TYPELIST ),
                  constrlist( TYPE, TYPELIST ), nil .
'type' TYPE     = datatype( Type: TERMINAL, Typevars: TYPELIST ),
                  builtin( POS, BUILTIN ),
                  funtype( POS, In: TYPE, Out: TYPE ),
                  array( EXPRESSION, TYPE ),
                  tupletype( TYPELIST ),
                  typevar( TERMINAL ), nil .
'type' BUILTIN  = integer, real, char, boolean, string, void .
```

Abbildung 3.5: Zusammenfassung der GZS Typen.

Die Bedeutung der Prädikate in der funktionalen Beispielsprache ist der Typenkontext von Typklassen und -instanzen, der mit einem Typnamen und eventuell Typvariablen angegeben wird. Dies entspricht gerade dem Funktor „`datatype()`“, also kann der abstrakte Typ „`Predicate`“ auf „`Type`“ abgebildet werden. Da nur ein einfacher Kontext zugelassen wird, ist der abstrakte Typ „`Predlist`“ für Kontextlisten überflüssig, ebenso die dazugehörigen Funktoren.

In der funktionalen Beispielsprache gibt ein Konstruktor an, wie ein benutzerdefinierter Datentyp konstruiert wird, indem ein neuer Typname und Typvariablen in normaler oder Infixschreibweise angegeben werden. Der normale Konstruktor kann wie in 3.6 auf „`datatype()`“ abgebildet werden, der Infixkonstruktor ebenfalls.

Da diese Funktoren auf „`datatype()`“ übertragen werden, wird ihr vormaliger Typ „`Constructor`“ neu durch „`Type`“ ersetzt. Nun kann aber eine Liste von Konstruktoren nicht einfach als Typliste interpretiert werden, weil sie mit '|' unterteilt wird, also disjunktiv ist. Um beim Abbilden die Struktur zu erhalten, wird ein neuer Funktor für Konstruktorlisten eingeführt: „`constrlist()`“.

```
'nonterm' Constructor(-> TYPE)                                  -- constructor
   'rule' Constructor(-> datatype(desig(P, conop(O)), typelist(L, typelist(R, nil)))):
          Type(-> L)  Conop(-> O) @(-> P) Type(-> R)                    -- infix constructor
   'rule' Constructor(-> datatype(desig(P, conid(C)), Ts)):
          Conid(-> C) @(-> P)  TTypelist(-> Ts)                         -- constructor, n>=0
```

Abbildung 3.6: Abbildung des funktionalen „`Constructor`“ auf „`datatype()`“.

Der Funktor „`builtin()`“ aus Abbildung 3.5 bezeichnet vordefinierte und in die Sprache eingebaute Datentypen. Die objektorientierte und die imperative Grammatik lassen vordefinierte Datentypen zu. Diese Datentypen sind nur eine Teilmenge der Datentypen, die die funktionale Grammatik anbietet. Es wird ein neuer Typ definiert , der diese vordefinierten Datentypen umfasst: „`Builtin`“; Funktoren werden von „`Type`“ der objektorientierten Syntax übernommen.

„`array()`“ beschreibt eine Liste von Werten des selben Typs, ein Ausdruck gibt Anfangswert und Länge der Liste an. Imperative und objektorientierte Arrays und funktionale Listen („`listtype()`“) werden auf diesen Funktor abgebildet.

Die folgenden Funktoren werden momentan nur von der funktionalen Syntax verwendet:

„`funtype()`“ bezeichnet den Input- und den Outputtyp von Funktionen. „`tupletype()`“ bezeichnet Datentypen, die aus mehreren Datentypen bestehen, ähnlich wie Records. Der

Funktor „`typevar()`“ bezeichnet Typenvariablen, also Variablen, die im Zusammenhang mit Typausdrücken verwendet werden. „`unit()`“ wird auf „`typelist( nil, nil )`“ abgebildet [Jon91].

Mit den vorgestellten abstrakten Typen erhält die generelle Zwischensprache ein mächtiges Instrument zur Konstruktion beliebig komplexer Datentypen. Der abstrakte Typ „`Type`“ der Beispielsprachen wird in der generellen Zwischensprache beibehalten und um die funktionalen Funktoren erweitert.

3.3.5 Abbildung der Anweisungen

Eine Anweisung ist eine Arbeitsvorschrift, die sich an diejenige Funktionseinheit richtet, welche die Arbeit ausführen soll; sie lenken den Progammfluss. Die GZS berücksichtigt bedingte Anweisungen wie Verzweigung und Wiederholung, und unbedingte Anweisungen wie Zuweisung, Sprungbefehl, Ein- und Ausgabe.

Abbildung 3.7 führt die GZS für Anweisungen auf: „`Statement`“, derjenige für eine Reihe von Anweisungen „`Statementlist`“. Die Funktoren, die der imperativen oder objektorientierten Beispielgrammatik entstammen, bieten keine Probleme bei der Abbildung und werden daher nicht näher beschrieben.

Die Definition einer Variable vom Typ Objekt als Anweisung wird mit „`new()`“ bezeichnet und setzt sich aus dem Bezeichner der neuen Variable, ihrem Typ und einer Parameterliste zusammen. Dies ist ein Funktor für eine Deklaration, aber unter „`Statement`“.

```
'type' STATEMENTLIST = stmtseq( STATEMENT, STATEMENTLIST ), stmt( STATEMENT ), nil.

'type' STATEMENT     = assignment( POS, EXPRESSION, EXPRESSION ),
                       ifthenelse( POS, EXPRESSION, STATEMENT, STATEMENT ),
                       loop( POS, STATEMENT, EXPRESSION, STATEMENTLIST ),
                       compound( STATEMENTLIST ),
                       call( POS, TERMINAL, EXPRESSIONLIST ),
                       read( POS, TERMINAL ),
                       write( POS, EXPRESSION ),
                       return( POS, EXPRESSION ),
                       new( POS, TYPE, TERMINAL, TYPE, EXPRESSIONLIST ),
                       switch( POS, EXPRESSION, STATEMENTLIST, STATEMENTLIST ),
                       case( POS, EXPRESSIONLIST, STATEMENTLIST, DECLARATIONLIST ),
                       declstmt( DECLARATION ),
                       exprstmt( EXPRESSION ), nil( POS ).
```

Abbildung 3.7: Funktoren der GZS für Anweisungen.

Die abstrakte Syntax der funktionalen Beispielsprache muss in Anweisungen und Ausdrücke unterteilt werden, damit eine Abbildung auf die generelle Syntax möglich ist. Als Ausdrücke werden Operationen und in der BNF-Hierarchie (Abb. 3.2) tiefere Ausdrücke beibehalten, alle komplexeren Ausdrücke werden zu Anweisungen.

Abbildung von Switch

„`switch()`“ bezeichnet Auswahlanweisungen, deren verschiedene Fälle aus einer Reihe von Case-Anweisungen „`case()`“ bestehen. Diese Anweisungen kommen in der objektorientierten und der funktionalen Beispielsprache vor. Es folgt die Konstruktion eines generellen Funktors, der alle unterschiedlichen Aspekte berücksichtigt.

Die Reihe der Alternativen entspricht einer Reihe von Case-Anweisungen, da ein Fall ausgeführt wird, falls dessen Pattern erfüllt sind. Folglich entspricht das Pattern einem Label. Eine einfache Alternative ist eine auszuführende Anweisung, deren Wert zurückgegeben wird, eine bewachte Alternative kann in eine verschachtelte If-Anweisungssequenz übertragen werden. Da das objektorientierte Switch keine lokalen Deklarationen kennt, das funktionale aber schon, wird „`case()`“ um einen Parameter für Deklarationslisten erweitert. Der Standardfall „Default“ kommt nur in der objektorientierten Beispielsprache vor. Dank dieser Abbildung werden folgende abstrakte Typen der funktionalen Beispielsprache nicht mehr benötigt: „`Altrhs`“, „`Gdalts`“ und „`Gdalt`“. Die abstrakten Typen „`Rhs`“, „`GdRhs`“ und „`Guard`“ werden ebenfalls auf „`Statement`“ abgebildet, da eine einfache rechte Seite (einer Bindung) als einfache Anweisung und eine bewachte als if-Sequenz ummodeliert werden können.

Anonyme Funktionen

Einige funktionale und logische Konstrukte unter „`Statement`“ lassen sich als Methodendeklarationen abbilden, also wird ein neuer Funktor „`declstmt()`“ eingeführt, um eine Deklaration als Anweisung zu ermöglichen. Die meisten dieser Methoden sind anonym, das heisst, sie können nicht mit einem Bezeichner referenziert werden. Es ist eine Prozedur zum Aufrufen und Ausführen anonymer Methoden zu definieren, etwa mit den auf die Methode folgenden Werten als Parameter.

Das funktionale Konstrukt „`LET .. IN ..`“ beschreibt eine Liste von lokalen Definitionen, die für die folgenden Strukturen gelten. Dies kann als anonyme Methode betrachtet werden, mit der Deklarationsliste als lokale Deklarationen und den Anweisungen als Body der Methode. Da die Deklarationen nur in diesem Konstrukt gelten, ist eine andere Möglichkeit, das Ganze als Compound-Anweisung zu betrachten, dadurch ist die begrenzte Gültigkeit gesichert. Nun müssen die Deklarationen und die Anweisungen zusammen als Anweisungsliste an „`compound()`“ übergeben werden. Da die Implemetierung von anonymen Methoden keine Probleme bieten sollte wird die Abbildung auf „`method()`“ gewählt.

Der Lambda Ausdruck der funktionalen Beispielsprache ist eine Deklaration einer anonymen Funktion, meist gefolgt von einem impliziten Aufruf durch Angabe der aktuellen Parameter. Abbildung 3.8 zeigt links einen Lambda Ausdruck und seine Abbildung auf eine anonyme Funktion, bei der `<Pattern>` die Parameter angibt und `<Expression>` den Funktionsrumpf. Ein Lambda Ausdruck kann in Gofer auch mit einer lokalen Definition formuliert werden, wobei „`newName`“ ein freier Variablenname ist [Jon91]. Eine funktionale Quellsprache, bei der Lambda Ausdrücke nicht erlaubt sind, verliert also keine Funktionalität gegenüber einer Sprache mit Lambda Ausdrücken.

```
\ <Pattern> -> <Expr>     ==>    LET { newName <Pattern> = <Expr> } IN newName
```

Abbildung 3.8: Umwandlung eines Lambda-Ausdruckes in einen Let-In-Ausdruck.

Mit *explicitly typed expressions* kann man in der funktionalen Beispielsprache den Datentyp einer Anweisung ausdrücklich angeben. Entweder wird der Funktor „`stmttype()`“ in die generelle abstrakte Syntax aufgenommen oder man bildet die Anweisungen mit explizitem Typ ab, aber ohne Angabe des Kontexts, als anonyme Methode. Diese haben an Stelle des Rückgabetyps den ganzen Typausdruck für die Anweisung. Also müsste dies speziell behandelt und generelle Typausdrücke zugelassen werden.

```
'nonterm' Stmt(-> STATEMENT)              -- new rule and type for statement
   'rule' Stmt(-> declstmt(declare(P, nil,                    -- lambda expression
                  method(concrete, private, nil, Pats, nil, exprstmt(Exp))))):
          "\" @(-> P) Patlist(-> Pats) "->" Expr(-> Exp)
   'rule' Stmt(-> declstmt(declare(P, nil,                    -- local definition
                  method(concrete, private, nil, nil, Ds, stmtseq(S, nil))))):
          "LET" @(-> P) "{" Decllist(-> Ds) "}" "IN" Stmt(-> S)
   'rule' Stmt(-> stmttype(P, S, C, T)):                      -- typed statement a)
          "::" Stmt(-> S) "::" @(-> P) Context(-> C) Type(-> T)
   'rule' Stmt(-> declstmt(declare(P, nil,                    -- typed statement b)
                  method(concrete, public, T, nil, nil, stmtseq(S, nil)))):
          "::" Stmt(-> S) "::" @(-> P) Type(-> T)
   'rule' Stmt(-> exprstmt(X)):                               -- expression statement
          Expr(-> X)
```

Abbildung 3.9: Abbildung der funktionalen Ausdrücke auf „`Statement`“ der GZS.

Logische Anweisungen

Das Abbildung der Bodies in Abbildung 3.10 der logischen Beispielsprache basiert auf der if-Anweisung und den Funktoren für die Wahrheitswerte „`true/fail`“, die als Resultat zurückgegeben werden.

Die Benennung eines Body mit einem Modulnamen (3.10, 1.Regel) entspricht einer Anweisung, die importiert wird oder einem *native call* und kann als Methodendeklaration abgebildet werden.

Zwei disjunktiv verknüpfte Bodies bedeuten (3.10, 2.Regel), dass entweder der erste Body wahr ist oder er ist falsch und dann kann der zweite wahr sein oder falsch. Dieses Verhalten kann auf eine if-Anweisung übertragen werden, wobei die Bedingung, also der erste Body, eine Anweisung und nicht ein Ausdruck ist. Falls der erste Body wahr ist, resultiert wahr, sonst wird der zweite Body ausgewertet. Die Disjunktion von Bodies kann auch umschrieben oder ohne funktionelle Einschränkung der Grammatik weggelassen werden.

Ein Goal (3.10, 5.Regel) ist ein auszuwertender Term, der als Bedingung einer if-Anweisung betrachtet werden kann. Falls der Term erfüllt ist, wird „`true`“ weitergegeben, sonst im Elseteil „`fail`“. Ein *notprovable* Term (3.10, 6.Regel) ist falsch, falls der Term selbst eine Lösung hat, ansonsten ist er wahr. Dies ist also keine richtige Negation, im Sinne, dass der Term falsch ist, sondern bedeutet, dass der Term nicht beweisbar ist. Dies kann auf eine if-Anweisung mit dem Term als Bedingung und „`fail`“ im Thenteil und „`true`“ im Elseteil abgebildet werden.

Bildung eines allgemeinen Funktors für Schleifen

In der imperativen und der objektorientierten Beispielsprache finden sich drei Arten von Schleifen, die jeweils mit einem eigenen Funktor bezeichnet werden. Es wird mit Hilfe von Abbildung 3.11 die Bildung eines generellen Funktors beschrieben, auf den alle vorkommenden Schleifen abgebildet werden können:

Welche Funktionalität muss ein Funktor haben, der alle Eigenschaften von Schleifen umfasst? Eine generelle Schleife „`loop()`“ besteht aus einer Anweisung, die vor der Bedingung ausgeführt wird, der Bedingung selbst und einem Anweisungsblock, der je nach Erfüllung der Bedingung ausgeführt wird. Nun werden die einzelnen Schleifen auf die allgemeine Form übertragen.

```
'nonterm' Body(-> STATEMENT)
   'rule' Body(-> declstmt(declare(P, M, method(concrete, public, nil, nil, nil, Bs)))):
          Module(-> M) @(-> P) ":" Bodies(-> Bs)                          -- named body
   'rule' Body(-> ifthenelse(P, stmtexp(L), exprstmt(true), R)):          -- disjunction
          "(" @(-> P) Body(-> L) ";" Body(-> R) ")"
   'rule' Body(-> ifthenelse(P, G, B1, B2)):                              -- if then else
          @(-> P) Term(-> G) "-> " Body(-> B1) ";" Body(-> B2)
   'rule' Body(-> ifthenelse(P, G, B, exprstmt(false))):                  -- if then, without else
          @(-> P) Term(-> G) "-> " Body(-> B)
   'rule' Body(-> ifthenelse(P, T, exprstmt(true), exprstmt(false))):     -- goal
          @(-> P) Term(-> T)
   'rule' Body(-> ifthenelse(P, B, exprstmt(false), exprstmt(true))):     -- not provable goal
          "\+" @(-> P) Term(-> B)
```

Abbildung 3.10: Abbildung der logischen Bodies auf generelle Anweisungen.

Die Whileschleife beginnt in der allgemeinen Form mit einer leeren Anweisung, ihre Bedingung wird übernommen, danach wird der Anweisungsblock ausgeführt.

Für die Repeatschleife wird zuerst der Anweisungsblock einmal ausgeführt. Nun muss die Bedingung erfüllt sein, damit die allgemeine Schleife weiterhin durchgeführt wird. Deshalb wird getestet, ob die ursprüngliche Bedingung falsch ist, danach folgt wiederum der Anweisungsblock.

Zur Übertragung der For-Anweisung in die generelle Schleife wird im ersten Anweisungsblock die Laufvariable mit dem Startwert initiiert. Danach wird als Bedingung geprüft, ob diese Laufvariable kleiner als der Endwert ist. Falls dies erfüllt ist, so werden im zweiten Anweisungsblock die Anweisungen der For-Schleife ausgeführt und die Laufvariable um die angegebene Schrittweite erhöht.

```
-- imperative grammar loops as statement-rules
-- mapped to general abstract functor 'loop(Pos, Stmt, Expr, Stmtlist)'
-- no code-positions (Pos) for visual reasons

'nonterm' Statement(-> STATEMENT)
 -- while do
        'rule' Statement(-> loop(nil, Cond, stmt(S))):
                "WHILE" Expression(-> Cond) "DO" Statement(-> S)
 -- repeat until
        'rule' Statement(-> loop(S, relative(Cond, desig(op(eq)), false), stmt(S))):
                "REPEAT" Statement(-> S) "UNTIL" Expression(-> Cond)
 -- for-loop
        'rule' Statement(-> loop(
                        assignment(termexp(V), Start),
                        relative(termexp(V), desig(op(lt)), End),
                        stmtseq(S, stmt(assignment(termexp(V), plus(termexp(V), Step))))
                        )):
                "FOR" Variable(-> V) ":=" Expr3(-> Start) "TO" Expr3(-> End)
                "STEP" Expr3(-> Step) "DO" Statement(-> S)
```

Abbildung 3.11: Abbildung von (imperativen) Schleifen auf den generellen Funktor „`loop()`“.

Dank dieser Generalisierung erübrigen sich die spezialisierten Funktoren in der GZS.

3.3.6 Abbildung der Ausdrücke

Die Ausdrücke einer Programmiersprache werden aus terminalen Bezeichnern, wie Variablen, Konstanten oder Funktionen, und Operatoren rekursiv aufgebaut: Jeder terminale Bezeichner ist ein Ausdruck. Die Verknüpfung zweier Ausdrücke durch einen Infixoperator ergibt wiederum einen Ausdruck. Präfixoperatoren stehen vor einem einzelnen Ausdruck. Ausdrücke können geklammert sein.

Der generelle abstrakte Typ für Ausdrücke ist „`Expression`“, derjenige für Reihen von Ausdrücken „`Expressionlist`“, siehe Abbildung 3.12. Da eine Ausdrucksliste endlich oder sogar leer ist, kann ein Wert vom Typ „`Expressionlist`“ auch gleich „`nil`“ sein. Eine Liste von Ausdrücken wird ohne Angabe der Position angegeben, da jeder Ausdruck selbst schon eine Positionsangabe besitzt.

```
'type' EXPRESSIONLIST = exprlist( EXPRESSION, EXPRESSIONLIST ), nil.

'type' EXPRESSION = operation( POS, EXPRESSION, TERMINAL, EXPRESSION ),
                    relation( POS, EXPRESSION, TERMINAL, EXPRESSION ),
                    sectionleft( POS, EXPRESSION, TERMINAL ),
                    sectionright( POS, TERMINAL, EXPRESSION ),

                    and( POS, EXPRESSION, EXPRESSION ),
                    or( POS, EXPRESSION, EXPRESSION ),
                    not( POS, EXPRESSION ),
                    true, false,
                    plus( POS, EXPRESSION, EXPRESSION ),
                    minus( POS, EXPRESSION, EXPRESSION ),
                    neg( EXPRESSION ),
                    mult( POS, EXPRESSION, EXPRESSION ),
                    div( POS, EXPRESSION, EXPRESSION ),
                    intdiv( POS, EXPRESSION, EXPRESSION ),
                    mod( POS, EXPRESSION, EXPRESSION ),

                    termexp( TERMINAL ),
                    parent( EXPRESSIONLIST ),
                    stmtexp( STATEMENTLIST ),
                    list( EXPRESSIONLIST ),

                    funexp( POS, EXPRESSION, TERMINAL, EXPRESSIONLIST ),
                    arithseq( POS, EXPRESSION, EXPRESSION, EXPRESSION ),
                    nil.                    -- empty expression parameter
```

Abbildung 3.12: Ausdrücke in der GZS.

Die imperative, die objektorientierte und die logische Beispielsprache verwenden Ausdrücke, die diesem Modell genau entsprechen. In der letzteren werden sie mit „`Term`“ bezeichnet. Die funktionale Beispielsprache dagegen hat ausdrucksähnliche Konstrukte unter verschiedenen abstrakten Typen. So können „`Pattern`“ und „`Function`“ auf „`Expression`“ abgebildet werden, „`Expr`“ aber muss man zuerst in Ausdrücke und Anweisungen unterteilen. Im folgenden werden einige Abbildungen beschrieben, die nicht trivial sind oder Probleme ergeben haben.

Die Beispielsprachen verwenden zwei verschiedene Ansätze, um Operationen auszudrücken:

- Die imperative und die objektorientierte Grammatik haben für jede Operation eine Regel und einen Funktor, die Anzahl der Operationen ist fix.

- Die logische und die funktionale Grammatik dagegen lassen dem Anwender freie Hand bei der Definition neuer Operatoren. Alle Operationen werden mit einer einzigen Regel geparst und einem allgemeinen Funktor „`operation()`“ zugewiesen.

Restriktion: Operationen werden generell als Infixoperationen von gleicher Priorität und Linksassoziativität behandelt. Wegen der zwei Ansätze kommen in der generellen Zwischensprache Funktoren vor, deren Bedeutung sich überschneidet. Um die Konstruktionsmöglichkeiten der Grammatiken nicht einzuschränken, werden diese Funktoren nicht zusammengelegt.

Die im folgenden aufgeführten Funktoren entstammen alle der Definition in Abbildung 3.12. „`true`“ und „`false`“ sind die logischen Wahrheitswerte, die bei Relationen resultieren. In JAVA sind es Literale, keine Schlüsselwörter, in GOFER werden sie in der Prelude definiert, in PROLOG sind es Kontrollausdrücke, also Schlüsselwörter und Resultat von Abfragen. Der Ausdruck „`false`“ steht in der generellen Zwischensprache auch für das logische „`fail`“, obwohl etwas nicht notwendigerweise falsch ist, wenn der Beweis davon nicht zustande kommt. „`termexp()`“ ersetzt alle Funktoren, die terminale Elemente bezeichnen. „`parent()`“ steht für eine geklammerte Ausdrucksliste, die in der funktionalen Sprache mit „`tuple()`“ bezeichnet wird. „`stmtexp()`“ für eine (geklammerte) Anweisungsliste als Ausdruck und ist somit genereller als „`body()`“ der logischen Sprache, das nur eine einzige Anweisung enthält. „`list()`“ bezeichnet eine Liste von Ausdrücken und wird in der funktionalen und der logischen Sprache als Liste verwendet. Die leere Liste „`[]`“ wird auf auf „`list(nil)`“ und Unit „`()`“ wird auf „`parent(nil)`“ abgebildet. Der Methodenaufruf „`funexp()`“ besteht in der allgemeinsten Form aus einem (Klassen)-Objekt „`Expression`“, gefolgt vom eigentlichen Namen der Methode „`Terminal`“ und einer Liste von zu übergebenden Parametern „`Expressionlist`“. Für einen Ausdruck, der einen Wertebereich angibt, wird „`arithseq()`“ verwendet, da er die Start- und Endausdrücke und zusätzlich die Schrittweite der Folge festhält. Das objektorientierte „`range()`“ wird zu diesem Funktor generalisiert.

Der allgemeine Type „`Expression`“ kommt bereits in allen Beispielsprachen vor, somit beschränkt sich die Abbildung auf ein Zusammenlegen der Funktoren.

3.3.7 Abbildung der terminalen Elemente

Am unteren Ende jeder Syntaxdefinition der hier untersuchten Programmiersprachen befinden sich Elemente, deren weitere Aufteilung nicht sinnvoll ist, nämlich Zahlen, Bezeichner oder Operatoren. Diese terminalen Elemente oder Token fallen unter den generellen abstrakten Typ „`Terminal`“ (Abbildung 3.13), der in allen Beispielsprachen neu eingeführt wird. Es gibt nun syntaktische Elemente, die einerseits in der Syntaxanalyse als „Terminale“ auftreten, andererseits aber aus terminalen Elementen zusammengesetzt sind. So zum Beispiel Arrayvariablen, die aus ihrem Namen und dem Index bestehen, und Listen von Variablen oder Operatoren, die mit den gleichen Regeln wie einzelne Variablen oder Operatoren geparst werden. Mit den generellen Typen „`Desig`“ (*designator*) und „`Compterm`“ (*composite terminal*) werden einfache von zusammengesetzten terminalen Elementen unterschieden. „`nil`“ wird verwendet, falls kein Terminal vorkommt.

```
'type' TERMINAL = desig( POS, DESIG ),
                  compterm( POS, COMPTERM ), nil.
```

Abbildung 3.13: Einfache und zusammengesetzte Terminale.

Abbildung 3.14 listet die einfachen terminalen Elemente und ihre Funktoren auf. Als Parametertypen werden die Tokennamen verwendet, die abhängig von der jeweils verwendeten Sprache sind und so in einen allgemeinen Zusammenhang gestellt werden. Die abstrakten Typen dieses Abschnitts ermöglichen es, die lexikalischen Elemente der Paradigmasprachen generell zu handhaben und unabhängig von ihrer konkreten Form darzustellen.

Der Funktor „`varid()`“ steht für Variablen und Bezeichner. Jede Paradigmasprache hat eine lexikalisch andere Form von Variablen, semantisch unterscheiden sie sich jedoch nicht.[2] Da zum Beispiel die logische Beispielsprache konstante Namen lexikalisch von Variablen unterscheidet, werden Konstanten auf „`constid()`“ abgebildet.

„`symbol()`“ steht für spezielle Symbole, deren Bedeutung sprachabhängig ist. „`cut`“ ist ein spezielles Symbol der logischen Beispielgrammatik.

```
'type' DESIG = varid( VAR ),
               constid( WORD ),
               number( INT ),                -- ganze Zahlen, Integer
               real( DOUBLE ),               -- reelle Zahlen, Real, Double, Float
               char( CHAR ),                 -- ein Buchstabe in Anfuehrungszeichen
               string( TSTRING ),            -- Strings in Anfuehrungszeichen
               relop( RELOP ),               -- relative Operatoren
               operator( OPERATOR ),         -- sonstige Operatoren
               symbol( SYMBOL ),             -- spezielle Symbole
               conid( CONID ),               -- speziell fuer funktionale BspPL
               conop( CONOP ),               -- speziell fuer funktionale BspPL
               cut.
```

Abbildung 3.14: Einfache Bezeichner.

In der imperativen und der objektorientierten Beispielsprache werden Arrays verwendet, die aus einem terminalen Namen und einem Indexausdruck bestehen (Abbildung 3.15):

Der Funktor „`list()`“ wird für eine Reihe von terminalen Elementen verwendet. So in der funktionalen Beispielsprache für Listen von Variablen oder Operatoren. Der erste Parameter ist ein beliebiger einfacher Terminal (Varid, Varop, Conop), der zweite sinnvollerweise der rekursive Aufruf dieser Liste.

Weitere zusammengesetzte Terminals sind möglich. Sie sind in den Beispielgrammatiken nicht implementiert sind, da es sich ja um minimale Teilsprachen handelt: Records, Enumerated Types, Union, Procedure, Method, Classtype.

```
'type' COMPTERM = array( TERMINAL, EXPRESSION ),
                  list( DESIG, TERMINAL ).
```

Abbildung 3.15: Zusammengesetzte Bezeichner.

Durch die generellen abstrakten „`Terminal`“, „`Desig`“ und „`Compterm`“ werden in der imperativen Beispielsprache die abstrakten Typen „`Variable`“ und „`Relop`“ eliminiert, in der objektorientierten zusätzlich „`Logop`“, in der funktionalen „`Operator`“, „`Identifier`“, „`Literal`“,

[2] Logische Variablen werden einmal instantiiert und danach (ausser beim Backtrackingvorgang) nicht mehr verändert. Sie können ohne Deklaration einen beliebigen Datentyp annehmen, sind also untyped.

„Oplist“ und „Varlist“, und in der logischen „Operator“, „Relop“, „List“, „Constant“, „Atom“ und „Module“.

3.4 Diskussion ausgewählter Probleme

Es folgen einige Anmerkungen und Beschreibungen von Problemfällen, die nicht eindeutig einem der vorangehenden Absätze zugeordnet werden konnten oder die von untergeordneter Bedeutung sind und nicht zum Verständnis der abstrakten Typen beitragen. Weiter einige Konstrukte, die sich nicht auf die generelle Zwischensprache abbilden lassen.

List comprehension

List Comprehension, also die Konstruktion einer Liste, kontrolliert durch verschiedene Qualifiers, kann nach „Introduction to Gofer“ [Jon91] mit bereits vorhandenen Sprachkonstrukten gebildet werden. Dies funktioniert bei einfachen Qualifiers wie folgt: Ein Generatorqualifier wird als Funktion geschrieben, die für jedes Element des Ausdrucks auf der rechten Seite des Pfeils testet, ob es auf das Pattern der linken Seite passt. Falls diese Bedingung erfüllt ist, wird das Element an die Ausgabeliste angehängt, sonst nicht. Ein Filterqualifier testet, ob die boolesche Bedingung erfüllt ist, funktioniert also wie eine If-Anweisung. Ein Qualifier mit lokaler Definition kann als Let-In-Anweisung betrachtet werden. Soweit besteht kein Problem, die Qualifier der List Comprehension auf Konstrukte der generellen Zwischensprache abzubilden. Aber die funktionale Beispielsprache erlaubt auch die Angabe von mehreren Qualifiern, die in einer erweiterten List Comprehension zusammengefasst werden. Dieser generelle Fall bietet einige Probleme bei der Abbildung, da die Qualifierarten miteinander kombiniert werden dürfen. Somit müsste im Parser für jede Kombination von Qualifiern eine Regel stehen, was den Rahmen des Parsers und der Arbeit sprengen würde.

```
list comprehension: [ e | qs ]
transformation of single qualifiers:
 - generator        : qs := pat <- exp    ==>     loop exp
                                                  WHERE loop []       = []
                                                        loop (pat:xs) = e : loop xs
                                                        loop ( _ :xs) = loop xs
 - filter           : qs := condition     ==>     IF condition THEN [e] ELSE []
 - local definition : qs := pat = exp     ==>     LET { pat = exp } IN  e

several qualifiers  : [ e | qs1, qs2 ]    ==>     concat [[ e | qs2 ] | qs1 ]
```

Wie soll man nun bei der Abbildung der List Comprehension auf die generelle Zwischensprache vorgehen?

- Man behält die drei ursprünglichen Funktoren bei und verbindet mehrere Qualifier erst bei der Codeerzeugung. Dieser Ansatz wird nicht berücksichtigt, weil die Eigenschaft der Sprachunabhängigkeit verloren geht.
- Der Parser löst die Qualifier auf und übergibt sie als allgemeine Konstrukte. Dies ist, wie gesagt, ziemlich aufwendig.
- Die List Comprehension kann als Methode betrachtet werden, die einen Ausdruck als Parameter und die Qualifier als Body zur Erzeugung der Liste hat. Damit wird aber das Problem mehrerer Qualifier nicht gelöst.
- Die List Comprehension wird in dieser Arbeit auf einfache Qualifier eingeschränkt, oder der Anwender einer funktionalen Sprache muss die beschriebenen Transformationen in seinem Programm vornehmen.

Funktionale Pattern

Für die funktionalen Pattern, die nicht gleich einem Ausdruck sind, ist ein Abbildung auf die generelle Zwischensprache festzulegen: Eine *wildcard* wird in den Zeichensatz der Variablen aufgenommen oder als spezielles Symbol mit „`symbol()`“ abgebildet. Die Rolle von *wildcard* beim „Pattern Matching“ ist zu beachten. *Irrefutable* Pattern kommen kaum vor, können mit einer Regel umschrieben werden und brauchen daher keinen eigenen Funktor.

3.5 Diskussion

Mit der vorgestellten Zwischensprache können die meisten paradigmaspezifischen Konstrukte abgebildet werden. Wo dies nicht funktioniert, werden Problemlösungsstrategien angegeben. Für Speziallfälle braucht es ein „User-defined Concept“, das nicht-abbildbare Konstrukte wie *native* Code einbindet.

Die Abbildung der abstrakten Syntax auf die GZS ändert nichts an der Nonterminal- und Regelstruktur der jeweiligen Grammatiken. Der Parser liefert mit der abstrakten Syntax GZS eine Interpretation der konkreten Syntax. Diese Interpretation hat dank der vorliegenden Abbildung für jede Beispielsprache die gleiche Form. Die Typen und Funktoren der GZS werden bei der Codeerzeugung (Kapitel 7) nicht nach den ursprünglichen Paradigmasprachen unterschieden, nur ihre Bedeutung ist wichtig.

Im Gegensatz zu der abstrakten Syntax von CooL[3], für welche ein lauffähiger Compiler mit verständlichem Code gebaut wurde, gewichtet diese Arbeit die allgemeine Verwendbarkeit der Zwischensprache stärker als die Lesbarkeit. Es werden deshalb nicht viele spezielle, aussagekräftige Konstrukte verwendet, sondern die abstrakte Syntax ist aus ein paar wenigen generellen Typen und Funktoren zusammengesetzt.

Der Ansatz einer generellen Zwischensprache sollte nicht aus dem Lehrbuch oder einer grossen Grammatik konstruiert werden, sondern der Anwender einer Programmiersprache oder eines Paradigmas soll eine Grammatik aufstellen: dieser relevante Kern kann iterativ aufgebaut werden. Auch die Anforderungen, die das Backend an die Zwischensprach stellt, sind bei der Konstruktion der GZS nicht bekannt gewesen. Hier muss ein Zusammenspiel zwischen diesen beiden Komponenten eine Validierung und Verbesserung der Zwischensprache hervorbringen.

Erkenntnisse zum Erstellen einer Zwischensprache

Funktoren und Typen sind eindeutig zu benennen, damit ihr Sinn nicht nur im Zusammenhang mit anderen Konstrukten, sondern auch losgelöst, klar aus ihrem Namen zu erkennen ist. Funktoren, die denselben Typ weiterleiten, zu dem sie gehören, können aus Sicht der GZS weggelassen werden, da ihr Zweck nur dem Verständnis der Grammatik und der Beschreibung der Syntaxumgebung dient. Diese Funktoren kann man als Knoten im abstrakten Syntaxbaum betrachten, die zusammengefasst werden können. Generell ist eine minimale Anzahl Funktoren mit maximaler Bedeutung anzustreben, damit einerseits die Zwischensprache übersichtlich und verständlich bleibt, und andererseits jeder Funktor in Zielcode übersetzt werden soll.

Eine Vereinfachung, die in der vorliegenden GZS wegen der schlechteren Verständlichkeit nicht vorgenommen wurde, ist die Entfernung aller Typen, die Listen bezeichnen. Somit müsste der dazugehörige Typ den Funktor für eine Reihe von Elementen übernehmen und es wären keine Übergänge zwischen diesen Typen mehr nötig.

[3] Abstrakte Syntax von CooL in „`ast.g`“ auf ftp://ftp.fu-berlin.de/pub/unix/languages/cool/CooL.tar.gz

Kapitel 4

Virtuelle Maschine von Java

> In zunehmendem Masse werden Maschinen durch andere Maschinen gesteuert und ersetzen damit die einfacheren Formen menschlicher Intelligenz.
>
> *John Kenneth Galbraith, Die moderne Industriegesellschaft*

Dieses Kapitel bietet einen kurzen Einblick in das Konzept und den Aufbau der Virtuellen Maschine von Java, im folgenden kurz „JVM“ genannt. Besonderes Gewicht wird auf das `class` File Format gelegt, das die Form der Programme beschreibt, die eine JVM ausführt. In Kapitel 5 wird auf der hier vorgestellten Grundlage ein Programm zur Analyse und Manipulation von `class` Files realisiert. Die hier beschriebene Version 1.0.2 der JVM wurde 1996 von Tim Lindholm und Frank Yellin in „The Java Virtual Machine Specification“ eingeführt [Lin97].

4.1 Konzept

Die Idee einer allgemein verwendbaren Programmiersprache, die einfach und sicher, deren ausführbare Form klein und robust ist, ist sowohl für Software im allgemeinen als auch für deren Verbreitung über das Internet geeignet. Diese Vision von Allgegenwärtigkeit und freier Portabilität verlangt nach einer präzisen Definition der Sprache und ihrer Umgebung. Wichtig ist ihre Unabhängigkeit von Betriebssystemen und Plattformen. Java löst diese Forderungen durch die Spezifikation eines abstrakten Rechners, auf dem die Sprache läuft: die virtuelle Maschine.

Diese Spezifikation ist verbindlich, sowohl für Compiler, die die JVM als Ziel haben, als auch für Implementationen kompatibler virtueller Maschinen. Es wird aber nur ein Interface und die Verhaltensweise vorgeschrieben, die für die Plattformunabhängigkeit der JVM notwendig sind. Die Implementation einer JVM kann entweder die Designvorschläge der Autoren befolgen oder aber Modifikationen und Optimierungen vornehmen, solange die Spezifikatio erfüllt ist. So kann eine JVM-Implementierung den Bytecode in die Instruktionsmenge einer anderen virtuellen Maschine oder einer konkreten CPU übersetzen oder *just-in-time* compilieren[1].

Die JVM Spezifikation beschreibt neben dem abstrakten Design der JVM die Files, die von der JVM akzeptiert und ausgeführt werden. Es handelt sich dabei um strenge Bedingungen an das Format und die Struktur der sogenannten „`class` Files“. Jedes Programm, das gemäss diesen Regeln als gültiges `class` File formuliert wird, kann von jeder JVM ausgeführt werden. Zitat von Lindholm und Yellin [Lin97] zur Beziehung zwischen JVM und `class` Files:

[1] Just-in-time Compiler von JDK.

> Eine JVM ist eine virtuelle Maschine, die dem abstrakten Design der Spezifikation entspricht, die das `class` File Format einlesen und alle darin enthaltenen Aktionen ausführen kann.

Die Verwendung einer virtuellen Maschine und deren offenes Design sind im wesentlichen verantwortlich für die Plattformunabhängigkeit der Sprache Java. Da die JVM hauptsächlich zur Ausführung von Java Programmen entworfen wurde, sind ihre Konzepte und das Vokabular verwandt mit dem von Java in „The Java Language Specification“ [Gos96]. Die JVM hat aber keine Kenntnis von Java, sondern akzeptiert alle Eingaben im `class` File Format, das von einem beliebigen Compiler aus einer beliebigen Programmiersprache erzeugt werden kann.

Dieser Vorteil wurde rasch von Sprachkonstrukteuren erkannt, die die JVM als Instrument zur Verbreitung ihrer Nicht-Java-Sprachen benutzen[2]

- Eine Zusammenstellung von etwa 60 Programmiersprachen für die JVM findet man auf: `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`
- „Programming the Internet in Ada95“ von Tucker Taft: `http://www.inmet.com/~stt/adajava_paper/`.
- Pizza, eine Erweiterung von Java um funktionale Eigenschaften: `http://wwwipd.ira.uka.de/~pizza/`
- Kawa, ein Compiler, der Scheme nach Bytecode übersetzt: `http://www.cygnus.com/~bothner/kawa.html`

Auch die vorliegende Arbeit benutzt die JVM als robuste und plattformunabhänigige Maschine für die generelle Zwischensprache, indem das Compilerbackend mit dem `class` File Format arbeitet.

4.2 Struktur

Eine JVM besteht aus einem Runtime Datenbereich, einer Menge von Instruktionen und strukturierten Werten.

Der Runtime Datenbereich einer JVM, der *Heap*, kann dynamisch vergrössert und verkleinert werden oder besitzt eine fixe, vom Benutzer angegebene Grösse. Ein Teil des Heaps, die *Method Area*, hält während der ganzen Ausführung Klassenstrukturen fest. Dazu gehört eine Runtime Darstellung des `ConstantPool` (siehe Abschnitt 4.5), Daten und Code für Fields, Methoden und die speziellen Klassen und Instanzkonstruktoren (Abschnitt 4.8). Weiter existiert auf dem Heap ein *Frame Stack* und Speicherplatz für alle Instanzen und Arrays.

Die JVM ist Stack-basiert, benutzt keine Register, um Argumente zu übergeben oder zu erhalten. Einzig zur Angabe der Adresse der momentan ausgeführten Instruktion der aktuellen Methode wird ein Register verwendet, der *Program Counter*. Die Bytecodes operieren mit Werten auf dem Operandenstack und legen lokale Variablen in einem Array ab.

Jede Methode hat einen eigenen *Frame*, welcher ihre Ausführungsumgebung, die Daten und Resultate speichert. Die lokalen Variablen der Methode werden in einem Array, die Operanden für die Instruktionen als Stack gespeichert.

Der Frame wird auch zum dynamischen Linken der Methode und zur Behandlung von Exceptions verwendet. Es ist jeweils ein Frame aktiv, falls die laufende Methode eine andere Methode aufruft, wird der laufende Frame auf den Stack gelegt und ein neuer Frame aktiviert.

[2] Die folgenden Adressen waren zum Zeitpunkt der Drucklegung dieser Arbeit im Mai 1999 gültig.

Beim Beenden einer Methode übergibt der aktive Frame dem vorhergehenden den Rückgabewert (falls vorhanden) und wird aufgegeben. Nun stellt die JVM den Frame der aufrufenden Methode wieder her. Dazu setzt sie den Program Counter auf die Instruktion nach dem Methodenaufruf und schiebt den eventuellen Rückgabewert auf den Operanden Stack, worauf sie mit der Ausführung dieser Methode fortfährt.

4.3 Datentypen

Die JVM verwendet die folgenden Datentypen, wobei angenommen wird, dass der Vorgang des Typecheckings vom Compiler vorgenommen wurde. Die numerischen Typen werden *primitive* Datentypen genannt: `byte, short, int, float, long, double, char`. Dazu gehört auch der Datentyp, der für Returnadressen im Opcode verwendet wird. Es gibt keinen boolschen Typ, `true` wird als 1, `false` als 0, und die entsprechenden Operationen auf ganzen Zahlen vom Typ `int` durchgeführt. Der *reference* Datentyp `reference` enthält Zeiger auf dynamisch erzeugte Klasseninstanzen, Arrays oder Klassen, die ein Interface implementieren.

Passend zu den Datentypen werden die Wertetypen *primitive* und *reference* von der JVM zum Speichern in Variablen, zur Übergabe von Argumenten, zur Rückgabe bei Methoden und zum Ausführen von Operationen verwendet. Jede Instruktion operiert nur mit ihrem spezifischen Typ, so gibt es zum Beispiel vier Operationen zum Addieren zweier Werte: `iadd, ladd, fadd, dadd`. Deren Operanden müssen beide vom Typ `int`, respektive `long`, `float` oder `double` sein. Es existieren keine speziellen Instruktionen für `byte`, `short` und `char`, ausser bei der Arraybehandlung, da die JVM diese Typen intern als `int` behandelt, deren Wertebereiche eingeschränkt sind. So wird die Gruppe der `int`-Instruktionen darauf angewendet.

Zwei-Wort-Datentypen

Die Datentypen `long` und `double` werden als Zwei-Wort-Datentypen bezeichnet und unterscheiden sich von den übrigen Datentypen.

Ein Zwei-Wort-Datentyp belegt zwei Einträge im `ConstantPool` und im Array der lokalen Variablen. Der Zugriff muss über den kleineren Index erfolgen, der zweite Index darf nicht direkt manipuliert werden. Zwei-Wort-Datentypen füllen auch zwei Einträge des Operanden Stacks, welche nicht voneinander getrennt durch Instruktionen manipuliert werden dürfen.

4.4 Instruktionen

Eine JVM Instruktion besteht aus einem Opcode von 1 Byte Länge, der die Operation identifiziert, gefolgt von keinem bis mehreren Operanden, die Argumente oder Daten für die Operation liefern. Jeder Operand ist mindestens ein Byte lang und *big-endian*. Falls ein Operand länger ist, wird das höchste Byte zuerst gespeichert. Die Instruktionen müssen gewissen statischen und strukturellen Bedingungen (4.7) genügen, da sonst der Zustand der JVM undefiniert ist.

Die Instruktionen lassen sich in drei Gruppen teilen: Normale Instruktionen sind von 0 bis 201 nummeriert, es folgen von 203 bis 228 die `_quick`-Pseudoinstruktionen. Um ein Optimierung der Perfomance zu erreichen, werden die normalen Instruktionen dynamisch durch die Pseudoinstruktionen ersetzt. Sie sind nicht Teil der JVM Spezifikation sondern von Suns JVM. Sie erscheinen nicht in `class` Files, ebensowenig wie die reservierten Instruktionen zum internen Gebrauch der JVM (Tests, Debugging) mit Nummern 202, 254 und 255. Die restlichen Opcodes haben in der aktuellen Version der JVM keine Bedeutung.

Die Aufzählung und Beschreibung der Instruktionen würde den Rahmen dieser Arbeit sprengen. Die bei der Codegenerierung verwendeten Instruktionen werden falls nötig im konkreten Teil erklärt. An dieser Stelle eine Anmerkung zu Instruktionen mit erweiterten Indices.

Wide Bytecodes

Instruktionen mit erweiterten Indices werden verwendet, falls ein Operand von einem Byte Länge nicht ausreicht, um eine lokale Variable, eine Opcode Adresse oder einen `ConstantPool` Eintrag zu adressieren. Die JVM löst dieses Problem auf zwei Arten:

Durch Voranstellen des Opcodes „*wide*“ operieren die Instruktionen *iinc, ret, <type>load, <type>store* mit einem 16bit Index, der aus den zwei nach der Instruktion folgenden Bytes zusammengesetzt wird. Ausser dem erweiterten Index verhalten sich die Instruktionen gleich wie ihre nicht modifizierte Form, obwohl die *wide* Instruktion die modifizierte Instruktion als einen ihrer Operanden behandelt. Diese *wide*-Instruktionen werden bei mehr als 256 lokalen Variablen oder einer Adresse grösser als 256 eingesetzt.

Die andere Art von Instruktionen mit erweiterten Indices verwendet nicht die gleichen Opcodes und modifiziert sie, sondern es wird eine andere Instruktion eingesetzt, die die gleiche Operation mit erweitertem Index ausführt. Die Instruktionen *goto_w, jsr_w* werden für Sprünge an Opcode Adressen verwendet, für die ein zwei Byte Offset nicht ausreicht. Der so erreichte 4Byte Offset kann in der aktuellen Version der JVM aber nicht ausgenützt werden, da der Code einer Methode auf 65535 Bytes eingeschränkt ist, weil die Indices der beteiligten Attribute fix zwei Bytes lang sind. *ldc2_w* lädt einen Wert vom Typ `long` oder `double` aus dem `ConstantPool` auf den Operanden Stack. Diese Instruktion hat bewusst keine Form mit normalem Index. Alle anderen Datentypen werden mit *lcd_w* aus dem `ConstantPool` auf den Stack geladen. Diese Erweiterungen sind durchaus sinnvoll, da der `ConstantPool`, begrenzt durch seinen 2Byte Zähler auf 65535 Einträge, mit dem normalen 1Byte Index der *ldc* Instruktion nicht vollständig adressiert werden kann.

Nach der Beschreibung der Speicherbereiche, der Behandlung der Methoden, der Datentypen und der Instruktionsmenge einer JVM wird nun das Format der Eingaben für JVM betrachtet: das `class` File Format.

4.5 class File Format

Compilierter Code, der von der JVM akzeptiert und ausgeführt werden soll, wird als binäres File in einem plattformunabhängigen Format abgelegt, dem `class` File Format. Ein `class` File beschreibt genau einen Objekttyp, üblicherweise eine Java Klasse oder ein Interface. Es kann aber auch eine Klasse oder ein Programm einer anderen Quellsprache sein.

Das 4. Kapitel der „Java Virtual Machine Specification“ [Lin97] bietet eine präzise Definition für den Inhalt solcher Files. Details und konkrete Erfahrungen im Umgang mit den hier beschriebenen Strukturen finden sich im Kapitel 8 mit dem Beispiel.

Ein `class` File besteht aus einer Reihe von 8bit Bytes, die sequentiell als Strukturen gemäss der Spezifikation interpretiert werden. Diese Strukturen werden weder ausgerichtet noch auf eine bestimmte Länge ergänzt, sondern sind alle von variabler Länge. Einheiten von 16bit, 32bit oder 64bit werden durch Aneinanderreihen von 2, 4 oder 8 Bytes konstruiert. Im folgenden werden Daten von 8bit mit `u1` bezeichnet. 16bit, also 2 Bytes, werden als `u2` angegeben; 32bit, also 4 Bytes als `u4`. 64bit werden als zwei nacheinander folgende `u4` betrachtet, deren spezielle Behandlung in Abschnitt 4.3 erläutert wird.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;

    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];

    u2 access_flags;
    u2 this_class;
    u2 super_class;

    u2 interfaces_count;
    u2 interfaces[interfaces_count];

    u2 fields_count;
    field_info fields[fields_count];

    u2 methods_count;
    method_info methods[methods_count];

    u2 attributes_count;
    attribute_info attrributes[attributes_count];
}
```

Abbildung 4.1: Struktur eines `class` Files

Nach Spezifikation muss ein `class` File genau aus einer `ClassFile` Struktur bestehen, wie sie in Abbildung 4.1 dargestellt wird.

Die ersten drei Elemente der `ClassFile` Struktur dienen der Identifikation des `class` File Formats und haben in jedem `ClassFile` die gleichen Werte, nämlich `magic = 0xCAFEBABE`, `minor_version = 3` und `major_version = 45`.

Der `ConstantPool` ist eine Tabelle von unterschiedlich langen Teilstrukturen, die Konstanten und Referenzen zum Aufbau der `ClassFile` Struktur darstellen. Der erste Eintrag im `ConstantPool` ist für den internen Gebrauch der JVM reserviert und im `ClassFile` nicht vorhanden. Die Anzahl Einträge im `ConstantPool` ist daher `constant_pool_count - 1`.

Die folgenden fünf Elemente enthalten Informationen des Deklarationsheaders der Klasse: Das `access_flags` Element gibt die Zugriffsmöglichkeiten und die Art der Klasse an (`ACC_PUBLIC, ACC_FINAL, ACC_SUPER, ACC_INTERFACE, ACC_ABSTRACT`). Der Name der Klasse wird von `this_class` mittels einer Klassenreferenz festgehalten, `super_class` gibt, ebenfalls anhand einer Klassenreferenz, den Namen des direkten Vorgängers in der Klassenhierarchie an. `interfaces_count` ist die Anzahl der Elemente im `interfaces` Array, das Klassenreferenzen auf die von der Klasse implementierten Interfaces enthält.

Nun folgen die Elemente, die den Body der Klasse im `class` File definieren: In `fields_count` wird die Anzahl der Elemente in der `fields` Tabelle festgehalten. Diese enthält für jede Klassen- und Instanzvariable eine `field_info` Struktur. `methods_count` gibt die Anzahl der Methodenstrukturen in der `methods` Tabelle an. Jede in dieser Klasse explizit deklarierte Methode muss durch eine Methodenstruktur beschrieben werden. Diese Beschreibung beinhaltet den JVM Code für die Methode.

Ein `ClassFile` schliesst mit einer Tabelle `attributes` für allgemeine Attribute der Klasse, die `attributes_count` Elemente hat. Üblicherweise handelt es sich dabei nur um ein Attribut, das `SourceFile_attribute`, welches den Namen des Sourcefiles angibt, aus dem das vorliegende `class` File compiliert wurde.

4.6 Ausführung

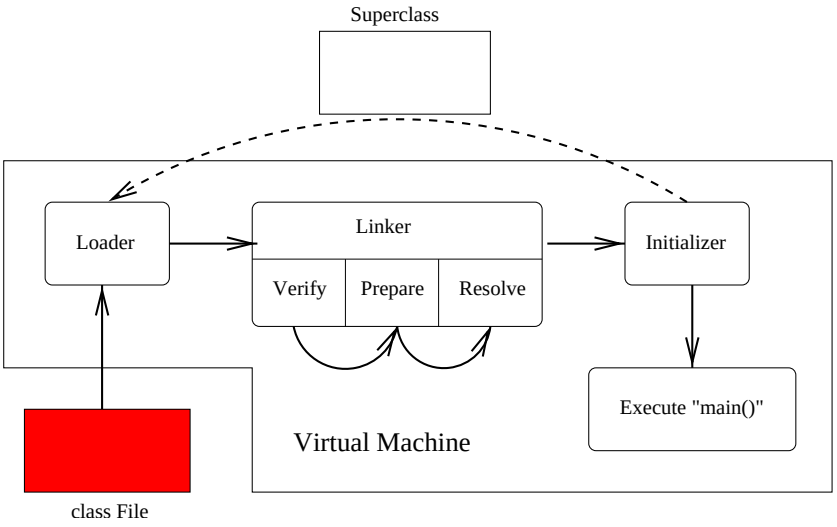

Abbildung 4.2: Ausführung eines `class` Files durch die JVM.

In diesem Abschnitt wird die Ausführung eines `class` Files anhand der Figur 4.2 betrachtet: Eine JVM beginnt die Ausführung eines angegebenen `class` Files, indem sie dessen Methode `main` aufruft, die `public`, `static` und `void` sein muss. Beim Aufruf wird als Argument ein String Array verwendet, der Eingaben des Anwenders übergibt. Um `main` ausführen zu können, muss die angegebene Klasse zuerst mit einem **ClassLoader** geladen werden. Dieser sucht und lädt die Klasse als binäres File, üblicherweise im `class` File Format.

Die nächsten Schritte der Verifikation, Präparation und Resolution sind gemeinhin als **Linking** bekannt. Es muss verifiziert werden, ob die binäre Form der Klasse den strukturellen und semantischen Bedingungen der JVM genügt (Abschnitt 4.7). Falls sie dies erfüllt, wird den statischen Konstrukten der nötige Speicherplatz zugewiesen und interne Datenstrukturen wie Methodentabellen vorbereitet (Präparation). Im folgenden Schritt, der Resolution, werden die symbolischen Referenzen aufgelöst, die im `ConstantPool` enthalten sind. Dazu werden die dort erwähnten Klassen zuerst geladen, danach auf Korrektheit der Referenzen kontrolliert und üblicherweise durch direkte Referenzen ersetzt, die einen effizienteren Zugriff erlauben.

Jetzt werden die Initialisierungsprozeduren der Klassenvariablen und `<init>` (4.8) ausgeführt. Damit dieser **Initialisierungs**schritt möglich ist, muss zuerst die direkte Superklasse dieser Klasse initialisiert werden, und so weiter bis zur Klasse `Object`. Dadurch wird eventuell eine Reihe von rekursiven Lade-, Link- und Initialisierungsphasen ausgelöst, bis endlich `main` ausgeführt werden kann (**Execution**). Während jedem der beschriebenen Schritte können Fehler auftreten, die den Ausführungsprozess unterbrechen.

Die JVM beendet ihre Aktivitäten, wenn entweder alle Threads sich im terminalen Zustand befinden oder wenn ein Thread die Methode `exit` aufruft.

Ausführung von Bytecode

Der Bytecode Interpreter, sozusagen die Software CPU der JVM, untersucht jeden Opcode der laufenden Methode. Falls dieser Operanden benötigt, werden die obersten Werte auf dem

Stack verwendet, damit wird die Aktion für diesen Opcode ausgeführt. Das heisst, der Opcode wird auf richtige Mikroprozessor Instruktionen abgebildet.

Die JVM unterscheidet Operanden nicht nach vom Compiler generierten und zur Runtime erzeugten Operanden, sondern führt ihre Instruktionen mit den Werten auf dem Operanden Stack unabhängig ihrer Herkunft aus. Diese implizite Verwendung der Operanden auf dem Stack führt im Gegensatz zur expliziten Codierung zu kompakterem Bytecode.

4.7 Bedingungen und Einschränkungen für JVM Code

Um die Sicherheit der (Java) Programme auf der JVM zu gewährleisten, werden strenge Forderungen an Format und Struktur des Bytecodes gestellt. Im folgenden werden diejenigen Bedingungen und Einschränkungen betrachtet, die beim Compilieren für die JVM zum Einsatz kamen. Die vollständige Aufzählung findet sich im Abschnitt 4.8 „Constraints on JVM Code“ [Lin97].

Static Constraints

Die statischen Einschränkungen für Bytecode dienen der Wohlgeformtheit der `ClassFile` Struktur. *Code Array* ist ein Byte Array im Codeattribut einer Methode.

- Der Opcode der ersten Instruktion im Code Array beginnt bei Index 0. Das letzte Byte der letzten Instruktion muss bei Index `code_length - 1` sein. Es dürfen nur Instruktionen aus Abschnitt 6.4 der JVM Spezifikation verwendet werden. Der Index des folgenden Opcodes ergibt sich aus dem Index der aktuellen Instruktion plus die Länge dieser Instruktion, einschliesslich ihrer Operanden. Das Ziel einer Sprung- oder Branchinstruktion muss innerhalb des Code Arrays dieser Methode liegen. Das Code Array darf nicht leer sein, es muss mindestens eine `return` Instruktion enthalten sein.
- Die Operanden der Instruktionen, die Werte aus dem `ConstantPool` auf den Stack laden, müssen immer gültige Indices in den `ConstantPool` sein und
  - für `ldc` vom Typ `CONSTANT_Integer`, `CONSTANT_Float` oder `CONSTANT_String`.
  - für `ldc2_w` vom Typ `CONSTANT_Long` oder `CONSTANT_Double`.
  - für Instruktionen, die Fields betreffen vom Typ `CONSTANT_Fieldref`.
  - für *invoke*-Instruktionen vom Typ `CONSTANT_Methodref`. Ausnahme: *invokeinterface* vom Typ `CONSTANT_InterfaceMethodref`.
  - für *instanceof, checkcast, new, anewarray, multinewarray* vom Typ `CONSTANT_Class`.
- Eine *new* Instruktion darf weder zum Erzeugen eines Arrays, noch eines Interfaces, noch einer Instanz einer abstrakten Klasse verwendet werden, sondern nur zum Erzeugen einer Instanz einer konkreten Klasse.
- *anewarray* muss zur Erzeugung von Arrays mit Referenztypen verwendet werden, ansonsten muss *newarray* verwendet werden. Ein Array darf nicht mehr als 255 Dimensionen haben.
- Die Methode zur Instanzinitialisierung `<init>` darf nur von *invokespecial* aufgerufen werden. Es ist verboten, eine andere Methode mit erstem Zeichen „`<`“ aufzurufen, speziell die Methode zur Klasseninitialisierung `<clinit>`, die von der JVM selbst implizit verwendet wird.

- Die Indices der Instruktionen, die den lokalen Stack betreffen, dürfen nie grösser als `max_locals - 1` sein.

Structural Constraints

Die strukturellen Bedingungen für das Code Array spezifizieren die Beziehungen zwischen den JVM Instruktionen.

- Jede Instruktion darf nur mit den angegebenen Typen und der passenden Anzahl Argumente auf dem Operanden Stack und in den lokalen Variablen ausgeführt werden.
- Die Reihenfolge der Werte eines Typen mit zwei Worten (`double, long`) darf nie umgekehrt werden, diese Werte dürfen auch nicht getrennt oder individuell bearbeitet werden.
- Der Operanden Stack darf nie mehr als `max_stack` Worte enthalten. Auch dürfen ihm nicht mehr Worte entnommen werden als vorhanden sind.
- Die *invokespecial* Instruktion muss eine private Methode, eine Methode in der Superklasse oder `<init>` aufrufen. Beim Aufruf von `<init>` muss eine uninitialisierte Instanz einer Klasse auf dem Stack sein.
- Die Instanz einer Klasse muss initialisiert sein, um deren Methoden oder Variablen zu bearbeiten. Die Typen und die Anzahl Argumente bei einem Methodenaufruf müssen mit dem Descriptor der Methode übereinstimmen.
- Die *return* Instruktionen dürfen nur den Typ zurückgeben, der als Rückgabetyp ihrer Methode angegeben ist. Ein Wert vom selben Typ muss sich auf dem Operandenstack befinden.
- Der Wertetyp beim Speichern in ein Field muss dem Typ im Descriptor des Fields entsprechen.
- Die Ausführung verlässt das Code Array nie.

Eine JVM muss jedes `class` File, das sie ausführen soll, auf diese Forderungen und Einschränkungen testen. Dieser Verifikationsschritt ist unabhängig von der Sprache des Quellprogramms und dem Compiler, der das File erzeugt hat.

4.8 Spezielle Methoden zur Initialisierung

Die JVM behandelt jeden Konstruktor als Instanzinitialisierungsmethode mit dem geschützten Namen `<init>`. Diese Methoden dürfen nur mit der *invokespecial* Instruktion auf einer nichtinitialisierten Klasseninstanz aufgerufen werden. Der Aufruf von `<init>` kann also nie aus einem Programm heraus geschehen. `<init>` enthält vor dem eigentlichen Code einen Aufruf einer anderen Instanzinitialisierungsmethode aus dieser Klasse oder der Superklasse und die Initialisierung der Fields, die einen Initialwert erhalten.

Eine Klasse oder ein Interface wird von der JVM durch Aufrufen der Klasseninitialisierungsmethode `<clinit>` initialisiert. Falls das `class` File statische initialisierte Fields enthält, muss der Compiler `<clinit>` generieren. `<clinit>` darf weder vom Programm noch vom Bytecode, sondern nur indirekt von der JVM aufgerufen werden.

Kapitel 5

ClassFile-Analyse-Tool

> Zu viel Beiwerk ist bei jeder Art von Produktion ein Fehler.
>
> *David Hume, Of simplicity and refinement in writing*

Nachdem im vorangehenden Kapitel 4 die JVM und das `class` File Format theoretisch eingeführt worden sind, wird in diesem Kapitel die Behandlung von `class` Files in der Praxis betrachtet. Laut Problemstellung ist ein Ziel dieser Arbeit die Generierung von Bytecode, der von einer JVM ausgeführt werden soll. Zur Analyse und Manipulation von `class` Files wurde ein „ClassFile-Analyse-Tool“ implementiert. Es kann `class` Files einlesen, textuell darstellen und, was für die Codegenerierung (Kapitel 7) am wichtigsten ist, die `ClassFile` Struktur repräsentieren und in binärer Form ausgeben. Der ClassFile-Analyzer ist in Java implementiert.

5.1 Design

Die Idee dieses objektorientierten Ansatzes ist es, das `class` File Format in ein System von Klassen zu übertragen. Durch Vererbung, Instanzierung und Einbindung soll eine Klassenhierarchie entstehen, die alle Konstrukte der `class` File Spezifikation abdeckt.

Jede Struktur des `class` File Formats wird auf eine Klasse abgebildet, die mindestens die Elemente enthält, die in der spezifizierten Struktur vorgegeben sind. Die Behandlung von Teilstrukturen soll weitergeleitet werden, so dass nur die oberste Struktur aktiviert werden muss, die restlichen werden über Objektverkettung aufgerufen. Dabei müssen nur die untersten Strukturen, also die einzelnen Bytes des Bytecodes, tatsächlich manipuliert werden.

Jede dieser Klassen muss die Möglichkeit haben, den Bytecode zu manipulieren, der ihre Struktur betrifft. Dazu gehört das Konstruieren von neuen Elementen, das Setzen und Aktualisieren der Werte, das Schreiben und das Lesen der Elemente im binären `class` File Format, und nicht zuletzt, die Ausgabe der Elemente und ihrer Werte in einer benutzerfreundlichen Form. Mit einem solchen Klassensystem lässt sich ein `class` File setzen, speichern, darstellen und analysieren.

5.2 Resultat

Die Umsetzung dieser Vorgaben resultiert in einer Reihe von Java Klassen, deren Hierarchie als Klassendiagramm 5.1 gemäss dem UML-Standard abgebildet ist [Boo99]. Die gerahmten Namen sind Klassen. Ein geschlossener Pfeil zeigt eine Generalisierung, ein offener Pfeil von

einem Rhombus aus eine Aggregation an. Die Anzahl dieser Beziehungen wird durch das daneben stehende Symbol angegeben: „*“ bedeutet eine optionale, „n“ mehrere und „1“ eine Instanz. Drei Punkte zeigen mehrere Verbindungen an, die der Übersichtlichkeit halber weggelassen wurden. So werden auch die verschiedenen Typen von `ConstantPool` Einträgen nicht explizit gezeichnet.

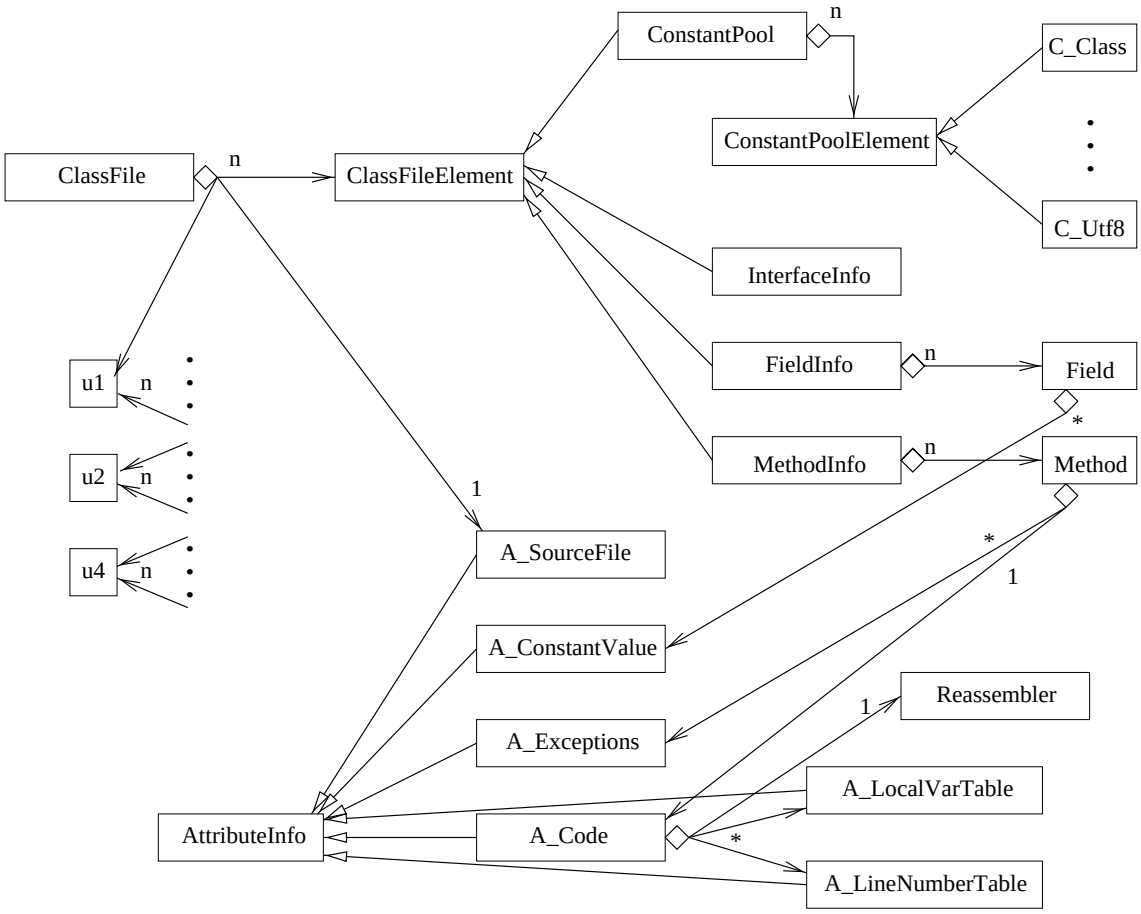

Abbildung 5.1: Klassendiagramm des ClassFile-Analyzers; entspricht der `ClassFile` Struktur.

Die folgenden Funktionalitäten sind in jeder Klasse implementiert[1]:

- Der Konstruktor ist nach dem `class` File Format gebildet, das heisst, alle Elemente einer Struktur werden zu Fields dieser Klasse.
- Ein `class` File wird als Inputstream gelesen und strukturweise interpretiert: `read()`
- Werte können mit `set()` gesetzt oder aktualisiert werden.
- Die Elemente werden im binären Format auf einen Outputstream ausgegeben: `flush()`

[1] Die Klasse `Reassembler` entspricht keiner `class` File Struktur sondern ist eine Helperklasse für `A_Code` zur textuellen Darstellung der Opcodes.

- Für die Analyse werden die Strukturen mit zusätzlichen Informationen auf den Standard Output geschrieben: `write()`. Dabei muss der `ConstantPool` übergeben werden, der als Symboltabelle dient.

Die ausführbare Topklasse des ClassFile-Analyzer ist `ClassFile`, sie bietet ein Interface zur Manipulation aller anderen Klassen. Sie enthält genau die Klassen, die in der Spezifikation für die Struktur `ClassFile` vorgegeben sind.

Weiter soll die Klasse `ConstantPool` erwähnt werden, die als Symboltabelle für das `class` File dient. Sie implementiert ein Containment für die Elemente im `ConstantPool`, die Instanzen der `CONSTANT`-Klassen sind. Um solche Elemente nachzuschauen, besitzt `ConstantPool` die Methode `lookup()`. Dabei werden Referenzen in einen Hashtable mit statischem Typedispatching aufgelöst: Die Elemente des Hashtables werden generell als `ConstPoolElement` betrachtet, der Superklasse aller `CONSTANT`-Klassen, da der konkrete Typ des Inhaltes beim Zugriff auf einen Hashtableeintrag nicht bekannt ist. Dieser Typ muss daher getestet und anschliessend mit *cast* umgewandelt werden. Dieses Prinzip findet auch bei der Auflösung der `Attribute`-Klassen Anwendung. Eine elegantere Lösung bietet das *Command*-Pattern [Bus96]. Das Einfügen in den `ConstantPool` geschieht mit einer speziellen Methode für jeden Elementtyp, da beim Setzen durch den Compiler Typ und Funktion des Elementes bekannt sind (umgekehrte `ConstantPool`-Resolution).

`Field` und `Method` entsprechen den Strukturen `field_info` und `method_info`, die Klassen mit diesen Namen verwalten die Zähler und die Inhalte, gehören daher logisch zur `ClassFile`-Struktur.

Da der Aufruf von Methoden an die Teilstrukturen weitergeleitet wird, muss nur die oberste Struktur, die Klasse `ClassFile`, aktiviert werden. Nur die elementaren Einheiten in der `ClassFile` Struktur, nämlich `u1, u2, u4`, werden wirklich auf den Outputstream geschrieben oder vom Inputstream gelesen.

5.3 Vergleich

In diesem Abschnitt werden einige Programme verglichen, die `class` Files bearbeiten. Eine ausführliche Liste wird von der Belgischen Java User Gruppe geführt:
`http://meurrens.ml.org/ip-Links/Java/CodeEngineering/`

- Die `java.lang.reflect`-Package bietet Klassen an, die eine Inspektion der geladenen Java Klassen erlauben.

- Disassembler erzeugen aus einem binären File eine lesbare Form. Der Input ist ein `class` File, das gemäss der JVM Spezifikation interpretiert und dargestellt wird. Der bekannteste Disassembler wird mit Java (JDK) mitgeliefert: `javap -c`. Shawn Silvermann entwickelte D-Java, einen Disassembler der sowohl mit Java als auch mit Jasmin funktioniert.

- Viel Material findet sich zu Decompilern, die einen Schritt weiter gehen und aus Bytecode ein Java File erzeugen. Dazu werden die Opcodes zu Anweisungen der Java Syntax zusammengesetzt. Beispiele: Mocha von Hanpeter Van Vliet, mittlerweile Teil des JBuilder von Borland[2], darauf aufbauend Jasmine von SRCTec[3] oder unzählige kommerzielle Versionen.

[2] http://www.borland.com/jbuilder/
[3] http://members.tripod.com/~SourceTec/jasmine.htm

- Aktuell sind einige Programme zur Verschleierung oder Korruption von Bytecode auf dem Internet zu finden, die ein Disassemblieren verunmöglichen sollen.

Der vorliegende ClassFile-Analyzer umfasst sowohl die Aufgaben, die von einem Disassembler übernommen werden, als auch die Manipulation von `class` Files. Es kann ein File behandelt werden, das lokal vorhanden sein muss. Die Ausgabe der `ClassFile` Struktur(en) erfolgt sehr ausführlich, so wird der gesamte `ConstantPool`, alle Fields, alle Methoden inklusive Code und alle Attribute dargestellt. Referenzen werden aufgelöst. Jede Struktur, die vom Compilerbackend verwendet wird, kann auch erzeugt werden. Das Programm eignet sich daher für die Analyse und das Schreiben von Bytecode.

5.4 Diskussion

Das verwendete Prinzip hat sich bewährt, es konnten alle Strukuren des `class` File Formats abgebildet werden. Jede Klasse behandelt bei einem Aufruf ihre Elemente und gibt danach die Kontrolle weiter. Die Strukturen müssen daher nicht explizit traversiert werden. Der Programmcode bleibt relativ klein und überschaubar.

Nur die Methode *write* in `Reassembler.java`, das heisst die textuelle Erklärung der Opcodes, ist nicht vollständig implementiert. Im Moment werden die Opcodes behandelt, die bei der Codegenerierung erzeugt werden oder häufig in `class` Files auftreten. Das Vorgehen zum Ergänzen von *write* ist simpel, so muss pro Opcode eine `case`-Anweisung in den Code gefügt werden, die den Opcode und die Argumente einliest und einen erklärenden Text ausgibt.

Die JVM Spezifikation ist wörtlich zu befolgen, jedes Byte muss den Constraints genügen und im Zusammenhang sinnvoll sein. Geschieht ein Fehler oder eine Unachtsamkeit beim Einlesen eines `class` Files, so sind die folgenden Bytes undefiniert und bedeutungslos. Fehler beim Schreiben von Bytecode wirken sich spätestens bei der Verifikation durch die JVM aus, die fehlschlägt.

So ist zum Beispiel der Behandlung von Zwei-Wort-Typen besondere Beachtung zu schenken, da diese ja zwei Einträge im `ConstantPool`, im Array der lokalen Variablen oder auf dem Operanden Stack benötigen. Diese Erfahrung machte unter anderen auch Shawn Silverman (der Entwickler von D-Java), der vorschlägt, einen Zwei-Wort-Typen im `ConstantPool` von einem leeren Eintrag folgen zu lassen.

Wie verschiedene Tests mit `class` Files unterschiedlicher Herkunft und der kurze Vergleich gezeigt haben, ist das hier realisierte Programm durchaus in der Lage, mit Tools zum Lesen und Schreiben von `class` Files mitzuhalten. Als Disassembler übertrifft es in der Ausführlichkeit sogar *javap -c*, wobei dessen *-verify* Option unerreicht bleibt.

Kapitel 6

Einführung Compilerbau

> See the little phrases go, watch their funny antics.
> The men who make them wiggle so, are teachers of semantics.
>
> *Frederick Winsor, Space Child's Mother Goose*

Als Grundlage für die praktische Umsetzung werden die verschiedenen Phasen eines Compilers kurz eingeführt. Die Unterteilung eines Compilers kennt je nach Autor verschiedene Schwerpunkte, die Beschreibung von Frontend und Backend stützt sich vor allem auf die Klassiker von Aho, Sethi und Ullman „The Theory of Parsing, Translation and Compiling“ [Aho72], „Compilers: Principles, Techniques and Tools“ [Aho86] und das Lehrbuch zur Compilerimplementation von Andrew Appel „Modern Compiler Implementation in Java“ [App97]. Weitere verwendete Literatur: John Nichols „The Structure and Design of Programming Languages“ [Nic75] und die Lecture Notes in „Compiler Construction“ [Bau74].

Um ein Programm von einer Quellsprache in eine Zielsprache zu übersetzen, muss es ein Compiler zuerst auseinandernehmen und seine Struktur und Bedeutung verstehen, und es danach anders wieder zusammensetzen. Das Frontend eines Compilers führt die Analyse durch, das Backend die Synthese. Jeder Teil kann in verschiedene Phasen unterteilt werden, die Abschnitte dieses Kapitels folgen dem Weg, den ein Programm im Compiler durchläuft, siehe Abbildung 6.1. Automatisierung und Optimierung sind nicht an bestimmte Phasen des Compilerbaus gebunden.

6.1 Frontend

Das Frontend erhält ein Quellprogramm in einer bestimmten Programmiersprache als Input, analysiert dieses Programm und generiert daraus ein Zwischenprogramm für das Backend. Das Zwischenprogramm kann als Zwischencode vorliegen oder als abstrakte Syntax, wobei die letzte Phase (6.1.4) ausgelassen wird.

6.1.1 Lexikalische Analyse

Die lexikalische Analyse unterteilt den Input in einzelne Worte oder Token. Es kommen reguläre Ausdrücke, deterministische oder nicht-deterministische endliche Automaten und automatische Analyseprogramme *Lexer* zum Einsatz. Der erste und bekannteste Lexer stammt von M. E. Lesk: Lex [Les75].

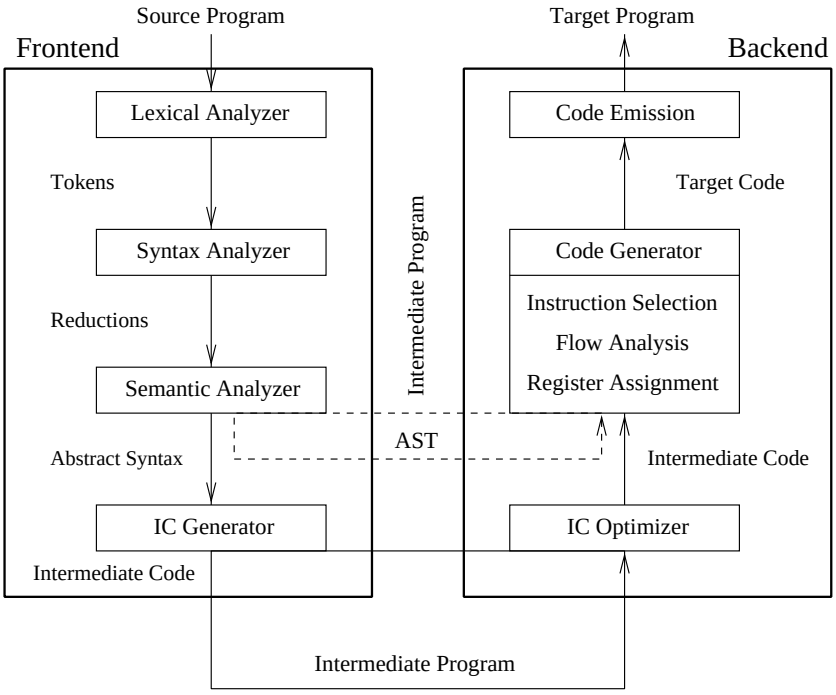

Abbildung 6.1: Phasen und Zwischenprogrammdarstellungen eines Compilers

6.1.2 Syntaktische Analyse

Die syntaktische Analyse parst die Satzstruktur, die als Sequenz von Token vorliegt. Das heisst, es wird mit Hilfe einer Grammatik der Quellsprache kontrolliert, ob die Eingabe syntaktisch korrekt ist. Es kommen verschiedene Typen von Grammatiken und Parsern zum Einsatz. Die automatische Unterstützung bei der Konstruktion eines Parsers wird von Compiler Compilern geleistet (siehe Abschnitt 6.3). Johnson entwickelte „YACC: Yet Another Compiler-Compiler“ ebenfalls 1975 bei Bell [Joh75].

6.1.3 Semantische Analyse

Nachdem die syntaktische Analyse einen Satz als zu einer Grammatik einer Programmiersprache zugehörig erkannt hat, muss eine sinnvolle, semantische Aktion mit diesem Satz geschehen. Es sind drei verschiedene Vorgehen bei der **semantischen Analyse** möglich:

- Man kann einen ganzen Compiler in die syntaktischen Aktionen eines Parsers einbauen. Ein solcher Compiler ist jedoch schwierig zu verstehen und zu unterhalten und zur Analyse des Programmes in genau der Reihenfolge gezwungen, in der es geparst wird.
- Mit den semantischen Aktionen wird ein *Parse Tree* produziert, der für jedes Inputtoken ein Blatt und für jede Grammatikregel, die reduziert wurde, einen Knoten hat. Dieser Baum stellt die **konkrete** Syntax eines Programmes dar, ist aber ungünstig zur Traversierung und Behandlung, da er zu viel konkrete Informationen beinhaltet, wie Interpunktion, Syntaxredundanz und abhängig von der Grammatik ist.
- Besser ist ein **abstrakter** Syntax Baum, *abstract syntax tree* oder „AST“, der als saubere Schnittstelle zwischen dem Parser und dem restlichen Compiler verwendet wird. Ein

AST beinhaltet die Satzstruktur des Eingabeprogrammes, befreit von Parsingthemen. Mit Hilfe dieser Datenstruktur wird das Programm semantisch interpretiert.

Nebst der Produktion eines AST und der Kontrolle der Typenkorrektheit der Ausdrücke werden während der semantischen Analyse Symboltabellen oder *Environments* erstellt. Diese können als globaler Hashtable, als Array pro Definitionsbereich oder als binäre Suchbäume implementiert werden [Wai74, App97].

6.1.4 Zwischencode Generation

Diese Phase übersetzt die abstrakte Syntax nach abstraktem Maschinencode, der aus einfachen Basiselementen besteht. Die komplexen Komponenten der abstrakten Syntax werden auf diese maschinenunabhängige Zwischensprache abgebildet, anschliessend wird daraus für jede erwünschte Zielmaschine konkreter Maschinencode erzeugt. Compilerkomponenten, die für verschiedene Quellsprachen und verschiedene Zielmaschinen verwendet werden sollen, kann man durch diese unabhängige Zwischensprache als portable Module aufbauen. Ein Versuch einer Definition von universal verwendbarem Zwischencode wurde mit „UNCOL“ (Unified Compiler Language) gemacht [Str58].

6.2 Backend

Das Backend erhält ein Zwischenprogramm von einem Frontend und generiert daraus ein Zielprogramm für eine Zielmaschine. Falls das Zwischenprogramm als abstrakte Syntax vorliegt oder der Zwischencode nicht optimiert werden soll, kann die erste Phase des Backends übersprungen werden.

6.2.1 Reduktion des Zwischencodes

Der Zwischencode, der vom Frontend übergeben wird, wird in dieser Phase optimiert. Es existieren verschiedene Verfahren, die Transformationen an der Baumdarstellung oder einer Zwischencodesequenz (*basic blocks*) vornehmen (6.4). Falls der Compiler keinen abstrakten Maschinencode verwendet, wird diese Phase nicht umgesetzt.

6.2.2 Codegeneration

Bei der **Instruktionswahl** wird der Zwischencode nach Maschinencode übersetzt. Dieser Vorgang kann als *tiling* des abstrakten Syntaxbaumes mit passenden Instruktionspattern betrachtet werden. Ein Abdeckung (*tile*) ist eine Menge von Baumstrukturen, die sich auf eine Maschineninstruktion abbilden lassen. Der Baum muss vollständig mit nicht überlappenden Abdeckungen belegt werden. Ziel ist eine optionale Instruktionssequenz bezüglich Länge, Zeit oder sonstiger Kosten. Es existieren verschiedene Algorithmen zur Selektion der Instruktionen: *Maximal Munch*, *Dynamic Programming* oder *Fast Matching*.

Falls kein abstrakter Maschinencode vorliegt, wird die abstrakte Syntax direkt nach konkretem Maschinencode übersetzt. Für jedes Element der abstrakten Syntax muss eine Reihe von Instruktionen gewählt werden.

Die **Zuweisung von konkreten Registern** kann unabhängig von der Wahl der Instruktionen erfolgen, die in diesem Falle mit temporären oder abstrakten Registern arbeiten. Falls die Anzahl der Register auf der Zielmaschine begrenzt ist, dient die **Lebendigkeitsanalyse** zur Einschränkung der verwendeten Register. Dabei wird mit Hilfe eines Kontrollflussgraphen

für Anweisungen analysiert, welche Variablen in Zukunft verwendet werden und ob eventuell nicht-initialisierte Variablen vorkommen. Die Register von nicht länger verwendeten Variablen können wiederverwendet werden, uninitialisierte Variablen werden als Fehler gemeldet. Da eine Variable als *lebendig* bezeichnet wird, wenn sie in Zukunft noch gebraucht wird, wird die Analyse von der Zukunft des Kontrollflusses nach der Gegenwart durchgeführt. Die minimale Anzahl Register wird durch *graph coloring* Algorithmen gefunden. Diese Phase bietet viel Raum für Optimierungen.

6.2.3 Code Emission

Zum Schluss werden die temporären Namen in den Instruktionen durch die ausgewählten Maschinenregister ersetzt. Die konkreten Instruktionen werden zu einem Maschinenprogramm zusammengefasst. Die Zielsprachen und ihre Ausführung lassen sich in drei Gruppen unterteilen: Absolute Maschinensprache, die direkt nach der Compilation ausgeführt wird. Maschinensprache in einem *object module*, einem frei platzierbaren File, das vor der Ausführung geladen und eventuell mit anderen Modulen gelinkt wird. Assemblersprache, die auf einer Maschinensprache aufbaut und bei der Codegeneration komfortabler ist. Assemblerprogramme müssen zusätzlich assembliert werden.

6.3 Automatisierung

Die verschiedenen Phasen des Compilerbaus können mehr oder weniger automatisiert werden.

Am üblichsten ist die Automatisierung bei der Generation von Parsern. Eine Grammatik, die die Syntax der Sprache spezifiziert, wird mittels der LALR(1) Technik zu einem effizienten Parser umgebaut. Yacc ist ein Beispiel eines LALR(1)-Parsers. Dieser Typus wurde von DeRemer in „Practical Translators for LR(k) Languages“ eingeführt [DeR69]. DeRemer schlägt auch einen Compiler vor, der vollständig auf Baumtransformationen basiert [DeR74].

Die Zwischensprache eines Compilers hat meist eine operationale Semantik. Die Bedeutung der Sprachelemente wird als Operationen einer abstrakten Maschine ausgedrückt (ungleich der Zielmaschine). Im Gegensatz dazu hat durch eine funktionale Semantik jeder Knoten eine statische Bedeutung. Darauf aufbauend kann ein Codegenerator spezifiziert werden, der die funktionale Darstellung nach Maschinencode übersetzt, wie Appel in „Semantics-Directed Code Generation“ zeigt [App84]. Lee konstruiert in „Realistic Compiler Generation“ einen Zwischencode Generator, der auf funktionaler Semantik basiert [Lee89].

In der vorliegenden Arbeit wird Gentle (Abschnitt 2.2) zur Automatisierung verwendet. Gentle kann sowohl bei der Konstruktion eines Frontends als auch eines Backends eingesetzt werden. Die Analyse arbeitet mit Lex und dem LALR(1)-Parser Yacc zusammen. Es wird eine Attributgrammatik verarbeitet, die kontextsensitive Bedingungen der Eingabesprache zusammen mit der kontextfreien Syntax angibt. Dies ergibt einen dekorierten Parsebaum. Hier können beliebige Regeln angefügt werden, die die weiteren Phasen behandeln. Bei diesem Vorgehen konstruiert man einen Compiler, der in einem Durchgang und in der Reihenfolge des geparsten Inputs ein Zielprogramm generiert.

6.4 Optimierung

Maschinell erzeugter Zwischen- oder Zielcode ist meist weniger effizient als von einem Programmierer geschriebener. Zur Steigerung der Effizienz eines Compilers werden Optimierungstechniken verwendet, die optimalen Code erzeugen sollen, der mit handcodiertem vergleichbar ist.

Um Optimierungen sinnvoll durchzuführen, muss der Erfolg einer Transformation abgeschätzt werden. Allgemein wird eine Verkürzung der Ausführungszeit oder eine Amelioration compilierter Programme angestrebt.

Es existieren unzählige Artikel über Optimierungstechniken. Einige allgemein gehaltene Aufzählungen und Vergleiche findet man in „A Catalogue of Optimizing Transformations“ von Frances Allen und John Cocke [All71], in „Comparison of Optimation Techniques“ von Patricia Goldberg [Gol72], in „Design and Optimization of Compilers“ von Randall Rustin [Rus72] und im Kapitel 10 von Aho et al [Aho86].

Die Techniken lassen sich in maschinenabhängige und -unabhängige Optimierungen unterteilen, je nachdem, ob Eigenschaften der Zielmaschine berücksichtigt werden oder nicht.

Maschinenunabhängige Transformationen

Maschinenunabhängige Optimierung umfasst Transformationen aller Zwischenprogrammdarstellungen, da diese nicht an eine Zielmaschine gebunden sind. Optimierung kann während der semantischen Analyse (6.1.3) und der Behandlung von Zwischencode (6.1.4, 6.2.1) eingesetzt werden. Zuerst wird der Datenfluss analysiert, die resultierenden Informationen über die Verwendung von Daten im Programm sind die Grundlage für Transformationen. Maschinenunabhängige Transformationen betrachten oft nicht das ganze Zwischenprogramm sondern nur *basic blocks*, das sind Ausschnitte des Zwischencodes ohne Sprünge und Labels.

Algebraische Transformationen ändern Zwischenprogramme, ohne Informationen zu verlieren. Dies kann Umbenennen oder Auswechseln, Verschieben von Code nach schwach frequentierten Regionen, Ersetzen von Operationen durch effizientere Operationen oder sogar die Elimination von redundanten Ausdrücken und nie erreichtem Code beinhalten.

Nichtalgebraische Transformationen: *Constant folding* ersetzt die Verwendung von Variablen, die auf einen konstanten Wert gesetzt wurden, durch diese Konstante.

Schleifen können speziell optimiert werden, sobald sie mit Hilfe von *dominators* gefunden sind: eine Schleife wird aufgelöst (*unrolled*) und durch die wiederholte Codesequenz ihres Bodys ersetzt, wenn die Datenflussanalyse zeigt, dass diese Schleife nur eine kleine Anzahl Durchgänge hat. Falls die Bedingungen zweier Schleifen gleich sind, kann man diese zusammenlegen (*fusion*).

Probleme ergeben sich, wenn Daten mit verschiedenen Namen referenziert werden oder Funktionen Seiteneffekte haben können. Konservative Optimierungsmethoden nehmen in solchen Fällen keine Transformationen vor.

Maschinenabhängige Transformationen

Maschinenabhängige Optimierung wird auf der Zwischenprogrammdarstellung der letzten Phasen durchgeführt (6.2.2), die auf eine konkrete Zielmaschine ausgerichtet sind.

Bei der Peephole Optimization wird ein kleines, bewegliches Fenster über das Zielprogramm bewegt und die untersuchten Instruktionen nach Möglichkeit durch kürzere oder schnellere ersetzt. Es lassen sich die meisten Transformationen anwenden, die bei der maschinenunabhängigen Optimierung beschrieben sind, jetzt aber mit konkreten Instruktionen und Wissen über den Aufbau der Zielmaschine. Je nach Scheduling der Zielmaschine können Instruktionen so angeordnet werden, dass die Ausführungszeit minimiert wird.

Die effiziente Verwendung von high-speed Registern und die Speicherzuweisung an Variablen sind maschinenabhängig. Kennedy optimiert die Registerzuweisung [Ken72], Beatty präsentiert einen Algorithmus zur globalen Registerzuweisung [Bea72].

Kapitel 7

Codegenerierung - Implementation

> Calvin: [...] I made some modifications. The box is on it's side now. It's a duplicator! It combines the technologies of the transmogrifier and a photocopier, so instead of merely making a reproduction on paper, this machine creates a real duplicate!
> Hobbes: So our financial worries are over?
>
> *Bill Watterson, Scientific Progress goes „Boink“*

In Kapitel 6 wurden die theoretischen Grundlagen des Compilerbaus eingeführt. Das vorliegende Kapitel zeigt das Design, das Umfeld und die Implementation eines Compilerbackends auf. Dieses Backend (Abbildung 7.1) erhält einen AST nach der GZS (Kapitel 3) als Input und generiert daraus Bytecode für die JVM (Kapitel 4). Das Zwischenprogramm wird vom Parser/Builder umgeformt und an den ABCompiler (AST-Bytecode-Compiler, „ABC“) übergeben; beide sind in Java implementiert.

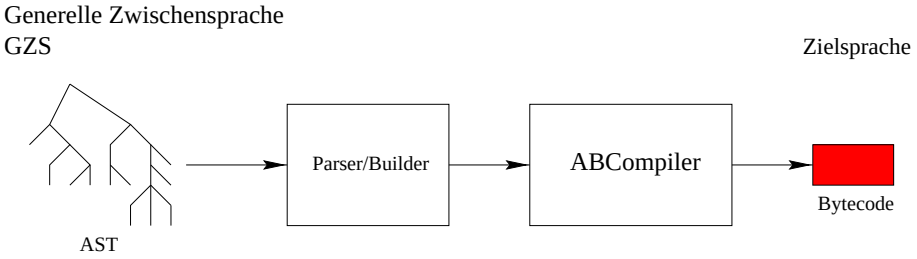

Abbildung 7.1: Komponenten des implementierten Backends.

7.1 Umgebung des Compilers

Ein Compiler hängt von verschiedenen Faktoren ab. Wie werden Ziel und Zweck des Compilers definiert? Soll effizienter, optimierter Zielcode generiert werden oder ist schnelle Compilation mit sinnvollen Fehlermeldungen wichtiger? Die Art der Quellsprache, die Zielsprache, die Zielmaschine, das Operationssystem und nicht zuletzt die Implementationssprache bilden das Umfeld, das sich auf den Bau des Compilers auswirkt. Dieser Abschnitt diskutiert die Wahl der äusseren Faktoren für das vorliegende Compilerbackend.

7.1.1 Frontend

Es wird für jede der untersuchten Paradigmasprachen (imperative (2.3), objektorientierte (2.4), funktionale (2.5), logische (2.6)) ein Frontend erstellt, das Zwischenprogramme als abstrakte Syntaxbäume im GZS-Format liefert.

Diese Frontends werden in der Compilerbeschreibungssprache Gentle (2.2) implementiert. Da Gentle sowohl die lexikalische als auch die syntaktische Analyse automatisch mit LEX [Les75], respektive YACC [Joh75] durchführt, wird auf diese Phasen nicht näher eingegangen. (Siehe Abbildung (2.1) zum Zusammenhang der Gentle Komponenten.)

Die semantische Analyse wird in den Gentleprogrammen beim Parsen ausgeführt, es wird kein spezieller Parsebaum erstellt, sondern direkt ein abstrakter Syntaxbaum. Dieser AST wird ohne weitere Modifikationen an das Backend übergeben, insbesondere wird kein abstrakter Zwischencode erzeugt, da keine Portabilität nach verschiedenen Zielsprachen verlangt ist.

Nur syntaktisch korrekte Programme passieren diese Phase, Fehler werden von den Frontends ausgegeben. Da keine Typenchecks oder sonstige semantischen Tests vorgenommen werden, werden generell semantisch korrekte Inputprogramme für den Compiler vorausgesetzt.

7.1.2 Zwischensprache

Das Compilerbackend ABC, das die Codegenerierung vornimmt, erhält ein Programm in einer Zwischendarstellung als Eingabe. Kapitel 3 beschreibt den Aufbau und die Motivation für die GZS als Zwischenprogrammdarstellung dieses Compilerframeworks.

Es sind alle Programme erlaubt, die mit der generellen abstrakten Syntax GZS als Grammatik aufgebaut werden (Appendix A.5). Das Backend arbeitet folglich unabhängig von der Quellsprache des eingegebenen Programmes. Um GZS im Textformat einzulesen und zu verarbeiten, wird der *JavaCC* Parser PARSER/BUILDER als Schnittstelle eingesetzt (7.2). Dieser bildet die abstrakten Typen auf abstrakte Klassen und die Funktoren auf konkrete Unterklassen ihres Typs ab und instanziiert so einen Baum von AST-Klassen. Der ABCOMPILER ergänzt diese templateartigen AST-Klassen um Methoden zur Generation von Bytecode.

Eine Vereinfachung der Zwischenprogramme wäre vorstellbar und wünschenswert, solange sie die Struktur der GZS nicht verändert. Im Abschnitt 6.4 werden einige optimierende Transformationen skizziert, die maschinenunabhängigen könnte man bei der semantischen Analyse (6.1.3) und der Reduktion des Zwischencodes (6.2.1) anwenden. Für die Codegenerierung macht es keinen Unterschied, ob ein Zwischenprogramm optimierend transformiert wurde oder nicht.

7.1.3 Zielsprache

Ziel dieser Arbeit ist die Generation von portablem Code. Die generierten Programme sollten möglichst auf allen Plattformen und unter allen Betreibssystemen ausführbar sein. Es muss daher eine Zielsprache gewählt werden, die allgegenwärtig ist. Eine Möglichkeit, diesem Ideal in realistischer Weise nahe zu kommen, ist die Zusammenarbeit mit der JVM. Zusätzlich zur hohen Portabilität hat die JVM eine weitere Eigenschaft, die dieser Arbeit zu Gute kommt: sie ist mit einer standardisierten Spezifikation klar umschrieben.

Nun stehen zwei Ansätze zur Auswahl, wie Zwischenprogramme für die JVM compiliert werden können. Einerseits kann ein Compiler direkt Bytecode, die Maschinensprache der JVM, generieren, andererseits kann er ein Zwischenprogramm in ein Javaprogramm übersetzen, welches anschliessend mit einem Standardcompiler nach Bytecode compiliert wird. Was sind die Vorteile und Nachteile dieser zwei möglichen Zielsprachen?

Java ist eine bekannte objektorientierte Programmiersprache, Bytecode dagegen eine tiefe Maschinensprache. Mit Java und seinen umfangreichen Packages lassen sich die Aktionen bei der Codegeneration komfortabel umschreiben, für die Generierung von Bytecode müssen sie zuerst in einzelne Elemente zerlegt und eventuell umgeformt werden. Der Komfort von Java wird mit einem zusätzlichen Compilerschritt bezahlt, eben von Java nach Bytecode.

Bytecode kann nach Belieben generiert werden, die JVM lässt sich bewusst steuern. Dagegen ist der Java Compiler eine Blackbox, der generierte Bytecode nicht kontrollierbar.

Java ist in „The Java Language Specification“ [Gos96] definiert, die Sprache wird im Laufe der Zeit erweitert und erneuert. Die erwünschten Eigenschaften Robustheit, Sicherheit und Portabilität verdankt Java der JVM, diese Punkte treffen daher auch auf Bytecode zu. Bytecode ist ebenfalls genau spezifiziert, jedoch kaum Änderungen unterworfen.

Die Eigenschaften lassen sich wie folgt zusammenfassen. **Bytecode** ist standardisiert, klar umschrieben und bietet trotz Assemblerniveau eine Fülle von Instruktionen und Hilfsmitteln an. Er ist der Schlüssel zur JVM. **Java** erzeugt einen zusätzlichen Schritt, der nicht manipuliert werden kann. Zudem kann sich sich der Komfort negativ auswirken, wenn man funktionale Konstrukte auf das objektorientierte Paradigma abbilden will. Die Nähe zur JVM wird höher gewichtet als der eventuelle Komfort. Deshalb wird **Bytecode** als Zielsprache für das Compilerbackend ausgewählt.

Klassifizierung von Bytecode

Einerseits passt ein `class` File, das den Bytecode enthält, auf die Beschreibung eines *object modules* von Aho et al. [Aho86]: ein *object module* ist ein vom Compiler erzeugtes File, das ein Maschinensprachprogramm enthält, das vor der Ausführung geladen werden muss; im Gegensatz zu einem absoluten Maschinenprogramm, das nach der Compilation direkt ausgeführt wird. Ein `class` File wird geladen (4.6), kann mit anderen `class` Files zusammen ausgeführt werden und dazu die Aufrufmechanismen der JVM verwenden.

Andererseits verhält sich die Menge der JVM Instruktionen wie eine Assemblersprache, die symbolische Instruktionen und Hilfsfunktionalitäten zur Verfügung stellt. So wird aus dem Zwischenprogramm nicht direkt ein Maschinenprogramm erzeugt, das auf der Plattform ausgeführt wird, sondern nur Anweisungen an die JVM.

Also betrachtet man Bytecode als Vereinigung der Eigenschaften beider Definitionen. Die Generierung von Bytecode wird durch diese mehrdeutige Zuweisung nicht beeinflusst.

7.1.4 Backend

Die klassischen Phasen in einem Backend sind nach Abbildung 6.1 die Optimierung des Zwischencodes, die Instruktionswahl, die Registerzuweisung und die Code Emission.

Das vorliegende Backend übersetzt AST nach Bytecode. Es befasst sich nicht mit der Zwischencodeoptimierung, weil kein abstrakter Zwischencode vorliegt. Die abstrakte Syntax wird direkt nach konkretem Maschinencode übersetzt, für jedes Element der abstrakten Syntax wird eine Reihe von Instruktionen gewählt. Weil die JVM eine relativ grosse Zahl (65535) an lokalen Variablenplätzen zur Verfügung stellt, wird die Registerzuweisung ohne Beschränkung vorgenommen. Das heisst, es findet keine Lebendigkeitsanalyse statt, nicht mehr *lebendige* Variablen behalten ihren Registerplatz. Mit Algorithmen für minimale Registerbelegung könnte aber der Speicherplatz auf dem Heap reduziert werden. Zum Schluss werden die konkreten Instruktionen und Strukturen zu einem `class` File zusammengefasst.

Die JVM bietet einen automatisch verwalteten Heap, übernimmt die Speicherung der Frames auf dem Stack und die nötigen Schritte beim Wechseln der laufenden Methode, wie schon

in Abschnitt 4.2 beschrieben. Die Codegeneration ist somit von diesen Themen befreit.

7.1.5 Implementationssprache

Die JVM ist die Zielmaschine dieses Compilers, ein JDK (Java Developers Kit) kann daher bei der Anwendung des vorliegenden Frameworks vorausgesetzt werden. Ein Framework ist ein spezielle Klassenstruktur, die nach Belieben verwendet, adaptiert und ausgebaut werden kann [Joh88, Fay97]. Die Implementationssprache sollte dabei keine einschränkende Voraussetzung bilden, deshalb wird Java [Gos96, Fla96] verwendet, die Sprache des JDK, somit wird auch der Compiler selbst auf der JVM ausgeführt und ist plattformunabhängig.[1]

Java ist objektorientiert, erlaubt die Modularisierung verschiedener Phasen und Komponenten, die Unterteilung von abstrakten und konkreten Klassen, die Zugriffsbeschränkung für Teilmengen des Systems und die Anwendung von Subtyping. Weiter existiert eine Reihe von Packages im JDK mit verschiedenartigster Funktionalität; das Compilerframework wird in das JDK-Framework eingebunden.

Das Backend könnte unter anderem auch mit Gentle implementiert werden. Die semantischen Aktionen (6.1.3) würden das Zwischenprogramm in einem einzigen Schritt zu Code transformieren. Ein Vorteil von Gentle ist die Unterstützung von zusätzlichen Prädikaten und Kosten für das Regelsystem. Nachteile sind die fehlende Modularität und die geringe Verbreitung von Gentle (im Vergleich zu Java). Die Implementation der Frontends wird in dieser Arbeit mit Gentle vorgenommen, kann aber, wie auch die des Backends, in einer beliebigen Programmiersprache durchgeführt werden.

Das Umfeld des Compilers ist gegeben, die Vorgaben daraus ebenfalls. In den nächsten Abschnitten wird das Innere des Compilerbackends beschrieben.

7.2 Schnittstelle zwischen GZS und Backend

In Abschnitt 2.3.4 wurde das Prinzip eingeführt, wie ein Konstrukt der GZS auf die entsprechende Java Klasse abgebildet wird:

Abbildung der GZS auf Java Klassen

Jeder Typ der abstrakten Syntax in der GZS entpricht einer abstrakten Klasse im ABCompiler. Die Funktoren eines Typs entsprechen konkreten Klassen, die von der abstrakten Klasse den Typ erben. Dabei hat jede konkrete Klasse einen Konstruktor, dessen Parametertypen analog zu ihrem Funktor sind. Diese Parameter werden zur Initialisierung der Fields oder Instanzvariablen der konkreten Klassen verwendet. Dieses Prinzip wird auf alle Typen der abstrakten Syntax angewendet, ausser denjenigen, deren Funktoren keine Parameter haben. Hier ist eine abstrakte und für jeden Funktor konkrete Klasse überflüssig, da diese keine Fields hätte und somit auch keine darauf aufbauenden spezifischen Funktionen. Deshalb wird für sie eine konkrete Klasse erstellt, die die verschiedenen Funktoren als Aufzählungstyp festhält.

Ein Ansatz, die GZS auf den ABCompiler abzubilden, ist die direkte Konstruktion von Java Klassen aus dem Gentle Regelsystem. Die vorliegende Arbeit realisiert diese Schnittstelle durch Parsen des Zwischenprogramms, mit semantischen Aktionen. Die Eingabe von Datenkonstruktoren erwies sich als Sackgasse.

[1] *javac*, der Java Compiler des JDK, ist in Java geschrieben, die virtuelle Maschine in C.

Erzeugen des ABCompiler durch Datenkonstruktoren

In einem ersten Schritt wurden Programme durch Anwendung von Datenkonstruktoren dargestellt, da keine Möglichkeit zur Eingabe der GZS im Textformat in das Compilerbackend bestand. Schon kleine Testprogramme verwenden aber eine grosse Zahl Konstruktoren, zudem ist diese Methode sehr unflexibel bei Änderungen der Testprogramme. Eine Zwischenschicht drängte sich auf, die eingegebene AST automatisch in eine zur Codegenerierung günstige Form bringt und eine einfache Kombination der Frontends der Paradigmasprachen mit dem Backend erlaubt.

Erzeugen des ABCompiler durch Parsen des Zwischenprogramms

Mit dem Parser/Builder wird das oben beschriebene Vorgehen automatisiert. Der Parser/Builder ist mit *JavaCC*, dem Java Compiler Compiler von Sun, und dem dazugehörigen *tree building preprocessor* JJTree[2] implementiert.

Dazu wird eine Grammatik der Eingabesprache GZS aufgestellt. JavaCC parst diese und generiert als semantische Aktion für jedes Nonterminal ein Skelett `AST<Nonterminal>.java`, das „parserintern“ das Zwischenprogramm repräsentiert. Diese Klassen erben alle von `SimpleNode.java`, einer von JavaCC generierten Klasse zur Manipulation von Parsebäumen. Von Hand wird diese flache Klassenhierarchie entsprechend der Hierarchie der GZS umgeformt.

JJTree unterstützt das *Visitor* Design Pattern [Bus96], indem er ein Visitor-Interface implementiert und in jeden generierten Knoten eine Methode einfügt, die den Visitor akzeptiert. Die Java Klasse `Builder.java` implementiert dieses Interface, indem für jede `AST<Nonterminal>`-Klasse in einer Methode angegeben wird, wie das entsprechende Template (Abschnitt 2.3.4) gebildet wird: `Builder` traversiert den AST und baut eine konkrete Kopie davon mit Java Klassen auf, indem von jedem Knoten (= Funktor) die Sohnknoten oder Argumente des Funktors instanziiert werden. Terminale Elemente sind die Token des AST und Funktoren ohne Parameter (wie `ABSTRACT` mit `abstract, concrete`), die als Strings zurückgegeben werden.

Mit der GZS aus Kapitel 3 als Vorlage wird so eine Reihe von Java Klassen konstruiert, die als Darstellung von Zwischenprogrammen im Compiler dienen. Diese Klassen machen den ABCompiler aus. Dabei wird die abstrakte Syntax der GZS als Datenstruktur verwendet, die die Satzstruktur von Programmen im Compiler festhält und deren Bearbeitung zulässt. Dazu gehört auch die Traversierung der Baumstruktur mittels Rekursion. Jede Klasse des ABCompiler kann ihre Elemente verarbeiten und traversieren; das Design wird im nächsten Abschnitt beschrieben.

7.3 Design des Backends

Dieser Abschnitt beschreibt das Design des Compilerbackends ABCompiler. Beim Betrachten des dazugehörigen Klassendiagramms in Abbildung 7.2 fällt sofort auf, dass der Aufbau gleich demjenigen der generellen Zwischensprache in Abbildung 3.2 ist. Die GZS ist ausführlicher, es sind nicht alle entsprechenden Klassen abgeleitet und implementiert worden; die vorliegende Arbeit führt die Konstrukte ein, die für die imperative Quellsprache benötigt werden.

Die Codegenerierung findet in den ABC-Klassen statt, die vom Parser/Builder (Abschnitt 7.2) instanziiert werden. Jede Klasse hat Methoden zur Manipulation und Auswertung des Knotens, den sie repräsentiert. Sämtliche Klassen schreiben ihre Informationen in das gleiche `class` File, dessen Verwaltung und Strukturen vom ClassFile-Analyse-Teil (Kapitel 5) des Frameworks übernommen werden. Die `ClassFile` Struktur wird in zwei Schritten gesetzt,

[2] Informationen über JavaCC: `http://www.suntest.com/JavaCC/`

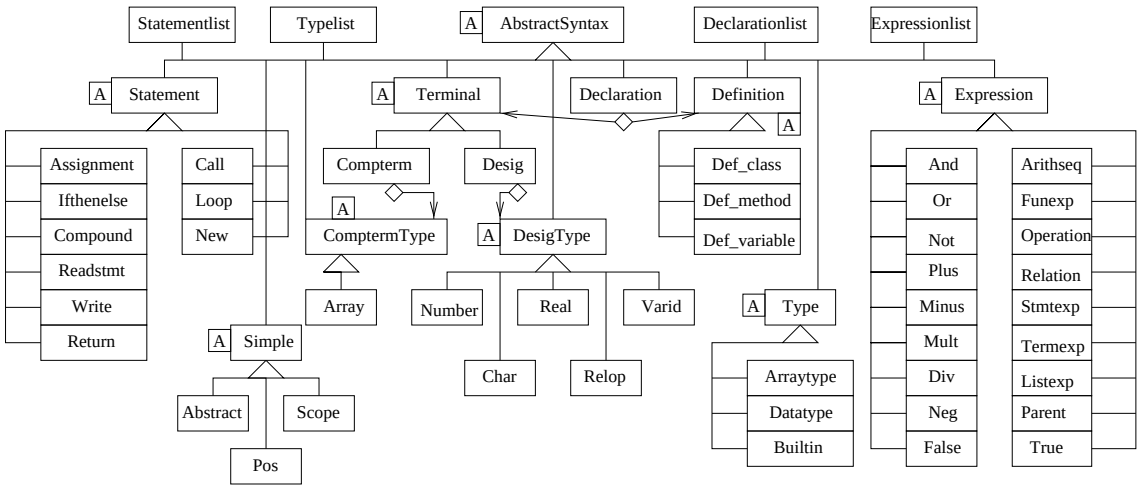

Abbildung 7.2: Klassendiagramm der AST-Klassen (Java).

mittels zweier Methoden, die in jeder Klasse implementiert sind: `check`, die den `ConstantPool` und statische Informationen bearbeitet, und `generate`, die die restlichen und die von der Symboltabelle abhängigen Strukturen generiert. Für die Ausführung wird zuerst die Methode `check`, danach die Methode `generate` des ABCompiler aufgerufen. Beide Methoden, beziehungsweise die ganze Codegenerierung, gehen nach dem gleichen Prinzip vor.

1. Zuerst wird kontrolliert, ob die Inhalte der Fields in ihrem Wertebereich liegen. Neue Variablen werden initialisiert, Defaultwerte gesetzt.
2. Danach wird die Methode an jeden Nachfolger im AST weitergegeben. Meist wird für jedes Field die laufende Methode aufgerufen.
3. Mit den Rückgabewerten der Nachfolger werden die Strukturen ergänzt oder berechnet.
4. Die vollständigen Strukturen werden im `ClassFile` abgelegt.
5. Zum Schluss werden, falls nötig, Werte aktualisiert und ein Rückgabewert an die aufrufende Klasse zurückgegeben.

Nachdem die oberste Klasse des ABCompiler `generate` beendet hat, sind alle Informationen des Zwischenprogramms in der `ClassFile` Struktur gesetzt. Die Ausgabe der Elemente in das konkrete `class` File erfolgt in der spezifizierten Reihenfolge. Mit der ClassFile-Analyse-Methode `flushCF` wird jede Struktur durch implizite Rekursion in das binäre File geschrieben.

Die zwei Phasen der Bytecodegenerierung werden in den nächsten Abschnitten detailliert betrachtet.

7.3.1 Erster Schritt: check

Im ersten Durchgang der Bytecodegenerierung werden spezifische Informationen gesammelt und unabhängige, statische Werte im `ConstantPool` gesetzt. Alle Strukturen des `ClassFile` werden mit Zeiger auf `ConstantPool` Elemente zusammengestellt. Der `ConstantPool` ist der einzige Ort im `ClassFile`, wo konkrete Werte vorkommen, abgesehen von Zählern und Zahlwerten mit bestimmten Bedeutungen wie Tags oder Flags. Er wird von der JVM als Symboltabelle verwendet. Weil die Standardinformationen im `ConstantPool` für die Codegenerierung

nicht ausreichen, wird während der `check`-Phase je eine Hashtabelle für die lokalen und die class Variablen des Zwischenprogramms und deren Attribute (Place, Type, Access, ClassRef, ArrayType) erstellt.

Konkret bedeutet dies, dass die folgenden spezifischen Informationen gebildet und die folgenden Elemente im `ConstantPool` gesetzt werden:

- Die Deskriptoren für Arrays, Methoden und Fields werden nach Spezifikation gebildet.
- Datentypen und Zugriffsmodi werden nach JVM-internen Flags konvertiert.
- Variablen, Bezeichner, Zahlen und Strings werden in den `ConstantPool` eingefügt.
- Der Name des Eingabeprogramms und eine Klassenreferenz auf diesen Eintrag kommen in den `ConstantPool`. (Die Referenz wird im `ClassFile` dem Feld `this_class` zugewiesen.) Analog wird der Name der Superklasse mit `super_class` festgehalten.
- Die Instanzinitialisierungsmethode `<init>` wird kreiert und der Liste der Methoden hinzugefügt.
- Alle Methoden, ausser `main`, werden per Methodenreferenz im `ConstantPool` gesetzt. Ebenso alle Fields per Fieldreferenz.
- Die Attributnamen, die im `class` File Format verwendet werden, kommen als Elemente in den `ConstantPool`: `SourceFile, ConstantValue, Code, Exceptions, LineNumberTable, LocalVariableTable`

Im Prinzip sollten während dieser Phase auch die u2-Felder `access_flags, this_class, super_class` des `ClassFile` gesetzt werden, da diese statisch bestimmbar sind. Den `check` Methoden wird aber nur eine Referenz auf den `ConstantPool` übergeben, daher ist in dieser Phase kein Zugriff auf `ClassFile` möglich.

7.3.2 Zweiter Schritt: generate

Bei der zweiten Traversierung des Zwischenprogramms wird aus den im ersten Schritt gesammelten Informationen und mit Hilfe der Symboltabellen ein `ClassFile` erstellt, das dem Zwischenprogramm entspricht. Dafür werden die class Variablen des Inputs als `Field_info` beschrieben und im `ClassFile` festgehalten.

Für jede Methode wird in einer `Method_info` Struktur der Name und der Descriptor mit Indices auf `ConstantPool` Einträge, der Zugriffsmodus als Integerflag, ein (leeres) Exceptionattribut und das Codeattribut festgehalten. Pro Methode wird ein Codevektor erstellt. Der Codevektor enthält eine Sequenz von Opcodes, die nur innerhalb dieser Methode verwendet werden dürfen. Deshalb werden nach der Generierung einer konkreten Methode der Codevektor und alle methodenlokalen Zähler zurückgesetzt.

Auch die Hauptmethode „`main`“ wird wie jede andere Methode behandelt. `main` muss `public, static` sein, einen Array von Strings als Argument erwarten und keinen Rückgabewert haben. Die Hauptmethode eines Zwischenprogramms wird um diese Eigenschaften ergänzt, falls sie nicht der Spezifikation entsprechen.

Gemäss Definition der JVM hat jedes `class` File mindestens eine Instanzinitialisierungsmethode `<init>` (siehe auch Abschnitt 4.8). Sie steht für den Konstruktor oder die Konstruktoren im Quellcode und macht als erste Anweisung einen Aufruf einer anderen Instanzinitialisierungsmethode. Falls im Quellprogramm kein Konstruktor vorhanden ist, wird ein ein Defaultkonstruktor mit dem geschützten Namen „init“ erzeugt, der weder Parameter noch einen Rückgabewert hat, sondern nur `<init>` der Superklasse aufruft und eventuell nichtstatische Fields initialisiert. Diese Funktionalität befindet sich in der Methode `generateInit`.

Im zweiten Schritt werden auch die nötigen Attributstrukturen aufgestellt und eingefügt. Die wichtigste davon ist das `Code_Attribute`, das für jede Methode generiert wird. Dabei werden die folgenden Hilfsstrukturen verwendet:

- Der Code wird intern als Vektor bearbeitet, der Index gibt den Wert des *program counters* an, auf jedes Element kann einzeln zugegriffen werden, die Länge wächst dynamisch.
- Um die maximale Anzahl Variablen und die Anzahl Stellen auf dem Operandenstack an `max_locals` und `max_stack` zu übergeben, werden alle Operationen auf je einem Stack nachgebildet.
- Die lokalen Variablen werden zusätzlich in einer Hashtabelle verwaltet, ebenso die class Variablen oder Fields, wobei der Name jeweils als Schlüssel dient.

Es folgt eine genauere Betrachtung des Vorgehens bei der Instruktionswahl und der Generierung des Code Arrays.

Code Array

Die Wahl der Instruktionen folgt einem trivialen Schema, das zwar korrekt den ganzen AST abdeckt, aber nicht optimal effektiv ist: Für jede konkrete ABCompiler-Klasse wird ein Codeskelett aufgestellt, das zusammen mit Werten aus Feldern oder Rückgabewerten der Sohnknoten den Bytecode für dieses Konstrukt ergibt. Dieses wird in den Codevektor eingefügt, der in einem weiteren Schritt mit (abhängigen) Transformationen optimiert werden könnte. Der Bytecode wird in der Reihenfolge erzeugt, in der der konkrete AST traversiert wird.

Da die JVM Bytecode auf einem Operandenstack bearbeitet, dessen Bewegungen implizit in den Instruktionen enthalten sind, wird zuerst der Code der Sohnknoten generiert, damit sich die Operanden auf dem Stack befinden. Deren Typ ist nicht ersichtlich. Um typ-passende Instruktion zu wählen, sollten die Typen dem Compiler aber bekannt sein, deshalb haben ABCompiler-Klassen, die Ausdrücke darstellen, eine Funktion zur Angabe ihres Datentyps. In Abhängigkeit der Typen dieser Operanden wird danach eine Opcodesequenz ausgewählt, die semantisch der AST-Klasse entspricht. Resultate werden auf dem Operandenstack für die nächste Operation bereitgestellt, in lokale Variablen abgespeichert werden sie nur bei Deklarationen und Zuweisungen.

Das verwendete Prinzip des Backends ist umrissen, die Umsetzung im ABCompiler (Abbildung 7.2) lässt sich in drei Gruppen teilen, die von den Superklassen `Terminal, Statement/Expression, Declaration/Definition` abgeleitet und charakterisiert werden:

- **Terminale** checken ihren Wert im `ConstantPool` und den Hilfsstrukturen und laden ihn bei der Generierung passend auf den Stack.
- **Anweisungen** und **Ausdrücke** generieren passenden Bytecode.
- Die **Deklaration** bestimmt je nach Definition die Art des Checks. **Definitionen** sammeln die Informationen für die `class` File Strukturen und lösen die Generierung aus.

Es werden in der Folge einige Erfahrungen und Probleme bei der Codegenerierung diskutiert. Das konkrete Vorgehen wird an einem Fallbeispiel in Kapitel 8 gezeigt.

7.3.3 Implementationsdiskussion

Es folgen Erfahrungen, die mit der Handhabung des `ConstantPool` gemacht wurden, und Implementationsprobleme bei der Bytecodegenerierung.

Handhabung des `ConstantPool`

In einem ersten Ansatz wurden alle ganzen Zahlen, die im Zwischenprogramm vorkommen, in den `ConstantPool` eingefügt und mit der *load constant value* Instruktion „`ldc`“ bei Bedarf auf den Stack geladen. Die Analyse verschiedener `class` Files, die mit dem Java Compiler erzeugt wurden, hat den folgenden Algorithmus zum Speichern von Integers in Bytecode ergeben:

1. Ganze Zahlen, deren Absolutwert grösser als 255 ist, werden als `CONSTANT_Integer` im `ConstantPool` gespeichert und mit `ldc` geladen, die restlichen werden direkt im Opcode angegeben.
2. Für die am häufigsten verwendeten Integers bietet die JVM je einen eigenen Opcode an: `iconst_<i>` (*push integer constant to stack*) mit $i \in [-1, 0, 1, 2, 3, 4, 5]$.
3. Grössere Zahlen werden als ein Byte mit `bipush` (*push byte to stack*),
4. respektive als zwei Bytes mit `sipush` (*push short to stack*) auf den Stack geschoben.

Dieses Vorgehen gilt sinngemäss auch für Zahlen der Typen *byte, short* in einem Zwischenprogramm, da diese bei der Bytecodegenerierung als *integer* betrachtet werden.

Im Gegensatz zu Integers, die je nach Wert unterschiedlich behandelt werden, müssen Zahlen der Datentypen *float, double, long* grundsätzlich im `ConstantPool` abgelegt und mit `ldc` auf den Stack geladen werden, da keine speziellen Opcodes wie für Integers existieren. Eine Ausnahme sind die Konstanten 0 und 1, die in jedem Typ einen kurzen Opcode haben.

Die Speicherung als `CONSTANT` Eintrag erfolgt für *float* mit zwei Bytes, für *double* und *long* mit vier Bytes. Bei der Codegenerierung kann für die Konvertierung nach Bytes zum Beispiel `java.lang.Float.floatToIntBits(float value)` verwendet werden.

Der vorliegende Ansatz legt alle Bezeichner im `ConstantPool` ab, ohne zwischen Klassen-, Methoden-, Field- und lokalen Variablennamen zu unterscheiden. Letztere werden aber in einem `class` File nicht über den `ConstantPool` referenziert, sondern als Eintrag im Array der lokalen Variablen. Die Variablennamen im `ConstantPool` sind daher überflüssig, beeinträchtigen jedoch die Funktionalität der JVM nicht. Falls man diese Einträge im `ConstantPool` nicht wünscht, kann durch Angabe des Kontexts für Bezeichner in der Klasse `Varid.java` diese Unterscheidung durchgeführt werden.

Einige Implementationsprobleme

Jede Methode endet mit der Rückgabe eines Wertes gemäss ihrem Descriptor, also einer `return` Instruktion. Falls aber die letzte Anweisung der Methode ein Return war, ist das Defaultreturn überflüssig. In der Praxis kommen beide Fälle vor, keiner darf weggelassen werden. Lösung bietet eine Helper-Methode, die immer bei der Generierung von `return` verwendet wird. Sie merkt sich die aktuelle Methode und den Programcounter des letzten Returns. Falls dies nur einen Schritt zurückliegt, kann das Defaultreturn unterdrückt werden.

Neben `<init>`, der Instanzinitialisierungsmethode existiert eine weitere, Bytecode-interne Methode, die in den analysierten `class` Files nur selten vorkommt: `<clinit>`, die Klasseninitialisierungsmethode (Abschnitt 4.8). Sie muss nur generiert werden, falls initialisierte statische Fields im Zwischenprogramm vorkommen. Dies gilt unbedingt für Arrayfields, sonst kann auch das `ConstantValue` Attribut des Fields verwendet werden.

Bei Schleifen im Code Array lohnt es sich, wie in Abbildung 7.3 links, den Test für die Bedingung am Ende des Codes für die Schleife zu plazieren. Dadurch wird zwar zu Beginn dieses Codeabschnitts ein unbedingter Sprung „`2) goto 8`“ benötigt, um zum Test zu gelangen.

Doch bei jeder Wiederholung der Schleife wird eine Instruktion eingespart, die bei der (rechten) Version mit Anfangstest wieder hoch springt: „`14) goto 2`“. Der zusätzliche Sprung „`8) goto 17`“ aus der Schleife kann wie in Abbildung 7.3 ganz rechts durch Umkehren der Bedingung vermieden werden: „`5) if_icmpge 14`“.

```
-- end-condition ----------------------------- start-condition ---------------
 0 iconst_0      - put 0 on stack             0 iconst_0
 1 istore_1      - store 0 in t1              1 istore_1
 2 goto 8        - goto absolut label         2 iload_1
 5 iinc 1 1      - t1 = t1 + 1                3 bipush 100
 8 iload_1       - load t1 to stack           5 if_icmplt 11    5 if_icmpge 14
 9 bipush 100    - put 100 on stack           8 goto 17         8 iinc 1 1
11 if_icmplt 5   - t1 < 100 goto label       11 iinc 1 1       11 goto 2
14 return        - return, loop finished     14 goto 2         14 return
                                             17 return
```

Abbildung 7.3: Bytecode für `int i = 0; while (i < 100) { i++; };` Version mit End- und Anfangsbedingung. Die Sprungadressen sind hier ausnahmsweise absolut angegeben.

Neben dem Einsparen einer Instruktion und dem kürzeren Bytecode haben Schleifen mit Endbedingung eine weiteren Vorteil. Beim Rückwärssprung ist das Label, das angesprungen werden soll, bereits bekannt und kann konkret eingesetzt werden. Bei Vorwärtssprüngen muss mit einem Platzhalter und einer Ersetzungsmethode operiert werden. Aus diesen Gründen wurden bei der Codegenerierung Schleifen mit Endbedingungen verwendet.

7.4 Diskussion der Codegenerierung

Im vorliegenden Kapitel 7 wird das Design und das Umfeld eines Compilerbackends und die entsprechende Implementation beschrieben. Der resultierende ABCompiler erzeugt aus einem Programm in GZS Bytecode für die JVM. Damit wird aus Quellcode der imperativen Beispielsprache (2.3) korrekter Zielcode erzeugt (7.4.1). Aus Zeitgründen wurden nur diejenigen Konstrukte der GZS behandelt, die imperativ verwendet werden. Um objektorientierte Quellprogramme zu compilieren, kann der vorliegende Ansatz ohne Probleme weiterverfolgt werden, weil die „objektorientierten“ Konstrukte der GZS bereits im ABCompiler vorhanden sind. Für die logische Beispielsprache muss vor der Codegenerierung das Problem des fehlenden Interpreters gelöst werden. Die funktionale Beispielsprache verwendet einige Konstrukte, wie zum Beispiel Konstruktorlisten (3.3.4), deren Bedeutung in Bytecode noch geklärt werden muss.

Der Ansatz, der in der Problemstellung vorgeschlagen wird, nämlich die Generierung von Bytecode aus einer generellen Zwischensprache, kann wie hier exemplarisch gezeigt wird, in die Praxis umgesetzt werden. Leider reichte die Zeit nicht für die vollständige Behandlung der GZS , da der prinzipielle Aufbau des ABCompiler aufwendig war und einige Probleme bei der Zusammenarbeit mit der JVM aufwarf. Für zukünftige Arbeiten ist die Struktur und Manipulation von `class` Files mit dem ClassFile-Analyzer gewährleistet.

Dieses Kapitel schliesst mit einigen Anmerkungen zur Verifikation von `class` Files, den Freiheiten, die die JVM bei der Codegenerierung gewährt, der Klassifizierung von Bytecode und dem Vorgehen bei der Implementation des Backends.

7.4.1 Verifikation von class Files

Die Implementation der JVM von SUN testet beim Laden alle `class` Files auf die statischen und strukturellen Beschränkungen. Es werden nur diejenigen `class` Files ausgeführt, die alle Bedingungen erfüllen. Das heisst, ein `class` File, das von der JVM von SUN ausgeführt wird, ist gültig und korrekt gemäss der Spezifikation. Diese Verifikationsmethode wird zur Kontrolle der mit dem ABCOMPILER generierten `class` Files verwendet.

Falls die Verifikation nicht erfolgreich verläuft, liegt irgendwo ein Fehler vor; leider gibt die JVM keine Fehlermeldungen aus. Daher folgt hier eine Aufzählung von potentiellen Fehlerquellen bei der Bytecodegenerierung:

- Der Klassenname des `class` Files darf nicht direkt in `this_class` mit dem Eintrag des Namens abgelegt werden. Korrekt ist eine Klassenreferenz in `this_class`.
- Die angegebenen Werte für `max_stack, max_local` müssen mit der Anzahl wirklich verwendeter Werte übereinstimmen. Durch Verwendung einer Subklasse von `java.util.Stack`, die den Zähler und seinen Maximalwert zentral verwaltet, konnte die Codegenerierung davon befreit werden. Es ist zu beachten, dass bei einem Methodenaufruf `this` übergeben wird, womit auch Methoden ohne Parameter bereits zu Beginn eine lokale Variable besitzen.
- Werte der Typen *double* und *long* brauchen immer **doppelte** Einträge, sowohl im `ConstantPool`, als auch auf dem Stack und im Array der lokalen Variablen.
- Die Spezifikation der JVM [Lin97] erwähnt nur am Rande, dass die Klasseninitialisierungsmethode `<clinit>` (4.8) vom Compiler zur Verfügung gestellt werden muss, falls statische initialisierte Fields im Quellcode vorkommen.
- Klassennamen werden intern mit „/“ anstelle des sonst in Java üblichen Punktes getrennt.

Trotz strengen Bedingungen akzeptiert die JVM einiges:

Die Reihenfolge von gleichen Strukturen ist unwichtig, nur der Aufbau jeder Struktur muss genaustens befolgt werden. So spielt es keine Rolle, welche Methoden, Fields (und Interfaces) zuerst generiert und in das `class` File eingefügt werden. Auch der `ConstantPool` kann nach Belieben gefüllt oder geordnet werden. Es dürfen Einträge vorkommen, die in den restlichen Strukturen und im Opcode keine Verwendung haben.

Der Name des Quellprogramms im `SourceFile` Attribut ist unwichtig, ebenso das Attribut selbst. Derjenige von `this_class` muss korrekt sein.

Auch die Attribute `LineNumberTable, LocalVariableTable` sind optional, wie in der Spezifikation angegeben. Das Attribut `Exceptions` dagegen muss nicht für jede Methode vorhanden sein, nur falls tatsächlich *Exceptions* aufgeworfen werden.

7.4.2 Vorgehen bei der Codegenerierung

Beim ABCOMPILER wird nicht wie bei Aho et al [Aho86] oder Appel [App97] mit Registern und einem Speicherbereich gearbeitet, sondern nur mit einer grossen Anzahl lokaler Variablen, die als Stellvertreter der Variablen im Quellcode betrachtet werden können. Die Themen „Registerallocation“ und „Activation Records“ haben, dank der JVM, keine grosse Relevanz im vorliegenden Compilerbackend. Da die Übergabe der Frames und damit auch der lokalen Variablen bei einem Methodenaufruf von der JVM übernommen wird, sind mit dem vorliegenden

Ansatz immer alle Variablen während der ganzen Methode gültig, also weder *caller-save* noch *callee-save*.

Auch die *Garbage Collection* muss nicht vom Compiler vorgenommen werden, der Heap wird von der JVM bei Bedarf geräumt.

Implementation des Parser/Builders mit funktionalem Programmierstil

Bei der Implementation des Parser/Builder wurde Appels Ratschlag befolgt, einen funktionalen Programmierstil zu verwenden [App97]. Das heisst, ohne Seiteneffekte durch Zuweisungen, die Variablen und Datenstrukturen verändern, nur mit Zuweisung bei der Initialisierung von Datenstrukturen. Jede Klasse hat einen Konstruktor, der diese Initialisierungen der Fields vornimmt. Danach kann *instanceof* zum Bestimmen der Subklasse (=Funktor) eines zu untersuchenden Objekts von einem generellen Typ verwendet werden, worauf dieses Object auf die Subklasse ge*cast*ed wird (*static dispatch* mit *typecase* nach Abadi und Cardelli [Aba96]).

7.4.3 Future Work

Die durch die generelle Zwischensprache vorgegebenen Klassen konnten aus Zeitgründen nicht vollständig implementiert werden. Einige Punkte, die beim Weiterführen dieser Arbeit von Interesse sind:

- In der 2. Phase würden von der Idee her auch Interfaces und Exceptions bearbeitet, weil beide Konzepte in der generellen Zwischensprache nicht verwendet werden, sind sie im vorliegenden Backend nicht implementiert. Im Hinblick auf ein möglichst generelles Backend sind Interfaces und Exceptions sicher sinnvoll; beide Konzepte werden von der JVM unterstützt.
- Im Moment wird keine Kurzfassung des Zugriffs auf lokale Variablen verwendet, weil das immer mit einem Typtest verbunden wäre. Falls der Compiler möglichst kompakten Bytecode generieren soll, würde der Testaufwand durch den Opcode ohne Operanden entschädigt.
- Der Opcode `iinc` könnte für die Inkrementation einer Variablen verwendet werden, dadurch liessen sich 2 Instruktionen einsparen: `iinc x 1` anstelle von `get x, get1, iadd`.
- Der Einsatz einer einfachen Peephole-Optimierung zur Elimination von redundanten Instruktionen ist lohnend, wie Vergleiche von selber generiertem Code mit Java Bytecode gezeigt haben. Auch weitere Optimierungen sind vorstellbar, wie in Abschnitt 6.4 erwähnt wurde.
- Falls ein Zwischenprogramm explizite Konstruktoren hat (zum Beispiel mit der objektorientierten Beispielsprache) müssen diese auf `<init>` abgebildet werden. Dafür sollte man die Behandlung von `<init>` in einer Methode lokalisieren.

Kapitel 8

Beispiel

> Lang ist der Weg durch Lehren (der Theorie), kurz und erfolgreich durch Beispiele.
>
> *Seneca*

Dieses Kapitel zeigt an einem Fallbeispiel Schritt für Schritt auf, wie Bytecode generiert wird. Die Zustände und Umformungen eines imperativen Programmes werden für jeden praktischen Teil dieser Arbeit beschrieben. Besonderes Gewicht wird auf das Backend von Kapitel 7 gelegt.

8.1 Input

Das Beispiel beginnt mit einem einfachen Programm in einer imperativen, Pascal-ähnlichen Sprache. Das Programm von Abbildung 8.1 berechnet mit Hilfe zweier Prozeduren die Potenzreihe einer reellen Zahl und legt die Resultate in einem Array ab. Das File, das den Programmtext enthält entspricht der Grammatikdefinition von Abschnitt 2.3.3, welche die Beispielsprache für das imperative Paradigma festlegt. Es ist wichtig, dass die Eingabe semantisch korrekt ist, da dies in keiner Phase getestet wird.

Die Nummerierung der Zeilen wird von der Grammatik nicht akzeptiert, hier aber zum Erklären eingefügt. Die auskommentierten Anweisungen auf Zeile 5, 20, 32 und 42 müssen in den Programmtext einbezogen werden, um ein sinnvolles Resultat zu erhalten. Sie sind ausgeklammert, weil die imperative Grammatik keine Zuweisungen von Anweisungen, sondern nur von Ausdrücken kennt. Dies ist ein Schwachpunkt des imperativen Frontends. Die fehlenden Konstrukte werden von Hand in den AST eingetragen, das heisst, man kann die auskommentierten Anweisungen in die Compilation miteinbeziehen.

8.2 Frontend

Da hier ein imperatives Programm betrachtet wird, dient `impgensyn` (A.2 ist die nichtgeneralisierte Version!), das Gentleprogramm für die imperative Beispielsprache, als Frontend. Die Eingabe wird geparst, also eingelesen und analysiert. Die einzelnen Schritte, die Gentle dabei vornimmt, sind in Abschnitt 2.2 beschrieben.

Der Input ist syntaktisch und semantisch korrekt[1], ist daher den Regeln der Grammatik nach gültig. Als semantische Aktion erzeugt `impgensyn` eine Zwischenprogrammrepräsentation

[1] Falls der Input nicht der Grammatik entspricht, wird das Programm abgebrochen und das Token als fehlerhaft gemeldet, das nicht in den bisher eingelesenen Text passt.

```
 1 PROGRAM powerToArray ;
 2
 3 DECLARE
 4   result : REAL;
 5   test : INTEGER; (* := 10;  *)
 6   aField : ARRAY [0..10] OF REAL;
 7
 8   PROCEDURE xToN ( VAR base : REAL; VAR exponent : INTEGER );
 9
10   DECLARE
11     result : REAL;
12
13     BEGIN
14       result := 1;
15       WHILE exponent > 0 DO
16         {
17          result := result * base;
18          exponent := exponent - 1
19         };
20       WRITE(result) (* RETURN result *)
21   END;
22
23   PROCEDURE fillArray ;
24
25   DECLARE
26     i : INTEGER;
27
28   BEGIN
29     FOR i := 0 TO 10 STEP 1 DO
30       {
31        xToN(2.5, i);
32        aField[i] := xToN; (* aField[i] := xToN( ); *)
33        WRITE(aField[i])
34       }
35   END;
36
37 BEGIN (* main program *)
38
39   IF test <= 0 THEN
40     {
41      xToN(2.5, 4);
42      result := xToN; (* result := xToN( ); *)
43      WRITE(result)
44     }
45     ELSE
46       {
47        fillArray
48       }
49   ENDIF
50
51 END.
```

Abbildung 8.1: Quellcode des imperativen Beispielprogramms

des Quellprogramms. Dieses Zwischenprogramm nach ist ein abstrakter Syntaxbaum im Textformat, einen Ausschnitt davon zeigt Abbildung 8.2. Da die abstrakte Syntax der imperativen Grammatik auf die generelle abstrakte Syntax abgebildet wurde, haben wir einen AST in GZS vor uns, wie sie in Kapitel 3 beschrieben ist.

Der Befehl, der diesen Schritt auf einem Unixterminal ausführt, lautet:

```
impgensyn powerToArray > powerToArray.gs
```

Exemplarisch für die Abbildung von konkretem Inputtext nach der korrespondierenden abstrakten Syntax wird die Bearbeitung der *While*-Anweisung (Zeilen 15 - 19) in der Prozedur *xToN* durch `impgensyn` betrachtet. Nach dem Parsen des Hauptprogramms und seiner Deklarationen wird das Token „`WHILE`“ vom Lexer geliefert, darauf trifft (nur) die folgende Regel der imperativen Beispielgrammatik (generalisierte Version) zu:

```
'nonterm' Statement(-> STATEMENT)                        -- while loop
   'rule' Statement(-> loop(P, nil, E, stmtseq(S, nil))):
          "WHILE" @(-> P) Expression(-> E)
          "DO" Statement(-> S)
```

Die Schlüsselwörter werden eliminiert und nach erfolgreichem Parsen des Ausdrucks und der Anweisung wird die semantische Aktion nach „−>“ aktiviert. Diese wählt den Funktor `loop` vom Typ `STATEMENT` aus und instanziiert ihn mit den Werten seiner Teilausdrücke. So wird sukzessive ein Element der abstrakten Syntax konstruiert, das aus Funktoren und Terminals besteht. Der gesamte Output ergibt einen relativ grossen AST, daher wird hier nur der Teilbaum für die While-Anweisung in Abbildung 8.2 explizit dargestellt. Der Funktor `loop` beginnt auf der ersten Zeile und endet auf der letzten; dazwischen befinden sich seine Argumente.

Damit ein AST wie in Abbildung 8.2 entsteht, ist ein kleiner manueller Zwischenschritt nötig. Die Namen der Bezeichner sind im Zwischenprogramm als Pointer auf eine Tabelle festgehalten und müssen von Hand durch die im Eingabeprogramm verwendeten Namen ersetzt werden.

Anmerkung: *Eventuell könnte man das Gentleprogramm umstellen oder die Bezeichner bleiben in dieser unleserlichen Form, was die Ausführung durch die JVM nicht stört. Was sicher umgeformt werden muss, sind Strings, Real-Zahlen und* `main`.

An dieser Stelle müssen ebenfalls die folgenden Konstrukte in das Zwischenprogramm eingefügt werden, die in der imperativen Grammatik nicht vorkommen, für die Codeerzeugung aber notwendig sind. (Beim Erstellen der Grammatik waren die spezifischen Bedingungen der JVM noch nicht bekannt, in einer nächsten Version des Frontends kann man diese Punkte problemlos einbeziehen.)

- Die Zugriffsflags der Methoden und der Hauptroutine werden von `nil` oder `public` auf `static` gesetzt, da die imperative Beispielgrammatik nur statisch funktioniert.
- Die Prozeduren erhalten einen Rückgabetyp in dem dafür vorgesehenen Feld, dieser müsste beim Parsen von den Funktionsargumenten getrennt werden.
- Die auskommentierten Zuweisungen von Anweisungen werden von Hand eingefügt.

8.3 Zwischensprache

Das Beispielprogramm liegt nun in GZS als Textfile vor und muss für die Codeerzeugung in Instanzen der ABCompiler-Klassen umgeformt werden, da dies vom Frontend nicht gemacht

```
stmtseq(loop(1000019005,                              -- position
             nil,                                     -- preparation statement
             relation(1000019020,                     -- condition = relation expression
                termexp(desig(1000019011,              -- terminal expression
                   varid(<<exponent>>))),              -- variable identifier
                desig(1000019020,                      -- terminal
                   relop(gt)),                         -- relation operator
                termexp(desig(1000019022,              -- terminal expression
                   number(0)))),                       -- integer number
             stmtseq(compound(stmtseq(                -- loop body = statementsequence
                assignment(1000021007,                 -- assignment statement
                   termexp(desig(1000021007,           -- terminal expression
                      varid(<<result>>))),             -- variable identifier
                   mult(1000021024,                    -- multiplication expression
                      termexp(desig(1000021017,        -- terminal expression
                         varid(<<result>>))),          -- variable identifier
                      termexp(desig(1000021026,        -- terminal expression
                         varid(<<base>>))))),          -- variable identifier
                stmtseq(                               -- statement sequence
                   assignment(1000022007,              -- assignment
                      termexp(desig(1000022007,        -- terminal expression
                         varid(<<exponent>>))),        -- variable identifier
                      minus(1000022028,                -- subtraction expression
                         termexp(desig(1000022019,     -- terminal expression
                            varid(<<exponent>>))),     -- variable identifier
                         termexp(desig(1000022030,     -- terminal expression
                            number(1))))),             -- integer number
                   nil))),                             -- nil statement
             nil)), ...                                -- nil statement, end of loop
```

Abbildung 8.2: Abstrakter Syntaxbaum für die While-Schleife.

wird. Deshalb folgt wieder ein Parsingschritt, diesmal mit dem *JavaCC* Parser/Builder (7.2).

Der textuelle AST wird nach den Regeln einer Grammatik für die generelle abstrakte Syntax eingelesen, anschliessend instanziiert der Parser/Builder (→ Visitorpattern) die zum Eingabeprogramm passenden Klassen des ABCompiler, indem er deren Konstruktoren mit den nötigen (und gültigen) Elementen aufruft. Diese Elemente sind wiederum AST-Klassen oder Literale. Der AST in des Beispielprogramms resultiert daraus in Java-Klassen-Form. Ausgeführt wird dieser Schritt mit der folgenden Eingabe am Terminal, wobei `powerToArray.gs` die umgeformte Zwischenprogrammdarstellung ist: `java Parser powerToArray.gs`

```
void Statement() #Statement : {}  { Assignment()  | Ifthenelse()  | Loop()  }

void Loop() #Loop : {}
  {
    "loop" "(" Pos() "," Statement() "," Expression() "," Statementlist() ")"
  }
```

Abbildung 8.3: Loop Regeln der Grammatik für die generelle abstrakte Syntax in Parser.jjt.

Die While-Anweisung wird in der GZS mit dem Funktor `loop` dargestellt (Abb. 8.2), dar-

auf passt die Regel des PARSER/BUILDER in Abbildung 8.3. An der Stelle eines `Statements` kann ein Nonterminal für eine Anweisung stehen: `Assignment()`, `Ifthenelse()` oder `Loop()`. Die Methode für das Nonterminal `Loop` parst die Token in den Anführungszeichen und erwartet Rückgabewerte von `Pos(), Statement(), Expression(), Statementlist()`. Danach instanziiert sie einen Knoten, dessen Namen sich aus „`AST`“ und dem Namen nach „#“, also „Loop“, zusammensetzt und den geparsten Funktor `loop` repräsentiert: `ASTLoop.java`, Abbildung 8.4.

`ASTLoop.java` ist eine von *JJTree* automatisch generierte Klasse, die gemäss der Hierarchie der generellen abstrakten Syntax angepasst wurde, damit sie als Untertyp von `(AST)Statement` erkannt wird. Die zwei Konstruktoren werden zur internen Verknüpfung verwendet, die interessante Methode ist `jjAccept`, die den *Visitor* akzeptiert.

```
/* Generated By:JJTree: Do not edit this line. ASTLoop.java */

public class ASTLoop extends  ASTStatement {

  public ASTLoop(int id) { super(id); }

  public ASTLoop(Parser p, int id) { super(p, id); }

  /** Accept the visitor. **/
  public Object jjtAccept(ParserVisitor visitor, Object data) {
    return visitor.visit(this, data);
  }
}
```

Abbildung 8.4: ASTLoop.java

Die Klasse `Builder.java`, die die Visitorschnittstelle implementiert, hat für jedes Nonterminal der Grammatik in `Parser.jjt` eine `visit`-Methode. Diejenige für `ASTLoop`, die die While-Anweisung repräsentiert, ist in Abbildung 8.5 dargestellt. Beim Travesieren des Loop-Knotens wird für jeden Unterknoten explizit `visit` aufgerufen; `node.getChildNode(i)` liefert den i-ten Unterknoten des aktuellen Knotens, jedoch vom Typ `SimpleNode`, daher muss der korrekte Typ gecastet werden. Dies geht problemlos, da die Typen der Unterknoten gleich wie in der GZS aufgebaut sind. Der „Besuch“ bei `ASTLoop` resultiert in einer neuen Instanz von `Loop.java`, der Zwischenprogrammdarstellung auf Klassenebene, die die Codegenerierung enthält.

```
public Object visit(ASTLoop node, Object data)
  {
    Integer p        = (Integer) visit((ASTPos)node.getChildNode(0), null);
    Statement s      = (Statement) visit((ASTStatement)node.getChildNode(1), null);
    Expression e     = (Expression) visit((ASTExpression)node.getChildNode(2), null);
    Statementlist ss = (Statementlist) visit((ASTStatementlist)node.getChildNode(3), null);
    return new Loop(p, s, e, ss); // return new instance
  }
```

Abbildung 8.5: Methode `visit` für `ASTLoop`-Knoten. In `Builder.java`

Die vollständige Instanziierung der While-Anweisung ist in Abbildung 8.6 aufgeführt. Diese

Sequenz von Objektkreationen ergibt sich aus der eben beschriebenen Traversierung der AST-Knoten mit dem Visitor.

Die While-Anweisung wird zu einem Objekt der Klasse `Loop.java`, das keine Vorbereitungsanweisung hat, eine Bedingung und einen Body. Die Bedingung setzt sich aus einem terminalen Ausdruck, einem Relationsoperator und wieder einem terminalen Ausdruck zusammen. Der erstere ist ein Bezeichner, wird daher mit einem `Varid`-Objekt dargestellt, das mit dem String „exponent“ initialisiert ist. Der letztere ist ein Objekt vom Typ `Number` mit dem Wert „0“. Auf diese Art und Weise wird der AST nachgebildet, die Funktoren werden zu gleichnamigen Klassen, die terminalen Elemente zu Integers, Floats, Characters oder Strings mit spezieller Bedeutung.

```
new Statementlist(
    new Loop(                                                    // while
        null,
        new Relation(
            new Termexp(
                new Desig(
                    new Varid("exponent"))),                     // exponent
            new Desig(
                new Relop(">")),                                 // >
            new Termexp(
                new Desig(
                    new Number( 0 )))),                          // 0
        new Statementlist(
            new Compound(
                new Statementlist(
                    new Assignment(
                        new Termexp(
                            new Desig(
                                new Varid("result"))),
                        new Mult(                                // *
                            new Termexp(
                                new Desig(
                                    new Varid("result"))),       // result
                            new Termexp(
                                new Desig(
                                    new Varid("base"))))),       // base
                    new Statementlist(
                        new Assignment(
                            new Termexp(
                                new Desig(
                                    new Varid("exponent"))),
                            new Minus(                           // -
                                new Termexp(
                                    new Desig(
                                        new Varid("exponent"))), // exponent
                                new Termexp(
                                    new Desig(
                                        new Number( 1 ))))),     // 1
                        null))),
        null)), ...
```

Abbildung 8.6: AST für die While-Schleife als Java Instanzen im ABCompiler.

8.4 Backend

Aus den Instanzen der ABCompiler-Klassen wird nun ein `class` File für das ursprüngliche Inputprogramm generiert. Die Klassen werden nach dem Designprinzip (7.3) durchlaufen, indem jede Klasse ihre Kinder oder Elemente aufruft. Abbildung 8.7 skizziert die Hauptroutine des Parser/Builders, die das ganze Compilerbackend steuert. Zuerst werden mit `check()` statische Informationen im `ClassFile`, insbesondere im `ConstantPool` gesetzt. Danach werden mit `generate()` die abhängigen Informationen nachgeliefert und der Bytecode generiert. Die Variable `theAST`, die das Zwischenprogramm enthält, wird dafür mit `check()` und `generate()` aufgerufen. „`theAST`“ ist in diesem Beispiel der konkrete ABCompiler.

```
PARSER_BEGIN(Parser)
class Parser {

  public static void main(String args[]) {

    Parser theParser = new Parser(inStream);

    try {
      ASTPrgr top = theParser.Prgr();                  // toplevel of input prog
      Builder b = new Builder();                       // create visitor
      ClassFile classfile = new ClassFile();           // the ClassFile

      // start & accept visitor, returns instantiated AST
      AbstractSyntax theAST = (AbstractSyntax) top.jjtAccept(b, null);

      theAST.check(constantPool);                      // check theAST, fill ConstPool
      theAST.generate(classfile);                      // generate rest of classFile
      classfile.writeCF();                             // write to standardout
      classfile.flushCF(outfile);                      // binary output
      }  catch (Exception e) {
         System.err.println(e);
         }
  }
}
PARSER_END(Parser)
```

Abbildung 8.7: Hauptroutine des Parser/Builder.

8.4.1 Check der statischen Informationen

Anhand des Kontrollflusses durch das konkrete Zwischenprogramm (Abb. 8.8) wird im folgenden die `check`-Phase des ABCompiler detailliert betrachtet. Die *depth-first* Kontrolle ist im ersten Teilbaum durch gestrichelte Kontrollpfeile explizit angegeben. Tabelle 8.1 zeigt den Aufbau des `ConstantPool`; Abbildung 8.11 zeigt den resultierenden `ConstantPool` für das `class` File des Beispielprogramms.

Die folgenden Punkte beziehen sich auf die Schritte in Tabelle 8.1. Die Nummerierung in der folgenden Aufzählung und der Aktionstabelle ist identisch zu der Nummerierung in der Kontrollflussabbildung 8.8.

1. Der Bezeichner mit dem Namen des Beispielprogramms „PowerToArray“ wird als `Utf8`-String in den `ConstantPool` eingetragen. Die Klassendeklaration macht mit `C_Class` eine

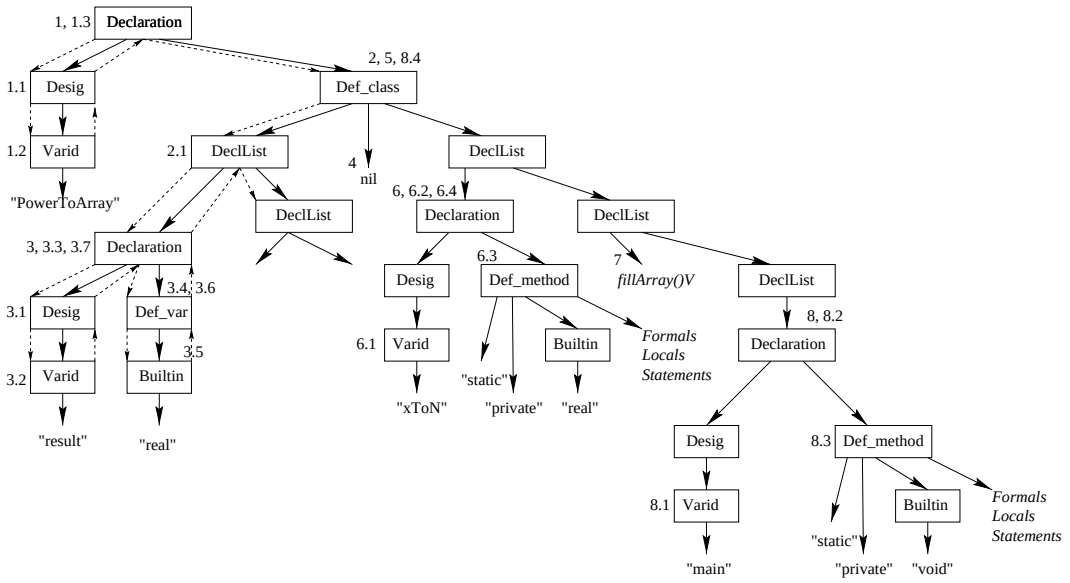

Abbildung 8.8: Kontrollfluss durch die konkrete Zwischenprogrammdarstellung.

Referenz auf diesen String und merkt sich ihn sich als Name der laufenden Klasse.

2. Die Klassendefinition checkt die class Variablen „result“, „test“ und „aField“ mit der speziellen Methode `checkField` als Fields. Da das imperative Beispielprogramm nicht instantiiert werden kann, sind Fields `private` und `static`.

3. Für „result“ wird der Bezeichner als `CONSTANT_Utf8` in den `ConstantPool` eingetragen, der Deskriptor des Typs abgeleitet und eine `CONSTANT_Field`-Referenz mit Indices auf die aktuelle Klasse, den Namen und den Typ generiert (Schritt 3.7).

4. Da das Beispielprogramm keine Superklasse kennt, wird die Standardsuperklasse der Java Klassenhierarchie verwendet: `java/lang/Object`

5. Mit Datentypkonstruktoren wird die Instanzinitialisierungsmethode `<init>` generiert, gecheckt und mit einer Methodenreferenz in den `ConstantPool` gefügt. Anschliessend wird der dritte Parameter der Klassendefinition gecheckt, eine Deklarationsliste, die die Methoden der Klasse enthält.

6. Die Prozedur „xToN“ erzeugt eine Methodenreferenz.

7. Die Prozedur „fillArray“ erzeugt ebenfalls eine Methodenreferenz.

8. Die Methode „main“ wird in einer speziellen Prozedur gecheckt. Sie muss `public` und `static` sein, hat keinen Rückgabewert. Als formaler Parameter wird ein String Array definiert, das Zwischenprogramm darf keine eigenen Formals verlangen. Nach dem Check der lokalen Variablen und dem Anweisungsblock von „main“ ist die erste Phase beendet.

Der `ConstantPool`-Zähler wird mit jedem Eintrag inkrementiert und bleibt auf dem Höchstwert stehen, im vorliegenden Fall ist `Constant_Pool_Count = 61`.

Nr	Klasse	Aktion	Eintrag im ConstantPool
1	Declaration	check Terminal → Desig	
1.1	Desig	check DesigType → Varid	
1.2	Varid	cp.addUtf8("PowerToArray")	1. Utf8 = "PowerToArray"
1.3	Declaration	cp.addClass("PowerToArray")	2. ClassRef = 1
		check Definition → Def_class	
2	Def_class	check 1. DeclList als Fields	
2.1	Declarationlist	checkField für alle Elemente	
3	Declaration	check Terminal → Desig	
3.1	Desig	check DesigType → Varid	
3.2	Varid	cp.addUtf8("result")	3. Utf8 = "result"
3.3	Declaration	check Definition → Def_variable	
3.4	Def_variable	check Type → Builtin	
3.5	Builtin	construct typeID ⇒ 6 (real)	
3.6	Def_variable	construct descriptor ⇒ F	
3.7	Declaration	cp.addField("result", "PtA", "F")	4. Utf8 = "F"
			5. NameAndType = 3, 4
			6. FieldRef = 2, 5
4	Def_class	keine explizite Superklasse, daher	16. Utf8 = "java/lang/Object"
		cp.addClass("java/lang/Object")	17. ClassRef = 16
5	Def_class	new <init>()V	18. Utf8 = "<init>"
		cp.addMethod("<init>",	19. Utf8 = "()V"
		"java/lang/Object", "()V")	20. NameAndType = 18, 19
			21. MethodRef = 17, 20
		check 2. DeclList (Methoden)	
6	Declaration	check Terminal → Desig → Varid	
6.1	Varid	cp.addUtf8("xToN")	22. Utf8 = "xToN"
6.2	Declaration	check Definition → Def_method	
6.3	Def_method	static + private ⇒ 0x0010	
		construct descriptor ⇒ (FI)F	
		check formals, locals, statements	23. Utf8 = "base"
	Declarationlist	analog zu 3 - 3.6	24. Utf8 = "exponent"
6.4	Declaration	cp.addMethod("xToN", "PtA",	25. Utf8 = "(FI)F"
		"(FI)F")	26. NameAndType = 22, 25
			27. MethodRef = 2, 26
7	Declaration	"fillArray" analog zu 6 - 6.4	47. MethodRef = 2, 46
8	Declaration	check Terminal → Desig → Varid	
8.1	Varid	cp.addUtf8("main")	48. Utf8 = "main"
8.2	Declaration	spezielles check Main	
8.3	Def_method	public + static ⇒ 0x0009	
		descriptor ⇒ ([Ljava/lang/String;)V	
		check locals, statements	
8.4	Def_class	check der Methoden beendet	

Tabelle 8.1: Aktionen in der `check`-Phase der Codegeneration. Notation: „→ “ leitet einen konkreten Subtyp einer abstrakten Klasse ab. „⇒ “ gibt das Resultat einer Berechnung an. Die Zahlen im `ConstantPool` referenzieren Pool-Einträge.

8.4.2 Generation von abhängigen Informationen und Bytecode

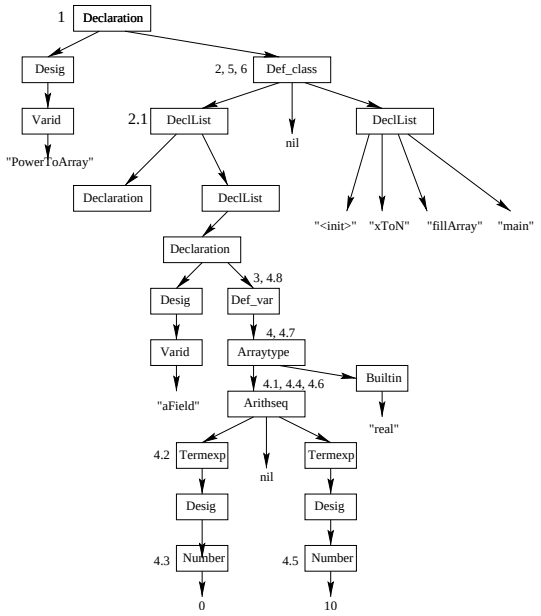

Abbildung 8.9: Kontrollfluss für die `generate`-Phase.

Die Traversierung des ABCompiler in der `generate`-Phase erzeugt den Kontrollfluss in Abbildung 8.9. Tabelle 8.2 beschreibt zusammen mit der folgenden Aufzählung den Aufbau der Strukturen im `ClassFile` für unser Beispielprogramm.

1. Die oberste Deklaration setzt im `ClassFile` die Variable `this_class` auf die `C_Class`-Referenz, die in der `check`-Phase eruiert wurde.

2. Die Klassendefinition setzt `super_class` und `access_flags` im `ClassFile` und generiert ihre class Variablen mit der speziellen Methode `generateField`.

3. Für jedes Field wird eine `field`-Struktur in `ClassFile` eingetragen. Für „aField“ setzt sich das Zugriffsflag aus `private` und `static` zusammen; der Name und der Descriptor werden als Referenzen auf `ConstantPool`-Einträge übergeben. In der Hashtabelle für Fields wird nebst diesen Werten auch die Klassenreferenz und der Arraytyp (*float*) abgelegt. Da es sich bei „aField“ um ein statisches, initialisiertes Field handelt, muss die Klasseninitialisierungsmethode „`<clinit>`“ generiert werden: `static <clinit>()V`

4. Der Arraytyp wird generiert, das heisst, die Generierungsmethode wird bis zum terminalen Element weitergereicht. Hier wird zum ersten Mal in diesem Beispiel Code generiert: `Number` setzt die Werte als Byte oder als Konstante auf den Operanden Stack, danach geht der Kontrollfluss wieder zurück zu `Arithseq`, welches die beiden Zahlen subtrahiert. Das Resultat wird als Grösse des Arrays verwendet und auf dem Operanden Stack gelassen. Dort wird es von `Arraytype` um den Typ ergänzt: ein neues Array mit 10 *float*-Elementen wird geschaffen. Nun beendet `Def_variable` die `<clinit>`-Methode, das Array wird als statisches Field abgelegt und ein `return` ohne Rückgabewert produziert. Die Maximalwerte für den Operanden Stack und die lokalen Variablen werden zusammen

mit dem Code Array als `Code_attribute` in `clinit` abgelegt, anschliessend werden die Stacks und der Codevektor zurückgesetzt.

5. Die intern erzeugte Spezialmethode „`<init>`“ wird analog zur in Schritt 4 behandelten Methode mit einer `method`-Struktur im `ClassFile` gesetzt. Die Methoden „xToN“, „fillArray“ und „main“ werden ebenfalls nach dem gleichen Schema gesetzt; der erzeugte Bytecode ist im Anhang A.6.2 zu finden.

6. Die Klassendefinition setzt das `SourceFile` Attribut mit dem Hinweis, dass das vorliegende `class` File aus der generellen Zwischensprache GZS compiliert wurde.

Nr	Klasse	Aktion	Eintrag im ClassFile
1	Declaration	cf.thisClass.set() generate Definition → Def_class	this_class = 2
2	Def_class	cf.superClass.set() cf.accessFlags.set() generate 1.DeclList als Fields	super_class = 17 access_flags = PUBLIC | SUPER
2.1	DeclList	checkField für alle Elemente	
3	Def_variable	cf.setField(0x10, 12, 13) fieldHash.setAttributes() cp.addUtf8("<clinit>") cf.setMethod(0x8, 56, 19) generate Type → Arraytype	PRIVATE | STATIC, "aField", "[F" 56. Utf8 = "<clinit>" STATIC, "<clinit>", "()V"
4	Arraytype	generate Expression → Aseq	
4.1	Arithseq	generate 3.Exp → Termexp	
4.2	Termexp	gen Terminal → Desig → Number	
4.3	Number	10 < 128 ⇒ bipush 10	CodeVec = [16, 10]
4.4	Arithseq	generate 1.Exp → → Number	
4.5	Number	0 < 5 ⇒ iconst_0	CodeVec = [16, 10, 3]
4.6	Arithseq	⇒ isub	CodeVec = [16, 10, 3, 100]
4.7	Arraytype	⇒ newarray FLOAT	CodeVec = [16, 10, 3, 100, 188, 6]
4.8	Def_variable	⇒ putstatic FieldRef ⇒ return VOID cf.methods.setCode(opStack.max(), locStack.max())	CodeVec = [..., 188, 6, 179, 0, 15] CodeVec = [..., 179, 0, 15, 177] max_stack = 2, max_locals = 1 code_length = 10
5	Def_class	cf.setMethod(0x1, 18, 19) cf.setMethod(0x10, 22, 25) cf.setMethod(0x10, 28, 19) cf.setMethod(0x9, 48, 58)	PUBLIC, "<init>", "()V" PRIVATE | STATIC, "xToN", "(FI)F" PRIVATE | STATIC, "fillArray", "()V" PUBLIC | STATIC, "main", "([LString;)V"
6	Def_class	cf.setSourceFile()	59. Utf8 = "SourceFile" "generated from GZS"

Tabelle 8.2: `generate`-Phase: Klassenstruktur und `<clinit>`

8.4.3 Ausgabe des binären class Files

Zum Schluss werden (in Abbildung 8.10) die drei Werte gesetzt, die in jedem `class` File identisch sind: die Versionsnummern und die `magis`chen ersten vier Bytes (`0xcafebabe`). Der ClassFile-Analyzer schreibt alle generierten Strukturen im Binärformat in „`PowerToArray.class`“, das `class` File, das das compilierte Beispielprogramm beinhaltet. Eine vollständige Textversion des soeben generierten Zielprogrammes findet man im Anhang A.6.2.

```
Magic: cafebabe
Minor Version: 3
Major Version: 45
Constant_Pool:
Access Flags: ACC_PUBLIC | ACC_SUPER
This  class: index = 2, class = "PowerToArray"
Super class: index = 17, class = "java/lang/Object"
0 Interface(s):
Fields count: 3
  0. Field: ACC_PRIVATE | ACC_STATIC
       Name: "result", Descriptor: "F", Attributes count: 0
  1. Field: ACC_PRIVATE | ACC_STATIC
       Name: "test", Descriptor: "I", Attributes count: 1
       Attribute : "ConstantValue", length = 2
                "Int 10"
  2. Field: ACC_PRIVATE | ACC_STATIC
       Name: "aField", Descriptor: "[F", Attributes count: 0
Methods count: 5
  0. Method: ACC_STATIC
       Name: "<clinit>", Descriptor: "()V", attrCount = 1
       Attribute : "Code", length = 22
       Max stack: 2, Max locals: 1, Code length: 10
       0 : 16  bipush        10        byte value
       2 : 3   iconst_0
       3 : 100 isub
       4 : 188 newarray       type 6
       6 : 179 putstatic     #015      Field  PowerToArray/aField:[F
       9 : 177 return
       Exception Table length: 0
       Code Attributes count: 0
  1. Method: ACC_PUBLIC
       Name: "<init>", Descriptor: "()V", attrCount = 1
       Attribute : "Code", length = 17
       Max stack: 1, Max locals: 1, Code length: 5
       0 : 42  aload_0
       1 : 183 invokespecial #021      Method java/lang/Object/<init>:()V
       4 : 177 return
       Exception Table length: 0
       Code Attributes count: 0
    ...
Class Attributes count: 1
       Attribute : "SourceFile", length = 2
 --> "generated from general IR"
```

Abbildung 8.10: Fields und Initialisierungsmethoden des `ClassFile`.

```
Magic: cafebabe
Minor Version: 3
Major Version: 45
Constant_Pool:
 Constant_Pool_Count: 61
  1: C_Utf8: length = 12, bytes = "PowerToArray"
  2: C_Class: name_index = 1
  3: C_Utf8: length = 6, bytes = "result"
  4: C_Utf8: length = 1, bytes = "F"
  5: C_NameAndType: name_index = 3, descriptor_index = 4
  6: C_Fieldref: class_index = 2, nameAndType_index = 5
  7: C_Utf8: length = 4, bytes = "test"
  8: C_Integer: bytes = 10
  9: C_Utf8: length = 1, bytes = "I"
  10: C_NameAndType: name_index = 7, descriptor_index = 9
  11: C_Fieldref: class_index = 2, nameAndType_index = 10
  12: C_Utf8: length = 6, bytes = "aField"
  13: C_Utf8: length = 2, bytes = "[F"
  14: C_NameAndType: name_index = 12, descriptor_index = 13
  15: C_Fieldref: class_index = 2, nameAndType_index = 14
  16: C_Utf8: length = 16, bytes = "java/lang/Object"
  17: C_Class: name_index = 16
  18: C_Utf8: length = 6, bytes = "<init>"
  19: C_Utf8: length = 3, bytes = "()V"
  20: C_NameAndType: name_index = 18, descriptor_index = 19
  21: C_Methodref: class_index = 17, nameAndType_index = 20
  22: C_Utf8: length = 4, bytes = "xToN"
  23: C_Utf8: length = 4, bytes = "base"
  24: C_Utf8: length = 8, bytes = "exponent"
  25: C_Utf8: length = 5, bytes = "(FI)F"
  26: C_NameAndType: name_index = 22, descriptor_index = 25
  27: C_Methodref: class_index = 2, nameAndType_index = 26
  28: C_Utf8: length = 9, bytes = "fillArray"
  29: C_Utf8: length = 1, bytes = "i"
  30: C_Float: bytes = 1075838976, value = 2.5
  31: C_Utf8: length = 16, bytes = "java/lang/System"
  32: C_Class: name_index = 31
  33: C_Utf8: length = 3, bytes = "out"
  34: C_Utf8: length = 21, bytes = "Ljava/io/PrintStream;"
  35: C_NameAndType: name_index = 33, descriptor_index = 34
  36: C_Fieldref: class_index = 32, nameAndType_index = 35
  37: C_Utf8: length = 19, bytes = "java/io/PrintStream"
  38: C_Class: name_index = 37
  39: C_Utf8: length = 7, bytes = "println"
  40: C_Utf8: length = 4, bytes = "(I)V"
  41: C_NameAndType: name_index = 39, descriptor_index = 40
  42: C_Methodref: class_index = 38, nameAndType_index = 41
  43: C_Utf8: length = 4, bytes = "(F)V"
  44: C_NameAndType: name_index = 39, descriptor_index = 43
  45: C_Methodref: class_index = 38, nameAndType_index = 44
  46: C_NameAndType: name_index = 28, descriptor_index = 19
  47: C_Methodref: class_index = 2, nameAndType_index = 46
  48: C_Utf8: length = 4, bytes = "main"
  49: C_NameAndType: name_index = 33, descriptor_index = 34
  50: C_Fieldref: class_index = 32, nameAndType_index = 49
  51: C_NameAndType: name_index = 39, descriptor_index = 40
  52: C_Methodref: class_index = 38, nameAndType_index = 51
  53: C_NameAndType: name_index = 39, descriptor_index = 43
  54: C_Methodref: class_index = 38, nameAndType_index = 53
  55: C_Utf8: length = 13, bytes = "ConstantValue"
  56: C_Utf8: length = 8, bytes = "<clinit>"
  57: C_Utf8: length = 4, bytes = "Code"
  58: C_Utf8: length = 22, bytes = "([Ljava/lang/String;)V"
  59: C_Utf8: length = 10, bytes = "SourceFile"
  60: C_Utf8: length = 13, bytes = "generated from general IR"
```

Abbildung 8.11: ConstantPool des `class` Files für das Beispielprogramm.

8.5 Output - Bytecode

Abbildung 8.10 zeigt die Fields und Initialisierungsmethoden, Abbildung 8.11 den vollständigen `ConstantPool` des Beispiels. Eine Abbildung des ganzen `ClassFile` findet sich im Anhang A.6.2.

Die Ausführung von „`PowerToArray.class`“ auf der JVM verläuft mit der *-verify* Option problemlos, die Resultate werden korrekt berechnet, wie man in Abbildung 8.12 sehen kann.

```
% java -verify PowerToArray
1.0
2.5
6.25
15.625
39.0625
97.65625
244.14062
610.35156
1525.8789
3814.6973
%
```

Abbildung 8.12: Ausführung von PowerToArray.class mit der JVM.

Kapitel 9

Schlussbemerkungen

Jucundi acti labores. (Angenehm sind die erledigten Arbeiten.)

Cicero, de finibus

In dieser Diplomarbeit wird ein Ansatz für ein Compilerframework für die virtuelle Maschine von Java beschrieben. Im letzten Kapitel wird Rückschau gehalten und Bilanz gezogen. Die erwarteten Resultate werden mit den erreichten Zielen verglichen, der Ansatz und offene Probleme diskutiert. Schliesslich werden noch einige Punkte aufgelistet, die bei weiterführenden Arbeiten von Interesse sein könnten.

9.1 Diskussion

Im einleitenden Kapitel 1 wurde in Abbildung 1.2 ein Compilerframework vorgeschlagen. Das resultierende Compilerframework dieser Arbeit ist in Abbildung 9.1 dargestellt und wird im folgenden nach Kapiteln geordnet diskutiert.

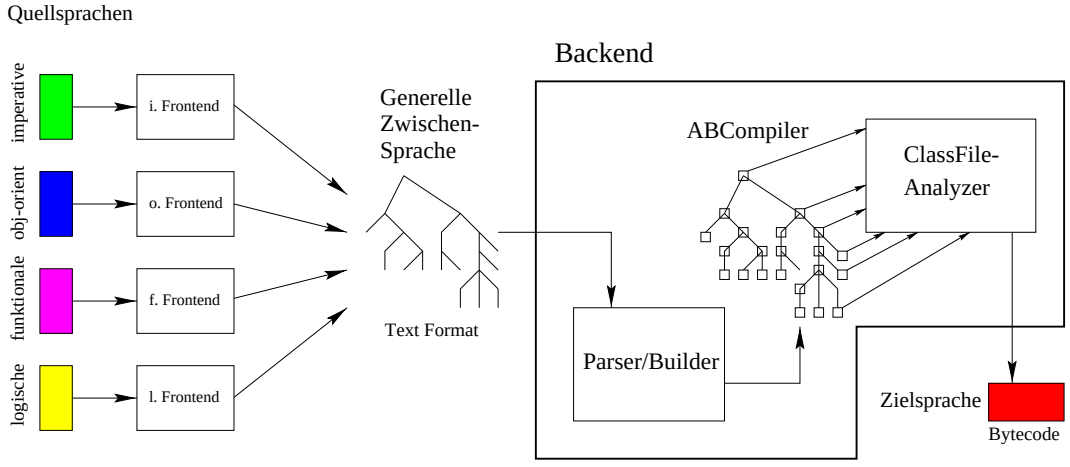

Abbildung 9.1: Resultierendes Compilerframework.

Kapitel 2: Programmiersprachen

In dieser Arbeit wurde eine Analyse verschiedener Programmiersprachen vorgenommen. Für die untersuchten Paradigmen wird jeweils das Konzept charakterisiert, die Grammatik und ein passender Parser einer Beispielsprache implementiert und die resultierende abstrakte Syntax beschrieben. Die imperative Sprache (2.3) entstand aus Pascal [Wir75] und Minilax [Wai89], die objektorientierte (2.4) baut auf der imperativen auf und erweitert sie mit objektorientierten Konstrukten, die von CooL und Java [Gos96] stammen. Die funktionale Beispielsprache (2.5) ist eine Teilmenge von Gofer [Jon91], die logische (2.6) eine Teilmenge von Prolog [Sic96].

Da in dieser Arbeit aus Zeit- und Komplexitätsgründen Beispielsprachen verwendet werden, musste man diese aus Teilmengen einer oder mehrerer bestehender Sprachen definieren. Dabei wäre eine weitere Phase oder ein iterativer Prozess zum Überarbeiten und Anpassen der Beispielsprachen an Erkenntnisse aus der Analyse oder Codegenerierung wünschenswert gewesen. Die in einem ersten Schritt gewählten Konstrukte reichten teilweise zu weit oder waren gar zu eng gefasst. So ist zum Beispiel die Syntax von Gofer recht umfangreich und komplex, eine sinnvolle Teilmenge zu finden war nicht trivial. Eine Alternative dazu wäre PureFun-Tiger, eine funktionale Sprache, die von Appel eingeführt wird [App97]. Die Syntax von Minilax dagegen ist zwar als Beispielsprache im Compilerbau geeignet, aber zum Erstellen von Programmen nur beschränkt einsetzbar, wie das konkrete Beispiel in Kapitel 8 gezeigt hat.

Nach dem Aufstellen der Grammatiken konnten Parser und abstrakte Syntax ohne nennenswerte Schwierigkeiten abgeleitet werden.

Hier ist eine Verschiebung des Arbeitsgewichtes anzumerken: Entgegen der ursprünglichen Problemstellung, in der vor allem das Backend betont wurde, befasste sich ein grosser Teil der praktischen Arbeit mit den Frontends und Parsingproblemen.

Kapitel 3: Generelle Zwischensprache

Die Konstrukte, die in jeder der resultierenden abstrakten Grammatiken vorkommen, bilden die Grundlage der generellen Zwischensprache GZS (Kapitel 3). Durch Generalisieren, Aufteilen oder Umbenennen konnte ein grosser Teil der restlichen Konstrukte auf die GZS abgebildet werden. Im Prinzip lassen sich das imperative, das objektorientierte und das funktionale Paradigma vereinen. Bei logischen Programmen fehlt der Interpreter im Hintergrund, um sie prozedural abzuwickeln.

Ein Nebeneffekt der Abbildungen ist eine Vereinfachung der Grammatiken. Unnötige oder komplizierte Konstrukte wurden weggelassen. Das Zusammenspiel zwischen Frontend und Zwischensprache liefert für beide Teile nützliche Informationen, die iterativ einbaut werden können. Das gilt auch für die Zusammenarbeit mit dem Backend. Dieses diktierte einige Änderungen an die GZS.

Kapitel 4: Die virtuelle Maschine von Java

Der Aufbau und das Konzept der virtuellen Maschine von Java wird in Kapitel 4 dargestellt, besonderes Gewicht wird auf das `class` File Format gelegt. Die JVM ist die Zielmaschine des vorliegenden Compilerframeworks, ihre Spezifikation von Lindholm und Yellin [Lin97] legt den Grundstock für den ClassFile-Analyzer (Kapitel 5) und den ABCompiler (7.3). Dadurch wird die Plattformunabhänigkeit der JVM auf diese Arbeit übertragen.

Zu Beginn dieser Diplomarbeit war die JVM für den Autor eine Blackbox; mehrmaliges Studium der Spezifikation, unzählige Analysen von `class` Files, die Implementation der beiden Bytecode-erzeugenden Tools und das Erstellen dieser Arbeitsdokumentation, die sich immer

wieder auf die JVM bezieht, hat dem Autor zum Durchblick [und dem Buch zu Kaffeeflecken und Eselsohren] verholfen.

Kapitel 5: ClassFile-Analyse-Tool

In Kapitel 5 wird das Design und die Implementation des ClassFile-Analyzers beschrieben. Dieses Tool wurde so aufgebaut, dass jede Klasse eine Struktur des `class` Files repräsentiert und die darin enthaltenen Elemente manipulieren kann. Dieses Design liess sich gut objektorientiert mit Java umsetzten. Der ClassFile-Analyzer wird zum Ablegen und Schreiben der Elemente eingesetzt, die vom ABCompiler generiert werden.

Kapitel 6: Einführung Compilerbau

In Kapitel 6 werden die Grundlagen des Compilerbaus aufgearbeitet und konzeptionelle Phasen eingeführt (Abbildung 6.1), dadurch wird die Abbildung 1.1 von Seite 2 unterteilt und die Aufgaben von Frontend, Backend und Zwischendarstellungen festgelegt. Die in Kapitel 6 definierten Begriffe werden in der ganzen Arbeit verwendet. Die Techniken zur Zwischenprogramm- und Codeoptimierung (6.4) konnten aus Zeitgründen nicht in die Praxis umgesetzt werden.

Kapitel 7: Codegenerierung

Vor der eigentlichen Codegenerierung wird in Kapitel 7 die Wahl von Bytecode als Zielsprache und von Java als Implementationssprache begründet.

Ein Zwischenprogramm in GZS wird als Textfile an das Compilerbackend übergeben und als Baum von Datenkonstruktoren zu Bytecode verarbeitet. Das Parsen der GZS, die Initialisierung der korrespondierenden Klassen und die Ausführung des Backends wird vom Parser/Builder (7.2) vorgenommen. Dieses Tool wurde mit *JavaCC* und *JJTree* implementiert, dadurch konnte nach der Konstruktion der paradigmaspezifischen Frontends mit Gentle eine weitere Compilerbausprache kennengelernt werden. Das Aufstellen und Parsen der Grammatik der GZS verlief problemlos, die durch JavaCC generierten Klassen bilden einen gewissen Overhead gegenüber dem Parserbau mit Gentle. Durch Anwendung des Visitorpatterns kann die Funktionalität zur Konstruktion der AST-Klassen in einer Klasse gebündelt werden.

Die Codegenerierung mit dem ABCompiler (7.3) findet in den AST-Klassen statt, jede Klasse stellt ein Konstrukt der GZS dar, dessen Bedeutung sie nach Bytecode überträgt. Dafür wurde in jeder Klasse eine Methode `check` (7.3.1) zum Setzen des `ConstantPool` und eine Methode `generate` (7.3.2) zum Generieren des restlichen `class` Files implementiert. Die generierten Einträge werden im ClassFile-Analyzer abgelegt; seine Strukturen falls nötig ergänzt. So reicht zum Beispiel die Symboltabelle des `class` Files, der `ConstantPool`, für die Codegenerierung nicht ganz aus, es müssen zusätzliche Attribute im ABCompiler abgelegt werden.

Die Übersetzung der GZS nach Bytecode verlief nach der Einpegelung auf das Assemblerniveau ohne grosse Schwierigkeiten. Ein Zwischenprogramm in GZS erfüllt aber nicht alle Anforderungen, die die JVM an ein `class` File stellt, deshalb muss es nicht nur umgeformt, sondern auch um Konstrukte ergänzt werden, die für Bytecode nötig sind. So muss eine Initialisierungsmethode vorhanden sein, Fields werden anders behandelt als lokale Variablen, Instruktionen werden nach Typen unterschieden und arbeiten nur mit typenkorrekten Operanden.

Die Restriktionen, die die JVM an `class` Files stellt, und ihre strenge Überprüfung davon dient als Verifikation des Bytecodes, der mit dem vorliegenden Compilerframework erzeugt wird. Bei der Bytecodegenerierung kann man folgende Themen ausser Acht lassen, da sie von

der JVM übernommen werden: Speicherverwaltung, Framehandling und Parameterübergabe bei Methodenaufrufen, Laden und Linken von externen (Super-)Klassen. Die Registerzuweisung und eine Lebendigkeitsanalyse erübrigen sich, weil das Array für lokale Variablen 2^{16} Einträge erlaubt. Rückblickend lässt sich sagen, dass die Verwendung von Bytecode als Zielsprache und der JVM als Zielmaschine eine gute Wahl war.

Kapitel 8: Beispiel

In Kapitel 8 wird die Anwendung der implementierten Tools und ihr Zusammenspiel an einem Beispielprogramm aufgezeigt (siehe auch Abbildung 9.1). Der Quellcode ist ein imperatives Programm, das vom imperativen Frontend geparst und in GZS ausgegeben wird. Anschliessend wird es vom Parser/Builder geparst und im ABCompiler aufgebaut. Das Backend traversiert den AST und setzt die entsprechenden Strukturen im ClassFile-Analyzer. Schliesslich schreibt der ClassFile-Analyzer den binären Bytecode in ein `class` File, das von der JVM ausgeführt wird.

9.2 Erreichte Ziele

Es folgt ein Vergleich der in der Problemstellung erwarteten Resultate (1.4) mit den erreichten Zielen.

- Die Wahl der zu analysierenden Sprachen fiel auf eine imperative, eine objektorientierte, eine funktionale und eine logische Programmiersprache. Durch die Wahl von vier (Teil-)Sprachen aus vier Paradigmen steht eine breite Palette von Sprachkonstrukten zur Verfügung, um daraus eine allgemeine Zwischensprache zu definieren. Die Implementation von je einem Frontend für jede Beispielsprache ermöglicht die Abstraktion von ihren grammatikalischen Eigenschaften. Auf dieser Stufe zeigen sich gemeinsame Elemente, auf die, nach geeigneten Umformungen, jede Sprache abgebildet werden kann. Es sind vor allem terminale Bezeichner, Zahlen, Ausdrücke und einige Anweisungen, die in jeder Sprache vorkommen. Andere Elemente sind sehr sprachspezifisch, wieder andere werden zwar gleich bezeichnet, haben aber verschiedene Bedeutungen. Dies betrifft vor allem Deklarationen und Anweisungen.

- Als Zwischensprache haben sich abstrakte Syntaxbäume bewährt, da sie, wie oben erwähnt, von der Grammatik einer Sprache losgelöst und auch vom Backend unabhängig sind. Die Wahl der Zwischensprachart war aber kein Problem in dieser Arbeit, eher ihre Definition und die Abbildung der paradigmaspezifischen Konstrukte. Prinzipiell lassen sich das imperative, das objektorientierte und das funktionale Paradigma vereinigen, beim logischen besteht das erwähnte Problem des fehlenden Interpreters.

- Die JVM und ihr `class` File Format bringen die geforderte Plattformunabhängigkeit in dieses Compilerframework. Ihre strengen Bedingungen müssen bei der Codegenerierung eingehalten werden, dienen aber auch als Garantie für korrekten Bytecode.

- Das in der Problemstellung vorgeschlagene Compilerframework (Abbildung 1.2) konnte in die Praxis umgesetzt werden. Abbildung 9.1 zeigt den resultierenden Aufbau der Komponenten, der sich kaum von dem in Kapitel 1 unterscheidet. Dank der Unterteilung von Frontend, Zwischensprache und Backend kann der Compiler auch mit einem beliebigen Frontend für eine hier nicht behandelte Quellsprache verwendet werden. Die aufgestellte Zwischensprache GZS ist umfangreich und beinhaltet Konstrukte aus den vier Paradigmen; falls trotzdem zusätzliche Konstrukte nötig sind, müssen diese nicht nur

in der GZS sondern auch im Backend hinzugefügt und implementiert werden. Die imperative Beispielsprache wurde bereits von Front- bis Backend in das Compilerframework eingebunden. Die komplette Behandlung aller vier Beispielsprachen sollte die Elemente der GZS validieren und einen konsistenten Zustand erzeugen.

- Der allgemeine Ansatz dieser Arbeit konnte realisiert werden, es entstand ein Compilerframework für die JVM. Dieses besteht aus einem Frontend pro Quellsprache, der Aufbau von Grammatiken, Parsern und semantischen Aktionen wurde vorgeführt. Weiter wird eine generelle Zwischensprache GZS konstruiert und die Beispielsprachen darauf abgebildet. Die Schnittstelle des Backends ist der Parser/Builder, der Zwischenprogramme in Instanzen des ABCompiler umwandelt, welcher sie nach Bytecode compiliert. Die Manipulation von `class` Files geschieht mit dem ClassFile-Analyzer.

  Das resultierende Compilerframework ist dank der JVM plattformunabhängig und bietet mit der GZS einen weitgehend sprachunabhängigen Ansatz.

9.3 Offene Probleme

Dieser Abschnitt führt Punkte auf, die nicht oder nur am Rande behandelt wurden. Dies ist einerseits auf den zeitlich begrenzten Rahmen der Arbeit und das komplexe Gebiet zurückzuführen, andererseits traten unerwartete Probleme auf, deren Lösung hier nicht angegangen werden konnte.

- Ansatz zur Abbildung einer logischen Sprache auf die GZS.
- Validierung der GZS mit den anderen Beispielsprachen.
- Optimierung von Zwischensprache und Bytecode.

9.4 Ausblick

Die in dieser Arbeit erreichten Resultate sind ermutigend für die Verwendung von Bytecode als Zielsprache und lassen sich überall anwenden, wo eine JVM vorhanden ist. Die Arbeit zeigt aber auch Schwachpunkte des Ansatzes und problematische Stellen auf; es war nicht zu erwarten, dass sich im ersten Durchgang alles lösen lässt.

Um die vorliegende Arbeit zu bewältigen, hat sich der Autor mit der JVM, verschiedenen Programmiersprachen und dem Compilerbau beschäftigt, sein Wissen darin aufgefrischt und vertieft. Er hat Erfahrungen mit Java, Gentle, Unix und Latex gesammelt, verschiedene Disassembler getestet und OO-Prinzipien und Pattern angewendet. In der Hoffnung, diese Erkenntnisse seien auch für andere Interessierte in den behandelten Gebieten nützlich, sind sie so umfassend wie möglich festgehalten worden. Natürlich wurden weitere Themen behandelt, die nicht weniger interessant waren, aber nichts zum Verständnis des Vorliegenden beitragen. Deshalb wurde einiges weggelassen. Für Fragen oder um unklare Punkte zu klären, steht der Autor jederzeit gerne zur Verfügung.

Anhang A

A.1 Gentle Syntax

Gentle Syntax Repräsentation aus „Gentle, Language Reference Manual“ [Sch97].

```
Module = ["'module'" Ident] [UseClause] [ExportClause] Declarations .
UseClause = "'use'" Ident{ "," Ident} .
ExportClause = "'export'" Ident{ "," Ident} .
Declarations = {Declaration} .
Declaration = TypeDecl | PredicateDecl | VariableDecl | TableDecl | RootDef .
TypeDecl = "'type'" Ident [["="] TermSpec{ [","] TermSpec}] .
TermSpec = IdentLC ["("[ParamSpec{ "," ParamSpec}]")"] .
ParamSpec = [Ident ":"] Ident .
VariableDecl = "'var'" Ident ":" Ident .
TableDecl = "'table'" Ident "(" Ident ":" Ident{ "," Ident ":" Ident} ")" .
PredicateDeclaration = Category Ident Signature Rules .
Category = "'nonterm'" | "'token'" | "'action'"
       | "'condition'" | "'choice'" | "'sweep'" .
Signature =
        ["(" [ParamSpec{ "," ParamSpec}]
        ["->"[ParamSpec{ "," ParamSpec}]]")"] .
Rules = {Head ":" Body "."} | {"'rule'" Head [":"] Body ["."]} .
Head = Ident ["(" [Pattern{ "," Pattern}]
        ["->" [Expression{ "," Expression}]] ")"] .
Pattern = IdentUC [":" Pattern]
       | Functor ["("[Pattern{ "," Pattern}]")"]
       | "_" .
Expression = Expr2 | Expression ( "+" | "-" ) Expr2 .
Expr2 = Expr3 | Expr2 ( "*" | "/" ) Expr3 .
Expr3 = IdentUC | Number | String
       | Functor ["("[Expression{ "," Expression}]")"]
       | ( "+" | "-" ) Expr3 | "(" Expression ")"
Functor = [Ident "'"] IdentLC .
Body = {Member} [CostSpec]
CostSpec = "$$" Number .
Member = Ident ["(" [Expression{ "," Expression}]
        ["->" [Pattern{ "," Pattern}]] ")"]
       | ContextDesignator "<-" Expression
       | ContextDesignator "->" Pattern
       | IdentUC "::" Ident
       | String
       | "(|" {Member} "||" {Member} {"||" {Member}} "|)"
       | "[|" {Member} "|]" .
ContextDesignator = [IdentUC "'"] Ident .
RootDef = "'root'" {Member} .
```

A.2 Parser für die imperative Grammatik

```
---------
-- ROOT
---------
'root' Program(-> P) print(P)  -- print abstract syntax tree
-------------------------------------------------------------------------------
-- TERMINALS ------------------------------------------------------------------
-------------------------------------------------------------------------------
'type' DECLARATION
       declare( IDENT, DEFINITION ).

'type' DECLLIST
       decllist( DECLARATION, DECLLIST ), nil

'type' DEFINITION
       variable( TYPE ),
       valueparam( TYPE ),
       varparam( TYPE ),
       proc( DECLLIST, DECLLIST, STMTSEQ )

'type' TYPE
       integer, real, char, boolean, none,
       array( INT, INT, TYPE )

'type' STMTSEQ
       stmt( STATEMENT ),
       seq( STATEMENT, STMTSEQ )

'type' STATEMENT
       assignment( VARIABLE, EXPRESSION ),
       ifelse( EXPRESSION, STATEMENT, STATEMENT ),
       while( EXPRESSION, STATEMENT ),
       repeat( STATEMENT, EXPRESSION ),
       for( VARIABLE, EXPRESSION, EXPRESSION, EXPRESSION, STATEMENT ),
       compound( STMTSEQ ),
       call( IDENT, EXPRLIST ),
       read( VARIABLE ),
       write( EXPRESSION ),
       nil

'type' EXPRLIST
       exprlist( EXPRESSION, EXPRLIST ), nil

'type' EXPRESSION
       rel( RELOP, EXPRESSION, EXPRESSION ),
       and( EXPRESSION, EXPRESSION ),
       or( EXPRESSION, EXPRESSION ),
       not( EXPRESSION ),
       mult( EXPRESSION, EXPRESSION ),
       div( EXPRESSION, EXPRESSION ),
       intdiv( EXPRESSION, EXPRESSION ),
       mod( EXPRESSION, EXPRESSION ),
       plus( EXPRESSION, EXPRESSION ),
       minus( EXPRESSION, EXPRESSION ),
       neg( EXPRESSION ),
       num( INT ),
       double( DOUBLE ),
       var( VARIABLE ),
       true, false

'type' LOGOP
```

```
       and, or, not

'type' RELOP
       eq, ne, lt, le, gt, ge

'type' VARIABLE
       id( IDENT ),
       array( VARIABLE, EXPRESSION )

'type' IDENT
'type' DOUBLE

-------------------------------------------------------------------------
-- NONTERMINALS ---------------------------------------------------------
-------------------------------------------------------------------------
------------------------
-- Program Declarations ------------------------------------------------
------------------------

'nonterm' Program(-> DECLARATION)  -- top level program
   'rule' Program(-> declare(I, proc(nil, Ds, S))) :
    "PROGRAM" Ident(-> I) ";" "DECLARE" DeclList(-> Ds)
  "BEGIN" StmtSeq(-> S) "END" "."

'nonterm' Declaration(-> DECLARATION)  -- declarations
   'rule' Declaration(-> declare(I, variable(T))) : -- variable
  Ident(-> I) ":" Type(-> T)
   'rule' Declaration(-> declare(I, proc(Fs, Ds, S))) : -- procedure
       "PROCEDURE" Ident(-> I)  FormalPart(-> Fs) ";"
         "DECLARE" DeclList(-> Ds) "BEGIN" StmtSeq(-> S) "END"

'nonterm' DeclList(-> DECLLIST) -- several declarations
   'rule' DeclList(-> decllist(D, Ds)) :
  Declaration(-> D) ";" DeclList(-> Ds)
   'rule' DeclList(-> decllist(D, nil)) :
  Declaration(-> D)

'nonterm' FormalPart(-> DECLLIST) -- formal part
   'rule' FormalPart(-> nil) :  -- nothing
   'rule' FormalPart(-> Fs) :  -- parameter list
  "(" FormalList(-> Fs) ")"

'nonterm' FormalList(-> DECLLIST)  -- several formals
   'rule' FormalList(-> decllist(F, nil)) :
  Formal(-> F)
   'rule' FormalList(-> decllist(F, Fs)) :
  Formal(-> F) ";" FormalList(-> Fs)

'nonterm' Formal(-> DECLARATION)  -- formal parameters
   'rule' Formal(-> declare(I, varparam(T))) :  -- variables
  "VAR" Ident(-> I) ":" Type(-> T)
   'rule' Formal(-> declare(I, valueparam(T))) :  -- (return) values
  Ident(-> I) ":" Type(-> T)

'nonterm' Type(-> TYPE) -- types
   'rule' Type(-> integer) : "INTEGER" -- integer
   'rule' Type(-> real) : "REAL"  -- real
   'rule' Type(-> boolean) : "BOOLEAN"  -- boolean
   'rule' Type(-> array(Lwb, Upb, T)) :  -- array
  "ARRAY" "[" Number(-> Lwb) ".."
  Number(-> Upb) "]" "OF" Type(-> T)
```

```
--------------
-- Statements ------------------------------------------------------------
--------------

'nonterm' StmtSeq(-> STMTSEQ) -- several statements
   'rule' StmtSeq(-> stmt(S)): Statement(-> S)
   'rule' StmtSeq(-> seq(S, SS)): Statement(-> S) ";"  StmtSeq(-> SS)

'nonterm' Statement(-> STATEMENT) -- statements
   'rule' Statement(-> assignment(V, E)):  -- assignement
  Variable(-> V) ":=" Expression(-> E)
   'rule' Statement(-> ifelse(E, S1, S2)):  -- if then else
  "IF" Expression(-> E) "THEN" Statement(-> S1) Else(-> S2)
   'rule' Statement(-> while(E, S)):  -- while do
  "WHILE" Expression(-> E) "DO" Statement(-> S)
   'rule' Statement(-> repeat(S, E)):  -- repeat until
  "REPEAT" Statement(-> S) "UNTIL" Expression(-> E)
   'rule' Statement(-> compound(S)):  -- compound stmt
  "{" StmtSeq(-> S) "}"
   'rule' Statement(-> for(V, E, L, I, S)) :  -- for-loop
          "FOR" Variable(-> V) ":=" Expr3(-> E)
  "TO" Expr3(-> L) "STEP" Expr3(-> I) "DO" Statement(-> S)
   'rule' Statement(-> call(I, Es)) :  -- procedure call
  Ident(-> I) "(" ExprList(-> Es) ")"
   'rule' Statement(-> call(I, nil)) :  -- without parameters
  Ident(-> I)
   'rule' Statement(-> read(V)):  -- read-in
  "READ" "(" Variable(-> V) ")"
   'rule' Statement(-> write(E)):  -- write-out
  "WRITE" "(" Expression(-> E) ")"

'nonterm' Else(-> STATEMENT) -- else-parts
   'rule' Else(-> S2):  -- else
  "ELSE" Statement(-> S2) "ENDIF"
   'rule' Else(-> nil):  -- no else
  "ENDIF"
-- NOTE: avoid if-then-else-conflict by closing stmt with keyword "ENDIF"

---------------
-- Expressions -----------------------------------------------------------
---------------

'nonterm' ExprList(-> EXPRLIST)  -- several expressions
   'rule' ExprList(-> exprlist(E, nil)) :
          Expression(-> E)
   'rule' ExprList(-> exprlist(E, Es)) :
  Expression(-> E) "," ExprList(-> Es)

'nonterm' Expression(-> EXPRESSION) -- expressions
   'rule' Expression(-> X): Expr1(-> X)
   'rule' Expression(-> rel(Op, X,Y)):  -- relative operations
  Expression(-> X)  Relop(-> Op) Expr1(-> Y)
   'rule' Expression(-> and(X,Y)):  -- logical and
  Expression(-> X) "AND" Expr1(-> Y)
   'rule' Expression(-> or(X,Y)):  -- logical or
  Expression(-> X) "OR" Expr1(-> Y)
   'rule' Expression(-> not(X)):  -- logical not
  "NOT" Expr1(-> X)

'nonterm' Expr1(-> EXPRESSION)
   'rule' Expr1(-> X): Expr2(-> X)
   'rule' Expr1(-> plus(X,Y)):  -- addition
```

```
  Expr1(-> X) "+" Expr2(-> Y)
   'rule' Expr1(-> minus(X,Y)):  -- subtraction
  Expr1(-> X) "-" Expr2(-> Y)

'nonterm' Expr2(-> EXPRESSION)
   'rule' Expr2(-> X): Expr3(-> X)
   'rule' Expr2(-> mult(X,Y)):  -- multiplication
  Expr2(-> X) "*" Expr3(-> Y)
   'rule' Expr2(-> div(X,Y)):  -- division
  Expr2(-> X) "/" Expr3(-> Y)
   'rule' Expr2(-> mod(X,Y)):  -- modulo
    Expr2(-> X) "MOD" Expr3(-> Y)
   'rule' Expr2(-> intdiv(X,Y)):  -- integer division
  Expr2(-> X) "DIV" Expr3(-> Y)

'nonterm' Expr3(-> EXPRESSION) -- atomic expressions
   'rule' Expr3(-> num(X)):  -- number = integer
  Number(-> X)
   'rule' Expr3(-> double(X)):  -- double = real, float
  Double(-> X)
   'rule' Expr3(-> var(X)):  -- variables
  Variable(-> X)
   'rule' Expr3(-> neg(X)):  -- unary negation
  "-" Expr3(-> X)
   'rule' Expr3(-> X):  -- unary positive
  "+" Expr3(-> X)
   'rule' Expr3(-> X):  -- parentheses
  "(" Expression(-> X) ")"

'nonterm' Relop(-> RELOP) -- relative operators
   'rule' Relop(-> eq): "=" -- equal
   'rule' Relop(-> ne): "!=" -- not equal
   'rule' Relop(-> lt): "<" -- less than
   'rule' Relop(-> le): "<=" -- less or equal
   'rule' Relop(-> gt): ">" -- greater than
   'rule' Relop(-> ge): ">=" -- greater or equal

'nonterm' Variable(-> VARIABLE) -- variables
   'rule' Variable(-> id(I)):  -- identifiers
  Ident(-> I)
   'rule' Variable(-> array(V, E)):  -- arrays
          Variable(-> V) "[" Expression(-> E) "]"

-------------------------------------------------------------------------------
-- TOKENS -----------------------------------------------------------------------
-------------------------------------------------------------------------------
'token' Number(-> INT)
'token' Ident(-> IDENT)
'token' Double(-> DOUBLE)
```

A.3 Konkrete Grammatiken der Paradigmasprachen

Die imperative Grammatik wird hier nicht nochmals aufgeführt. Man findet sie im Abschnitt 2.3.3 und im obigen Parser A.2.

A.3.1 Grammatik der objektorientierten Paradigmasprache

```
<Class> ::= {{ <Class> }}                          -- several classes
          | CLASS Ident [[ EXTENDS Ident ]] {
            <InstVarDecl> <Method> }               -- class declaration
```

```
<Method> ::= <Method> {{ <Method> }}                  -- several methods
           | [[ ABSTRACT ]] [[ <Scope> ]]
             [[ <Type> ]] Ident
             ( [[ <FormalParam> ]] )  {
             <InstVarDecl> <Statement> }              -- classmethod decl

<FormalParam> ::= <FormalParam> {{, <FormalParam> }}  -- formal parameters
                | Ident : <Type>                      -- value parameter

<Scope> ::= PUBLIC
          | PRIVATE
          | PROTECTED
          | PRIVATE PROTECTED

<InstVarDecl> ::= {{ <InstVarDecl> }}                 -- several declarations
                | VAR Ident : <Type> ;                -- variable declaration
                | CONST <Type> Ident = <Expression> ; -- constant declaration
<Statement> ::= {{ <Statement> }}                     -- several statements
              | <Expression> = <Expression> ;         -- assignment
              | { <Statement> } ;                     -- compound statement
              | Ident [[ ( [[ <ExprList> ]] ) ]] ;    -- call statement
              | IF <Expression> THEN <Statement>
                [[ ELSE <Statement> ]] ENDIF ;        -- selection
              | WHILE <Expression> DO <Statement> ;   -- while loop
              | REPEAT <Statement>
                UNTIL <Expression> ;                  -- repeat loop
              | FOR <Variable> := <Expression>
                TO <Expression> STEP <Expression>
                DO <Statement> ;                      -- for loop
              | RETURN <Expression> ;                 -- return value
              | <Type> <Variable> =
                NEW <Type> ( [[ <ExprList> ]] ) ;     -- create new object
              | SWITCH ( <Expression> ) {
                {{ CASE <Label> : <Statement> }}
                [[ DEFAULT : <Statement> ]] } ;       -- switch selection

<Label> ::= <Label> {{, <Label> }}
          | <Expression> [[ .. <Expression> ]]

<ExprList> ::= <Expression> {{, <Expression>}}        -- several expressions

<Expression> ::= String                               -- string
               | Character                            -- character
               | <Expression> . Ident
                 ( [[ <ExprList> ]] )                 -- method call

<Variable> ::= Ident                                  -- variable
             | <Variable> [ <Expression> ]            -- array

<Type> ::= VOID                                       -- void
         | INTEGER                                    -- integer
         | REAL                                       -- real
         | CHAR                                       -- character
         | BOOLEAN                                    -- boolean
         | STRING                                     -- string
         | Ident                                      -- object type
         | ARRAY [ <Expression> .. <Expression> ]
           OF <Type>                                  -- array
```

A.3.2 Grammatik der funktionalen Paradigmasprache

```
<Module>      ::= { <Topdeclaration> }               -- top-level module
<Interpreter> ::= <Expr> [[ <Where> ]]              -- top-level expression

-- DECLARATIONS

<Topdeclaration> ::= {{ <Topdeclaration> ; }}        -- several declarations
            | DATA Conid <Varlist> = <Constructor>   -- userdefined datatype
            | TYPE Conid <Varlist> = <Type>          -- synonym type
            | INFIXL [[ Integer ]] <Oplist>          -- fixity left
            | INFIXR [[ Integer ]] <Oplist>          -- fixity right
            | INFIX  [[ Integer ]] <Oplist>          -- fixity nonassociative
            | CLASS [[ <Context> => ]]
              <Pred> [[ <Where> ]]                   -- type class
            | INSTANCE [[ <Context> => ]]
              <Pred> [[ <Where> ]]                   -- type instance
            | { <Declaration> }                      -- value declarations

<Constructor> ::= <Constructor> [[ | <Constructor> ]] -- several constructors
                | <Type> Conop <Type>                -- infix constructor
                | Conid {{ <Type> }}                 --

<Declaration> ::= {{ <Declaration> ; }}              -- multiple declarations
                | <Varidlist> ::
                  [[ <Context> => ]] <Type>          -- type declaration
                | <Fun> <Rhs> [[ <Where> ]]           -- function binding
                | <Pattern> <Rhs> [[ <Where> ]]       -- pattern binding

<Rhs> ::= {{ | <Expr> = <Expr> }}                    -- guarded rhs
        | = <Expr>                                   -- simple righthand side

<Where> ::= WHERE { <Declaration> }                  -- local definitions

<Fun> ::= <Qvarid>                                   -- constant
        | <Pattern> <Qvarop> <Pattern>               -- infix operator
        | ( <Pattern>  <Qvarop> )                    -- section notation
        | ( <Qvarop>  <Pattern> )                    -- right side
        | <Fun> <Pattern>                            -- fun with argument
        | ( <Fun> )                                  -- parenthesised fun

-- TYPES

<Type> ::= <Type>  ->  <Type>                        -- type of a function
         | Conid {{ <Type> }}                        -- predefined datatype
         | Varid                                     -- type variable
         | [ <Type> ]                                -- list type
         | ( [[ <Type> {{ , <Type> }} ]] )           -- tuple type

<Context> ::= <Pred>                                 -- single context
            | ( <Pred> {{ , <Pred> }} )              -- general form

<Pred> ::= Conid <Type> {{ <Type> }}                 -- predicate

-- EXPRESSIONS

<Expr> ::= \ <Pattern> {{ <Pattern> }} -> <Expr>     -- lambda expression
         | LET { <Declaration> } IN <Expr>           -- local definition
         | IF <Expr> THEN <Expr> ELSE <Expr>         -- conditional expr
         | CASE <Expr> OF { <Alternative> }          -- case expression
         | :: <Expr> :: [[ <Context> => ]] <Type>    -- typed expression
         | <Expr> <Qop> <Expr>                       -- operator application
         | - <Expr>                                  -- negation
```

```
         | <Expr> <Expr>                                -- function application
         | <Qvarid>                                    -- (qualified) variable
         | <Gcon>                                      -- general constructor
         | <Literal>                                   -- literals
         | ( <Expr> {{ , <Expr> }} )                   -- tuples
         | [ <List> ]                                  -- list expression
         | ( <Expr> <Qop> )                            -- left section
         | ( <Qop> <Expr> )                            -- right section

<List> ::= <Expr> {{, <Expr> }}                        -- (enumerated) list
         | <Expr> "|" <Qualifier>                      -- list comprehension
         | <Expr> [[ , <Expr> ]] .. [[ <Expr> ]]       -- arithmetic sequence

<Qualifier> ::= <Qualifier> {{ , <Qualifier> }}        -- multiple qualifiers
              | <Pattern> <- <Expr>                    -- generator
              | <Pattern> = <Expr>                     -- local definition
              | <Expr>                                 -- boolean guard

<Alternative> ::= <Alternative> {{ ; <Alternative }}   -- multipel alternatives
                | <Pattern> <AltRhs> [[ <Where> ]]     -- alternative

<AltRhs> ::= -> <Expr>                                 -- single alternative
           | {{ | <Expr> --> <Expr> }}                 -- guarded alternatives

-- PATTERNS

<Pattern> ::= <Pattern> <Qconop> <Pattern>             -- operator application
            | <Qvarid> + Integer                       -- (n+k) pattern
            | <Pattern> <Pattern>                      -- application patterns
            | <Qvarid>                                 -- variable
            | <Qvarid> @ <Pattern>                     -- as pattern
            | ~ <Pattern>                              -- irrefutable pattern
            | _                                        -- wildcard
            | <Gcon>                                   -- constructor
            | <Literal>                                -- literal
            | ( <Pattern> <Qconop> )                   -- left section
            | ( <Qconop> <Pattern> )                   -- right section
            | ( <Pattern> {{, <Pattern> }} )           -- tuple (1 to n)
            | [ <Pattern> {{, <Pattern> }} ]           -- list (1 to n)

-- VARIABLES, IDENTIFIERS, OPERATORS

<Varlist> ::= [[ Varid {{, Varid }} ]]                 -- several pure vars

<Varidlist> ::= <Qvarid> {{, <Qvarid> }}               -- several varids

<Literal> ::= Integer
            | Float
            | Char
            | String

<Gcon> ::= ( )                                         -- unit
         | [ ]                                         -- empty list
         | <Qconid>                                    -- qualified constr. id

<Oplist> ::= <Qop> {{ , <Qop> }}                       -- multiple operators

<Qop> ::= <Qvarop>
        | <Qconop>                                     -- qualified operators

<Qvarid> ::= Varid                                     -- var/fct identifier
           | ( Varop )                                 -- mapped operator
```

```
<Qvarop> ::= Varop                                  -- var/fct operator
           | ‘ Varid ‘                              -- mapped identifier

<Qconid> ::= Conid                                  -- userdef identifier
           | ( Conop )                              -- mapped operator

<Qconop> ::= Conop                                  -- userdef operator
           | ‘‘ Conid ‘‘                            -- mapped identifier

-- EXTERNAL TYPES

Integer; String;  Varid; Varop; Conid; Conop;
```

A.3.3 Grammatik der logischen Paradigmasprache

```
<Program> ::= {{ <Sentence> }}                      -- several expressions

<Sentence> ::= <Module> : <Sentence>                -- named sentence
             | <Term> [[ :- <Body> ]] .             -- clause
             | :- <Body> .                          -- command
             | ?- <Body> .                          -- query

<Body> ::= <Body> {{ , <Body> }}                    -- several bodies
         | <Module> : <Body>                        -- named body
         | <Body> ; <Body>                          -- disjunction
         | <Term> -> <Body> [[ ; <Body> ]]          -- if then else
         | \+ <Term>                                -- not provable goal
         | <Term>                                   -- goal

<Module> ::= <Atom>                                 -- module designator

<Term> ::= <Term> <Op> <Term>                       -- operation
         | <Atom> ( <Term> {{ , <Term> }} )         -- compound term
         | <Terminal>                               -- terminal
         | <Constant>                               -- constant
         | Var                                      -- variable
         | ( <Term> {{ , <Term> }} )                -- parenthesised terms

<List> ::= []                                       -- empty list
         | [ <Term> {{ , <Term> }} ]                -- normal list
         | [ <Term> {{ , <Term> }} | <Term> ]       -- variable tail

<Op> ::= is | =:= | =\\= | < | <= | > | >=          -- relational operators
       | = | + | - | * | / | // | /\ | \/ | #       -- arithmetic, logic

<Terminal> ::= <List>                               -- list
             | String                               -- string

<Constant> ::= Integer                              -- integer
             | Float                                -- float
             | <Atom>                               -- atom

<Atom> :: = Word                                    -- constant words
          | Symbol                                  -- symbols
          | Char                                    -- 'something'
          | ! | ;                                   -- solo chars
```

A.4 Abstrakte Grammatiken der Paradigmasprachen

Die abstrakte Syntax der imperativen Grammatik ist in Abbildung 2.3.

A.4.1 Abstrakte Syntax der objektorientierten Grammatik

```
'type' DECLARATION = declare( IDENT, DEFINITION ) .

'type' DECLLIST = decllist( DECLARATION, DECLLIST ), nil .

'type' DEFINITION = variable( TYPE ), valueparam( TYPE ),
       const( TYPE, EXPRESSION ), class( DECLLIST, IDENT, DECLLIST ),
       method( ABSTRACT, SCOPE, SUPERTYPE, DECLLIST, DECLLIST, STMTSEQ ) .

'type' ABSTRACT = abstract, concrete .

'type' SCOPE = public, private, protected, privateprotected, default .

'type' STMTSEQ = stmt( STATEMENT ), seq( STATEMENT, STMTSEQ ), nil .

'type' STATEMENT = assignment( EXPRESSION, EXPRESSION ), compound( STMTSEQ ),
       call( IDENT, EXPRLIST ),ifthenelse( EXPRESSION, STATEMENT, STATEMENT ),
       while( EXPRESSION, STATEMENT ), repeat( STATEMENT, EXPRESSION ),
       for( VARIABLE, EXPRESSION, EXPRESSION, EXPRESSION, STATEMENT ),
       return( EXPRESSION ), new( TYPE, VARIABLE, TYPE, EXPRLIST ),
       switch( EXPRESSION, STMTSEQ, STMTSEQ ), case( EXPRLIST, STMTSEQ ), nil.

'type' EXPRLIST = exprlist( EXPRESSION, EXPRLIST ), nil .

'type' EXPRESSION = relative( EXPRESSION, RELOP, EXPRESSION ),
       and( EXPRESSION, EXPRESSION ), or( EXPRESSION, EXPRESSION ),
       not( EXPRESSION ), mult( EXPRESSION, EXPRESSION ),
       div( EXPRESSION, EXPRESSION ), intdiv( EXPRESSION, EXPRESSION ),
       mod( EXPRESSION, EXPRESSION ), plus( EXPRESSION, EXPRESSION ),
       minus( EXPRESSION, EXPRESSION ), neg( EXPRESSION ),
       var( VARIABLE ), num( INT ), double( DOUBLE ), string( TSTRING ),
       dot( EXPRESSION, IDENT, EXPRLIST ), true, false .

'type' RELOP = eq, ne, lt, le, gt, ge .

'type' TYPE = array( EXPRESSION, EXPRESSION, TYPE ),
       void, integer, real, character, boolean, string, object( IDENT ).

'type' VARIABLE = id( IDENT ), array( VARIABLE, EXPRESSION ) .

'type' SUPERTYPE = id( IDENT ), nil .

'type' IDENT .

'type' DOUBLE .
```

A.4.2 Abstrakte Syntax der funktionalen Grammatik

```
'type' TOP = module(TOPDECLLIST), interp(EXPR, DECLLIST).

'type' TOPDECLLIST = topdecllist(TOPDECL, TOPDECLLIST), nil.

'type' TOPDECL = data(TYPE, CONSTRLIST), simple(TYPE, TYPE),
                 fixity(FIXITY, DIGIT, OPLIST), class(PREDICATE, DECLLIST),
                 classcontext(CONTEXT, PREDICATE, DECLLIST),
                 instancecontext(CONTEXT, PREDICATE, DECLLIST),
```

```
                instance(PREDICATE, DECLLIST), decls(DECLLIST).

'type' CONSTRLIST = oneconstr(CONSTRUCTOR),
                    constrlist(CONSTRUCTOR, CONSTRLIST).

'type' CONSTRUCTOR = infix(TYPE, CONOP, TYPE), construct(CONID, TYPELIST),
                     unit, list, qcon(IDENTIFIER).

'type' FIXITY = left, right, infix.

'type' DIGIT = digit(INT).

'type' CONTEXT = context(PREDICATE), contextlist(PREDLIST).

'type' PREDLIST = onepred(PREDICATE), predlist(PREDICATE, PREDLIST).

'type' PREDICATE = predicate(CONID, TYPELIST).

'type' TYPELIST = typelist(TYPE, TYPELIST), onetype(TYPE), notype,
                  atypes(TYPE, TYPELIST), oneatype(TYPE).

'type' TYPE = simple(CONID, VARLIST), funtype(TYPE, TYPE), datatype(CONID),
         synoym(CONID, TYPELIST), var(VAR), unit, tuple(TYPELIST),list(TYPE).

'type' DECLLIST = decls(DECL, DECLLIST), body(DECLLIST), nil.

'type' DECL = memfun(VARLIST, TYPE), typedecl(VARLIST, CONTEXT, TYPE),
           funbind(FUNCTION, RHS, DECLLIST), patbind(PATTERN, RHS, DECLLIST).

'type' RHS = rhs(EXPR), guardrhs(GDRHS).

'type' GDRHS = onegdrhs(GUARD), multgdrhs(GUARD, GDRHS).

'type' GUARD = guard(EXPR, EXPR).

'type' FUNCTION = funvar(IDENTIFIER), funinfix(PATTERN, OPERATOR, PATTERN),
                  funsecl(PATTERN, OPERATOR), funsecr(OPERATOR, PATTERN),
         funarg(FUNCTION, PATTERN).

'type' EXPRLIST = exprlist(EXPR, EXPRLIST), nil.

'type' EXPR = lambda(PATLIST, EXPR), let(DECLLIST, EXPR),
              ifthenelse(EXPR, EXPR, EXPR), case(EXPR, ALTS),
              exprtypecontext(EXPR, CONTEXT, TYPE), exprtype(EXPR, TYPE),
              opapp(EXPR, OPERATOR, EXPR), neg(EXPR), funapp(EXPR, EXPR),
              var(IDENTIFIER), constr(CONSTRUCTOR), literal(LITERAL),
              parent(EXPRLIST), list(EXPR), lsection(EXPR, OPERATOR),
              rsection(OPERATOR, EXPR), enumerated(EXPRLIST), nil,
              listcomp(EXPR, QUALLIST),
              arithseq(EXPRLIST), arithseq2(EXPRLIST, EXPRLIST).

'type' QUALLIST = qlist(QUAL, QUALLIST), aqual(QUAL).

'type' QUAL = generator(EXPR, EXPR), localdef(EXPR, EXPR), guard(EXPR).

'type' ALTS = altlist(ALT, ALTS), onealt(ALT).

'type' ALT = alt(PATTERN, ALTRHS, DECLLIST).

'type' ALTRHS = single(EXPR), guarded(GDALTS).

'type' GDALTS = gdalts(GDALT, GDALTS), onegdalt(GDALT).
```

```
'type' GDALT = guard(EXPR, EXPR).

'type' PATTERN = application(PATLIST, OPERATOR, PATTERN),
       successor(IDENTIFIER, INT), applist(PATLIST), var(IDENTIFIER),
       aspattern(IDENTIFIER, PATTERN), irrefutable(PATTERN), wildcard,
       gcon(CONSTRUCTOR), literal(LITERAL), patsecl(PATTERN, OPERATOR),
       patsecr(OPERATOR, PATTERN), tuple(PATLIST), list(PATLIST).

'type' PATLIST = patlist(PATTERN, PATLIST), onepat(PATTERN).

'type' VARLIST = varlist(VAR, VARLIST), var(VAR),
         varidlist(IDENTIFIER, VARLIST), varid(IDENTIFIER), nil.

'type' LITERAL = int(INT), string(STRING).

'type' OPLIST = operators(OPERATOR, OPLIST), oneop(OPERATOR).

'type' OPERATOR = qvarop(OPERATOR), qconop(OPERATOR), varop(VAROP),
                  conop(CONOP), varid(VAR), conid(CONID).

'type' IDENTIFIER = varid(VAR), varopParent(VAROP), conid(CONID),
            conopParent(CONOP).

'type' CONID; 'type' VAR; 'type' VAROP; 'type' CONOP
```

A.4.3 Abstrakte Syntax der logischen Grammatik

```
'type' SENTENCES = sentences(SENTENCE, SENTENCES), onesentence(SENTENCE) .

'type' SENTENCE = named(MODULE, SENTENCE),
       clause(CLAUSE), directive(DIRECTIVE) .

'type' CLAUSE = nonunit(TERM, BODYLIST), unit(TERM) .

'type' DIRECTIVE = command(BODYLIST), query(BODYLIST) .

'type' BODYLIST = bodies(BODY, BODYLIST), body(BODY) .

'type' BODY = named(MODULE, BODY), or(BODY, BODY), goal(TERM),
       ifthenelse(TERM, BODY, BODY), notprovable(TERM), nil .

'type' TERMINAL = list(LIST), string(TSTRING) .

'type' MODULE = atom(ATOM) .

'type' TERMLIST = termlist(TERM, TERMLIST), oneterm(TERM) .

'type' TERM = relation(TERM, RELOP, TERM), arithmetic(TERM, OPERATOR, TERM),
       constant(CONSTANT), variable(VAR), terminal(TERMINAL),
       compound(ATOM, TERMLIST), parent(TERMLIST) .

'type' CONSTANT = number(INTEGER), atom(ATOM) .

'type' ATOM = word(WORD), symbol(SYMBOL), quoted(CHAR),
       cut, semicolon, brackets .

'type' LIST = emptylist, normallist(TERMLIST), variabletail(TERMLIST, TERM) .

'type' RELOP = is, eq, ne, lt, leq, gt, geq .

'type' OPERATOR = unification, plus, minus, mult, div, intdiv, and, or, xor .
```

```
'type' SYMBOL .
'type' TSTRING .
'type' INTEGER .
'type' WORD .
'type' CHAR .
'type' VAR .
```

A.5 Grammatik der generellen Zwischensprache

Es folgen alle Typen und Funktoren der GZS.

```
'type' DECLARATIONLIST = decllist( DECLARATION, DECLARATIONLIST ), nil.

'type' DECLARATION     = declare( POS, TERMINAL, DEFINITION ),
                         decls( DECLARATIONLIST ).

'type' DEFINITION      = variable( TYPE, EXPRESSION ),
                         datatypedef( TERMINAL, TYPELIST ),
                         class( DECLARATIONLIST, SUPERTYPE, DECLARATIONLIST ),
                         method( ABSTRACT, SCOPE, TYPE, DECLARATIONLIST,
                                 DECLARATIONLIST, STATEMENTLIST ),
                         binding( EXPRESSION, STATEMENT, DECLARATIONLIST ),
                         infixleft( TERMINAL ),
                         infixright( TERMINAL ),
                         nonassoc( TERMINAL ),
                         resolve( STATEMENTLIST, EXPRESSION ).

'type' STATEMENTLIST   = stmtseq( STATEMENT, STATEMENTLIST ),
                         stmt( STATEMENT ), nil.

'type' STATEMENT       = assignment( POS, EXPRESSION, EXPRESSION ),
                         ifthenelse( POS, EXPRESSION, STATEMENT, STATEMENT ),
                         loop( POS, STATEMENT, EXPRESSION, STATEMENTLIST ),
                         compound( STATEMENTLIST ),
                         call( POS, TERMINAL, EXPRESSIONLIST ),
                         read( POS, TERMINAL ),
                         write( POS, EXPRESSION ),
                         return( POS, EXPRESSION ),
                         new( POS, TYPE, TERMINAL, TYPE, EXPRESSIONLIST ),
                         switch( POS, EXPRESSION, STATEMENTLIST, STATEMENTLIST ),
                         case( POS, EXPRESSIONLIST, STATEMENTLIST, DECLARATIONLIST ),
                         declstmt( DECLARATION ),
                         exprstmt( EXPRESSION ),
                         nil( POS ).

'type' EXPRESSIONLIST  = exprlist( EXPRESSION, EXPRESSIONLIST ), nil.

'type' EXPRESSION      = operation( POS, EXPRESSION, TERMINAL, EXPRESSION ),
                         relation( POS, EXPRESSION, TERMINAL, EXPRESSION ),
                         sectionleft( POS, EXPRESSION, TERMINAL ),
                         sectionright( POS, TERMINAL, EXPRESSION ),
                         and( POS, EXPRESSION, EXPRESSION ),
                         or( POS, EXPRESSION, EXPRESSION ),
                         not( POS, EXPRESSION ),
                         true, false,
                         plus( POS, EXPRESSION, EXPRESSION ),
                         minus( POS, EXPRESSION, EXPRESSION ),
                         neg( EXPRESSION ),
                         mult( POS, EXPRESSION, EXPRESSION ),
                         div( POS, EXPRESSION, EXPRESSION ),
```

```
                        intdiv( POS, EXPRESSION, EXPRESSION ),
                        mod( POS, EXPRESSION, EXPRESSION ),
                        termexp( TERMINAL ),
                        parent( EXPRESSIONLIST ),
                        stmtexp( STATEMENTLIST ),
                        list( EXPRESSIONLIST ),
                        funexp( POS, EXPRESSION, TERMINAL, EXPRESSIONLIST ),
                        arithseq( POS, EXPRESSION, EXPRESSION, EXPRESSION ),
                        nil.

'type' TYPELIST       = typelist( TYPE, TYPELIST ),
                        constrlist( TYPE, TYPELIST ), nil .

'type' TYPE           = datatype( Type: TERMINAL, Typevars: TYPELIST ),
                        builtin( POS, BUILTIN ),
                        funtype( POS, In: TYPE, Out: TYPE ),
                        array( EXPRESSION, TYPE ),
                        tupletype( TYPELIST ),
                        typevar( TERMINAL ), nil .

'type' BUILTIN        = integer, real, char, boolean, string, void .

'type' TERMINAL       = desig( POS, DESIG ),
                        compterm( POS, COMPTERM ).

'type' DESIG          = varid( VAR ), constid( WORD ),
                        number( INT ), real( DOUBLE ),
                        char( CHAR ), string( TSTRING ),
                        operator( OPERATOR ), relop( RELOP ),
                        symbol( SYMBOL ),
                        conid( CONID ), conop( CONOP ),
                        cut, nil.

'type' COMPTERM       = array( TERMINAL, EXPRESSION ),
                        list( DESIG, TERMINAL ).

'type' ABSTRACT       = abstract, concrete.

'type' SCOPE          = public, private, protected, privateprotected,
                        default.

'type' SUPERTYPE      = id( IDENT ), nil.

'type' RELOP          = eq, ne, lt, le, gt, ge.
```

A.6 Beispiel aus Kapitel 9

A.6.1 Zwischenprogramm des Fallbeispiels

```
declare(1000006001,
  desig(1000006009,
    varid(<<PowerToArray>>)),
  class(decllist(declare(1000009003,
        desig(1000009003,
          varid(<<result>>)),
        variable(builtin(1000009012,
            real),
          nil)),
        decllist(declare(1000010003,
```

```
        desig(1000010003,
          varid(<<test>>)),
        variable(builtin(1000010007,
            integer),
            termexp(desig(1000018015,
            number(10)))))),
      decllist(declare(1000012003,
          desig(1000012003,
            varid(<<aField>>)),
          variable(array(arithseq(1000012020,
                termexp(desig(1000012019,
                    number(0))),
                nil,
                termexp(desig(1000012022,
                    number(10)))),
              builtin(1000012029,
                real)),
            nil)),
      nil))),
  nil,
  decllist(declare(1000012003,
      desig(1000012003,
        varid(<<xToN>>)),
      method(static,
        private,
        builtin(1000010007,
            real),
        decllist(declare(1000012019,
            desig(1000012023,
              varid(<<base>>)),
            variable(builtin(1000012034,
                real),
              nil)),
          decllist(declare(1000012043,
              desig(1000012047,
                varid(<<exponent>>)),
              variable(builtin(1000012054,
                  integer),
                nil)),
            nil)),
        decllist(declare(1000015005,
            desig(1000015005,
              varid(<<result>>)),
            variable(builtin(1000015014,
                real),
              nil)),
          nil),
        stmtseq(assignment(1000018005,
            termexp(desig(1000018005,
                varid(<<result>>))),
            termexp(desig(1000018015,
                number(1)))),
          stmtseq(loop(1000019005,
              nil,
              relation(1000019020,
                termexp(desig(1000019011,
                    varid(<<exponent>>))),
                desig(1000019020,
                  relop(gt)),
                termexp(desig(1000019022,
                    number(0)))),
              stmtseq(compound(stmtseq(assignment(1000021007,
```

```
                    termexp(desig(1000021007,
                        varid(<<result>>))),
                    mult(1000021024,
                      termexp(desig(1000021017,
                          varid(<<result>>))),
                      termexp(desig(1000021026,
                          varid(<<base>>))))),
                  stmtseq(assignment(1000022007,
                      termexp(desig(1000022007,
                          varid(<<exponent>>))),
                      minus(1000022028,
                        termexp(desig(1000022019,
                            varid(<<exponent>>))),
                        termexp(desig(1000022030,
                            number(1))))),
                      nil)),
              nil)),
          stmtseq(return(1000024005,
              termexp(desig(1000024011,
                  varid(<<result>>)))),
            nil))))),
  decllist(declare(1000031003,
      desig(1000031003,
        varid(<<fillArray>>)),
      method(static,
        private,
        nil,
        nil,
        decllist(declare(1000034005,
            desig(1000034005,
              varid(<<i>>)),
            variable(builtin(1000034009,
                integer),
              nil)),
          nil),
        stmtseq(loop(1000037005,
            assignment(1000037009,
              termexp(desig(1000037009,
                  varid(<<i>>))),
              termexp(desig(1000037014,
                  number(0)))),
            relation(1000037009,
              termexp(desig(1000037009,
                  varid(<<i>>))),
              desig(1000037005,
                relop(lt)),
              termexp(desig(1000037019,
                  number(10)))),
            stmtseq(compound(stmtseq(assignment(1000040015,
                      termexp(compterm(1000040015,
                          array(desig(1000040009,
                              varid(<<aField>>)),
                            termexp(desig(1000040016,
                                varid(<<i>>)))))),
                      stmtexp(stmtseq(
                    call(1000039009,
                    desig(1000039009,
                      varid(<<xToN>>)),
                    exprlist(termexp(desig(1000039014,
                          real(<<2.5>>))),
                      exprlist(termexp(desig(1000039019,
                          varid(<<i>>))),
```

```
                        nil))), nil))),
                    stmtseq(write(1000041009,
                        termexp(compterm(1000041021,
                            array(desig(1000041015,
                                varid(<<aField>>)),
                              termexp(desig(1000041022,
                                  varid(<<i>>)))))))),
                      nil))),
              stmtseq(assignment(1000037005,
                  termexp(desig(1000037009,
                      varid(<<i>>))),
                  plus(1000037009,
                    termexp(desig(1000037009,
                        varid(<<i>>))),
                    termexp(desig(1000037027,
                        number(1))))),
                nil))),
          nil))),
    decllist(declare(1000028001,
        desig(1000028001,
          nil),
        method(static,
          public,
          nil,
          nil,
          nil,
            stmtseq(ifthenelse(1000049003,
                relation(1000049011,
                  termexp(desig(1000049006,
                      varid(<<test>>))),
                  desig(1000049011,
                    relop(le)),
                  termexp(desig(1000049014,
                      number(0)))),
                compound(stmtseq(
                      assignment(1000031003,
                      termexp(desig(1000031003,
                      varid(<<result>>))),
                      stmtexp(stmtseq(
                      call(1000030003, desig(1000030003,
                      varid(<<xToN>>)),
                      exprlist(termexp(desig(1000030008,
                      real(<<2.5>>))),
                      exprlist(termexp(desig(1000030013,
                      number(4))),
                       nil))), nil))),
                      stmtseq(write(1000054007,
                          termexp(desig(1000054013,
                              varid(<<result>>)))),
                        nil))),
                compound(stmtseq(call(1000058006,
                      desig(1000058006,
                        varid(<<fillArray>>)),
                      nil),
                    nil))),
              nil))),
        nil)))))
```

A.6.2 `class` File des Fallbeispiels

```
Magic: cafebabe
Minor Version: 3
Major Version: 45
Constant_Pool:
 Constant_Pool_Count: 61
  1: C_Utf8: length = 12, bytes = "PowerToArray"
  2: C_Class: name_index = 1
  3: C_Utf8: length = 6, bytes = "result"
  4: C_Utf8: length = 1, bytes = "F"
  5: C_NameAndType: name_index = 3, descriptor_index = 4
  6: C_Fieldref: class_index = 2, nameAndType_index = 5
  7: C_Utf8: length = 4, bytes = "test"
  8: C_Integer: bytes = 10
  9: C_Utf8: length = 1, bytes = "I"
  10: C_NameAndType: name_index = 7, descriptor_index = 9
  11: C_Fieldref: class_index = 2, nameAndType_index = 10
  12: C_Utf8: length = 6, bytes = "aField"
  13: C_Utf8: length = 2, bytes = "[F"
  14: C_NameAndType: name_index = 12, descriptor_index = 13
  15: C_Fieldref: class_index = 2, nameAndType_index = 14
  16: C_Utf8: length = 16, bytes = "java/lang/Object"
  17: C_Class: name_index = 16
  18: C_Utf8: length = 6, bytes = "<init>"
  19: C_Utf8: length = 3, bytes = "()V"
  20: C_NameAndType: name_index = 18, descriptor_index = 19
  21: C_Methodref: class_index = 17, nameAndType_index = 20
  22: C_Utf8: length = 4, bytes = "xToN"
  23: C_Utf8: length = 4, bytes = "base"
  24: C_Utf8: length = 8, bytes = "exponent"
  25: C_Utf8: length = 5, bytes = "(FI)F"
  26: C_NameAndType: name_index = 22, descriptor_index = 25
  27: C_Methodref: class_index = 2, nameAndType_index = 26
  28: C_Utf8: length = 9, bytes = "fillArray"
  29: C_Utf8: length = 1, bytes = "i"
  30: C_Float: bytes = 1075838976, value = 2.5
  31: C_Utf8: length = 16, bytes = "java/lang/System"
  32: C_Class: name_index = 31
  33: C_Utf8: length = 3, bytes = "out"
  34: C_Utf8: length = 21, bytes = "Ljava/io/PrintStream;"
  35: C_NameAndType: name_index = 33, descriptor_index = 34
  36: C_Fieldref: class_index = 32, nameAndType_index = 35
  37: C_Utf8: length = 19, bytes = "java/io/PrintStream"
  38: C_Class: name_index = 37
  39: C_Utf8: length = 7, bytes = "println"
  40: C_Utf8: length = 4, bytes = "(I)V"
  41: C_NameAndType: name_index = 39, descriptor_index = 40
  42: C_Methodref: class_index = 38, nameAndType_index = 41
  43: C_Utf8: length = 4, bytes = "(F)V"
  44: C_NameAndType: name_index = 39, descriptor_index = 43
  45: C_Methodref: class_index = 38, nameAndType_index = 44
  46: C_NameAndType: name_index = 28, descriptor_index = 19
  47: C_Methodref: class_index = 2, nameAndType_index = 46
  48: C_Utf8: length = 4, bytes = "main"
  49: C_NameAndType: name_index = 33, descriptor_index = 34
  50: C_Fieldref: class_index = 32, nameAndType_index = 49
  51: C_NameAndType: name_index = 39, descriptor_index = 40
  52: C_Methodref: class_index = 38, nameAndType_index = 51
  53: C_NameAndType: name_index = 39, descriptor_index = 43
  54: C_Methodref: class_index = 38, nameAndType_index = 53
```

```
  55: C_Utf8: length = 13, bytes = "ConstantValue"
  56: C_Utf8: length = 8, bytes = "<clinit>"
  57: C_Utf8: length = 4, bytes = "Code"
  58: C_Utf8: length = 22, bytes = "([Ljava/lang/String;)V"
  59: C_Utf8: length = 10, bytes = "SourceFile"
  60: C_Utf8: length = 25, bytes = "generated from GZS"
Access Flags: ACC_PUBLIC | ACC_SUPER
This  class: index = 2, class = "PowerToArray"
Super class: index = 17, class = "java/lang/Object"
0 Interface(s):
Fields count: 3
  0. Field: ACC_PRIVATE | ACC_STATIC
       Name: "result", Descriptor: "F", Attributes count: 0
  1. Field: ACC_PRIVATE | ACC_STATIC
       Name: "test", Descriptor: "I", Attributes count: 1
       Attribute : "ConstantValue", length = 2
                "Int 10"
  2. Field: ACC_PRIVATE | ACC_STATIC
       Name: "aField", Descriptor: "[F", Attributes count: 0
Methods count: 5
  0. Method: ACC_STATIC
       Name: "<clinit>", Descriptor: "()V", attrCount = 1
       Attribute : "Code", length = 22
       Max stack: 2, Max locals: 1, Code length: 10
       0 : 16  bipush        10        byte value
       2 : 3   iconst_0
       3 : 100 isub
       4 : 188 newarray       type 6
       6 : 179 putstatic     #015      Field  PowerToArray/aField:[F
       9 : 177 return
       Exception Table length: 0
       Code Attributes count: 0
  1. Method: ACC_PUBLIC
       Name: "<init>", Descriptor: "()V", attrCount = 1
       Attribute : "Code", length = 17
       Max stack: 1, Max locals: 1, Code length: 5
       0 : 42  aload_0
       1 : 183 invokespecial #021      Method java/lang/Object/<init>:()V
       4 : 177 return
       Exception Table length: 0
       Code Attributes count: 0
  2. Method: ACC_PRIVATE | ACC_STATIC
       Name: "xToN", Descriptor: "(FI)F", attrCount = 1
       Attribute : "Code", length = 41
       Max stack: 2, Max locals: 3, Code length: 29
       0 : 4   iconst_1
       1 : 134 i2f
       2 : 56  fstore        t2        store float into loc var
       4 : 167 goto          16         => this line + offset
       7 : 23  fload         t2        load float from local variable
       9 : 23  fload         t0        load float from local variable
       11 : 106 fmul
       12 : 56  fstore        t2       store float into loc var
       14 : 21  iload         t1       load int from local variable
       16 : 4   iconst_1
       17 : 100 isub
       18 : 54  istore        t1       store int into local variable
       20 : 21  iload         t1       load int from local variable
       22 : 3   iconst_0
       23 : 163 if_icmpgt     goto255240       if int1  > int2
       26 : 23  fload         t2       load float from local variable
       28 : 174 freturn
```

```
       Exception Table length: 0
       Code Attributes count: 0
  3. Method: ACC_PRIVATE | ACC_STATIC
       Name: "fillArray", Descriptor: "()V", attrCount = 1
       Attribute : "Code", length = 57
       Max stack: 4, Max locals: 1, Code length: 45
       0 : 3   iconst_0
       1 : 54  istore        t0        store int into local variable
       3 : 167 goto          34         => this line + offset
       6 : 178 getstatic     #015      Field  PowerToArray/aField:[F
       9 : 21  iload         t0        load int from local variable
       11 : 18  ldc           #30      Float 2.5
       13 : 21  iload         t0       load int from local variable
       15 : 184 invokestatic  #027     Method PowerToArray/xToN:(FI)F
       18 : 81  fastore
       19 : 178 getstatic     #036     Field  java/lang/System/out:Ljava/io/PrintStream;
       22 : 178 getstatic     #015     Field  PowerToArray/aField:[F
       25 : 21  iload         t0       load int from local variable
       27 : 48  faload
       28 : 182 invokevirtual #045     Method java/io/PrintStream/println:(F)V
       31 : 21  iload         t0       load int from local variable
       33 : 4   iconst_1
       34 : 96  iadd
       35 : 54  istore        t0       store int into local variable
       37 : 21  iload         t0       load int from local variable
       39 : 16  bipush        10       byte value
       41 : 161 if_icmplt     goto255221       if int1  < int2
       44 : 177 return
       Exception Table length: 0
       Code Attributes count: 0
  4. Method: ACC_PUBLIC | ACC_STATIC
       Name: "main", Descriptor: "([Ljava/lang/String;)V", attrCount = 1
       Attribute : "Code", length = 52
       Max stack: 3, Max locals: 1, Code length: 40
       0 : 178 getstatic     #011      Field  PowerToArray/test:I
       3 : 3   iconst_0
       4 : 164 if_icmple     goto07    if int1 <= int2
       7 : 3   iconst_0
       8 : 167 goto          4  => this line + offset
       11 : 4   iconst_1
       12 : 153 ifeq          goto 024 => this line + offset, if int1 == 0
       15 : 18  ldc           #30      Float 2.5
       17 : 7   iconst_4
       18 : 184 invokestatic  #027     Method PowerToArray/xToN:(FI)F
       21 : 179 putstatic     #06      Field  PowerToArray/result:F
       24 : 178 getstatic     #050     Field  java/lang/System/out:Ljava/io/PrintStream;
       27 : 178 getstatic     #06      Field  PowerToArray/result:F
       30 : 182 invokevirtual #054     Method java/io/PrintStream/println:(F)V
       33 : 167 goto          6         => this line + offset
       36 : 184 invokestatic  #047     Method PowerToArray/fillArray:()V
       39 : 177 return
       Exception Table length: 0
       Code Attributes count: 0
Class Attributes count: 1
       Attribute : "SourceFile", length = 2
 --> "generated from general IR"
```

Literaturverzeichnis

[Aba96] Martin Abadi, Luca Cardelli: *A Theory of Objects*, Springer Verlag, 1996.

[Aho72] Alfred V. Aho, Jeffrey D. Ullman: *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, 1972.

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[All71] Frances E. Allen, John Cocke: *A Catalogue of Optimizing Transformations*, Research Report RC 3548, IBM, Yorktown Heights, 1971.

[App84] Andrew W. Appel: *Semantics-Directed Code Generation*, Proc. Twelfth ACM Symposium on Principles of Programming Languages, 1985.

[App97] Andrew W. Appel: *Modern Compiler Implementation in Java: Basic Techniques*, Cambridge University Press, 1997.

[Bac57] J. W. Backus et al: *The FORTRAN Automatic Coding System*, Proc. WJCC 11, 1957, pp. 188-198.

[Bal94] Henri E. Bal, Dick Grune: *Programming Language Essentials*, Addison-Wesley, 1994.

[Bar96] John Barnes: *Programming in Ada 95*, Addison-Wesley, 1996.

[Bau74] Friedrich L. Bauer, Johann Eickel: *Compiler Construction, An Advanced Course*, Springer-Verlag, 1974.

[Bea72] J. C. Beatty: *A Global Register Assignment Algorithm*, in „Design and Optimization of Compilers“, Prentice-Hall, 1972.

[Ber66] Edmund C. Berkeley, Daniel G. Bobrow, Editors: *The Programming Language LISP: Its Operation and Applications*, MIT Press, 1966.

[Bir73] G. Birtwistle, Ole Johan Dahl, B. Myhrtag, Kristen Nygaard: *Simula Begin*, Auerbach Press, Philadelphia, 1973.

[Boo99] Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[Bou78] S.R. Bourne: *The UNIX Shell*, Bell System Technical Journal 57/6 (2), 1978.

[Bus96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, May 1996.

[Car85] Luca Cardelli, Peter Wegner: *On Understanding Types, Data Abstraction and Polymorphism*, ACM Computing Surveys, vol. 17, no. 4, Dec 1985, pp. 471-522.

[Car92] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson: *Modula-3 Language Definition*, ACM SIGPLAN Notices 27/8, Aug 1992, pp. 15-42.

[Car94] Luca Cardelli, John C. Mitchell: *Operations on Records*, in „Theoretical Aspects of Object-Oriented Programming“, MIT Press, 1994.

[Coc70] John Cocke: *Global Common Subexpression Elimination*, ACM SIGPLAN Notices, Vol. 5, No. 7, 1970.

[DeR69] F. L. DeRemer: *Practical Translators for LR(k) Languages*, Ph.D. Dissertation, Department of Electrical Engineering, MIT, Cambridge MS, 1969.

[DeR74] F. L. DeRemer: *Transformational Grammars*, in „Compiler Construction“, Springer-Verlag, 1974.

[Dij62] E. W. Dijkstra: *A Primer of ALGOL 60 Programming*, Academic Press, 1962.

[Dyb87] Kent Dybvig: *The SCHEME Programming Language*, Prentice-Hall, 1987.

[Ell95] Margaret A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, April 1995.

[Fay97] Mohamed E. Fayad, Douglas C. Schmidt: *Object-Oriented Application Frameworks*, Communications of the ACM 40/10, Oct 1997, pp. 39-42.

[Fla96] David Flanagan: *Java in a Nutshell*, O'Reilly, 1996.

[Gol72] Patricia C. Goldberg: *A Comparison of Certain Optimization Techniques*, in „Design and Optimization of Compilers“, Prentice-Hall, 1972.

[Gol89] Adele Goldberg, David Robson: *Smalltalk-80: the language*, Addison-Wesley, 1989.

[Goo83] G. Goos, J. Harmanis, Editoren: *The Programming Language Ada Reference Manual*, Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[Gos96] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification*, Addison-Wesley, 1996.

[Hol91] Urs Holzle, Bay-Wei Chang, Craig Chambers and David Ungar: *The SELF Manual*, Computer Systems Laboratory of Stanford University, 1991.

[Hud89] Paul Hudak: *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys 21/3, 1989, pp 359-411.

[Hud92] Paul Hudak, Simon Peyton Jones und Philip Wadler (Editor): *Report on the Programming Language Haskell – A Non-strict, Purely Functional Language (Version 1.2)*, ACM SIGPLAN Notices 27/5, 1992.

[IBM66] IBM System/360 Operating System: *PL/I Language Specifications*, IBM Corp. Data Processing Division, 1966.

[Joh75] S.C. Johnson: *Yacc: Yet Another Compiler Compiler*, Computer Science Technical Report No 32, Bell Laboratories, Murray Hill, NJ, 1975.

[Joh88] Ralph E. Johnson, Brian Foote: *Designing Reusable Classes*, Journal of Object-Oriented Programming 1/2, 1988, pp. 22-35.

[Jon91] Mark P. Jones: *Gofer - Functional Programming Environment*, Version 2.20, 1991.

[Jon94] Mark P. Jones, Codearchiv: *ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer/*, Gofer Version 2.30a, 1994.

[Kam90] Samuel Kamin: *Programming Languages, an Interpreter-Based Approach*, Addison-Wesley, 1990.

[Ken72] Ken Kennedy: *Index Register Allocation in Straight Line Code and Simple Loops*, in „Design and Optimization of Compilers“, Prentice-Hall, 1972.

[Ker78] B.W. Kernighan, D.M. Ritchie: *The C Programming Language*, Prentice Hall Software Series, 1978.

[Kop88] Helmut Kopka: *LaTeX- Eine Einführung*, Addison-Wesley, 1988.

[Lee89] Peter Lee: *Realistic Compiler Generation*, MIT, 1989.

[Les75] M.E. Lesk, E. Schmidt: *Lex – A Lexical Analyzer Generator* , Computer Science Technical Report No 39, Bell Laboratories, Murray Hill, NJ, 1975.

[Lex97] Hans-Jochen Schneider, Editor: *Lexikon der Informatik und Datenverarbeitung*, Oldenbourg, 1997.

[Lin97] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[Lut96] Mark Lutz: *Programming Python*, O'Reilly, 1996.

[Mey88] Betrand Meyer: *Object-oriented Software Construction*, Prentice Hall, 1988.

[Mey92] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

[Mil90] Robin Milner, MadsTofte, Robert Harper: *The Definition of Standard ML*, MIT Press, 1990.

[Nau60] P. Naur, Editor: *Report on the Algorithmic Language ALGOL 60*, ACM 3/5, May 1960, pp. 299-314.

[Nic75] John E. Nichols: *The Stucture and Design of Programming Languages*, Addison-Wesley, 1975.

[Pae93] Andreas Paepcke, Editor: *Object-oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[Pey87] Simon L. Peyton Jones: *The implementation of functional programming languages*, Prentice Hall, 1987.

[Pol81] Wolfgang Polak: *Compiler Specification and Verification*, Springer Verlag, 1981.

[Ral92] Anthony Ralston, Edwin D. Reilly, Editoren: *Encyclopedia of computer science, third edition*, Van Nostrand Reinhold, 1992.

[Rei91] Martin Reiser, Niklaus Wirth: *The Oberon System, User Guide and Programmer's Manual*, ACM Press, 1991.

[Ros96] Guido van Rossum: *Python Reference Manual*, Stichting Mathematisch Centrum, Amsterdam, 1996.

[Rus72] Randall Rustin, Editor: *Design and Optimization of Compilers*, Courant Computer Science Symposium No 5, Prentice-Hall, 1972.

[Sam69] Jean E. Sammet: *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969.

[Sch94] David A. Schmidt: The Structure of Typed Programming Languages, MIT Press, 1994.

[Sch97] Friedrich Wilhelm Schröer: *The GENTLE Compiler Construction System*, Oldenbourg Verlag, 1997.

[Set89] Ravi Sethi: *Programming languages, concepts and constructs*, Addison-Wesley, 1989.

[Sic96] Intelligent Systems Laboratory, Swedish Institute of Computer Science: *SICStus Prolog User's Manual*, Release 3.5, Oct 1996.

[Str58] J. Strong et al: *The Problem of Programming Communication with Changing Machines: A Proposed Solution*, Comm. ACM 1/8 & 1/9, 1958.

[Thi94] Peter Thiemann: *Grundlagen der funktionalen Programmierung*, Teubner, 1994.

[USG65] U.S. Government, Department of Defense: *COBOL: Edition 1965*, Printing Office, Washington, D.C., Nov 1965.

[Wai74] W. M. Waite: *Semantic Analysis*, in „Compiler Construction", Springer-Verlag, 1974.

[Wai89] W. M. Waite, J. Grosch, F.W. Schröer: *Three Compiler Specifications*, GMD-Studien Nr. 166, Gesellschaft für Mathematik und Datenverarbeitung GmbH, 1989.

[Wal90] Larry Wall, Randal L. Schwartz: *Programming Perl*, O'Reilly, 1990.

[Wel97] Brent B. Welch: *Practical Programming in Tcl & Tk*, Prentice-Hall, 1992.

[Wir75] Niklaus Wirth, Kathleen Jensen: *PASCAL: User Manual and Report*, Springer-Verlag, 1975.