Diplomarbeit

Definition und Implementierung eines

Front-End-Generators für Oz

von

Leif Kornstaedt

September 1996

Betreuer:

Prof. Dr. rer. nat. Gert Smolka
Dipl.-Inform. Christian Schulte
Prof. Dr.-Ing. Hans-Wilm Wippermann

Fachbereich Informatik

Universität Kaiserslautern

Erklärung

Hiermit erkläre ich, Leif Kornstaedt, daß ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kaiserslautern, den 16. September 1996

Zusammenfassung

In der vorliegenden Diplomarbeit wird ein Front-End-Generator entwickelt, der die multiparadigmatische Sprache Oz als Zielsprache verwendet. Damit wird die Eignung von Oz als Implementierungssprache für Compiler demonstriert und die Reimplementierung des Oz-Compilers in Oz vorbereitet.

Das Werkzeug ist besonders auf die Übersetzung vollkompositionaler Sprachen ausgelegt. Das bedeutet, daß neben der lexikalischen und syntaktischen Analyse, bei der über sogenannte *Produktionsschemata* auch eigene EBNF-Operatoren definiert werden können, auch die Reduktion in eine Kernsprache von dem Werkzeug abgedeckt wird. Um letztere mächtig zu machen und die Implementierung von Baumtransformationen zu vereinfachen, bei denen keine Konflikte der Variablennamen auftreten dürfen, wird weiterhin eine automatisch durchgeführte Bindungsanalyse mit Umbenennung aller gebundenen Bezeichner von dem Werkzeug angeboten.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Zielsetzung

1.1 Motivation

Bedenkt man die Allgegenwärtigkeit von Software im heutigen Alltag und die zunehmende Tendenz, Softwaresystemen sicherheitskritische Aufgaben anzuvertrauen, so wird die Bedeutung mächtiger Entwicklungswerkzeuge offensichtlich. Durch ihren Einsatz kann eine höhere Qualität der Softwareprodukte erzielt werden, beispielsweise hinsichtlich ihrer Benutzerfreundlichkeit und Zuverlässigkeit. Zu diesen Werkzeugen zählen auch die verwendeten Programmiersprachen sowie ihre Programmiersysteme. Aus diesem Grunde ist die Forschung in diesen Bereichen noch immer sehr interessant.

Um eine Programmiersprache zuverlässig bei der Entwicklung großer Softwareprojekte einsetzen zu können, sollte sie präzise definiert sein und ein korrekter Compiler oder Interpreter für sie existieren. Hierzu müssen die Beziehungen zwischen dem definierenden Dokument der Sprache (dem *Sprachreport*) und ihrer Implementierung auf Quelltextebene deutlich erkennbar sein. Die zur Definition verwendeten Formalismen müssen also auf sehr hoher Ebene in ein ausführbares Programm transformiert werden können.

Für die Formalismen, die sich in der Vergangenheit bewährt haben – besonders für die Beschreibung der syntaktischen Struktur –, ist eine Vielzahl von sogenannten *Compiler-Generatoren* (oder auch *Compiler-Compilern*) entwickelt worden. Von diesen basieren die meisten auf den traditionellen imperativen Sprachen, die aus Algol 60 [N+63] hervorgegangen sind. Neben den imperativen Sprachen wird jedoch auch an Sprachen geforscht, die auf anderen Paradigmen aufbauen, zum Beispiel rein funktionalen mit *lazy evaluation* [PJ88], logischen [War77] oder multiparadigmatischen Sprachen [Smo94]. Für diese müssen häufig neue Modelle gefunden werden.

Weitreichendere Werkzeugunterstützung ist daher zwar nur für imperative Sprachen verfügbar, aber gerade bei diesen klafft eine besonders große Lücke zwischen der Spezifikation und der Ausdrucksfähigkeit der Implementierungssprache. Vollkompositionale Sprachen

hingegen sind *per se* schon gut zur Umsetzung mathematisch basierter Formalismen geeignet. Einige Argumente für die besondere Eignung neuerer Programmiersprachkonzepte für den Compilerbau finden sich in [War80] und [JPB94].

Das Ziel dieser Arbeit ist es, die bewährte Compiler-Generator-Technologie um Mechanismen zu erweitern, die die Übersetzung moderner Sprachen erleichtern. Hierzu soll ein Werkzeug für die multiparadigmatische Sprache Oz zur Verfügung gestellt werden, die von dem Forschungsbereich Programmiersysteme in Saarbrücken entwickelt wird. Das entwickelte Werkzeug soll später für eine Reimplementierung des Oz-Compilers verwendet werden. Insbesondere Aufgaben wie die lexikalische Bindungsanalyse und die Reduktion in eine Kernsprache, die in traditionellen Werkzeugen selten automatisch erledigt werden, sollen unterstützt werden.

Im vorliegenden Kapitel sollen die genauen Anforderungen an das zu entwickelnde Werkzeug und dessen Spezifikationssprache erarbeitet werden. Vorab wird in den folgenden Abschnitten zum besseren Verständnis des Aufgabenumfeldes zunächst auf den Aufbau von Compilern sowie auf die Besonderheiten von Oz eingegangen.

1.2 Die Struktur von Compilern

Ein Compiler transformiert eine Eingabe, die in einer bestimmten *Quellsprache* vorliegt, in eine semantisch äquivalente Ausgabe in einer *Zielsprache*. Üblicherweise, aber nicht notwendigerweise, liegt die Eingabe in Form einer Zeichenfolge vor.

Diese Aufgabe wird traditionell in mehrere logische Phasen unterteilt. Sie bieten eine gute Möglichkeit, die – oft sehr komplexen – Compiler zu strukturieren, um sie leichter verständlich zu machen. Abbildung 1.1 zeigt das Standard-Compilermodell, das dieser Arbeit zugrunde liegt (abgeändert aus [ASU86, S. 10]). Dabei stellen Rechtecke Phasen, abgerundete Rechtecke Komponenten des Compilers dar; Pfeile deuten Datenfluß an, durchgezogene Striche die Verwendung einer Komponente.

Die Phasen können zwei Gruppen zugeordnet werden, wie dies in der Abbildung durch die gestrichelte Linie getan wurde. Das *Front-End* übernimmt die Analysephasen, also diejenigen, die nur von der Quellsprache, aber nicht (oder nur geringfügig) von der Zielsprache abhängen. Hierzu zählen die lexikalische, syntaktische und semantische Analyse sowie die Zwischencodeerzeugung, weiterhin die Fehlerbehandlung und die Symboltabellenverwaltung. Das *Back-End* hingegen erledigt die Syntheseaufgaben. Die Phasen in diesem Teil hängen sehr stark von der Zielsprache beziehungsweise deren Ausführungsmaschine ab.

In den folgenden Abschnitten werden die einzelnen Phasen und Komponenten kurz beschrieben. Die Implementierung des Back-Ends ist nicht Gegenstand der Untersuchung in der vorliegenden Arbeit, daher werden nur die Aufgaben des Front-Ends detaillierter beschrieben und nur für diese Möglichkeiten zum Einsatz von Compiler-Generatoren angegeben.

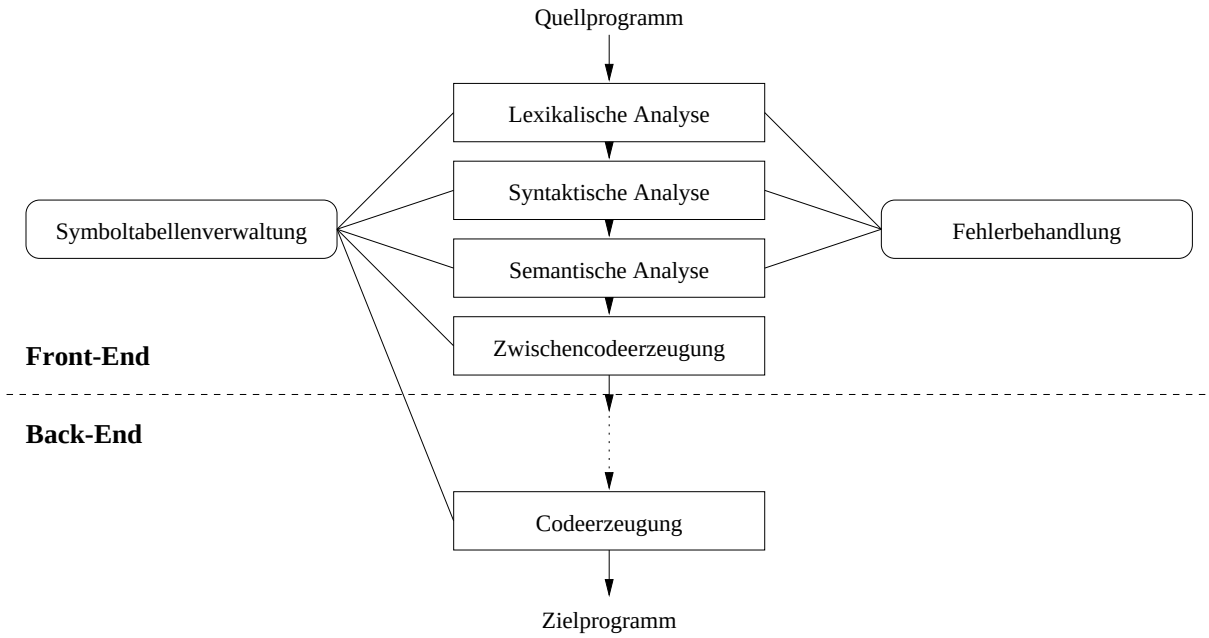

Abbildung 1.1: Die Phasen eines Compilers

1.2.1 Lexikalische Analyse

Die lexikalische Analyse, in der Literatur häufig auch *Scanning*, *Lexing*, *Tokenization* oder *lineare Analyse* genannt, faßt Teilfolgen des Eingabezeichenstroms zu *Token* zusammen, von denen jedes einen *Tokentyp* oder *Tokentag* zugeordnet bekommt (beispielsweise „Bezeichner“) und gegebenenfalls einen *Tokenwert* trägt (wie etwa die Repräsentation eines Bezeichners). Ausgabe ist der *Tokenstrom*, also eine Folge von Token.

Die genaue Darstellung der Tokentypen ist unbedeutend – wichtig ist nur, daß jeder Tokentyp eindeutig ist. Beliebige Folgen von Daten oder Ereignissen, die eindeutigen Tokentypen zugeordnet werden können, können ohne gesonderte lexikalische Analyse in die nächste Phase weitergereicht werden.

Für diese Phase gibt es gute Möglichkeiten der Werkzeugunterstützung: Gibt der Benutzer für jeden Tokentyp die Menge der Zeichenfolgen an, die ihm angehören, so kann aus diesen Informationen automatisch ein endlicher Automat erzeugt werden, der die Token erkennt. Solche Mengen können durch sogenannte *reguläre Ausdrücke* beschrieben werden.

1.2.2 Syntaktische Analyse

Diese Phase findet man in der Literatur auch unter den Bezeichnungen *Parsing* und *strukturelle* oder *hierarchische Analyse*. Entsprechend bestimmter Regeln werden Teilfolgen des Tokenstroms sukzessive durch neue Symbole ersetzt. Betrachtet man die ersetzten Symbole als Nachfolger des neuen, so wird der lineare Tokenstrom in einen *Syntaxbaum* (auch:

Strukturbaum) transformiert. Üblicherweise wird aber nur eine kompaktere Repräsentation davon aufgebaut, nämlich der *abstrakte Syntaxbaum*, der Ausgabe dieser Phase ist.

Auch für die Implementierung der syntaktischen Analyse ist Werkzeugunterstützung möglich, und zwar durch Angabe einer *kontextfreien Grammatik*. Der konkrete Syntaxbaum hängt von der genauen Formulierung der Grammatik ab. Das automatische Bestimmen einer geeigneten abstrakten Syntax aus der gegebenen Grammatik ist ein wichtiger Aspekt der Übersetzung, da hiervon die Komplexität der nachfolgenden Phasen abhängen kann. Da der verwendete Parsealgorithmus zusätzliche Einschränkungen an die Gestalt des konkreten Syntaxbaums bedingt, kann eine Darstellung, die von den Feinheiten in der Grammatik unabhängig ist, jedoch nicht ohne weitere Angaben durch den Benutzer bestimmt werden.

1.2.3 Semantische Analyse

In dieser Phase wird die Eingabe inhaltlich auf Validität überprüft werden. Hierzu müssen die durch den Sprachreport festgelegten Regeln formal für die gewählte abstrakte Syntax formuliert werden und bei einem Durchlauf des abstrakten Syntaxbaums angewendet werden. Dabei werden aus dem Kontext erschlossene Informationen in den Baumknoten gespeichert. Weiterhin können Konstrukte simplifiziert oder auf allgemeinere zurückgeführt werden. Ausgabe der Phase ist ein *annotierter abstrakter Syntaxbaum*.

Wenn in der Implementierungssprache komfortables *Matching* und Baum-Durchlaufsstrategien leicht formulierbar sind, läßt sich diese Phase einfacher umsetzen. Werkzeugunterstützung hat hier oft nur das Ziel, eine Sprache um derartige Funktionalität zu erweitern (*Pattern-Based Transformation Tools*, *Tree Walkers*). Auch für Simplifikationen ist die Bereitstellung von Hilfsmitteln interessant; denkbar ist die Angabe von Regeln, wie sie in Termersetzungssystemen Anwendung finden.

Lexikalische Bindungsanalyse

Für die semantische Analyse fast aller Sprachen ist es nötig, jedem Bezeichnerauftreten die zugehörigen Definition zuzuordnen. Dieser Schritt ist mit der unten beschriebenen Symboltabellenverwaltung eng gekoppelt und kann in den meisten Sprachen im großen und ganzen auf dieselbe Weise durch einer *lexikalischen Bindungsanalyse* gelöst werden. Werkzeugunterstützung würde hier bedeuten, daß einige häufig benötigte Funktionen einfach zur Verfügung gestellt werden können, wie das Erzeugen eines frischen (im Quelltext unbenutzten) Bezeichners und die konsistente Umbenennung aller Bezeichner, so daß keine zwei bindenden Auftreten eines selben Bezeichners vorkommen.

Dieses ist bei deklarativen Sprachen von Bedeutung, deren Semantikdefinition oftmals nicht das eingegebene Programm interpretiert, sondern dessen Quotient bezüglich Alpha-Renaming betrachtet. Als *Alpha-Renaming* bezeichnet man die konsistente Umbenennung eines gebundenen Bezeichners, also an den Orten seiner Definition sowie aller seiner Referenzen. Mit dieser Voraussetzung wird die Durchführung von Transformationen erleichtert,

da so von vorneherein garantiert ist, daß keine Bezeichnerkonflikte (sogenannte *name clashes*) auftreten können, solange keine Programmteile dupliziert werden.

1.2.4 Zwischencodeerzeugung

Oft wird aus der Repräsentation in abstrakter Syntax nicht direkt Code der Zielsprache generiert, sondern *Zwischencode*, der zwischen der Quell- und der Zielsprache angesiedelt ist. Dieser soll von der Zielsprache möglichst unabhängig sein, um spätere Übersetzungen in andere Zielsprachen zu erleichtern. In dieser Phase wird der annotierte abstrakte Syntaxbaum in die Zwischensprache übersetzt. Diese Transformation hängt sowohl von der abstrakten Syntax als auch von der Zielsprache ab und ist daher schwer durch Werkzeuge realisierbar.

1.2.5 Symboltabellenverwaltung

In der Symboltabelle werden während der Analyse Informationen zu den im Quelltext vorkommenden Bezeichnern gesammelt. Hierzu zählen beispielsweise der Ort ihrer Deklaration, die Definition des Objekts, das an den Bezeichner gebunden wird, oder sein Sichtbarkeitsbereich. Typische Operationen auf einer Symboltabelle sind das Eintragen eines Bezeichners mit seiner Definition oder das Nachsehen einer Definition in der aktuellen Umgebung.

1.2.6 Fehlerbehandlung

Im Falle eines Fehlers werden Informationen über diesen in lesbarer Form ausgegeben. Es muß entschieden werden, ob und wie mit dem Übersetzungsvorgang fortgefahren werden soll.

1.3 Die Programmiersprache Oz

In diesem Abschnitt werden einige der Eigenschaften der Programmiersprache Oz vorgestellt, die zum Verständnis der Arbeit wichtig sind, und ihre Bedeutung für den Compilerbau skizziert.

1.3.1 Logische Variablen

Eine logische Variable ist ein Platzhalter für einen (nicht änderbaren) Wert. Die Variable kann verwendet werden, bevor ihr Wert vollständig bekannt ist – im Gegensatz zu funktionalen Programmiersprachen, bei denen zum Zeitpunkt der Definition einer Variablen

bereits ihre komplette Berechnungsvorschrift angegeben werden muß, oder zu imperativen Sprachen, bei denen die Verwendung vor der Initialisierung meist einen Fehler mit unvorhersehbaren Konsequenzen darstellt.

Der Wert einer Variablen wird über *Constraints* spezifiziert. Beispielsweise kann bekannt sein, daß eine Variable X für einen Record mit bestimmten Teilbäumen steht. Deren Werte können jedoch noch unspezifiziert sein und sich erst aus späteren Berechnungen ergeben (oder gar nicht). Es gibt Konstrukte, die die das Entailment von Constraints prüfen können. Hierüber kann Matching von Werten realisiert werden, die logische Variablen enthalten, was in Compilern sehr häufig benötigt wird.

In einem Compiler-Front-End können logische Variablen nützlich eingesetzt werden: So könnten etwa im abstrakten Syntaxbaum logische Variablen stehen, die später die Annotationen aufnehmen, oder in der Symboltabelle könnten logische Variablen an den Stellen eingesetzt werden, wo noch nicht alle Informationen über einen Bezeichner bekannt sind – wie zum Beispiel im Falle eines deklarierten Bezeichners, dessen genaue Definition noch aussteht. Eine weitere Anwendung ist *Backpatching*: Wenn Code zu einem Zeitpunkt erzeugt werden muß, zu dem der Wert eines Operanden noch nicht feststeht, so kann über eine logische Variable ein Platzhalter eingefügt werden, der dann den endgültigen Wert aufnimmt.

1.3.2 Prozeduren höherer Ordnung

Oz unterstützt Prozeduren höherer Ordnung. Das bedeutet, daß zur Laufzeit *Closures* erzeugt (also Quantifizierungen über nicht-lokale Variablen vorgenommen) und Prozeduren Variablen zugewiesen werden können.

Prozeduren höherer Ordnung können im Compilerbau ebenso vielfältig eingesetzt werden wie in anderen Applikationen. Auf eine Anwendung, die vorgeschlagen worden ist, soll hier näher eingegangen werden.

Es gibt Parsetechniken, bei denen für den Aufbau eines Knotens des abstrakten Syntaxbaumes nur die Nachfolgerknoten referenziert werden können (also keine, die an anderen Stellen der Eingabe erzeugt werden). Dies schränkt Attributberechnungen auf sehr lokale Daten ein. Um diese Beschränkung aufzuheben, können Prozeduren höherer Ordnung verwendet werden [TA91], [May81]: Statt bei einer Knotenkonstruktion dessen Attributwerte sofort zu ermitteln, wird die Berechnungsvorschrift über die nicht verfügbaren Daten quantifiziert und eine prozedurale Abstraktion an der entsprechenden Stelle im abstrakten Syntaxbaum eingetragen. Zu dem Zeitpunkt, da die fehlenden Informationen zugreifbar sind, kann die Prozedur auf diese angewendet werden.

1.3.3 Nebenläufigkeit

Oz ist eine nebenläufige Sprache; es können also gleichzeitig mehrere Berechnungen in verschiedenen *Threads* durchgeführt werden.

Da die Struktur von Compilern eine einfache Kette von (eventuell verzahnten) Analyseschritten mit linearem Datenfluß ist, ist für die Grobstruktur eines Compiler-Front-Ends keine Nebenläufigkeit nötig. Threads können allerdings alternativ zu den obigen Prozeduren höherer Ordnung dazu verwendet werden, Berechnungen zurückzustellen, die noch nicht ausgeführt werden können.

1.3.4 Objektorientierte Programmierung

Ein weiteres Paradigma, das Oz unterstützt, ist die objektorientierte Programmierung. Klassen und Objekte werden zur Laufzeit erzeugt, wobei Mehrfachvererbung möglich ist. Es können private sowie öffentliche Methoden definiert werden und „befreundete“ Klassen vergleichbar zu C++ [Str87] existieren. Der Zustand eines Objekts kann in konstante (*Features*) und sich änderbare (*Attribute*) Anteile aufgeteilt werden; mehrere Objekte einer Klasse können sich dasselbe Feature teilen.

Objektorientierung kann im Compilerentwurf sehr gut eingesetzt werden. Sie ermöglicht es, die Compilerphasen gut zu strukturieren und mit klaren Schnittstellen zu versehen. Es ist also naheliegend, die von einem Compiler-Werkzeug generierten Programme durch Klassendefinitionen zu kapseln. Dann kann durch Vererbung das Verhalten einer Klasse modifiziert werden: Beispielsweise kann ein- und dieselbe Parserklasse sowohl zu einem Compiler-Front-End als auch zu einem Pretty-Printer erweitert werden, indem die Methoden überladen werden, die bei dem Erkennen grammatikalischer Regeln ausgeführt werden.

Weiterhin muß untersucht werden, welche Vorteile der Aufbau des abstrakten Syntaxbaumes aus Objekten bietet. Ein offensichtlicher Nachteil ist, daß Pattern-basierte Transformationsregeln nicht mehr so einfach durch existierende Konstrukte der Sprache formuliert werden können.

1.3.5 Namen

Viele Möglichkeiten der Sichtbarkeitsregelung, die im vorigen Abschnitt erwähnt wurden, werden in Oz durch sogenannte *Namen* ermöglicht. Es gibt eine Prozedur, die einen neuen, eindeutigen Namen generiert, der an eine logische Variable gebunden werden kann. Da Prozeduren, Methoden und Features durch Namen benannt werden können, sind alle Sichtbarkeitsregelungen realisierbar, die im Rahmen der lexikalischen Skopierung denkbar sind.

Die hierdurch gegebene Möglichkeit des *Information Hiding* [Par77] kann in Compilern – wie in allen Softwaresystemen – zugunsten einer besseren Wartbarkeit eingesetzt werden.

1.3.6 Die Ebenen der Definition von Oz

Die Sprachdefinition von Oz wird in mehreren Ebenen gegeben: Zuunterst wird *Kernel Oz* definiert [Smo94], wobei es sich um eine einfache, aber semantisch vollständige Teil-

sprache handelt. Dank ihrer Einfachheit ist sie axiomatisch definiert. Um eine komfortable Sprache zu erhalten, wird die *Oz-Notation* aufgesetzt [Hen95]. Die Konstrukte dieser Sprachebene können alle in Kernel Oz übersetzt werden. Auf der obersten Ebene kommen die *Standard Modules* [HMSW96] hinzu, die dem Benutzer Basisoperationen und häufig gebrauchte Funktionen zur Verfügung stellen.

Die Einteilung der Sprachdefinition in Ebenen bietet dem Sprachentwickler große Vorteile und macht dem Benutzer auch komplexe Sprachen leichter zugänglich. Kann diese Struktur auch für die Compilerkonstruktion übernommen werden, so bietet dies auch dem Compilerbauer Vorteile: Die Reduktionsregeln, die die Notation in Kernsprachenkonstrukte übersetzen, sind nur von der Grammatik, nicht aber von der abstrakten Syntax abhängig und müssen daher bei Modifikationen der letzteren nicht nachträglich angepaßt werden. Weiterhin erlaubt diese Definition inkrementelles Entwickeln und Testen des Compilers. Eine leichte Änderbarkeit dieser Regeln in der Implementierung hat für den Sprachentwickler wiederum den Vorteil, daß er sehr einfach mit neuen Notationen experimentieren kann.

1.4 Anforderungen an den Front-End-Generator

Nach dieser Übersicht können die Anforderungen an das Werkzeug formuliert werden. Zunächst wird festgelegt, welche Phasen genauer untersucht und – soweit der Umfang der Arbeit es zuläßt – unterstützt werden sollen:

Lexikalische Analyse. Diese Phase soll möglichst viel der Funktionalität verbreiteter Scanner-Generatoren bereitstellen, um das Portieren existierender Beschreibungen zu vereinfachen.

Syntaktische Analyse. Die Angabe der syntaktischen Regeln soll auf einem sehr hohen Niveau möglich sein. So sollen häufig auftretende syntaktische Muster abgekürzt werden können und von dem verwendeten Parsealgorithmus abstrahieren. Auf diese Weise soll eine einfache Portierbarkeit existierender Beschreibungen ermöglicht und eine hohe Übereinstimmung mit dem Sprachreport garantiert werden.

Lexikalische Bindungsanalyse. Die zur Durchführung dieser Phase notwendigen Informationen sollen direkt in der Grammatik angegeben werden können. Weiterhin soll die lexikalische Bindungsanalyse zu einer vollwertigen Symboltabellenverwaltung erweitert werden können, um die an dieser Stelle ohnehin schon vorhandenen Daten wiederverwenden zu können.

Reduktion in eine Kernsprache. Syntaktische Simplifikationen sollen durch dadurch unterstützt werden, daß Ersetzungsregeln angegeben werden können.

Aufbau einer abstrakten Syntax. Die Möglichkeit einer (semi-)automatischen Herleitung abstrakter Syntaxen aus der Grammatik soll bedacht werden.

Werkzeuge für Syntaxbaumtransformationen und Baumdurchläufe brauchen nicht ntersucht zu werden, da diese Aufgaben schon mit den in Oz verfügbaren Konstrukten leicht erledigt werden können.

Weiterhin werden einige Anforderungen an die Schnittstellen formuliert:

Unabhängige Verwendbarkeit. Die lexikalische Analyse einerseits und die Spezifikation der restlichen Phasen andererseits sollen unabhängig voneinander verwendet werden können. Dies erlaubt die Verwendung anderer Tokenquellen als den Scannergenerator und anderer Tokenkonsumenten als den Parsergenerator sowie eine freie Kombination mehrerer Scanner und Parser. Dazu ist eine klare Gliederung in den lexikalischen und den syntaktischen Anteil der Analyse erforderlich. Diese Trennung sollte auch die Übersichtlichkeit der Spezifikation erhöhen.

Hohe Integration. Sollen alle Werkzeuge genutzt werden, so müssen sie (trotz der geforderten klaren Trennung) gut integriert sein, um ohne Umstände aneinander angebunden werden zu können. Hierzu werden einfache, klare Schnittstellen zwischen den Einzelteilen gefordert, deren Kopplung durch das Werkzeug erfolgt.

Einbettung in Oz. Die Spezifikationssprache soll gut in Oz eingebettet sein, um die Ausdrucksfähigkeit dieser Sprache voll ausnutzen zu können.

Information Hiding. Das generierte Analyseprogramm soll von den Möglichkeiten der Abstraktion und Sichtbarkeitsregelungen, die Oz bietet, Gebrauch machen. Es bietet sich an, das Programm in eine Klasse einzubetten.

Interaktive Systeme. Das Werkzeug soll für interaktive Systeme verwendet werden können. Daher müssen die von der syntaktischen Analyse abhängigen Phasen verschränkt mit dieser ablaufen können.

1.4.1 Gliederung der Arbeit

Die Anforderungen haben festgelegt, welche Teile der zu entwickelnden Spezifikationssprache unabhängig voneinander sind. Entsprechend behandeln die Kapitel 2 bis 4 in dieser Reihenfolge die lexikalische Analyse, die syntaktische Analyse und deren Erweiterungen. Der Schwerpunkt liegt zunächst auf der Untersuchung und Bewertung verwandter Arbeiten, woraufhin jeweils die Entwurfsentscheidungen für den zu entwickelnden Front-End-Generator getroffen werden.

Nachdem die detaillierten Anforderungen an die Spezifikationssprache feststehen, befaßt sich das darauffolgende Kapitel mit ihrem Entwurf und der Umsetzung in ein Werkzeug. Die Bestandteile werden einzeln besprochen und ihr Zusammenspiel erläutert. Weiterhin wird ein Bootstrapping-Prozeß vorgestellt, der es ermöglicht, das System unter Verwendung seiner selbst zu implementieren. Für Leser, die nur am Ergebnis der Arbeit interessiert sind, ist Kapitel 5 wohl das wichtigste.

Kapitel 6 faßt die gesammelten Erkenntnisse zusammen und bewertet das neue Werkzeug. Dabei wird auf die Anforderungen aus Abschnitt 1.4 bezug genommen. Am Ende des Kapitels wird ein Ausblick auf eventuelle weiterführende Untersuchungen gegeben.

Kapitel 2

Lexikalische Analyse – Scannerdefinition

Die *lexikalische Analyse* ist der erste Schritt der strukturellen Analyse, in dem der eingegebene Zeichenstrom in einen Tokenstrom gewandelt wird. Da dieser von Parser konsumiert wird, wird die lexikalische Analyse oft als ein Klient-Prozeß der syntaktischen Analyse angesehen, vom dem der Parser die Token sukzessive anfordert; manchmal jedoch wird die Tokenisierung vollständig durchgeführt, bevor der Parsevorgang gestartet wird.

Die Aufgabe des Scanners besteht aus zwei Teilen. Der eine liest den Zeichenstrom und bestimmt das Ende der nächsten zu bearbeitenden Zeichenfolge, der andere generiert hieraus Token, die an den Tokenstrom angehängt werden. Hierbei können noch Präprozessoraufgaben wie Datei-Inklusion (beispielsweise in Oz über `\insert`), konditionale Compilierung oder Makroexpansion erledigt werden. Weiterhin können Symboltabelleneinträge erzeugt oder erfragt werden. Dies ist für Sprachen nötig, bei denen sich die Tokenklasse ein- und derselben Zeichenkette im Laufe der Analyse ändert. Ein Beispiel hierfür ist das Einführen eines neuen Typnamens in C und C++ über `typedef`, dessen Tokenklasse sich ab dem Zeitpunkt der Definition von ‚Bezeichner' in ‚Typname' ändert. Ist eine solche Anpassung nicht möglich, dann kann die Sprache nicht eindeutig geparst werden.

Entsprechend den beiden Teilaufgaben ist dieses Kapitel gegliedert: In Abschnitt 2.1 wird das Erkennen einer Zeichenfolge beschrieben, in Abschnitt 2.2 die Tokenerzeugung. Dabei werden die Lösungsansätze existierender Werkzeuge untersucht. Im letzten Abschnitt (2.3) wird festgelegt, wie der zu definierende Scannergenerator die Aufgaben angehen soll. Dessen Spezifikationssprache und die Schnittstellen werden in Abschnitt 5.2 definiert.

Einige der verwendeten Bezeichnungen sind aus [KVE94] übernommen.

2.1 Spezifikation der Eingabesprachen

Wie schon im Einleitungskapitel erwähnt wurde, wird der Erkennungsteil eines Scanners dadurch beschrieben, daß für jede Tokenklasse die Menge der ihr angehörenden Zeichenfol-

gen angegeben wird. Entsprechend diesen Angaben werden der eingegebene Zeichenstrom zerlegt und Tokenklassen zugeordnet. Zum Beispiel würde in Oz die Eingabe

```
case␣Xs␣of␣'|'(X1␣Xr)␣then...
```

in 14 Teile zerlegt: das Schlüsselwort `case`, ein Leerzeichen ␣, die Variable `Xs`, ein Leerzeichen, das Schlüsselwort `of`, ein Leerzeichen, das Atom `'|'` und so weiter.

Technisch wird die Zerlegung durch einen *endlichen Automaten* realisiert. Die Menge aller möglichen Eingabezeichen wird *Vokabular* oder *Alphabet* genannt, eine Menge von Zeichenfolgen (beispielsweise aller derjenigen, die einem Tokentyp angehören) wird als eine *formale Sprache* über diesem Vokabular bezeichnet. Solche Mengen können durch *reguläre Ausdrücke* beschrieben werden, die in Abschnitt 2.1.1 definiert werden. Ist eine Zeichenfolge in der durch einen regulären Ausdruck beschriebenen Menge enthalten, so spricht man von einem *Match*. Die Zeichenfolge heißt dann *Satz* der zugehörigen formalen Sprache oder auch – im Kontext von Scannern – das gematchte *Lexem*.

In Abschnitt 2.1.2 wird beschrieben, wie Scanner für Sprachen realisiert werden können, die in verschiedenen Teilsprachen andere lexikalische Regeln verwendetn. Hierzu wird das Konzept des *lexikalischen Modus'* eingeführt.

In der Praxis stellen die angegebenen regulären Ausdrücke keine Partitionierung der Menge aller Zeichenfolgen dar, die über dem Vokabular gebildet werden können. Zwei Fälle müssen dabei bedacht werden: Zum einen müssen Eingaben, die keinem regulären Ausdruck zugeordnet werden können, als fehlerhaft gemeldet werden, zum anderen können nichtleere Schnittmengen zwischen den Mengen zweier Tokenklassen existieren. Wann letzteres einen Fehler darstellt beziehungsweise wie derartige Konflikte aufgelöst werden können, wird in Abschnitt 2.1.3 untersucht.

2.1.1 Reguläre Ausdrücke

Reguläre Ausdrücke, die eine knappe Notation für die sogenannten *regulären Sprachen* darstellen (Grammatiken vom *Typ 3* in der Chomsky-Hierarchie [GW85]), sind ein Spezialfall der kontextfreien Sprachen (*Typ 2*) und zuerst von Kleene [Kle56] untersucht worden. Sie haben die Eigenschaft, daß sie von einem endlichen Automaten erkannt werden können. Algorithmen für die Erzeugung eines solchen Automaten finden sich beispielsweise in [ASU86, S. 113–146] oder [Gro87].

Üblicherweise wird zur Scanner-Erzeugung ein Satz regulärer Ausdrücke angegeben. Nur in seltenen Fällen werden dazu andere Techniken verwendet; Gegenbeispiele sind der Scanner- und Parsergenerator *Mango* [Age94] für die Programmiersprache Self sowie die Grammatik-Spezifikationssprache *GRAMOL*, bei denen die Techniken Anwendung finden, die hier in Kapitel 3 vorgestellt werden sollen.

Meist werden alle benötigten regulären Ausdrücke vom Benutzer angegeben. Das Werkzeug TXL [CCH95] ist jedoch so aufgebaut, daß ein Default-Satz an Regeln vorgegeben ist,

die teilweise modifiziert werden können. Eine Standardregel besagt beispielsweise, welche Zeichen in Bezeichnern verwendet werden dürfen; möchte man diese Zeichenmenge erweitern, so müssen die weiteren Zeichen über einen Kommandozeilenparameter angegeben werden.

In den existierenden Scanner-Generatoren sind viele verschiedene konkrete Syntaxen für reguläre Ausdrücke zu finden. Die verbreiteteste ist die von *Lex* [Les75] und dem abgeleiteten Werkzeug *flex* [Pax95]). Daher werden hier dessen Schreibweisen übernommen, um eine einfache Portierung von Spezifikationen zu ermöglichen. Alternativen findet man beispielsweise in [KVE94], [Dor96], [And95], [Gro92], [CCH95] und [GS88].

Basisausdrücke

Zunächst sollen die Basiskonstrukte dargestellt werden, aus denen reguläre Ausdrücke aufgebaut werden können. Der nachfolgende Abschnitt erweitert diese dann zu einer ausdrucksstärkeren Notation. Dort werden einige praxisrelevante Aspekte einbezogen.

Definition: Eine endliche nichtleere Menge $\mathcal{V}$ heißt **Vokabular**. Eine **Folge** der Länge n von Symbolen aus $\mathcal{V}$ ist eine Abbildung $\{1, \ldots, n\} \longrightarrow \mathcal{V}$. Die Menge aller solcher Folgen der Länge n wird mit $\mathcal{V}^n$ bezeichnet; die **leere Folge** wird durch ε notiert. Der **Kleene-Abschluß** von $\mathcal{V}$ wird definiert durch: $\mathcal{V}^* := \bigcup_{n \geq 0} \mathcal{V}^n$. □

Definition: Sei $\mathcal{V}$ Vokabular. Ein **regulärer Ausdruck** r und die formale Sprache (Menge der Folgen) $L(r)$, für die er steht, werden induktiv definiert durch:

- $r \equiv \varepsilon$ ist regulärer Ausdruck mit $L(r) = \{\varepsilon\}$.
- Sei $x \in \mathcal{V}$. Dann ist $r \equiv x$ regulärer Ausdruck mit $L(r) = \{x\}$.
- Sei s regulärer Ausdruck. Dann ist $r \equiv s*$ regulärer Ausdruck mit $L(r) = L(s)^*$.
- Sei s regulärer Ausdruck. Dann ist $r \equiv (s)$ regulärer Ausdruck mit $L(r) = L(s)$.
- Seien s und t reguläre Ausdrücke. Dann ist $r \equiv st$ regulärer Ausdruck mit $L(r) = \{uv \mid u \in s \wedge v \in t\}$.
- Seien s und t reguläre Ausdrücke. Dann ist $r \equiv s|t$ regulärer Ausdruck mit $L(r) = L(s) \cup L(t)$.

□

Einige Beispiele für reguläre Ausdrücke zeigt Tabelle 2.1.

Da reguläre Ausdrücke mit dem üblichen ASCII-Zeichensatz notierbar sein müssen und Metazeichen von Symbolen des Vokabulars unterscheidbar sein müssen, bestehen einige

r	$L(r)$
`a`	{`a`}
`ab`	{`ab`}
`a|b`	{`a`, `b`}
`a*`	{ε, `a`, `aa`, ...}
`ab*`	{`a`, `ab`, `abb`, ...}
`(a|b)*`	{ε, `a`, `b`, `ab`, `ba`, ...}

Tabelle 2.1: Beispiele für reguläre Basisausdrücke

Abweichungen zwischen der obigen Definition und der tatsächlichen Syntax. Zum einen gibt es die Möglichkeit, durch Anführungszeichen oder Fluchtsymbole auch Metazeichen als Symbole zu verwenden. Wird weiterhin ein regulärer Ausdruck $r \not\equiv \varepsilon$ notiert, so werden enthaltene ε-Zeichen üblicherweise weggelassen, zum Beispiel wird der Ausdruck `a(b|`ε`)`, der die Menge {`a`, `ab`} beschreibt, zu `a(b|)`.

Erweiterte Ausdrücke

Um reguläre Ausdrücke mächtiger beziehungsweise ausdrucksstärker zu machen, werden häufig einige der im folgenden vorgestellten Erweiterungen implementiert.

Zu den Konstrukten, die reguläre Ausdrücke effektiv mächtiger machen, zählen die *Zeilenstart-* und die *Zeilenendbedingung* sowie der *Nachfolgekontext* (*trailing context*).

- Wird einem regulären Ausdruck das Zeichen `^` vorangestellt, so wird er nur gematcht, wenn die aktuelle Position im Eingabezeichenstrom genau auf einen Zeilenanfang fällt (also entweder das erste Zeichen der Eingabe oder das auf ein Newline-Zeichen folgende Zeichen). Manche Scannergeneratoren (wie beispielsweise *lx* [Dor96] für Haskell) verallgemeinern dieses Konstrukt auf eine beliebige Zeichenmenge anstelle von Newline.

  Dieses Konstrukt wird beispielsweise benötigt, wenn – wie bei einigen C-Präprozessoren – Direktiven am Beginn einer Zeile stehen müssen (etwa das Doppelkreuz von `#include`).

- Werden zwei reguläre Ausdrücke s und t durch das Zeichen `/` verbunden, so wird ein s gematcht, aber nur, wenn der darauffolgende Text des Eingabestroms mit t matcht (Nachfolgekontext).

  Die Recordlabel in Oz sind ein Beispiel, wo dieses Konstrukt benötigt wird: Ein Atom oder eine Variable wird nur dann als Recordlabel erkannt, wenn direkt auf den Bezeichner eine öffnende runde Klammer folgt. Diese ist aber nicht Teil des Lexems.

- Die Zeilenendbedingung, die durch Anhängen des Zeichens `$` an einen regulären Ausdruck r notiert wird, ist eine Abkürzung für den regulären Ausdruck r mit einem Newline-Zeichen als Nachfolgekontext.

Andere Erweiterungen dienen nur einer kompakteren Schreibweise. Hierfür gibt es Zeichenmengen, Komplemente von Zeichenmengen, literale Zeichenketten, obligatorische Wiederholungen, Optionen, (i bis j)-fache Wiederholungen und Vereinbarungen benannter regulärer Ausdrücke (vergleichbar Makros).

Einige Beispiele für erweiterte reguläre Ausdrücke zeigt Tabelle 2.2. Hierbei wurde $\mathcal{V} = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ festgelegt.

r	$L(r)$
`[ab]`	$\{\texttt{a}, \texttt{b}\}$
`[^ab]`	$\{\texttt{c}\}$
`a+`	$\{\texttt{a}, \texttt{aa}, \ldots\}$
`a?`	$\{\varepsilon, \texttt{a}\}$
`a{2,4}`	$\{\texttt{aa}, \texttt{aaa}, \texttt{aaaa}\}$

Tabelle 2.2: Beispiele für erweiterte reguläre Ausdrücke

Zuletzt sollte noch ein spezieller regulärer Ausdruck erwähnt werden, nämlich die *End-Of-File*-Regel, notiert durch `<<EOF>>`. Diese wird nur durch das Ende der Eingabe gematcht und muß als Spezialfall gehandhabt werden.

2.1.2 Lexikalische Modi

Viele Sprachen besitzen eingebettete Teilsprachen, für die andere lexikalische Konventionen gelten. Beispiele hierfür sind Compilerdirektiven oder semantische Aktionen in Compilerbauwerkzeugen, die in der Zielsprache notiert werden und in eine Metasprache eingebettet sind. Um abweichende lexikalische Konventionen der Teilsprachen auseinanderhalten zu können, bieten mehrere Werkzeuge *lexikalische Modi* [And95] an, auch genannt *lexical classes* [PDC91], *start conditions* oder *start states* [Les75]. Prinzipiell wird für jeden lexikalischen Modus (und jeden Vorkontext, wie zum Beispiel `^`) ein eigener Erkenner generiert (wobei diese gemeinsame Zustände haben können).

Das Prinzip ist einfach: Jeder reguläre Ausdruck wird mit der Menge der lexikalischen Modi annotiert, in denen er gematcht werden kann (eventuell auch indirekt durch deren Komplement, wie dies bei dem Scannergenerator *Rex* [Gro92] möglich ist). Zu jedem Zeitpunkt befindet sich der Scanner in genau einem dieser Modi. Beim Erkennen bestimmter Token kann in einen anderen lexikalische Modus umgeschaltet werden. Zum Beispiel können C-Kommentare [KR78, S. 197] dadurch gescannt werden, daß bei Erkennen der Startsequenz `/*` in einen Kommentarmodus geschaltet wird, in dem der Kommentartext bis einschließlich der Sequenz `*/` überlesen wird. Daraufhin wird wieder in den ursprünglichen Modus zurückgekehrt.

Eine Möglichkeit, die Annotation der regulären Ausdrücke mit ihren zugehörigen gültigen Modi übersichtlicher zu machen, ist die *Vererbung* von lexikalischen Modi. Erbt ein

Modus B von einem Modus A, so matchen im Modus B nur die mit B präfixierten regulären Ausdrücke; im Modus A jedoch alle mit A oder B annotierten. (Die Vererbung von `<<EOF>>` erfährt hierbei wieder eine Spezialbehandlung.)

Mit *Lex* ist Vererbung nur sehr eingeschränkt möglich: Diejenigen regulären Ausdrücke ohne explizite Modusangabe werden als allen Modi angehörig angesehen, hierunter auch dem vordefinierten initialen Modus namens `INITIAL`.

Mit *flex* sowie in der POSIX-*Lex*-Spezifikation ist es möglich, bei der Definition eines lexikalischen Modus alle nicht-annotierten regulären Ausdrücke entweder zu erben (*inclusive*) oder nicht zu erben (*exclusive start condition*). Ab der Version 2.5.1 von *flex* können *start condition scopes* definiert werden, die es ermöglichen, eine Gruppe regulärer Ausdrücke gemeinsam einem lexikalischen Modus zuzuordnen; Vererbung ist hier durch Schachteln solcher Scopi realisierbar.

Ein anderes Werkzeug, das die Vererbung lexikalischer Modi erlaubt, ist *SAGA* [And95], der Parsergenerator für die multiparadigmatische Sprache AKL [Jan94]. Dort wird sie über eine Direktive der Form `:- inherits(`*derived*`,`*bases*`)` spezifiziert.

2.1.3 Auflösung von Konflikten

Es kann der Fall eintreten, daß zwei Sprachen, die mithilfe von regulären Ausdrücken definiert wurden, eine nichtleere Schnittmenge besitzen. Beispielsweise gilt:

$$L(\texttt{(a|b)*}) \cap L(\texttt{(a|c)*}) = L(\texttt{a*})$$

Taucht in der Eingabe also eine Folge von ‚`a`'s auf, so kann nicht entschieden werden, welche Regel anzuwenden ist. In diesem Fall spricht man von einem *Konflikt*. In einer typischen Sprachdefinition treten viele solche Konflikte auf; Schlüsselwörter sind zum Beispiel oft ausgezeichnete Bezeichner. Da es umständlich und unübersichtlich ist, reguläre Ausdrücke konfliktfrei zu formulieren, gibt es einen Satz von Regeln, mit denen Konflikte aufgelöst werden.

Als erstes Kriterium wird die *longest-match*-Regel angewendet: Gibt es zwei Matches für ein Präfix der Eingabe, so wird derjenige vorgezogen, der den längeren Text einbezieht (wobei die Längen des Lexems und des Nachfolgekontextes summiert werden). Die Umsetzung dieser Regel ist zwingend, sonst könnte beispielsweise bei einen regulären Ausdruck `aa*` nach dem ersten gelesenen `a` mit einem Match abgebrochen werden.

Die durch die *longest-match*-Regel definierte Partialordnung auf den Matches wird weiterhin entweder mit der *first-fit*- oder mit der *best-fit*-Regel lexikographisch kombiniert:

First-Fit. Matchen zwei reguläre Ausdrücke denselben Präfix der Eingabe, so wird der zuerst notierte der Ausdrücke ausgewählt. (Es sind natürlich auch andere Prioritätenvergaben denkbar als die rein absteigende Ordnung in Reihenfolge der Notation.)

R	r	S	s
{A}	`begin`	{A}	`[a-z]+`
{A}	`a`	{A, B}	`a`
{A}	`a`	{A, B}	`[a-z]+`

Tabelle 2.3: Beispiele für r spezieller als s

Diese Strategie wird von den meisten Werkzeugen (unter anderem von *Lex*, *Rex* aus Cocktail und *DLG* aus *PCCTS* [PDC91]) verfolgt. Hierbei können sämtliche Uneindeutigkeiten aufgelöst werden, es können aber auch einzelne reguläre Ausdrücke vollständig von den anderen verdeckt werden und somit nie zur Anwendung kommen.

Best-Fit. Angenommen, zwei reguläre Ausdrücke r und s matchen denselben Präfix der Eingabe. Seien R und S die Mengen der lexikalischen Modi, mit denen respektive r beziehungsweise s annotiert worden sind (nachdem Vererbungen explizit gemacht worden sind). Dann wird r genau dann gegenüber s vorgezogen, wenn $R \subseteq S \wedge L(r) \subseteq L(s)$ gilt – mit anderen Worten: wenn alle Präfixe der Eingabe, die von r gematcht werden, auch von s gematcht werden, s aber auch zusätzliche Präfixe abdeckt. r ist also echt *spezieller* als s. Einige Beispiele hierfür finden sich in Tabelle 2.3. Hat die hierdurch definierte Partialordnung für eine Eingabe kein kleinstes Element so wird ein Fehler in der Spezifikation gemeldet.

Diese Strategie findet man beispielsweise in *SAGA*. Ihr Vorteil besteht darin, daß die Reihenfolge der regulären Ausdrücke in der Spezifikation nicht mehr ausschlaggebend ist. Haben zwei Sprachen eine Schnittmenge, aber keine ist echte Teilmenge einer anderen, so muß die Schnittmenge durch andere Ausdrücke abgedeckt werden. Die Vererbung der Modi bewirkt hierbei übrigens, daß ein Ausdruck in einem abgeleiteten Modus demselben Ausdruck in den Basismodi vorgezogen wird. Realisiert werden kann diese Regel, indem die akzeptierenden Zustände des (deterministisch gemachten und minimierten) generierten Automaten betrachtet werden: Die Mengen der akzeptierenden Zustände müssen für alle regulären Ausdrücke paarweise auf die Teilmengeneigenschaft getestet werden.

Verwendet die Scannerdefinition kontextfreie statt regulärer Grammatiken, so sind diese Kriterien nicht ohne weiteres anwendbar. Eine Lösung für die sich daraus ergebenden Probleme findet man in der Sprache *GRAMOL*. Dort kann auch die Ada [Ada83]-Uneindeutigkeit bezüglich der Unterscheidung zwischen Quotes und Zeichenliteralen elegant gelöst werden.

2.2 Möglichkeiten der Tokenerzeugung

Wurden die Matches für die Präfixe der Eingabe bestimmt und einer von ihnen ausgewählt, so liegt lediglich die Information vor, welcher reguläre Ausdruck gematcht wurde und welche

Länge der betreffende Präfix hatte. Hieraus müssen nun Tokentyp und -wert bestimmt werden.

Es können drei Vorgehensweisen unterschieden werden:

- Die einfachste Möglichkeit, die dem Benutzer wenig Einfluß auf die Tokenerzeugung läßt, besteht darin, mit jedem regulären Ausdruck direkt einen Tokentyp zu assoziieren. Als Tokenwerte stehen dann meist nur das Lexem und eventuell dessen Position im Quelltext (Dateiname, Zeilennummer und eventuell die Spaltennummer, auch *Koordinaten* genannt) zur Verfügung. Sofern keine speziellen Ausnahmereglungen vorgesehen sind, können keine Kontextinformationen einbezogen werden und der Match nicht nachbearbeitet werden. Umschalten von lexikalischen Modi kann ausschließlich von dem gematchten Ausdruck abhängig gemacht werden (in SAGA wird dies beispielsweise durch eine angehängte, anstelle einer präfixierten, Annotation eines lexikalischen Modus' notiert).

  Diese Strategie wird beispielsweise von TXL [CCH95], SAGA, dem Scannergenerator für W-Lisp [Küh94] und dem von *Eli* [Com96a] verfolgt. Daß dieses Vorgehen in den betreffenden Tools als zu einschränkend empfunden worden ist, zeigt, daß hier nach anderen Mechanismen gesucht wurde, um die Mächtigkeit zu erhöhen: In SAGA wurden *Filter* eingeführt (die unten noch näher erläutert werden); der Scannergenerator für W-Lisp ermöglicht es, das Lexem textuell durch Einfügungen oder Löschungen zu modifizieren. Im Scannergenerator von Eli können *Auxiliary Scanners* und *Token Processors* durch die Namen von C-Funktionen angegeben werden.

- Bei der zweiten Möglichkeit wird für jeden regulären Ausdruck der Rumpf einer Funktion in der Zielsprache des Scannergenerators angegeben, genannt *semantische Aktion*. Nun können für die Berechnung des Tokentyps und -wertes beliebig komplizierte Ausdrücke ausgewertet werden. Das Token wird durch den Rückgabewert der Funktion gegeben. Soll kein Token zurückgegeben werden, so wird kein Rückkehrbefehl ausgeführt (in imperativen Sprachen, wie C im Falle von *Lex*) oder die generierte Scannerfunktion noch einmal rekursiv aufgerufen (wie im Scannergenerator ML-Lex für Standard ML [AMT92]).

  Nicht nur, was die Berechnung von Tokentyp und -wert angeht, ist dieses Verfahren mächtiger: In der semantischen Aktion kann auch der Match selbst manipuliert werden, wodurch das Folgeverhalten des Automaten beeinflußt wird. Beispielsweise kann der Match verkürzt werden, wobei die überschüssigen Zeichen wieder in den Eingabestrom zurückgegeben werden, es können zusätzliche Zeichen angefordert werden, der Match kann abgelehnt werden (wonach der nächstbeste Match gewählt wird) oder der Wert des Zeilenstartflags kann modifiziert werden.

- Die dritte Möglichkeit ist der zweiten sehr ähnlich. Sie erlaubt es aber gegebenenfalls, mehrere Token zurückzuliefern statt nur eines oder keines. Hierdurch kann zum Beispiel Makroexpansion realisiert werden oder literale Zeichenketten können als Listen von ganzen Zahlen geliefert werden, wie dies in der Oz-Definition getan wird.

Die Idee besteht darin, den generierten Scanner in eine Klassendefinition zu kapseln und jede semantische Aktion als Methode aufzufassen. Eine vordefinierte Methode dient dabei dazu, ein Token an den Tokenstrom anzuhängen; diese kann beliebig oft pro Aktion ausgeführt werden. Die Interaktion mit dem Automaten wird ebenfalls über Methodenapplikationen realisiert.

Bei der Auswahl der Strategie muß bedacht werden, daß die restliche syntaktische Analyse vereinfacht werden kann, wenn der Scanner Informationen aus dem Kontext einbeziehen kann; zudem ist dies sogar manchmal eine Vorbedingung für das Parsen. Da die semantische Analyse ohnehin die kompliziertesteste Phase ist, sollte ihr möglichst viel Arbeit von den semantischen Aktionen des Scanners abgenommen werden, indem auch die Tokenwerte keiner Nachbearbeitung mehr bedürfen.

Ein anderer Vorteil ist, daß das Interface zum Scanner (der Tokenstrom) einfacher und präziser zu definieren ist, wenn die Token bereits fertig verarbeitet worden sind. In Oz beispielsweise brauchen nichtgequotete und gequotete Atome nicht mehr unterschieden zu werden, wenn die Escape-Zeichen der letzteren bereits umgewandelt worden sind. Weitere Beispiele für anfallende nichttriviale Aufgaben werden in [Gro88b] angeführt.

Die Tendenz scheint dahin zu gehen, daß in Sprachen, bei denen die Scannerspezifikation mit den Sprachen für andere Werkzeugteile integriert wird, einfache Tokenerzeugungsmethoden gewählt werden. Dies kann daran liegen, daß hier versucht wird, jede zusätzliche Arbeit aus dem Bereich, den das Werkzeug abdeckt, herauszubewegen, oder auch daran, daß bei der Entwicklung mehr Wert auf die nachfolgenden Phasen gelegt wurde. Man sollte aber bedenken, daß die einfachen Tokenerzeugungsmöglichkeiten von den mächtigeren leicht und fast ebenso übersichtlich emuliert werden können und daß das zusätzliche Angebot – da es die Effizienz nicht beeinträchtigt – nur von Vorteil sein kann.

2.2.1 Filter

Einige Werkzeuge, wie beispielsweise SAGA oder Mango, bieten als Zwischenstufe zwischen Scanner und Parser noch eine *Filter*-Phase an, wie dies im Datenflußdiagramm in Abbildung 2.1 gezeigt wird. Diese durchsucht den Tokenstrom nach bestimmten Patterns (also Teilfolgen mit bestimmten Tokentypen) und erlaubt es, die entsprechenden Stellen nachzubearbeiten, indem sie durch neu erzeugte Tokenfolgen ersetzt werden. Auf diese Weise können beispielsweise Leerzeichen und Kommentare eliminiert oder Makroexpansionen durchgeführt werden.

Abbildung 2.1: Datenflußdiagramm bei Filterverwendung

Dieses Schema ist, gekoppelt mit einer der obigen Möglichkeiten der Tokenerzeugung, nicht mächtiger als die dritte Strategie: Die Methode, die ein Token an den Tokenstrom anhängt, kann gegebenenfalls überladen werden und durch eine ersetzt werden, die einen größeren Kontext erfaßt. Die einfachste Tokenerzeugungsstrategie ist, wenn Filter definiert werden können, aber durchaus mächtiger als das zweite vorgestellte Verfahren.

Die dritte Strategie von oben ist jedoch für viele Anwendungen übersichtlicher: Es brauchen keine zusätzlichen Tokentypen eingeführt zu werden, die lediglich die eine oder andere Nachbearbeitung erzwingen, wenn diese direkt in der zugehörigen Methode erledigt werden kann.

2.2.2 Interner Zustand

Ein anderer Aspekt von Scannern mit benutzerdefinierten semantischen Aktionen ist die Möglichkeit, einen internen Zustand zu verwalten. Hierbei kann es sich beispielsweise um eine Symboltabelle oder um den aktuellen lexikalischen Modus handeln. In *Lex* ist dies aufgrund der Zielsprache (C beziehungsweise C++) einfach möglich. Soll mit *flex* jedoch ein reentranter Scanner erzeugt werden, so muß der interne Zustand als Argument für die Tokenanfragefunktion immer mit übergeben werden. (Ein *reentranter* Scanner kann von mehreren Programmteilen gleichzeitig für das Scannen verschiedener Eingabetexte verwendet werden.) Ähnlich wird dies auch in Scannergeneratoren mit funktionalen Zielsprachen, wie ML-Lex, gehandhabt; SAGA erledigt dies über die globale Durchfädelung von Akkumulatoren.

Diese Weise scheint sehr umständlich. Wird der generierte Scanner allerdings in eine Klasse eingebettet, so ist die Verwaltung eines internen Zustands leicht durch die Definition zusätzlicher Attribute möglich.

2.3 Entwurfsentscheidungen

Aus der vorangegangenen Analyse existierender Werkzeuge und Methoden sowie den in Abschnitt 1.4 gegebenen Anforderungen werden nun die Entwurfsentscheidungen für den Scannerteil des Front-End-Generators für Oz getroffen:

- Die Syntax für reguläre Ausdrücke (ausgenommen die lexikalischen Modi) wird von *flex* übernommen. Sie ist (mit Ausnahme variablen Vorkontextes) die mächtigste und auf jeden Fall die verbreiteteste. Somit wird zum einen eine einfache Portierbarkeit existierender Beschreibungen möglich und zum anderen die Erlernbarkeit der Sprache verbessert.

- Für die Definition lexikalischer Modi wird das Prinzip der *start condition scopes* von *flex* ab Version 2.5.1 übernommen. Durch einen solchen Scopus wird der entsprechende lexikalische Modus implizit deklariert. Dabei kann angegeben werden, von welchen anderen lexikalischen Modi er erbt.

- Die Identifikation der lexikalischen Modi erfolgt durch Variablen. Diese werden in lokale Variablen der generierten Klasse umgesetzt, die an ganze Zahlen gebunden werden.
- Für die Konfliktauflösung kann zwischen den beiden Strategien *first-fit* und *best-fit* durch Compilerdirektiven gewählt werden.
- Für die semantischen Aktionen wird die methodenbasierte Möglichkeit gewählt. Ein großer Teil der Möglichkeiten von *flex*, den Automaten zu steuern, soll als vordefinierte Methoden zur Verfügung gestellt werden.
- Da die Methode, die ein Token an den Tokenstrom anhängt, leicht überladen werden kann, ist kein expliziter Mechanismus zur Filterrealisierung vorgesehen. Dieser kann aber bei Bedarf von Hand implementiert werden.
- Die Identifikation der Tokenklassen erfolgt durch Oz-Atome. Damit sind beliebige Bezeichnungen wählbar, insbesondere solche, die Satz- oder Sonderzeichen enthalten. Es muß keine Umsetzung in ganze Zahlen (wie beispielsweise bei *Lex*) vom Benutzer vorgenommen werden und die Tokentypen sind im Quelltext zugreifbar (sogar symbolisch errechenbar).

Die konkrete Syntax, die Schnittstellen und einige Bemerkungen über die Realisierung werden in Abschnitt 5.2 vorgestellt.

Kapitel 3

Syntaktische Analyse – Parserdefinition

Im vorliegenden Kapitel wird die Spezifikationssprache des Werkzeugs entwickelt, das die *syntaktische Analyse* der Eingabe vornimmt. Das Ziel ist es, die Struktur des Tokenstroms zu bestimmen und entsprechend weiterzuverarbeiten, um beispielsweise eine interne Repräsentation aufzubauen.

Die ersten Abschnitte untersuchen, was genau einen Parsergenerator ausmacht. Hierbei lassen sich grob vier Aspekte unterscheiden: 1) die Definition der Grammatik, 2) der dementsprechend ablaufende Parsevorgang, 3) in welcher Weise semantische Werte berechnet werden können und 4) welche Form das generierte Analyseprogramm hat. Entsprechend sind die ersten Teile des Kapitels gegliedert.

Abschnitt 3.1 behandelt die Definition der Grammatik, also die Angabe der *Terminale*, der *Nonterminale*, der *Produktionen* und des *Startsymbols*. Für die Produktionen sind möglichst ausdrucksstarke Sprachen erwünscht. Es wird auch untersucht, welche Uneindeutigkeiten die Spezifikation enthalten darf und welche Zusatzinformationen benötigt werden, um sie auflösen zu können.

Nach diesen rein deklarativen Aspekten wird betrachtet, wie die Spezifikation zur Durchführung eines Parsevorgangs verwendet werden kann. Abschnitt 3.2 gibt einen kurzen Einblick in einige der vielen existierenden Algorithmen und versucht zu identifizieren, welche Implikationen die Wahl einer Parsetechnik hat, etwa Einschränkungen, die sie an die Form der Grammatik macht. Außerdem ist nicht jeder Parsealgorithmus für jede Anwendung geeignet – bei interaktiven Systemen zum Beispiel ist Backtracking höchstens in begrenztem Maße erwünscht. Hier werden auch Möglichkeiten angegeben, wie auf fehlerhafte Eingaben reagiert werden kann.

Mit einer Grammatikdefinition allein ist lediglich eine Überprüfung der Eingaben auf Korrektheit möglich; um sie weiterverarbeiten zu können, müssen *semantische Aktionen* mit dem Erkennungsvorgang gekoppelt und *semantische Werte* berechnet werden. Abschnitt 3.3 stellt Möglichkeiten vor, wie dieses geschehen kann.

Der vierte Aspekt wird in Abschnitt 3.4 behandelt. Der von existierenden Parsergeneratoren generierte Code wird betrachtet, dessen Schnittstellen untersucht und die Form der Kapselung beurteilt.

Diese Betrachtungen werden dazu verwendet, in Abschnitt 3.5 die Eigenschaften des Parsergenerators für Oz festzulegen. Das Ergebnis wird später – in Abschnitt 5.3 – vorgestellt.

3.1 Spezifikation kontextfreier Grammatiken

Eine *Grammatik* beschreibt eine formale Sprache, also eine Menge von Symbolfolgen. Nachfolgend wird eine kurzgefaßte Definition für kontextfreie Grammatiken angegeben, angelehnt an [KVE94]. Einige der verwendeten Begriffe sind bereits in Abschnitt 2.1.1 eingeführt worden.

Definition: Eine **kontextfreie Grammatik** $\mathcal{G}$ wird definiert durch ein Quadrupel $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$. Dabei bezeichnet $\mathcal{V}$ das Vokabular und $\mathcal{T} \subset \mathcal{V}$ die Menge der Symbole, aus denen die Sätze der definierten formalen Sprache bestehen, genannt die **Terminalsymbole**. $\mathcal{N} := \mathcal{V} \backslash \mathcal{T}$ stehe für die übrigen Symbole, die dann **Nonterminale** heißen. Das Symbol $S \in \mathcal{N}$ ist ein ausgezeichnetes Nonterminal, genannt das **Startsymbol** der Grammatik. $\mathcal{P}$ ist die Menge der **Produktionen** der Grammatik. Jede Produktion aus $\mathcal{P}$ hat die Form (N, u) mit $N \in \mathcal{N}, u \in \mathcal{V}^*$, auch geschrieben als $N \to u$. □

Definition: Sei $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ eine Grammatik. Die **Ableitungsrelation** $\Rightarrow_{\mathcal{G}}$ wird definiert durch:

$$uNv \Rightarrow_{\mathcal{G}} uwv \quad \text{gdw.} \quad N \to w \in \mathcal{P}, \qquad \text{wobei } u, v, w \in \mathcal{V}^* \text{ und } N \in \mathcal{N}.$$

Eine **Ableitungsfolge** bezeichnet eine Folge von Sätzen $u_1, \ldots, u_n \in \mathcal{V}^*$, für die $u_i \Rightarrow_{\mathcal{G}} u_{i+1}$ gilt. Die durch G definierte formale Sprache $L(\mathcal{G})$ ist gegeben durch:

$$L(\mathcal{G}) := \{u \mid S \Rightarrow_{\mathcal{G}}^{+} u \ \wedge \ u \in \mathcal{T}^*\}.$$

Dabei bezeichnet $\Rightarrow_{\mathcal{G}}^{+}$ den transitiven Abschluß von $\Rightarrow_{\mathcal{G}}$. □

Mit diesen Bezeichnungen besteht die Aufgabe eines Parsers darin, zu einer gegebenen Folge von Terminalsymbolen festzustellen, ob diese ein Satz der durch die Grammatik definierten formalen Sprache ist, und eine entsprechende Ableitungsfolge zu bestimmen.

Die nachfolgenden Abschnitte befassen sich mit der Spezifikation derartiger Grammatiken für den praktischen Gebrauch in Compilergeneratoren.

3.1.1 Terminale

Die Eingabe des Parsers ist eine Terminalsymbolfolge. Wird diese von einem Scanner produziert, so entsprechen die Terminalsymbole dessen Tokentypen.

Im allgemeinen können hier aber beliebige Symbole oder Ereignisse stehen. Beispielsweise gibt es Werkzeuge, die das Verhalten von Objekten und deren internen Zustand durch eine kontextfreie Grammatik beschreiben, wie dies etwa *MUSKOX* tut [MAS96]. In diesem Fall entsprechen die Terminale Nachrichtentypen, die an ein Objekt gesendet werden können.

Wie es bereits die formale Definition einer Grammatik motiviert, müssen die Terminalsymbole deklariert werden. Intern werden Terminale (beziehungsweise Tokentypen) von den meisten Parsergeneratoren durch ganze Zahlen repräsentiert. Diese Darstellung ist für den Benutzer nicht immer transparent – sie kann beispielsweise dann relevant sein, wenn ein Programm zur Erzeugung von Terminalsymbolen von Hand geschrieben wird. Manche Werkzeuge ermöglichen es dem Benutzer, bei der Deklaration Terminalen explizit Zahlenwerte zuzuordnen; die übrigen werden danach automatisch numeriert. Dabei achten *Lalr* und *Ell* darauf, daß keine Konflikte mit benutzergewählten Zahlen vorkommen; *Yacc* [Joh75] und *Bison* [DS95] vergeben ungeachtet dessen aufeinanderfolgende Werte.

Interessant ist weiterhin die Benennung der Terminalsymbole in der Spezifikation. Es ist aus Gründen der Lesbarkeit sinnvoll, möglichst sprechende Namen zu verwenden. Zudem verwenden einige Werkzeuge die Terminalnamen in Fehlermeldungen, die zur Laufzeit ausgegeben werden, wie beispielsweise *PCCTS* [PDC91] oder *Bison*. In *Yacc* und *Bison* bis Version 1.24 kann man Terminale entweder als Ein-Zeichen-Literale, die ihren ASCII-Code als Wert haben, in der Form `'='` angeben oder mit Bezeichnern mit C-Syntax benennen. In der Version 1.25 wurden die *multi-character literal tokens* aus *AUIS-Bison* [DSH95] übernommen, die beispielsweise `":="` anstelle von `BECOMES` zulassen.

Literale, also einelementige formale Sprachen (*verbatim* durch das Element notiert), werden bei vielen Parsergeneratoren implizit durch ihre Verwendung deklariert, zum Beispiel in [KVE94]. Bei Systemen, in denen Scanner- und Parserspezifikation integriert sind, kann ein Terminal auch durch Verwendung eines regulären Ausdrucks deklariert werden, wie in *SAGA* [And95] oder *PCCTS*.

3.1.2 Nonterminale

Nonterminale sind die Symbole, die bei der Bildung einer Ableitungsfolge durch andere Symbolfolgen ersetzt werden. In Grammatiken sollten sie logisch abgeschlossenen Einheiten der beschriebenen Sprache entsprechen, wie zum Beispiel `expression`, `statement` oder `program`. In *MUSKOX*, das kontextfreie Grammatiken zur Beschreibung von Objektverhalten einsetzt, entsprechen Nonterminale Objektzuständen.

Für die Benennung der Nonterminale wird üblicherweise die Bezeichnersyntax der Zielsprache verwendet. Für den Benutzer des Parsers ist sie meist nicht sichtbar, im Gegensatz zu den Namen der Terminale.

Wie in der formalen Definition muß eines der Nonterminale als Startsymbol ausgezeichnet werden, auch genannt *Axiom* oder *sentence symbol*. Dies kann, wie bei *Bison* oder *ML-Yacc* [TA91], durch eine spezielle Direktive geschehen (`%start` *Nonterminalname*) oder, wie bei *Yacc*, das erste Nonterminal sein, für das in der Grammatik eine Produktion notiert wird. In *Eli* [Com96b] wird das Startsymbol als das (einzige) Nonterminal definiert, für das es nur eine Produktion gibt und das auf keiner rechten Seite erscheint.

Etwas verwirrend ist in dieser Hinsicht die Dokumentation des *ML-Yacc*, die empfiehlt, das Startsymbol auf keiner rechten Seite erscheinen zu lassen, weil sonst Konflikte beim Parsen auftreten könnten, wenn eine Produktion des Startsymbols erkannt wird.

3.1.3 Syntaxregeln

Für die Produktionen (oder *Syntaxregeln*) gibt es verschiedene Schreibweisen. In diesem Abschnitt sollen die verbreiteten Alternativen vorgestellt werden.

Die Backus-Naur-Form

Es können zwei Arten von Terminalen unterschieden werden, nämlich Literale (Elemente einelementiger formaler Sprachen) und sonstige benannte Terminale (auch *generische Token* [GS88] oder *variable Terminale* [Wad90]), wie dies in der *Backus-Naur-Form*, oder kurz *BNF*, getan wird. Für die Notation der Symbole gibt es dann zwei Möglichkeiten:

- Literale werden durch einfaches Hinschreiben notiert (im Druck können sie zusätzlich durch **Fettdruck** oder `Typewriter`-Zeichensatz kenntlich gemacht werden). Besteht die Gefahr einer Verwechslung mit Metasymbolen, so können sie in Anführungszeichen gesetzt werden. Um die benannten Terminale und die Nonterminale von den Literalen unterscheiden zu können, werden sie in spitze Klammern $\langle \ldots \rangle$ gesetzt. (Diese Schreibweise geht auf die ursprüngliche BNF zurück [N$^+$63].)
- Alle Literale müssen durch Anführungszeichen kenntlich gemacht werden. Dann können die benannten Terminale und die Nonterminale ohne zusätzliche Metasymbole geschrieben werden. Allerdings müssen sie eine Bezeichnersyntax haben, die nicht zu Konflikten mit Metasymbolen führt.

Als Separatoren der linken und der rechten Seite sind ‚::=', ‚→' und ‚:' gängig. Manchmal werden Regeln durch einen Punkt oder ein Semikolon abgeschlossen. Leere rechte Seiten werden durch ε oder gar nicht notiert. Es sei bemerkt, daß mehrere Produktionen dieselbe linke Seite haben können; manchmal können deren rechten Seiten dann zu einer einzigen Regel zusammengefaßt werden, indem sie durch ‚|' getrennt werden.

Diese oder eine ähnliche Notation wird unter anderem *Yacc* [Joh75] und Derivate (*Bison* [DS95], *AUIS-Bison* [DSH95] und *Bison++* [Coë93]), *Happy* [GM96], *ML-Yacc* [TA91] oder *Gentle* [Sch89] verwendet. Auch *definite clause grammars* (*DCGs*) [PW80] in Prolog basieren hierauf.

Der Vorteil der BNF ist, daß sie für die entsprechenden Werkzeuge leicht zu verarbeiten ist. Sie hat allerdings auch einige Nachteile: Spezifikationen werden sehr lang und verlieren damit an Lesbarkeit. Häufig auftretende Konstrukte wie separierte Listen müssen jedesmal ausformuliert werden – dies ist lästig und fehleranfällig und verschleiert das dahinterstehende Listenkonstrukt; es verliertd seinen Wiedererkennungswert und damit seine Prägnanz.

Die erweiterte Backus-Naur-Form

Aus diesen Gründen ist die EBNF, die *erweiterte Backus-Naur-Form*, eingeführt worden. Abarten von ihr werden beispielsweise von *Lalr* und *Ell* [GV92], *Eli* [Com96b], *SAGA* [And95], *PCCTS* [PDC91] und *GRAMOL* [GS88] verwendet. Allerdings kann es sein, daß ihre Unterstützung aufgrund der genannten Vorteile der BNF nur lückenhaft ist: So sind in *Eli* EBNF-Konstrukte nur eingeschränkt nutzbar, wenn eine abstrakte Syntax aus der konkreten hergeleitet werden soll.

Die EBNF ist nicht mächtiger als die BNF, aber besser lesbar und erlaubt gegebenenfalls eine effizientere Umsetzung. Ihre Semantik wird häufig durch Rückführung auf äquivalente Grammatiken in BNF definiert, was oft auch ihrer Implementierung entspricht. Bei einigen Werkzeugen, wie dem oben genannten *Eli*, ist die genaue Umsetzung auch für den Benutzer relevant, wenn die Beziehungen zwischen abstrakter und konkreter Syntax festgelegt werden.

In EBNF werden rechte Seiten von Produktionen nicht mehr als Folgen von Symbolen gesehen, sondern als Ausdrücke. Im folgenden werden die verbreiteten Operatoren aufgeführt; dabei sei A ein neues (also bisher in der Grammatik unbenutztes) Nonterminal, a Terminal, x, x_i EBNF-Ausdrücke). Es wird jeweils nur eine gängige Notation beschrieben. Es gibt viele alternative konkrete Syntaxen für dieselben Operatoren, die beispielsweise in [And95], [PW80], [PDC91], [Com96b], [KVE94], [GS88] oder [CCH95] nachgelesen werden können.

Sequenz. Die Sequenz entspricht der Symbolfolge in der BNF. Ihr Operator wird manchmal als Komma geschrieben, meist jedoch gar nicht notiert.

Alternative. Eine Alternative beschreibt verschiedene rechte Seiten für dieselbe linke Seite und wird meist durch ‚|‘ notiert. Es wird also

$$x_1 \mid \ldots \mid x_n$$

durch A ersetzt mit den Regeln

$$A \to x_1$$
$$\vdots$$
$$A \to x_n$$

Option. Eine Option ist eine 0- oder 1-fache Wiederholung. Sie wird hier durch eckige Klammern ‚[...]' notiert. Das BNF-Äquivalent von

$$[\ x\ ]$$

ist also das frische Nonterminal A mit den neuen Regeln

$$A \to \varepsilon$$
$$A \to x$$

Optionale Wiederholung. Optionale Wiederholungen bewirken die 0 bis n-fache Wiederholung eines EBNF-Ausdrucks. Sie werden in geschweifte Klammern eingeschlossen. Das Konstrukt

$$\{\ x\ \}$$

ist gleichbedeutend mit dem frischen Nonterminal A mit den Regeln

$$A \to \varepsilon$$
$$A \to A\ x$$

Obligatorische Wiederholung. Die obligatorische Wiederholung ist die 1 bis n-fache Wiederholung, notiert durch ‚{ ... }+'. Das Konstrukt

$$\{\ x\ \}+$$

ist gleichbedeutend mit dem frischen Nonterminal A mit den Regeln

$$A \to x$$
$$A \to A\ x$$

Separierte Wiederholung. Mittels einer separierten Wiederholung können beispielsweise Komma-getrennte Listen beschrieben werden. Hier wird dieser Operator durch ‚{ ... // ...}' notiert, wobei das zu wiederholende Element von dem Trennsymbol durch zwei Schrägstriche getrennt wird. Das Konstrukt

$$\{\ x\ //\ a\ \}$$

ist gleichbedeutend mit dem frischen Nonterminal A mit den Regeln

$$A \to x$$
$$A \to A\ a\ x$$

Klammerung. Um die Operatoren voll ausnutzen zu können, wird noch ein Gruppierungskonstrukt geboten, üblicherweise durch runde Klammern notiert. Also kann

$$(x)$$

in BNF transformiert werden, indem es durch ein frisches Nonterminal A ersetzt wird mit der Regel

$$A \to x$$

Diese Konstrukte zeichnen sich immer noch durch eine große Allgemeinheit aus. Andererseits kann es für manche Sprachen auch die Lesbarkeit und Knappheit der Spezifikation verbessern, wenn applikationsspezifische Konstrukte definiert werden können. Eine Technik, bei der dies getan werden kann, sind die *parsing combinators* [Fai87], die in funktionalen Sprachen Anwendung finden. Dort kann auf einer sehr niedrigen Ebene mit der Parsetechnik gearbeitet werden: Im Prinzip schreibt der Benutzer den gesamten Parser selbst, verfügt jedoch über einen Satz vordefinierter Funktionen. Die Anwendung dieses Konzeptes mit C als Zielsprache ist mit *PRECC* [BB92] versucht worden. Eine andere Möglichkeit, dies zu realisieren, wird in Abschnitt 5.3.3 unter dem Namen *Produktionsschemata* vorgeschlagen.

Strukturierte Grammatiken

Eine andere vorgeschlagene Notation sind die *strukturierten Grammatiken*. Auch sie können in äquivalente BNF-Regeln übersetzt werden. Als Vorteile werden erhöhte Lesbarkeit gegenüber *Yacc*, einfacheres Debugging der Spezifikation und die implizite Festlegung der abstrakten Syntax angeführt. Deren Bestandteile sind Objekte und nutzen Vererbungsmechanismen.

Strukturierte Grammatiken sind durch die folgenden beiden Punkte definiert:

- Für jedes Nonterminal existiert genau eine Produktion.
- Jede Produktion ist strukturiert, hat also eine der in Tabelle 3.1 angegebenen Formen. Dabei seien A Nonterminal, X_i, E Symbole, S Terminal. Die Tabelle zeigt auch das EBNF-Äquivalent jeder Produktion.

Strukturierte Grammatiken haben verschiedene Nachteile. Zum einen ist das Argument ihrer größeren Lesbarkeit gegenüber der BNF hinfällig, denn strukturierte Produktionen sind der EBNF unterlegen. Der Grund liegt darin, daß viele Teilausdrücke benannt werden müssen, sogar noch mehr als in der BNF. Diese haben aber selten eine besondere Bedeutung für einen Programmierer in der Sprache und sind somit nicht leicht mit guten Namen zu versehen. Weiterhin wird verschleiert, welche Nonterminale die eigentlich wichtigen sind.

Konstrukt	Strukturierte Form	EBNF-Äquivalent
Konstruktion	$A ::= X_1 \ \ldots \ X_n$	$A : X_1 \ldots X_n$
Alternative	$A ::\mid X_1 \ \ldots \ X_n$	$A : X_1 \mid \ \ldots \ \mid X_n$
Option	$A ::? E$	$A : [\ E\ ]$
Optionale Wiederholung	$A ::* E$	$A : \{\ E\ \}$
Optionale separierte Wiederholung	$A ::* E\ S$	$A : [\ \{\ E\ //\ S\ \}\ ]$
Obligatorische Wiederholung	$A ::+ E$	$A : \{\ E\ \}+$
Obligatorische separierte Wiederholung	$A ::+ E\ S$	$A : \{\ E\ //\ S\ \}$

Tabelle 3.1: Definitionen der Produktionen strukturierter Grammatiken

Ein anderer Nachteil betrifft die Implementierung strukturierter Grammatiken. Für ihre Umsetzung werden sie in BNF-Konstrukte transformiert, wobei die resultierende Grammatik unter Umständen Eigenschaften besitzt, die sie schwer parsebar machen (sogenannte *Konflikte* werden in Abschnitt 3.2 behandelt). Es ist also nötig, (semi-)automatische Transformationen zu implementieren, die diese Fehler korrigieren. *Mango* bietet hierzu ε-Elimination, Elimination von Nonterminalen mit nur einer Produktion, Inlining bestimmter Nonterminale und Expansion nicht-rekursiver Nonterminale.

Die Startproduktion

Dieser Abschnitt soll einige Feinheiten betrachten, was die Behandlung der Startproduktion angeht.

In dem Abschnitt über Nonterminale (3.1.2) wurde gefordert, daß eines davon als das Startsymbol ausgezeichnet werden muß. Es kann aber auch sinnvoll sein, den Parsevorgang mit mehreren verschiedenen Symbolen starten zu können. Beispielsweise könnte ein System mit integriertem Compiler und Debugger für den Übersetzungsvorgang ausschließlich vollständige Programme akzeptieren, beim Debuggen aber einzelne Anweisungen oder Ausdrücke parsen, um diese auszuführen beziehungsweise auszuwerten.

Eine bekannte Technik, die dies ermöglicht, geht folgendemaßen vor: Seien $S_1, \ldots, S_n$ die gewünschten Startsymbole. Seien weiterhin $a_1, \ldots, a_n$ neue (also bisher unbenutzte) Terminalsymbole und S ein neues Nonterminalsymbol. Dann wird folgende Produktion zu der Grammatik hinzugefügt:

$$S : a_1\ S_1 \mid \ \ldots \ \mid a_n\ S_n$$

Bei dem Aufruf des Parsers muß dann das gewünschte Startsymbol S_i angegeben werden, worauf als erstes Terminal vor den Eingabestrom das korrespondierende Symbol a_i eingefügt wird.

Diese Technik wird beispielsweise in der Anleitung zu ML-Yacc [TA91, Abschnitt 10.1] vorgeschlagen. Mehrere Startsymbole werden nur selten direkt unterstützt, beispielsweise von *GRAMOL* [GS88, Abschnitt 5.3.1].

Eine andere Frage, die mit der Startproduktion zusammenhängt, betrifft nicht den *Beginn* des Parsevorgangs, sondern dessen *Abschluß*. Üblicherweise wird ein neues Nonterminal S' mit folgender Produktion zu der Grammatik hinzugefügt (zum Beispiel von *Bison* oder *Lalr* [Gro88a]):

$$S' : S\ \$$$

Dabei sei S das ehemalige Startsymbol und \$ stehe für die Endmarkierung der Eingabe (ein ausgezeichnetes Terminal, in Yacc durch eine ganze Zahl ≤ 0 repräsentiert. In Bison kann dieses aufgrund eines Programmfehlers nicht notiert werden und bleibt damit ein internes Terminal). Wenn also die Startproduktion erkannt wurde, muß das Ende des Eingabestroms erreicht sein.

Nicht bei allen Sprachen ist dies ein wünschenswertes Verhalten. Bei Oberon-2 [MW91] beispielsweise können nach der ein Modul abschließenden Sequenz ‚`END.`' noch beliebige Daten folgen können, die im Oberon-System unter anderem als Kommandoaktivierungen oder Testdaten fungieren können und vom Compiler ignoriert werden sollen. Für diese Zwecke bieten *Yacc* und *Bison* den Befehl `YYACCEPT`, der den Parsevorgang sofort mit einer Erfolgsmeldung abbricht.

Bei *ML-Yacc* kann über die Direktive `%eop` (für *end-of-parse symbols*) die Menge der Terminale angegeben werden, die in der Eingabe auf das Startsymbol folgen dürfen.

3.1.4 Operatoren und Behandlung von Uneindeutigkeiten

Mit allen vorgestellten Formalismen ist es möglich, uneindeutige Grammatiken zu konstruieren. Die folgende Definition beschreibt, was dies bedeutet.

Definition: Eine Grammatik heißt **eindeutig**, wenn es für jede Symbolfolge aus der durch sie definierten formalen Sprache genau eine Ableitungsfolge gibt, aus der sie entsteht. Ist dies nicht der Fall, so heißt sie **uneindeutig**. □

Das Problem mit uneindeutigen Grammatiken liegt darin, daß für denselben Eingabetext mehrere Syntaxbäume existieren können, was von dem Sprachentwickler und dem Compilerbauer selten erwünscht ist. Für viele uneindeutige Grammatiken gibt es äquivalente eindeutige Grammatiken. Die Transformation in eine eindeutige Grammatik ist jedoch nicht trivial und oft nur auf Kosten von Übersichtlichkeit oder Effizienz realisierbar.

Klassische Probleme für uneindeutige Grammatiken sind:

Das ‚dangling else'. Dieses Problem ist nach einer Uneindeutigkeit benannt, die in vielen Sprachen existiert, zum Beispiel in Pascal, C und Haskell, aber im Sprachreport oft nur informell (natürlichsprachlich) aufgelöst wird. Es tritt aber auch in anderen

Kontexten auf, beispielsweise in Refus [EKVW94] bei der Definition einer Lambda-Funktion.

Gegeben sei folgende Grammatik für Anweisungen mit einem besonderen Augenmerk auf die `if`-Anweisung:

> Statement : `if` Expression `then` Statement
> Statement : `if` Expression `then` Statement `else` Statement
> Statement : Others

Dabei steht ‚Others' für alle anderen Anweisungsarten. Diese Grammatik ist uneindeutig: Etwa für den Satz ‚`if` x `then if` y `then` s_1 `else` s_2' macht die Grammatik keine Aussage darüber, zu welchem `if` das `else` gehört. Die Regel aus dem Sprachreport besagt üblicherweise, daß jedes `else` mit dem letzten `if` assoziiert werden soll, das keines besitzt.

Eine Möglichkeit, diese Grammatik eindeutig zu formulieren, besteht darin, zwischen „offenen" und „geschlossenen" `if`-Anweisungen zu unterscheiden [ASU86, S. 175]:

> Statement : Matched
> Statement : Unmatched
> Matched : `if` Expression `then` Matched `else` Matched
> Matched : Others
> Unmatched : `if` Expression `then` Statement
> Unmatched : `if` Expression `then` Matched `else` Unmatched

Dies wird offenbar sehr unübersichtlich.

Ausdrücke. Eine andere Uneindeutigkeit rührt daher, daß die Syntax von Ausdrücken im Sprachreport häufig folgendermaßen beschrieben wird:

> Expression : Expression Operator Expression
> Expression : Operand
> Operator : `+`
> Operator : `-`
> ⋮

Die Präzedenzen und Assoziativitäten werden dann in einer Tabelle angegeben.

Diese Grammatik ist offensichtlich uneindeutig, da sie zum Beispiel bei der Eingabe ‚$3 + 4 * 5$' nicht definiert, ob sie als ‚$(3 + 4) * 5$' oder als ‚$3 + (4 * 5)$' interpretiert werden soll; beide den Klammerungen entsprechenden Syntaxbäume sind möglich. Der Syntaxbaum soll aber die Präzedenzen und Assoziativitäten des arithmetischen Ausdrucks widerspiegeln.

Eindeutige Grammatiken für Infixausdrücke können relativ leicht und übersichtlich nach folgendem Verfahren konstruiert werden. Die Autoren von *PCCTS* beurteilen dieses zwar als das übersichtlichste, aber dazu gibt es kontroverse Meinungen.

i	A_i	o_i	Assoziativität
0	expr	compop : `=` | `<>`	nicht-assoziativ
1	sum	addop : `+` | `-`	linksassoziativ
2	term	mulop : `*` | `/`	linksassoziativ
3	factor	expop : `^`	rechtsassoziativ

Tabelle 3.2: Eine beispielhafte Operatortabelle

- Ordne jedem Präzedenzniveau i mit Operatoren o_i ein Nonterminal A_i zu. (Eines davon kann für ‚unsichtbare' Operatoren stehen, wie beispielweise die Applikation in der funktionalen Programmiersprache Haskell [HPJW92]. Dann ist $o_i : \varepsilon$.) Kleinere i entsprechen niedrigeren Präzedenzen.
- Jedem Präzedenzniveau wird eine Assoziativität zugeordnet. Man schreibe die Regel

  $$A_i : A_{i+1}$$

  und – abhängig von der Assozitivität – eine der folgenden:

  nicht-assoziativ: $A_i : A_{i+1}\ o_i\ A_{i+1}$
  linksassoziativ: $A_i : A_i\ o_i\ A_{i+1}$
  rechtsassoziativ: $A_i : A_{i+1}\ o_i\ A_i$

Man wähle beispielsweise die Operatortabelle aus Tabelle 3.2 und $A_4 =$ prim. Nach dem obigen Verfahren entstehen folgende Regeln:

expr : sum | sum compop sum
sum : term | sum addop term
term : factor | term mulop factor
factor : prim | prim expop factor

Der Nachteil ist, daß dies bei vielen Parsetechniken zu ineffizienten Lösungen führt. Bei LR-Parsern beispielsweise (siehe Abschnitt 3.2) kann der Aufwand für die Verarbeitung einer Eingabe als Summe von *Shift*- und *Reduce*-Aktionen angegeben werden. In der Sprache C können bis zu 15 Reduktionen für eine einfache Zuweisung nötig sein – verglichen mit einer einzigen, wenn die Uneindeutigkeiten anders aufgelöst werden. (Außerdem wird der Speicherverbrauch des generierten Parsers bei dieser Methode größer.)

Eine ausführliche Untersuchung über Präzedenzen und Assoziativitäten bei der Spezifikation und Implementierung von Programmiersprachen findet sich in [Aas92].

Sonderfall-Produktionen. Manchmal sollen in Sprachen besondere Verwendungen von Konstrukten nicht von einem allgemeinen Fall verarbeitet werden sondern eine Spezialbehandlung erfahren. Beispielsweise zählen hierzu Operatoren mit konstanten Operanden, die somit optimiert werden können, oder Konstrukte, für die der allgemeine

Fall nicht das gewünschte Ergebnis liefert. Als Beispiel hierfür seien Beschreibungen gedruckter mathematischer Formeln genannt: Ist für einen Ausdruck sowohl ein Index als auch ein nachfolgender Exponent angegeben (wie in TEX [Knu91] durch `x_i^2`), so sollen Index und Exponent übereinander (x_i^2) und nicht nebeneinander stehen ($x_i{}^2$). Dieser Fall kann über eine Sonderfall-Produktionen gehandhabt werden (abgewandelt aus [ASU86, S. 251]):

term : term index
term : term exponent
term : term index exponent

Offensichtlich liegt nun eine uneindeutige Grammatik vor. Die Sonderfall-Produktion sollte aber in jedem Fall vorgezogen werden, in dem sie anwendbar ist. Wird eine uneindeutige Grammatik gefordert, so muß der Sonderfall stattdessen in der semantischen Analyse explizit abgefragt werden.

Die Beispiele machen einige der Vorteile offensichtlich, die uneindeutige Grammatiken (mit separatem Mechanismus zum Auflösen der Uneindeutigkeiten) besitzen können. Wie diese Uneindeutigkeiten gehandhabt werden können, ist sehr stark vom verwendeten Parsealgorithmus abhängig und wird daher erst in Abschnitt 3.2.1 behandelt.

3.2 Parsetechniken

In diesem Abschnitt werden repräsentativ einige der vielen existierenden Parsetechniken skizziert, die anhand einer Grammatikspezifikation eine Eingabe analysieren. Da der Schwerpunkt der Arbeit auf der Definition der Spezifikationssprache und nicht auf ihrer Implementierung liegt, wird zunächst untersucht, welche Implikationen die Wahl der Parsetechnik hat. Die Vor- und Nachteile werden hauptsächlich aus Sicht des Benutzers erläutert.

Mächtigkeit der erkannten Sprache. Zwar ist es nicht erwünscht, die volle Mächtigkeit der BNF zu erlauben – so sollten Werkzeuge uneindeutige Grammatiken immer ablehnen –, aber die meisten Parsetechniken bedingen noch weitere Einschränkungen an die Form der Grammatik. Beispielsweise haben manche Algorithmen Schwierigkeiten mit linksrekursiven Produktionen wie der folgenden:

$$A : x \mid A\ x$$

In diesem Fall muß der Benutzer die Grammatik umformulieren, bevor das Werkzeug mit ihr arbeiten kann.

Für ein- und dieselbe formale Sprache kann es eine Vielzahl von Grammatiken geben. Diese können aber für die nachfolgenden Verarbeitungen unterschiedlich gut geeignet

sein, denn die Gestalt des Syntaxbaumes hängt maßgeblich von der Formulierung der Produktionen ab. So ist es zum Beispiel eine große Erleichterung, wenn die Präzedenzen und Assoziativitäten der Operatoren einer Sprache in die Konstruktion des Syntaxbaumes bereits eingegangen sind. Ebenso sollten sich hier die Skopierungsregeln der Sprache wiederfinden lassen: Der Sichtbarkeitsbereich eines Bezeichners sollte möglichst genau einem Teilbaum entsprechen.

Es wird deutlich, daß ein werkzeugbedingtes Umformulieren der Grammatik Probleme bereiten kann.

Auflösung von Uneindeutigkeiten. Für die Auswahl unter mehreren alternativen Syntaxbäumen können sehr schwer formale Regeln angegeben werden. Aus diesem Grund hängen die in existierenden Werkzeugen gefundenen Lösungen sehr stark von der verwendeten Parsetechnik ab – die Uneindeutigkeiten werden aufgelöst, indem das Laufzeitverhalten des Parsers auf Implementierungsebene beeinflußt wird.

Theoretische Grundlage. Um die Korrektheit eines Parsers nachweisen zu können, muß aus dem parsenden Programm beziehungsweise dessen Spezifikation eine formale Beschreibung der erkannten Sprache (in BNF) hergeleitet werden können. Dies ist nicht immer einfach.

Fehlerbehandlungsmöglichkeiten. Ebenso wie die Behandlung von Uneindeutigkeiten findet das Reagieren auf einen Fehler in der Eingabe auf einem sehr niedrigen Niveau des Parseverfahrens statt.

Mächtigkeit der semantischen Aktionen. Bei Ausführung einer semantischen Aktion ist nur ein Teil aller Informationen aus dem Syntaxbaum zugreifbar. Zu diesen gehören globale Variablen sowie einige benachbarte, bereits ausgewertete Knoten, manchmal Informationen aus den Elternknoten. Von der verwendeten Parsetechnik hängt ab, welche dieser Informationen in die Berechnungen einer Aktion einbezogen werden können.

So hat beispielsweise eine *S-attributierte Grammatik* die Eigenschaft, daß alle Attribute (Knotenannotationen des Syntaxbaums) in einem einzigen bottom-up-Durchlauf ausgewertet werden können. Für die Berechnung der Attribute sind also nur die Informationen der Nachfolgerknoten verwendbar. Dahingegen muß es bei einer *L-attributierten Grammatik* möglich sein, all ihre Attribute in einem einzigen top-down-, links-nach-rechts-Durchlauf auszuwerten. Es sind also die Informationen aus den höherliegenden und aus den weiter links im Parsebaum liegenden Knoten zugreifbar. (Vgl. [Wil79].)

Die Mächtigkeit der semantischen Aktionen wird noch weiter eingeschränkt, wenn ihr genauer Ausführungszeitpunkt nicht bekannt ist (beispielsweise aufgrund von Backtracking oder gewissen Transformationen der Grammatik). In diesem Fall können keine Seiteneffekte zur Steuerung des Scanners verwendet werden (sogenannte *lexical tie-ins* [DS95]).

Effizienz. Wie bereits in dem Abschnitt über Uneindeutigkeiten motiviert wurde, kann sich die Effizienz verschiedener Parsetechniken allein dadurch unterscheiden, wie die Grammatik umgesetzt wird (zusätzlich zu ohnehin durchzuführenden Komplexitätsanalysen). Beispielsweise können die Wiederholungskonstrukte von EBNF bei einigen Verfahren effizient direkt abgebildet werden, bei anderen müssen sie durch BNF-Äquivalente ersetzt werden. Daraufhin kann nicht mehr von ihrer bekannten Struktur profitiert werden.

Debugging. Der Debugging-Prozeß läuft inhärent auf einer sehr niedrigen Ebene ab. Debugging umfaßt hier sowohl das Debuggen der Grammatikspezifikation (beispielsweise das Eliminieren von Uneindeutigkeiten) als auch der benutzergeschriebenen semantischen Aktionen. Bei verschiedenen Verfahren muß hierfür an anderen Stellen angesetzt werden.

LL-Parsing

LL(k)-Parsing bezeichnet eine Top-Down-Parsetechnik, die die Eingabe von links nach rechts liest und eine umgekehrte *Linksableitung* bildet. (Eine Linksableitung ist eine Ableitungsfolge, bei der immer das am weitesten links stehende Nonterminal ersetzt wird.) Der Wert k gibt an, wieviele Terminale als sogenanntes *Lookahead* verwendet werden, das heißt, im Voraus gelesen und für die Parseentscheidungen einbezogen werden können. Meist ist $k = 1$, es gibt jedoch Argumente, größere Werte für k zu unterstützen [PQ95].

LL(k)-Parsing ist ein spezieller *Recursive-Descent-Parsing*-Algorithmus, der kein Backtracking benötigt (also ein *predictive parser*). Dafür muß jedoch die Grammatik bestimmten Kriterien genügen: Es darf keine (unmittelbare oder indirekte) Linksrekursion vorliegen und je zwei alternative Produktionen für ein Nonterminal dürfen keine Symbolfolge ableiten, die mit demselben Terminal beginnen. Daher kann aufwendiges Umformulieren nötig sein, bis der Generator die Spezifikation akzeptiert, was zu unnatürlichen Grammatiken führen kann. Abhilfe können hier *semantische Prädikate* verschaffen (siehe Abschnitt 3.2.1).

Semantische Aktionen können an beliebigen Stellen ausgeführt und Regel-lokale Variablen (die bei jedem Aufruf einer Produktion neu alloziert werden) definiert werden. L-attributierte Grammatiken können implementiert werden.

LL-Parser sind leicht zu implementieren und EBNF-Konstrukte können leicht und effizient umgesetzt werden. Beispielsweise sind separierte Wiederholungen effizienter als bei den meisten anderen Techniken. Es gibt aber nur wenige Möglichkeiten, Uneindeutigkeiten aufzulösen (siehe Abschnitt 3.2.1). Insbesondere können Operatorpräzedenzen nur durch Ausformulieren oder einen eingebetteten Parser, der eine andere Technik verwendet, umgesetzt werden.

Als Argument für LL-Parser wird häufig genannt, daß diese leicht zu debuggen sind, weil der generierte Code der Eingabegrammatik sehr ähnlich sieht: Zu jedem Nonterminal wird eine Funktion generiert; Sequenzen werden in Sequenzen umgesetzt, Alternativen

und Optionen in bedingte Anweisungen, Wiederholungen in Schleifen. Eigentlich sollten Parser aber nicht auf der Ebene des generierten Programmcodes, sondern der Spezifikation debugged werden. Außerdem macht dieses Argument den Grund zunichte, ein Werkzeug zum Generieren des Parsers zu verwenden: Wenn der Code der Grammatik ohnehin so ähnlich sieht, hätte man ihn auch von Hand schreiben können.

LR-Parsing

Das sogenannte *LR-Parsing* ist eine Bottom-Up-Parsetechnik ohne Backtracking, die üblicherweise als Tabellenparser implementiert wird (eine der wenigen Hard-Coded-Varianten ist in [BP95] beschrieben). Der eigentliche Parsealgorithmus kann je nach Tabellenkonstruktionsverfahren (SLR(k), LALR(k) oder LR(k), wobei k wieder für die Anzahl der Lookahead-Terminale steht) eine mehr oder weniger große Klasse Grammatiken erkennen. Bei LALR(k) und LR(k) ist diese größer als beim korrespondierenden LL(k), was häufig lesbarere und natürlichere Formulierungen der Grammatik zuläßt.

Die Grammatik wird in die Beschreibung eines Push-Down-Automaten transformiert. Anhand des aktuellen Zustands und der Lookahead-Terminale wird eine der Aktionen *Shift* oder *Reduce* ausgewählt. Bei *Shift*(s) wird ein Lookahead-Terminal auf den Parsestack geschoben und der Automat wechselt in den Zustand s. Bei *Reduce*(n) wird mit der (BNF-)Produktion mit Nummer n reduziert: Es werden soviele Symbole vom Parsestack durch das Nonterminalsymbol ersetzt, wie die rechte Seite der Produktion n enthält. Hieraus folgt, daß EBNF nur durch Transformation in eine äquivalente BNF-Darstellung unterstützt werden kann. Diese Parsetechnik kann für eine gegebene Grammatik kaum von Hand implementiert werden – ein Werkzeug ist hier unerläßlich.

Einschränkungen an die Grammatik bestehen bei dieser Parsetechnik darin, daß nicht gleichzeitig eine *Shift*- und eine *Reduce*-Aktion oder zwei verschiedene *Reduce*-Aktionen anwendbar sein dürfen. Es gibt mehrere Verfahren zum Auflösen derartiger Konflikte. Insbesondere können hierzu Operatorpräzedenzen elegant einbezogen werden (mehr hierzu findet sich in Abschnitt 3.2.1).

Das Auffinden der Ursachen für uneindeutige Grammatiken gestaltet sich häufig als sehr aufwendig, da die meisten Werkzeuge nur einen Dump des generierten Automaten als Hilfe anbieten. *ML-Yacc* kann auf Wunsch anstelle von LALR(1)-Tabellen vollständige (aber nicht mächtigere) LR(1)-Tabellen erzeugen, die das Debuggen des Automaten einfacher machen sollen. *Lalr* unterstützt die noch hilfreichere Ausgabe eines *Derivation Tree*, bei dem das Zustandekommen der Uneindeutigkeit leichter erkennbar ist.

Semantische Aktionen können nur mit *Reduce*(n)-Aktionen gekoppelt werden, was bedeutet, daß im generierten Parser eine umfangreiche Fallunterscheidung über den Wert von n gemacht wird. Zum Debuggen die semantischen Aktionen reicht ein einzelner Breakpoint auf diese Anweisung. Nur S-attributierte Grammatiken sind einfach realisierbar. Vererbte Attribute, also solche, die von Knoten stammen, die im Syntaxbaum weiter links stehen, können begrenzt simuliert werden, indem der Inhalt des Parsestacks nicht nur in

bezug auf die aktuelle Produktion, sondern auch auf weiter unten liegende Elemente einbezogen wird. Es können also Attribute gelesen werden, deren relative Position im Parsestack konstant und bekannt ist. Dies wird beispielsweise von *Yacc* und Derivaten durch negative Arrayindizierungen erlaubt. Eine Prüfung, ob diese Referenz korrekt ist (richtiger Index, Umwandlung in den richtigen Typ), bietet jedoch keines der untersuchten Werkzeuge.

Operator Precedence Parsing

Operator Precedence Parsing ist eine wenig mächtige und wenig verbreitete Technik, die aber für die Implementierung von Grammatiken für Infix-Ausdrücke einfach und effizient anwendbar ist. Es kann jedoch sein, daß die tatsächlich erkannte Sprache aus der Spezifikation nur schwer formal abgeleitet werden kann. Daher wird dieses Verfahren bestenfalls zur Realisierung eingebetteter Parser verwendet und auch selten von Werkzeugen angeboten.

Definite Clause Grammars

Definite Clause Grammars bezeichnen eine Erweiterung von Prolog, mit denen BNF-Produktionen spezifiziert werden können. Sie werden direkt in Prolog-Klauseln umgesetzt. Daher verwenden sie Backtracking mit allen daraus erwachsenden Nachteilen:

- Es sind keine interaktiven Systeme möglich, da bei diesen rasch und auf Anhieb richtig festgestellt werden muß, welche Produktion zum Parsen der Eingabe verwendet werden muß (zum Beispiel Benutzerschnittstellen oder Objektprotokolle).
- Die generierten Parser sind sehr ineffizient, wenn die Grammatik ungeschickt formuliert ist.
- Backtracking-Parser liefern oft sehr schlechte Fehlermeldungen.

Das Backtracking durch die „!“-Anweisung eingeschränkt werden, durch die alle Entscheidungen, die bis dahin gefällt worden sind, als definitiv gelten läßt. Dadurch können einige dieser Nachteile aufgehoben werden.

Allein durch ihre Umsetzung sind diese Grammatiken sehr an die Zielsprache Prolog gebunden. Sie sind hauptsächlich dann nützlich, wenn auf mächtiges Backtracking nicht verzichtet werden kann.

Combinator Parsing

Das *Combinator Parsing* ist eine Parsetechnik aus dem Bereich der funktionalen Programmiersprachen. Es handelt sich dabei um einen Satz Funktionen, die die Standardoperatoren zur Beschreibung und Erkennung von EBNF-Grammatiken implementieren. Da kein Backtracking möglich ist, wird immer eine Liste der möglichen Parseergebnisse mitgeführt,

wobei Alternativen entsprechend der Eingabe entfernt und entsprechend der Grammatik hinzugefügt werden. Am Ende bleibt genau ein Element übrig, wenn die Eingabe korrekt und die Grammatik eindeutig ist. Diese Technik wurde mit dem Werkzeug *PRECC* [BB92] für die Zielsprache C vorgeschlagen.

Die Realisierung von EBNF-Operatoren durch Funktionen erlaubt es dem Benutzer, eigene Operatoren zu definieren, die beliebig komplex sein können. Die Zielsprache kann dabei in ihrer vollständigen Mächtigkeit verwendet und beliebige Ausdrücke (und Teilparser) als Argumente verwendet werden. Der Preis für diese Flexibilität ist, daß die tatsächlich erkannte Sprache unter Umständen formal sehr schwer ableitbar sein kann. Weiterhin können Parser geschrieben werden, deren Termination nicht gesichert ist – überhaupt sind automatische Analysen der Grammatik nur bei Einhaltung strikter Richtlinien möglich. Die Technik ist zudem mäßig effizient bis sehr ineffizient. Der Vorteil ist allerdings, daß sind alle formalen Sprachen erkannt werden können [BB92].

Diese Vollständigkeit kann jedoch auch negative Auswirkungen auf den Sprachentwurf haben. Die Autoren von *PRECC* geben selbst ein Beispiel für legale, aber unübersichtliche Syntax ihres eigenen Werkzeugs an: Bei Formulierungen wie `@ a = )foo(TRUE)(b)(bar)(` muß bestimmt werden, ob es sich um

$$a = \texttt{)foo(TRUE)(}\ b\ \texttt{)(bar)(}$$

oder um

$$a = \texttt{)foo(}\ \text{TRUE}\ \texttt{)(b)(bar)(}$$

handelt, da sowohl die Klammerung durch (...) als auch die durch)...(eine Bedeutung hat. Diese Syntax ist nicht nur für Programme schwer zu parsen. Werkzeuge, die die Sprache etwas mehr Einschränken, können das Entstehen derartiger Syntaxen verhindern.

Zusammenfassung

Viele Parsetechniken fordern Einschränkungen an die Grammatik, die nicht immer mit den erwünschten Eigenschaften des Syntaxbaumes vereinbar sind. So hat beispielsweise eine Linksfaktorisierung der Grammatik, die für LL-Parser benötigt wird, zur Folge, daß verschiedene Konstrukte der Sprache schlecht auseinandergehalten werden können. Wird aus der konkreten Syntax automatisch eine abstrakte generiert, so zieht sich diese Verzahnung bis in die semantische Analyse und eventuell sogar die Codegenerierung durch. Eine Lösung besteht in der Verwendung einer mächtigen Parsetechnik, die die volle BNF implementieren kann – mit den entsprechenden Nachteilen. Eleganter ist es aber, von der Parsetechnik mehr zu abstrahieren. Da viele viele Transformationen auf Grammatiken angewandt werden können, die die erkannte Sprache nicht verändern, sollte der Benutzer die Grammatik auf einem hohen Niveau spezifizieren können und es dem Werkzeug überlassen, sie in eine parsebare Form zu bringen. Diese Transformationen müssen für den Benutzer

transparent sein. Insbesondere müssen Referenzen in semantischen Aktionen automatisch angepaßt werden – für den Benutzer wird also die konkrete Struktur des Syntaxbaums unwichtig, da die semantischen Aktionen sich auf die spezifizierte Form beziehen. Ein Werkzeug, das einen Schritt in diese Richtung macht, ist *Mango* [Age94].

3.2.1 Auflösung von Konflikten

In diesem Abschnitt sollen Lösungen angegeben werden, wie Grammatiken mit Uneindeutigkeiten umgesetzt werden können. In einem ersten Teil werden statische Auflösungen von Konflikten untersucht, also solche, die bereits bei der Parsergenerierung durchgeführt werden. Ein zweiter Teil beschreibt Möglichkeiten, erst zur Laufzeit – eventuell kontextabhängig – Parseentscheidungen zu treffen. Die Beispiele aus Abschnitt 3.1.4 werden hier wieder aufgegriffen.

Statische Auflösung von Konflikten

Bei LR-Parsing sind Uneindeutigkeiten gut untersucht. Folgende Möglichkeiten wurden bei existierenden Werkzeugen gefunden:

Operatorpräzedenz und -assoziativität. Unter anderem *Yacc*, *SAGA* und *Lalr* bieten dieses Verfahren an, das dazu dient, typische Uneindeutigkeiten beim Parsen von Infix-Ausdrücken zu eliminieren. Hierzu werden folgende (über die BNF-Produktionen hinausgehenden) Voraussetzungen benötigt:

- Jedem Terminal wird höchstens eine Präzedenz $\in \mathbb{N}$ und eine Assoziativität $\in$ {nicht-assoziativ, linksassoziativ, rechtsassoziativ} zugeordnet. (Hohe Präzedenzen entsprechen stärkerer Bindung.)
- Jeder Produktion wird höchstens eine Präzedenz $\in \mathbb{N}$ zugeordnet. (Als Default wird häufig die des letzten autretenden Terminals der Produktion verwendet.)

Tritt ein Konflikt auf, so wird die Entscheidung, ob eine *Shift*- oder eine *Reduce*-Aktion ausgeführt werden soll, durch einen Vergleich der Präzedenz des Lookahead-Terminals mit der der Produktion getroffen. Sind die Werte gleich, so wird anhand der Assoziativität entschieden [DS95].

Ausdrücke können dann beispielsweise von der Regel

Expression : Expression Operator Expression

durch die Angabe einer Operatortabelle wie in Tabelle 3.2 geparst werden.

Mit diesem Verfahren läßt sich auch das oben erläuterte *dangling else* lösen. Seien hierzu folgende Regeln gegeben:

> Statement : `if` Expression `then` Statement
> Statement : `if` Expression `then` Statement `else` Statement
> Statement : Others

Habe nun `else` eine höhere Präzedenz als `then`, die erste Regel die Präzedenz von `then` und die zweite die von `else`. Dann wird die Uneindeutigkeit korrekt aufgelöst – im Zweifelsfall wird das `else` geshiftet.

Shift-Annotationen. Bei einem *Shift/Reduce*-Konflikt in *SAGA* kann der Benutzer explizit fordern, ein Terminal zu shiften, indem er es `$shift` daran anhängt:

> Statement : `if` Expression `then` Statement
> Statement : `if` Expression `then` Statement `else$shift` Statement
> Statement : Others

Diese Annotation läßt sich ausschließlich für diese Parsetechnik umsetzen.

Modifikation des *exact right context*. Dieses Verfahren findet man beispielsweise in *Eli*. Hierbei können im Falle eines Konfliktes bestimmte Symbole aus der Menge der Terminale, die bei der Reduktion mit einer Produktion in der Eingabe folgen dürfen, entfernt werden. Beispielsweise kann das *dangling else* wie folgt gelöst werden:

> Statement : `if` Expression `then` Statement `$else`
> Statement : `if` Expression `then` Statement `else` Statement
> Statement : Others

Das `$else` in der ersten Produktion bedeutet hier, daß – wenn mit dieser Produktion reduziert werden soll – das Loohahead-Terminal kein `else` sein darf.

Der Nachteil all dieser Methoden ist, daß eine BNF-Version der erkannten Sprache unter Umständen schwer formal abgeleitet werden kann. Der Benutzer muß sich den generierten Automaten ansehen und nachprüfen, wie die Konflikte genau aufgelöst wurden.

Für LL-Parsing findet man weit weniger Möglichkeiten. Das Werkzeug *Ell* [Gro89, Abschnitt 3] verwendet folgenden Standardsatz an Regeln: Liegt ein Konflikt zwischen einer Option oder einer Wiederholung und dem nachfolgenden Teminal vor, so wird die Option beziehungsweise die Wiederholung gewählt. Liegt ein Konflikt zwischen zwei Alternativen vor, so wird die zuerst notierte gewählt. Entstehen hierbei „tote“ Äste in der Grammatik, die nie zur Anwendung kommen, dann wird ein Fehler gemeldet.

Für Combinator Parsing ist in [Hil94] die Frage untersucht worden, wie spezielle Kombinatoren für die Umsetzung von Operatorpräzedenzen implementiert werden können. Die Operatortabelle wird dann in diese Kombinatoren umgeformt.

Dynamische Auflösung von Konflikten

Wenn Konflikte erst bei der Ausführung aufgelöst werden, kann die Entscheidung von Kontextinformationen abhängig gemacht werden. Beispielsweise können so benutzerdefinierte Operatoren gehandhabt werden, wie in [PVGK93] vorgeschlagen. Der dort beschriebene Algorithmus ist auf LL- und LR-Parsern aufgesetzt und kann Sprachen mit dynamisch definierbaren Operatoren zu parsen, die folgende Eigenschaften besitzen:

- Jeder beliebige Bezeichner kann einen Operator benennen. Eventuell können auch vordefinierte Operatoren (wie `+`, `-`) neu definiert werden.
- Die syntaktischen Eigenschaften eines Operators verändern sich im Laufe des Parsevorgangs.
- Operatoren können anderen Operatoren als Argumente übergeben werden. Folglich ist beispielsweise der Satz „$\vee \times \wedge$“ legal, wenn „$\vee$“, „$\times$“ und „$\wedge$“ Infix-Operatoren darstellen.
- Operatoren können überladen werden, also dasselbe Symbol für mehrere syntaktisch unterschiedliche Operatoren verwendet werden. Zum Beispiel ist „`-`“ oft gleichzeitig Präfix- und Infix-Operator.

Das einzige, was zusätzlich zu der normalen Spezifikation benötigt wird, ist eine Deklaration, welche Symbole Operatoren darstellen. Jeder *Shift/Reduce*-Konflikt, in den dynamische Operatoren verwickelt sind, wird dann in eine sogenannte *Resolve*-Aktion umgewandelt, die ihre Entscheidung aufgrund der Laufzeit-Operatortabelle fällt.

Bei LL-Parsern gibt es noch die Möglichkeit, *semantische Prädikate* zu definieren, wie sie zum Beispiel von *PCCTS* [Par95, Abschnitt 2.6.1] unterstützt werden. Dies bedeutet, daß in die Grammatik Boole'sche Ausdrücke eingebettet werden, die zur Laufzeit ausgewertet werden. Liefern sie den Wahrheitswert *falsch*, so wird der zugehörige Zweig der Grammatik nicht gewählt. Entscheidungen sind also nicht nur aufgrund des Lookaheads möglich, sondern auch kontextabhängig realisierbar.

PCCTS bietet noch ein weiteres Konstrukt, die sogenannten *syntaktischen Prädikate* [Par95, Abschnitt 2.6.2]. Hierüber können Sonderfall-Produktionen sowie Uneindeutigkeiten wie zum Beispiel in C++ aufgelöst werden, wo Deklarationen initialisierter Variablen nicht immer von Funktionsprototypen unterschieden werden können. Die Lösung führt gezielt eingeschränktes Backtracking in einen LL-Parser ein.

3.2.2 Fehlerbehandlung

Die Fehlerbehandlung beschäftigt sich mit dem Verhalten des Parsers, wenn die Eingabe nicht der Grammatik entspricht, wenn also mit den Lookahead-Terminalen keine Parseaktion assoziiert ist (oder auch, wenn ein nichtassoziativer Operator zweifach in Folge

angewendet wird). Solche Fehler sollten möglichst früh erkannt werden, um eine präzise Fehlermeldung liefern zu können. Außerdem sollte bestimmt werden können, welche Produktionen die weitere Eingabe beschreiben, damit in einen wohldefinierten internen Zustand zurückgefunden und der Parsevorgang wieder aufgenommen werden kann. In diesem Abschnitt soll nur eine Kurzübersicht über Möglichkeiten gegeben werden, wie hierbei verfahren werden kann.

Fehlerbehandlung umfaßt folgende – gestaffelte – Aufgaben:

Reporting. Hiermit wird die Teilaufgabe bezeichnet, aufgetretene Parsefehler zu melden. Als Zusatzinformationen können geliefert werden:

- die Koordinaten des Lookahead-Tokens im Quelltext,
- der Text der Zeile, in der der Fehler erkannt wurde, mit einem Zeiger auf das aktuelle Zeichen,
- das Lookahead-Terminal und
- die möglichen (erwarteten) Terminale. Diese können noch in „Klassen“ gruppiert werden, wie *PCCTS* dies erlaubt: Definiert man beispielsweise eine Klasse „Zahl“ := {Integer, Real}, so erhält man statt der Meldung „Integer oder Real erwartet“ dann „Zahl erwartet“. Wenn das Werkzeug die (für den Benutzer unnatürliche) Benennung von Literalen durch Bezeichner vorschreibt, können auf diese Weise auch Tokennamen durch die entsprechenden Zeichenketten ersetzt werden, beispielsweise durch Definition einer Klasse „`!=`“ := {NOTEQUAL}.

Recovery. *Error Recovery* bezeichnet den Versuch, so viele Eingabeterminale zu überspringen, bis wieder in einem definierten internen Zustand weitergeparst werden kann. Diese Phase braucht nicht implementiert zu werden, wenn nach dem ersten Eingabefehler abgebrochen werden soll.

Der Parsevorgang kann an sogenannten *Restart Points* fortgesetzt werden. Entweder ermittelt das Werkzeug diese aus der Grammatik, wie *Ell* und *Lalr*, oder sie werden vom Spezifizierer explizit angegeben, wie dies in *Yacc* kontextabhängig durch das reservierte Terminal `error` möglich ist. Beispielsweise könnte in Modula-2 bei einem Parsefehler in einer Prozedur ab dem nächsten Semikolon in Erwartung einer Anweisung weitergeparst werden.

Wenn die semantischen Aktionen Seiteneffekte zulassen, ist es problematisch, wieder in einen definierten Zustand zurückzufinden. Bei der Verwendung von *Yacc* treten leicht Speicherlecks bei dem Überspringen von Terminalen auf, wenn diese explizit allozierte Werte tragen. *AUIS-Bison* unterstützt im Fehlerfall die Freigabe dieser Werte. Bietet die Zielsprache einen *Garbage Collector*, so tritt diese Schwierigkeit nicht auf (beispielsweise bei *JavaCup* [Hud96] für Java).

Repair. Einige Werkzeuge versuchen, die Eingabe zu korrigieren, anstatt Teile der Eingabe zu verwerfen. Hierbei wird versucht, durch eine minimale Anzahl von Einfügungen,

Löschungen oder Ersetzungen von Terminalen eine korrekte Eingabe zu konstruieren. Entweder wird nur das Lookahead-Terminal modifiziert (wie bei *Eli*) oder es werden auch die letzten n Terminale mit einbezogen (zum Beispiel 15 Terminale in *ML-Yacc*). Damit die Werte neu erschaffener Terminale nicht undefiniert sind, kann für jeden Tokentyp ein Default-Wert angegeben werden.

Zur Steuerung der möglichen Modifikationen kann in *ML-Yacc* eine Liste der Schlüsselwörter angegeben werden, bei denen eine Korrektur riskant ist, sowie eine Liste der für Einfügungen beziehungsweise Ersetzungen vorzuziehenden Terminale. *Eli* erlaubt weiterhin die Angabe der zulässigen Klammerungskonstrukte (beispielsweise `begin` ... `end`, „(...)“). Eine Reparatur gilt als erfolgreich, wenn in den nächsten n Terminalen kein weiterer Fehler auftritt (*Eli*: $n = 4$). Die semantischen Aktionen dürfen für diese Art der Fehlerkorrektur keine Seiteneffekte haben oder müssen rückgängig gemacht werden können (Backtracking).

Es ist bei Einfügungen wichtig, die Termination zu garantieren. Der Vorteil des Verfahrens ist, daß die nachfolgenden Compilerphasen *immer* eine konsistente Eingabe erhalten, da syntaktisch inkorrekte Eingaben gar nicht möglich sind.

Gerade bei interaktiven System ist es unerläßlich, Fehler abzufangen und weiterzuparsen, da bei einem einfachen Abbruch der interne Zustand verlorengeht.

3.3 Semantische Werte und semantische Aktionen

Um das Erkennen der Eingabe mit Berechnungen verknüpfen zu können, werden für alle Terminale und Nonterminale Slots für sogenannte *semantische Werte* vorgesehen. In statisch getypten Zielsprachen müssen diesen Werten noch Typen zugeordnet werden. Die Werte der Terminale sind – wenn ein Scanner zugrundeliegt – die Tokenwerte; die Werte der Nonterminale müssen berechnet werden (*attributiert*). In diesem Abschnitt wird untersucht, wie diese Aufgabe erledigt werden kann.

Eine einfache Möglichkeit ist die automatische Konstruktion konkreter oder abstrakter Syntaxbäume während des Parsevorgangs, die erst in einem zweiten Durchlauf annotiert werden. *ML-Yacc* beispielsweise verzögert die Ausführung der semantischen Aktionen, wenn diese Seiteneffekte besitzen, bis das Parsen abgeschlossen ist, damit es keine Probleme bei Fehlerkorrekturen gibt. Der Nachteil hiervon ist, daß keine Kontextinformationen verwendet werden können, um Uneindeutigkeiten in Scanner oder Parser aufzulösen, beispielsweise das `typedef`-Problem von C und C++ oder die Uneindeutigkeit bei Designatoren zwischen qualifizierten Bezeichnern und Recordfeldselektion in Oberon-2. Eine eingeschränkte Einbeziehung von Kontextabhängigkeiten schlägt [KVE94] vor: Hier kann mit einer Reduktion höchstens die spezielle Aktion gekoppelt werden, daß einem einelnen Lexem ein neuer Tokentyp zugewiesen wird. Ein Scanner, der dies unterstützt, wird in [Küh94] beschrieben.

Aus diesem Grund werden häufig *semantische Aktionen* direkt in die Syntaxregeln eingebettet. Diese sind genau in der Reihenfolge auszuführen, in der sich die Eingabe durch die Regeln der Grammatik ableiten läßt. Wie bereits bei der Diskussion der Parsetechniken in Abschnitt 3.2 erwähnt wurde, dürfen diese Aktionen nur an bestimmten Stellen der Grammatik stehen und es können mit ihrer Einfügung Parsekonflikte eingeführt werden.

Der wohl interessanteste Aspekt der semantischen Aktionen ist die Frage, auf welche Nonterminalwerte sie zugreifen können und wie. Zur Berechnung sind natürlich genau diejenigen Attribute verwendbar, deren Berechnung bereits abgeschlossen ist und die aus der Aktion referenziert werden können. Hängt die Berechnungsvorschrift eines Attributs ausschließlich von den Nachfolgern eines Knotens ab, so heißt das Attribut *synthetisiert*; wird es aus Werten berechnet, die aus Eltern- oder Geschwisterknoten stammen, so heißt es *vererbt* [ASU86, S. 280].

Bei *Yacc* wird auf die Werte vorangegangener Symbole über deren relative Position im (LR-)Parsestack zugegriffen. Dieser Zugriff über Indizes hat den Nachteil, daß bei der Modifikation von Produktionen sämtliche Referenzen in deren semantischen Aktionen nachgeprüft werden müssen; dieses wird leicht vergessen. Schlimmer wird dies noch bei der Verwendung vererbter Attribute, die nur ohne Überprüfung der korrekten Referenzierung möglich ist. Zum Beispiel könnte man in einer Grammatik für C-ähnliche Deklarationen den Typnamen vererben, wie in folgendem *Yacc*-Quelltext:

```
declaration: type declaredVariables ';';
declaredVariables: declaredVariable;
declaredVariables: declaredVariables ',' mark declaredVariable;
declaredVariable: Variable { enterVarDeclaration($1, $<Type>0); };
mark: { $$ = $<Type>-2; };
```

Mit den Indizes `$0` und `$-2` läßt sich auf vorangegangene Elemente im Parsestack zugreifen. Hierbei wird aber kein Legalitätscheck durchgeführt und auch die Casts in den entsprechenden Typ (`<type>`) werden nicht auf Korrektheit überprüft. Damit der Wert des Nonterminals ‚type' eine konstante relative Position im Parsestack hat, muß er von dem sogenannten *Marker*-Nonterminal ‚mark' wiederholt werden.

Möchte man außerdem EBNF-Konstrukte zulassen, so wird die lineare Bezeichnung durch Indizes unübersichtlich, wie in *Lalr*. Eine moderate Verbesserung ist die zweistufige Bezeichnung durch $\$scope.position$. Es scheint also unerläßlich, die Nonterminalwerte durch Namen zu bezeichnen. Bei *Ell* werden diese Namen automatisch zugewiesen; in der Regel

```
term : fact ( '*' fact | '/' fact )
```

beispielsweise sind die Werte unter den Namen ‚term0', ‚fact1', ‚fact2' und ‚fact3' zugreifbar. Eine individuelle Benennung erlaubt beispielsweise *JavaCup* [Hud96], das einen LR-

Parser generiert, wobei aber keine vererbten Attribute benannt werden können. Namen werden hier durch das Konstrukt `expr:e1` vergeben.

Die komfortabelste Lösung bieten LL-Parsergeneratoren durch die Angabe von formalen und aktuellen Parameterlisten, worüber auch leicht vererbte Attribute spezifiziert werden können. Zum Beispiel würde die Typvererbung auf dem obigen Beispiel in *PCCTS* folgendermaßen formuliert:

```
declaration: <<Type t;>> type > [t] declaredVariables[t] ';';
declaredVariables[Type t]: declaredVariable[t];
declaredVariables[Type t]: declaredVariables[t] ','
                           declaredVariable[t];
declaredVariable[Type t]: <<Variable v;>> Variable > [v]
                          <<enterVarDeclaration(v, t);>>;
```

Wählt man eine Schreibweise, die näher an Algol ist, so ist die Lesbarkeit auch besser.

Eine andere Frage ist, ob das Werkzeug Regel-lokale Variablen unterstützt. In *Yacc* können lokale Variablen nur durch einen Softwarestack implementiert werden. *PCCTS* hingegen erlaubt, wie im letzten Beispiel geschehen, sogenannte *init-actions* zur Allokation und Initialisierung lokaler Variablen auf dem Hardwarestack. Im allgemeinen können lokale Variablen in Tabellenparsern nur schwer gehandhabt, in einem Hard-Coded-Parser aber leicht implementiert werden.

3.4 Das generierte Programm

Die wohl wichtigste Eigenschaft des generierten Analyseprogramms ist dessen Kapselung. Diese hängt stark von den Möglichkeiten der Zielsprache zur Programmstrukturierung ab. In den gängigen Tools finden sich folgende Lösungen:

Programm-basiert. In *Yacc* sind die verwendeten globalen Symbole für die Parsefunktion etc. festgelegt. Aus diesem Grund ist es nicht möglich, mehr als einen generierten Parser in einem Programm zu verwenden; spätestens in der Link-Phase gibt es Symbolkonflikte.

Datei-basiert. In diese Kategorie fällt beispielsweise der *Bison*. Was ihn vom *Yacc* unterscheidet, ist die Möglichkeit, das Standardpräfix für die vordefinierten globalen Symbole (`yy-`) zu ändern. Diese Umbenennungen sind allerdings lästig: sie erfordern ein detailliertes Wissen über sämtliche globalen Symbole aller Programme, in denen die Parser verwendet werden sollen. (Dieses ist ein Grundproblem von C.)

Modul-basiert. Der Generator *Ell* für Modula-2 erzeugt ein eigenes Modul, in dem der Parser läuft. Aus diesem Grund steht die Koexistenz mehrerer generierter Parser im selben Programm nichts im Wege.

Objekt- oder Klassen-basiert. *Mango* für die Programmiersprache Self, *Bison++* für C++ und *JavaCup* für Java erzeugen Objekte beziehungsweise Klassen, die den Parser implementieren. Die Möglichkeiten zur Definition mehrerer Parser im selben Programm sind dieselben wie bei der Modul-basierten Lösung; dieses Verfahren bietet aber den zusätzlichen Vorteil, daß reentrante Parser realisierbar sind. Pro Eingabestrom kann ein Objekt geklont (Self) oder eine Klasse instanziiert werden (C++, Java).

Ein anderer Vorteil der Klassen-basierten Variante ist, daß das Verhalten des Parsers bei Erkennen von Produktionen durch Vererbung und Überladung bestimmt werden kann. So könnte beispielsweise dieselbe Grammatikspezifikation verwendet werden, um einen Interpreter, einen Compiler und einen Pretty-Printer für eine Sprache zu implementieren. Der einzige Generator, bei dem eine Erwähnung dieser Möglichkeit gefunden wurde, war *PCCTS*. Dort wird dieses Verfahren als die Anwendung von *trigger functions* bezeichnet.

Bei einigen Generatoren ist aus der Dokumentation nicht ersichtlich, welchen Restriktionen das generierte Programm obliegt. Es ist anzunehmen, daß relativ häufig die Programm-basierte Lösung verwendet wird.

3.5 Entwurfsentscheidungen

Nach dieser Untersuchung existierender Werkzeuge können die Entscheidungen in bezug auf die Spezifikationssprache des zu entwickelnden Werkzeugs getroffen werden. Sie orientieren sich auch an den Anforderungen aus Abschnitt 1.4. Die Eigenschaften des Parsergenerators sind:

- Der Benutzer soll sich nicht um die Umsetzung der Terminale in eine interne Repräsentation sorgen müssen. Aus diesem Grund werden sie nicht als ganze Zahlen dargestellt, sondern als Atome. Damit sind auch sprechende Namen möglich (wie `':='`). Dies entspricht der Repräsentation, die bereits in Abschnitt 2.3 für den Scannergenerator gewählt wurde. Ein-Zeichen-Literale entsprechen Atomen der Länge 1; für diese werden auch die ganzen Zahlen zwischen 1 und 255 akzeptiert. Ein-Zeichen-Literale werden implizit deklariert, alle anderen Terminale müssen explizit deklariert werden.

- Es wird die Definition mehrerer Startsymbole unterstützt. Nonterminale können durch Atome und Variablen bezeichnet werden. Entsprechend der Unterscheidung zwischen *public*- und *private*-Methoden in Oz gelten die Atome als Startsymbole. Das gewünschte Startsymbol wird durch Angabe seines Namens bei jeder Aktivierung des Parsers ausgewählt.

- Der Benutzer kann sowohl eigene als auch applikationsspezifische EBNF-Konstrukte definieren. Der Mechanismus hierzu ist das *Produktionsschema*: Operatoren werden

bei ihrer Anwendung durch neue Ausdrücke ersetzt, wobei die Argumente entsprechend substituiert werden. Variablen werden dabei eindeutig umbenannt. Neue Regeln können dadurch eingeführt werden. Verbreitete EBNF-Notationen werden durch vordefinierte Produktionsschemata im Prelude zur Verfügung gestellt.

- Nach der Expansion von Produktionsschemata werden einige simplifizierenden und optimierenden Transformationen auf die Grammatik angewendet. Zusammen mit der Möglichkeit, eigene Produktionsschemata zu definieren, kann leicht vom tatsächlich verwendeten Parsealgorithmus abstrahiert werden.

- Als Parsealgorithmus wird zwar LALR(1) gewählt, beim restlichen Entwurf wurde jedoch darauf geachtet, die Abhängigkeiten hiervon in der Spezifikationssprache möglichst gering zu halten. Die Gründe für diese Wahl waren folgende Eigenschaften:

  - LALR(1) ist mächtiger als LL(1) und führt damit zu natürlicheren Grammatikspezifikationen. LL(1) ist ein echter Spezialfall von LALR(1).
  - LALR(1) ist ein sehr verbreiteter Parsealgorithmus. Damit wird die Portierbarkeit existierender Grammatikbeschreibungen für Werkzeuge wie *Yacc* vereinfacht.
  - Operatorpräzedenzen sind direkt umsetzbar.
  - Der Algorithmus ist leicht um dynamische Operatortabellen erweiterbar.

- Präzedenz- und Assoziativitätsinformationen können bei der expliziten Deklaration von Terminalen angegeben werden. Präzedenzen von Regeln werden durch das reservierte Nonterminal `prec` mit einer positiven ganzen Zahl als Argument zugewiesen; das wird bei Simplifikationen wie ein Nonterminal behandelt wird.

- Bei vollkompositionalen Sprachen, für die der Parsergenerator ausgelegt ist, ist eine gute Recovery im Fehlerfall schwer möglich, da es wenige geeignete *Restart Points* gibt. Aus diesem Grund wird kein aufwendiges Fehlerkorrekturverfahren gesucht. Da es Fehlerbehandlungsmöglichkeiten gibt, die keine oder nur wenige zusätzliche Angaben in der Spezifikation benötigen (die eigentlich nichts mit der Sprachspezifikation, sondern nur mit dem Laufzeitverhalten zu tun haben), kann von einem solchen Verfahren ausgegangen werden.

- Auf Werte von Nonterminalen und Terminalen wird über Variablennamen zugegriffen. Nonterminale können beliebig viele Teilwerte besitzen. Die Syntax für Nonterminaldefinitionen und Applikationen von Grammatiksymbolen ist der Methodenanwendung sehr ähnlich; insbesondere wird nicht zwischen synthetisierten und vererbten Attributen unterschieden. Diese Information wird aus der Verwendung der Attribute hergeleitet; somit sind nachprüfbar korrekte Referenzierungen vererbter Attribute möglich (Marker-Nonterminale werden automatisch eingefügt).

- Bei Aufruf des Parsers wird eine Nonterminalapplikation als Record mit übergeben, die zusätzlich zum Startsymbol die initialen Werte aller vererbten Attribute der Startproduktion spezifiziert. Die synthetisierten Attribute werden nach Abschluß des Parsevorgangs mit den korrespondierenden Teilbäumen des Records unifiziert.

- Es werden lokale Variablen in Regeln unterstützt. Diese werden nicht auf dem Hardware-Stack gespeichert, wie bei Hard-Coded-Parsern, sondern auf dem bereits existierenden Softwarestack – dem Parsestack. Alle in geschachtelten EBNF-Ausdrücken verwendeten Variablen haben eine konstante relative Position im Parsestack, daher kann auf diese ohne zusätzliche Marker-Nonterminale zugegriffen werden. Folglich schränken sie nicht die Klasse der erkennbaren Sprachen des Algorithmus ein.

- Das generierte Programm soll – entsprechend dem bereits definierten Scannergenerator – Klassen-basiert sein. Die Kopplung von Scanner und Parser erfolgt durch Definition beider Teile in derselben Grammatik oder durch Definition in verschiedenen Grammatiken und Vererbung.

Konkrete Syntax, Schnittstellen und Realisierung werden in Abschnitt 5.3 vorgestellt.

Kapitel 4

Erweiterungen

In diesem Kapitel wird untersucht, welche weiteren Aufgaben eines Front-Ends durch Werkzeuge unterstützt werden können. Hierbei wird auf den bereits betrachteten Phasen aufgebaut.

In Abschnitt 4.1 wird eine spezielle Form von Baumtransformationen behandelt, nämlich die sogenannten *Ersetzungsregeln*. Sie eliminieren bestimmte Sätze der Sprache, indem sie sie durch äquivalente Konstrukte der Sprache ersetzen, die für den Compiler einfacher als die ursprüngliche Eingabe zu handhaben sind.

Abschnitt 4.2 sucht nach Möglichkeiten, die für alle lexikalisch skopierten Sprachen prinzipiell gleich ablaufende Analyse der Variablenbindungen zu automatisieren. Gleichzeitig soll die in Abschnitt 1.2 bereits motivierte eindeutige Umbenennung aller gebundenen Bezeichner durchgeführt werden.

In vielen Anwendungen ist das einzige Ergebnis der strukturellen Analyse eine interne Baumdarstellung der Eingabe. Abschnitt 4.3 untersucht, wie das Herleiten und Instanzieren geeigneter Strukturen durch das Werkzeug statt explizit durch den Benutzer vorgenommen werden kann.

Sind die Bezüge zwischen der konkreten und der abstrakten Syntax formal beschrieben, so kann das Werkzeug die abstrakte Repräsentation der Eingabe wieder in eine konkrete übersetzen. Dies ist Untersuchungsgegenstand von Abschnitt 4.4. Dies ist in Anwendungen, die Programme in andere in derselben Sprache transformieren (beispielsweise in eine Kernsprache reduzieren), von Bedeutung.

Der letzte Abschnitt (4.5) befaßt sich mit einer Erweiterung, die strenggenommen nicht weitere Compilerphasen unterstützen soll, sondern für den Entwurf relevant ist. Hier wird untersucht, wie die gesamte Spezifikation für ein Werkzeug, das eine oder mehrere der betrachteten Teilaufgaben unterstützt, modularisiert werden kann.

Die strukturelle Analyse aus den vorhergehenden Kapiteln ist Voraussetzung für jede der hier beschriebenen Erweiterungen. Es wird jeweils hervorgehoben werden, wie diese interagieren. Am Ende jedes Abschnittes werden die Entwurfsentscheidungen, ob und wie

die jeweils betrachtete Phase in dem zu entwickelnden Werkzeug realisiert werden soll, präsentiert.

4.1 Ersetzungsregeln

Die Programmiersprache Oz ist in mehreren Schichten definiert. Die unterste Ebene ist Kernel Oz [Smo94]. Dieser Kernsprache enthält alle Konzepte von Oz. Darauf setzt die Oz-Notation [Hen95] auf, die für gängige Programmierparadigmen ausdrucksstarke Konstrukte bietet. Beispielsweise gibt es in der Kernsprache kein `elsecase`, das stattdessen durch geschachtelte `case`-Anweisungen realisiert werden muß. Die Oz-Notation gibt dafür eine *Ersetzungsregel* wie die folgende an:

```
case EL1                    case EL1
elsecase EL2          ⇒     else
⋮                             case EL2
end                           ⋮
                              end
                           end
```

Derartig definierte Sprachen sollen in ihrer Übersetzung eine besondere Unterstützung erfahren.

Die Vorteile, die man sich davon verspricht, werden in Abschnitt 4.1.1 dargestellt. Gleichzeitig wird erläutert, welche Eigenschaften ein Werkzeug, das für das konkrete Problem Unterstützung anbieten soll, haben muß. Abschnitt 4.1.2 untersucht existierende Werkzeuge darauf, welche Ansätze sie zur Lösung dieser Aufgabe beizutragen haben. Die daraus entwickelte Lösung wird in Abschnitt 4.1.3 umrissen.

4.1.1 Ziele

Die folgenden Punkte erläutern, welche Vorteile die Unterstützung von Ersetzungsregeln für den Compilerbauer haben können. Es werden jeweils die nötigen Voraussetzungen beschrieben, die an das Werkzeug gemacht werden.

1. Ist es möglich, Ersetzungsregeln direkt niederzuschreiben, so kann eine bessere Verfolgbarkeit zwischen dem definierenden Dokument und der Implementierung gewährleistet werden, als sie bei einer Ausformulierung von Hand möglich wäre. Das macht ist die Implementierung weniger fehleranfällig und verbessert die Wartbarkeit.

2. Ersetzungsregeln fördern die schrittweise Programmentwicklung. In einem ersten Schritt braucht nur ein Compiler für eine Kernsprache geschrieben zu werden. Viele Konstrukte können zunächst über Ersetzungsregeln ineffizient, aber semantisch

korrekt übersetzt werden. Erst in späteren Verfeinerungen werden effiziente Implementierungen für diese Konstrukte hinzugefügt.

3. Kann für die Notation der Ersetzungsregeln statt einer abstrakten Syntax die konkrete Syntax der zu implementierenden Sprache verwendet werden, so sind die Ersetzungsregeln selbst invariant gegen Änderungen der Definition der abstrakten Syntax.

4. Unter derselben Voraussetzung wie unter Punkt 3 kann der Parsevorgang mit anderen Aktionen gekoppelt werden als lediglich mit der sonst erforderlichen Konstruktion des abstrakten Syntaxbaumes.

5. Unter derselben Voraussetzung wie unter Punkt 3 braucht eine Repräsentation weniger formal dokumentiert zu werden: Es braucht die abstrakte Syntax nicht *vor und nach* den vereinfachenden Transformationen beschrieben zu werden. Weiterhin kann unter Umständen eine geschicktere (einfachere) abstrakte Syntax gewählt werden, wenn für Konstrukte, die ohnehin eliminiert werden, keine gesonderten Knotentypen vorgesehen zu werden brauchen.

6. Wird eine spezielle Notation für Ersetzungsregeln vorgesehen, so kann statisch geprüft werden, ob die Transformation korrekt ist, denn nur korrekte (der Syntax entsprechende) Programmfragmente werden akzeptiert.

7. Können die Ersetzungen gekoppelt mit der strukturellen Analyse spezifiziert und durchgeführt werden, so bleiben Änderungen der Notation sehr lokal, nämlich auf die Parserspezifikation beschränkt. In dem restlichen Front-End braucht man sich der Konstrukte, die durch die Ersetzungsregeln eliminiert werden, gar nicht bewußt zu sein. Dies gilt natürlich nur, solange die Basissprache unverändert bleibt.

8. Unter derselben Voraussetzung wie unter Punkt 7 sind interaktive Systeme realisierbar, da die Ersetzungen nicht erst nach Abschluß des Parsevorgangs durchgeführt werden.

9. Unter derselben Voraussetzung wie unter Punkt 7 können Ersetzungen effizienter implementiert werden, da der abstrakte Syntaxbaum gleich in der definitiven Form aufgebaut werden kann. Es ist kein gesonderter kompletter Durchlauf des abstrakten Syntaxbaumes notwendig, da aus der Reduktion mit bestimmten Produktionen die Anwendbarkeit von Ersetzungsregeln direkt gefolgert werden kann.

Nach dieser Vorab-Untersuchung werden die gemachten Voraussetzungen noch einmal zusammengefaßt:

- Eine spezielle, der intuitiven Schreibweise ähnliche Notation für Ersetzungsregeln sollte zur Verfügung gestellt werden (*specification by example*).

- Die Spezifikation der Ersetzungsregeln sollte nicht von der abstrakten, sondern lediglich von der konkreten Syntax abhängen.

- Die Ersetzungen sollten gekoppelt mit dem Analysevorgang durchgeführt werden, so daß das Ergebnis des Parsens sich nicht von dem unterscheidet, wenn direkt die Normalform (also die maximal simplifizierte Form) der Eingabe bearbeitet worden wäre.

An dieser Stelle soll auf die Zusammenhänge und die Unterschiede zwischen Ersetzungsregeln, wie sie hier beschrieben werden, und Termersetzungssystemen [Ave95] eingegangen werden. Formal sind auch Ersetzungsregeln Definitionen von Äquivalenzen auf Termen, daher ist es wie bei Termersetzungssystemen die Termination für den Implementierer wichtig; gegebenenfalls muß er auch die Church-Rosser-Eigenschaft zusichern. In der Praxis müssen diese Eigenschaften bei Ersetzungsregeln aber – aufgrund ihrer Unentscheidbarkeit – nicht von dem Werkzeug nachgeprüft werden.

Anders als bei den Termersetzungssystemen, deren Regeln auf einer Termrepräsentation (also einer abstrakten Syntax) aufbauen, soll hier die konkrete Syntax zugrundegelegt werden. Außerdem wird nicht verlangt, daß Ersetzungsregeln die volle Mächtigkeit von Termersetzungssystemen haben: Es sollen nur Ersetzungsregeln unterstützt werden, deren Anwendbarkeit aus der Reduktion mit einer bestimmten Produktion gefolgert werden kann. (Insbesondere kann die verwendete Parsetechnik die Allgemeinheit der verwendbaren Ersetzungsregeln einschränken.) Die meisten Ersetzungsregeln in der Oz-Notation erfüllen dieses Kriterium, daher stellt es keine große Einschränkung dar.

4.1.2 Ansätze

In diesem Abschnitt werden existierende Mechanismen untersucht, mit denen sich Ersetzungsregeln formulieren lassen. Dabei wird hervorgehoben, welche der oben beschriebenen Forderungen der jeweilige Ansatz erfüllt.

Die meisten der Sprachen, die speziell für regelbasierte Transformationen entworfen wurden (wie *Gentle* [Sch89], [Sch], *Rigal* [Aug93] oder *puma* [Gro91a], [Gro91b]), arbeiten auf der Ebene einer abstrakten Syntax. Im Prinzip ist das einzige ausgezeichnete Konstrukt, das sie für die Formulierung von Ersetzungsregeln bieten, das Pattern-Matching mit Unifikation. Zudem muß der rekursive Abstieg mit den Regelanwendungen meist von Hand ausformuliert werden.

Die Sprache *TXL* [CCH95] hingegen übernimmt selbst den rekursiven Abstieg. Dort werden nach Abschluß des Parsevorgangs die vom Benutzer angegebenen Regeln angewendet. Mehrere Arten von Regeln können definiert werden:

- Transformationsfunktionen, die eine Ersetzung auf der obersten Ebene des Terms durchführen, auf den sie angewendet werden (in den oben erwähnten Werkzeugen ist dies die einzige angebotene Regelart),
- Suchfunktionen, die den Baum durchsuchen (*leftmost-innermost*) und die erstmögliche Ersetzung durchführen,

- Transformationsregeln, die den Baum durchsuchen und die Ersetzung an jeder Stelle durchführen, wo sie anwendbar ist. Dabei kann aus einer Ersetzung wieder ein Match für dieselbe Regel entstehen.

Die gewünschten Ersetzungsregeln entsprechen dabei den Transformationsregeln. Diese bieten aber nicht alle benötigten Eigenschaften: Die Anwendbarkeit der Regeln kann nicht statisch aufgelöst werden (also mit einer Reduktion mit bestimmten Produktionen gekoppelt werden), der Baum muß also dynamisch durchsucht werden, und dieses in den meisten Fällen sogar mehrfach – die Muster werden nicht in einen einzigen Vergleichsbaum gemischt, sondern strikt sequentiell in der von dem Benutzer angegebenen Weise angewendet.

Einen anderen Ansatz bietet der *Kimwitu Term Processor* [vEB93]. Dieser ist sehr nah an den Reduktionssystemen: Es wird ein Reduktionssystem definiert, aus dem Datentypen sowie Rewrite- und Ausgabe-Funktionen für Terme für die Programmiersprache C erzeugt wird. Diese können dann in Programmen verwendet werden, um zu einem Term dessen Normalform zu bilden. Die Regeln entsprechen den Transformationsregeln von *TXL*, aber die Anwendbarkeit aller Regeln wird gleichzeitig getestet.

Der vielversprechendste Ansatz ist der aus [CMA94]. Dort werden die Ersetzungsregeln als semantische Aktionen in der Grammatik angegeben, also insbesondere verschränkt mit der Analyse angewendet. Aus diesem Grund sind sie auch potentiell mit anderen semantischen Aktionen als nur dem Aufbau einer abstrakten Syntax koppelbar: Für Knotenkonstruktionen muß der Benutzer Funktionen liefern, die gleichzeitig auch andere Aktionen ausführen können. Die Ersetzungsregeln werden in der konkreten Syntax formuliert, indem ein Nonterminal angegeben wird, mit dem die rechte Seite geparst werden soll. Dabei werden entsprechende semantische Aktionen zur Übersetzzeit erzeugt, die dieselben Funktionen ausführen, als wäre die vereinfachte Form direkt eingegeben worden. Der Nachteil hiervon ist, daß eine vorgegebene lexikalische Struktur für die Umwandlung der konkreten Syntax in einen Tokenstrom vorausgesetzt wird (aufgrund der ASCII-Beschränkung sind Platzhalter – Metavariablen – nur mit zusätzlichen Forderungen von Bezeichnern der Sprache selbst unterscheidbar).

4.1.3 Entwufsentscheidungen

Die Lösung, die für das zu entwickelnde Werkzeug realisiert wurde, ist von dem Ansatz aus [CMA94] inspiriert. Dieser wurde modifiziert, um alle gewünschten Eigenschaften zu bieten. Das Ergebnis sieht wie folgt aus:

- Ersetzungsregeln dürfen überall dort stehen, wo in der Grammatik semantische Aktionen zugelassen sind. Sie werden bei der Parsergenerierung in semantische Aktionen übersetzt. Aus diesem Grund werden sie auch verschränkt mit dem Analyseprozeß angewendet.

- Die Produktion, mit deren Reduktion die Ersetzungsregel gekoppelt ist, liefert gleichzeitig deren linke Seite. Diese Verbindung wird durch ihre Plazierung in der Grammatik hergestellt. Das Verfahren ist einem Pattern-Matching-Prozeß gleichbedeutend, aber auf Ebene der konkreten Syntax. Das nachträgliche ineffiziente Durchsuchen des abstrakten Syntaxbaumes nach den Stellen, an denen Regeln angewendet werden können, entfällt. Die durch den Parsealgorithmus bedingten Einschränkungen an die Klasse der verwendbaren Ersetzungsregeln sind bei dieser Vorgehensweise leicht erfaßbar.

- Die rechte Seite der Ersetzungsregel wird durch das Nonterminal, mit dem sie geparst werden soll, sowie eine Folge von Applikationen gegeben. Jedes von diesen wird als Tupel repräsentiert: Das Label gibt den Tokentyp beziehungsweise das Nonterminal an, die Features die Parameter. Damit ist die rechte Seite nur von der konkreten Syntax abhängig, aber von lexikalischen Konventionen entkoppelt (wenn auch nicht von der Repräsentation der Token). Allerdings wird dadurch an Lesbarkeit eingebüßt. Die Tokentypen müssen statisch feststehen; für die Tokenwerte bleiben komplexe Berechnungen möglich.

Die konkrete Syntax und die Lösung werden zusammen mit einigen Beispielen in Abschnitt 5.4 im Detail vorgestellt.

4.2 Bindungsanalyse

Die *Bindungsanalyse* ist der Prozeß, der zu jedem Variablenauftreten die zugehörige Definition ermittelt. Diese Aufgabe ist Teil jedes Compilers, da die dabei ermittelte Information sowohl für die semantische Analyse (Überprüfung der Korrektheit von Referenzen, Type-Checking) als auch für die Codegenerierung (Speicher-/Registerzugriffe) unbedingt benötigt wird.

Üblicherweise wird die Bindungsanalyse von der Symboltabellenverwaltung übernommen. Eine Lookup-Funktion liefert zu einem Variablennamen dessen korrespondierenden Eintrag. Diese Komponente kann gleichzeitig alle gebundenen Bezeichner eindeutig umbenennen, da ihr die hierzu benötigten Informationen sowieso zur Verfügung stehen. Dies kann gegebenenfalls die Implementierung von Baumtransformationen unter Vermeidung von *name clashes* vereinfachen, wie beispielsweise Inlining von Prozeduraufrufen, Instanziierung von generischen Modulen oder Expansion von Typdefinitionen. Weiterhin könnte sie bei Bedarf frische (also im Quelltext nicht verwendete) Variablen erzeugen.

In Abschnitt 4.2.1 werden die Vorteile erläutert, die eine Automatisierung der Bindungsanalyse mit sich bringen kann, und die hierzu benötigten Eigenschaften herausgearbeitet. Abschnitt 4.2.2 stellt einige Ansätze aus der Literatur vor. Zuletzt werden in Abschnitt 4.2.3 die Entwurfsentscheidungen für das zu entwickelnde Werkzeug in bezug auf die Bindungsanalyse getroffen.

4.2.1 Ziele

Zunächst sollen einige Programmiersprachen repräsentativ daraufhin untersucht werden, wie bei ihnen Bezeichnerbindungen aussehen. Diese Sprachen bauen alle auf der lexikalischen Skopierung auf, was bedeutet, daß die Definition, auf die sich ein jeder Bezeichner im Quelltext bezieht, zur Übersetzungszeit ermittelt werden kann. Zusätzlich verwenden die meisten Sprachen, die in geschachtelte Sichtbarkeitsbereiche (*Blöcke*) strukturiert sind, noch die *most closely nested*-Regelung. Dort wird die von einer Variablen referenzierte Definition immer in den umgebenden Blöcken gesucht. Existieren Variablendefinitionen mit dem gesuchten Namen in mehreren umgebenden Blöcken, so ist die aus dem innersten Block die gültige.

Wird von diesem einfachen Modell ausgegangen, so ist einfach herzuleiten, welches Wissen ein Werkzeug, das die Bindungsanalyse automatisiert, über die zu übersetzenden Sprache haben muß: Zum einen muß spezifiziert werden, welche Konstrukte einen Block im obigen Sinne darstellen, zum anderen müssen die Variablenauftreten klassifiziert werden, indem zwischen bindenden und referenzierenden Auftreten unterschieden wird.

Es sollen nun einige Beispiele von Konstrukten aus existierenden Programmiersprachen angegeben werden, die über das oben beschriebene Verfahren hinausgehen. (Dabei sollen technische Feinheiten ignoriert werden.)

- Die Programmiersprache Pascal [JW75] bietet mit `with ... do` ein Konstrukt, daß in einem Block implizite Deklarationen einführt: Die Felder einer recordwertigen Variablen sind in diesem Block ohne Angabe des Variablennamens zugreifbar.

- Unter anderem in Oberon-2 gibt es vordefinierte globale Bezeichner (ohne, daß sie vorher aus einem anderen Modul importiert worden wären). Die Bindungsanalyse startet also nicht mit einer leeren Symboltabelle.

- In Oberon-2 können Symbole exportiert werden. Informationen über diese werden in für jedes Modul angelegten Symboldateien abgelegt. Die bei der Umbenennung erstellte Substitution des globalen Sichtbarkeitsbereiches muß hier explizit verarbeitet werden.

- Beispielsweise bei dem Modulkonzept von Ada [MW91] ist es möglich, alle (sichtbaren) Definitionen eines anderen Moduls *unqualifiziert* zu importieren, was bedeutet, daß sie unter ihrem Namen zugänglich sind, ohne mit ihrem Modulnamen gekennzeichnet zu werden. Dabei werden Bezeichner implizit deklariert, ohne daß ihr Name im Quelltext des importierenden Moduls bindend auftauchen muß.

- In W-Lisp (Version 3) [WKE96] werden Variablen implizit durch ihre Verwendung deklariert, wenn sie bisher unbekannt waren. Weiterhin dürfen Variablen nicht in eingeschachtelten Blöcken verdeckt werden.

- In Oz können Variablen, die private Features, Attribute oder Methoden bezeichnen, in der Klasse, der diese angehören, auch vor ihrer Definition verwendet werden.

Es muß also ein Formalismus gefunden werden, der mächtig genug ist, derartige Abweichungen vom einfachsten Modell der lexikalischen Skopierung zu spezifizieren.

Von einer automatischen Unterstützung der Bindungsanalyse werden folgende Eigenschaften erwartet:

- Es soll die eindeutige Umbenennung der gebundenen Variablen unterstützt werden. Dabei sollen die definitiven Namen möglichst schnell ermittelt werden, also nicht erst nach Abschluß des Parsevorgangs. Die Bindungsanalyse läuft somit verschränkt mit dem Parsen ab. Dies führt dazu, daß die Komponente auch zu beliebigen Momenten frische Variablen erzeugen kann, was die Ersetzungsregeln aus dem vorhergehenden Abschnitt deutlich mächtiger macht: Ersetzungsregeln können dann lokale Variablenbindungen einführen, ohne daß dabei *name clashes* resultieren könnten. Zum Beispiel gab es in Oz in der Version 1.1.1 folgende Ersetzungsregel:

$$\texttt{thread}\ E\ \texttt{end} \quad \Rightarrow \quad \begin{array}{l}\texttt{local}\ X\ \texttt{in}\\ \quad \texttt{if}\ X = \texttt{'go'}\ \texttt{then}\ E\ \texttt{fi}\\ \quad X = \texttt{'go'}\\ \texttt{end}\end{array}$$

  Dabei muß X eine frische Variable sein, damit keine Konflikte mit den freien Variablen aus dem Ausdruck E auftreten können.

- Die Bindungsanalyse muß zwar verschränkt mit dem Parsevorgang durchgeführt werden können, sollte aber nicht streng daran gebunden sein. Es sollte also möglich sein, ihre Funktionalität auch in handgeschriebenen (im Gegensatz zu generierten) Funktionen zu verwenden, beispielsweise, wenn ein Teil des abstrakten Syntaxbaumes mit gleichzeitiger eindeutiger Umbenennung aller gebundenen Variablen kopiert werden soll.

- Die Komponente, die die Bindungsanalyse durchführt, sollte zwecks Arbeitsersparnis durch den Benutzer zu einer voll funktionsfähigen Symboltabellenverwaltung erweitert werden können.

- Es soll die Unterstützung mehrerer verschiedener Variablensorten vorgesehen werden, für die verschiedene Sichtbarkeitsregelungen gelten können. Dies ist bei Sprachen der Fall, in die andere Sprachen eingebettet sind, wie beispielsweise Datenbankenanfragen in Erweiterungen von C oder Grammatikdefinitionen in Oz, wie sie von dem zu entwickelnden Werkzeug realisiert werden. Solch eine Möglichkeit ist auch anwendbar, wenn Namen von Recordfeldern oder Label wie in C oder Java global eindeutig umbenannt werden sollen.

- Das Werkzeug soll von der genauen Darstellung der Variablenterminale und ihrer Werte unabhängig bleiben. Dies hat den Vorteil, daß beliebige Zusatzinformationen von dem Benutzer in den Knoten gespeichert werden können, etwa Quelltextpositionen, die ursprünglichen Namen für Diagnoseausgaben oder *print names*, also zwar generierte, aber doch sprechende und informative Namen.

4.2.2 Ansätze

In diesem Abschnitt soll untersucht werden, welche in der Literatur gefundenen Ansätze die Zuordnung von referenzierenden und bindenden Auftreten von Bezeichnern durchführen.

Zur Spezifikation der Sichtbarkeitsregelungen von Programmiersprachen ist die Sprache *VisiCola* [Klu91] entwickelt worden. Dabei wurde versucht, möglichst alle Bestandteile zu identifizieren, aus denen sich eine derartige Spezifikation aufbauen läßt. Das Ergebnis ist eine reine Spezifikationssprache, bei der alles sehr viel detaillierter beschrieben werden muß, als es für die hier gestellte Aufgabe erforderlich ist. Der Autor führt einige Beispiele an: Die Spezifikation der Sichtbarkeitsregelungen von Pascal benötigen 5000 Zeilen, die von Modula-2 (mit Ausnahme der Aufzählungskonstanten) 1280 Zeilen. Dieser Ansatz ist also für das vorliegende Problem ungeeignet.

In der Sprache Scheme können sogenannte *hygienic macros* [HDB92] definiert werden. Diese entsprechen vom Prinzip her den Ersetzungsregeln gekoppelt mit Bindungsanalyse, um *name clashes* zu vermeiden. Zur eindeutigen Identifikation der Variablen wird ein Multi-Pass-Zeitstempel-Algorithmus angewandt. Dieser operiert auf der vollständig konstruierten abstrakten Syntax einer Scheme-Form. Dort ist die Situation einfacher als im vorliegenden Fall, da von einer festen Ausgangssprache ausgegangen werden kann, die zudem Variablenbindungen zuläßt, die sehr einfach ermittelt werden können (insbesondere existieren in Scheme die beiden unterschiedlichen Namensräume von Common-Lisp für Funktionen und Variablen nicht).

In [CMA94] wird ein Ansatz vorgestellt, der die Klassifikation der Variablenauftreten gekoppelt mit der Grammatikspezifikation vornimmt. Dort werden die den Variablen entsprechenden Terminale mit einer Annotation versehen, die sie als **`binder`**, **`variable`** (Referenz) oder **`label`** auszeichnen; es sind also nur zwei vordefinierte Sichtbarkeitsregelungen verwendbar – die für normale Variablen, die der Blockstruktur folgt, und die für Label, die auf einen einzigen globalen Scopus beschränkt ist. Blockgrenzen für Sichtbarkeitsbereiche können nicht explizit spezifiziert werden: Jede Produktion stellt einen Block dar. Frische Variablen können durch Angabe eines Variablenterminals mit der Annotation **`fresh`** erzeugt werden.

4.2.3 Entwurfsentscheidungen

Der hier realisierte Ansatz ist eine Modifikation des Ansatzes aus [CMA94]. Die Lösung sieht grob skizziert wie folgt aus:

- Über die Terminalannotationen **`binder`** und **`reference`** werden die Variablenauftreten klassifiziert. Zusätzlich kann der Name einer Variablensorte angegeben werden, der das Terminal angehört (zum Beispiel Programmvariable, Grammatiksymbol oder Label). Dadurch sind die beiden Variablensorten, die der Ansatz aus [CMA94] unterstützt, emulierbar, aber um beliebig viele weitere erweiterbar.

- Durch die Terminalannotation **`fresh`** wird eine frische Variable erzeugt.

- Die Grammatik wird um ein weiteres Klammerungskonstrukt erweitert, nämlich um **`scope`** ... **`end`**. Dabei kann eine Variablensorte angegeben werden. Das Konstrukt spezifiziert die Blockgrenzen eines Sichtbarkeitsbereiches der Variablensorte. Dadurch, daß es sich um eine Klammerung handelt, sind diese Blöcke immer geschachtelt. Dieses System ist weniger restriktiv als das aus [CMA94].

- Es existiert eine vordefinierte Klasse namens ‚BindingAnalysis', die die notwendigen Informationen speichert und verarbeitet. Sie wird einmal pro Variablensorte instanziiert. Sie erlaubt die operationale Umsetzung der deklarativen Grammatikannotationen und bietet die Operationen `openScope` und `closeScope` für die Angabe der Blockgrenzen. `mkBinder`, `mkLabel` und `mkFresh` entsprechen den Terminalannotationen; mit `enterSubstitution` können implizite Deklarationen bekanntgegeben werden.

- Durch Erweiterung der Klasse ‚BindingAnalysis' kann eine vollwertige Symboltabellenverwaltung implementiert werden.

- Der Benutzer muß in einer von ‚BindingAnalysis' abgeleiteten Klasse Methoden bereitstellen, die die Abstraktion von der Terminalrepräsentation übernehmen. Hierzu zählen folgende Operationen:

  anonymize wandelt ein Terminal in eines, bei dem für den Namen eine (nicht determinierte) Variable eingesetzt wird.

  nameToken ersetzt den Platzhalter in einem anonymisierten Token durch einen konkreten Namen (als Atom).

  generate erzeugt einen neuen Namen (als Atom). Diese Funktion kann aus einem vom Benutzer spezifizierten regulären Ausdruck erzeugt werden in dem Sinne, daß sie die Elemente der Sprache, die durch den Ausdruck definiert wird, aufzählt.

  tokenToAtom liefert ein Atom, das den (benutzergegebenen) Namen der Variablen identifiziert. Dieses wird zur internen Verwaltung der Substitutionen verwendet. Sind nicht alle Zeichen des Lexems relevant (beispielsweise in UCSD Pascal, wo nur die ersten 8 Zeichen gelten) oder gibt es mehrere Repräsentationen für dasselbe Objekt (wie in Oz, Atome mit und ohne Quotes), so können diese Abhängigkeiten von dieser Funktion transparent gemacht werden.

makeTokenFromAtom erzeugt aus einem Atom ein entsprechend benanntes Variablentoken. Diese Operation wird ausschließlich in Kombination mit ‚fresh' verwendet.

Die Vorteile dieses Verfahrens gegenüber [CMA94] sind offensichtlich. Zum einen ist die Bindungsanalyse nun orthogonal zu den Ersetzungsregeln. Weiterhin kann von der Repräsentation der Token abstrahiert werden, also können insbesondere Print-Names vorgesehen werden, anstelle einfach sequentiell die Bezeichner durchzunumerieren. Die Substitution kann als First-Class-Datenstruktur explizit gemacht und somit weiterverarbeitet werden (zum Beispiel für die Ausgabe einer Symboldatei). Durch die Trennung zwischen deklarativen Annotationen der Grammatik und dem operationalen Teil (in der Klasse ‚BindingAnalysis') kann letzterer auch unabhängig vom übrigen System verwendet werden.

Die konkrete Syntax der Annotationen und die Beschreibung der Klasse ‚BindingAnalysis' werden in Abschnitt 5.5 gegeben.

4.3 Automatische Konstruktion abstrakter Syntaxen

Häufig soll das Ergebnis der Parsephase eine interne, strukturierte Repräsentation des eingegebenen Programmes sein. In diesem Kapitel werden Möglichkeiten untersucht, diese Transformation durch ein Werkzeug zu unterstützen. Zunächst muß jedoch in einigen Vorüberlegungen identifiziert werden, welche Eigenschaften von einer geeigneten internen Darstellung erwartet werden.

Während der lineare Tokenstrom durch eine *konkrete Syntax* beschrieben wird, wird mit dem Begriff *abstrakte Syntax* eine Darstellung bezeichnet, die von Einschränkungen oder von Überspezifikation abstrahiert, die sich durch die Forderung der Parsebarkeit und/oder Lesbarkeit ergeben. Dieser Begriff kann etwas präziser definiert werden: Eine Syntax a gilt genau dann als abstrakter als eine Syntax b, wenn ein totale Funktion von a nach b existiert, die umgekehrte Relation jedoch keine Funktion ist [Bev93]. In der Praxis ist diese Form der Abstraktion gleichbedeutend mit dem Entfernen inessenzieller Details aus der Darstellung.

Es ist offensichtlich, daß es zu einer konkreten Syntax beliebig viele abstrakte Syntaxen geben kann. Diese können jedoch verschieden gut für die Verarbeitung in den nachfolgenden Phasen geeignet sein, analog zu den in Abschnitt 3.2 beschriebenen Auswirkungen verschiedener Formulierungen der Grammatik für die konkrete Syntax. Für die abstrakte Syntax werden sogar folgende Forderungen aufgestellt:

- Zwischen den Baumknoten der abstrakten Syntax und den semantischen Konzepten der Sprache soll eine eins-zu-eins-Beziehung bestehen. Diese kann in der konkreten Syntax üblicherweise nicht hergestellt werden.
- In die Struktur einer Syntax können unterschiedlich viele Kontextbedingungen einer Sprache (auch *statische Semantik* genannt) kodiert werden. Eine abstrakte Syntax

soll hiervon möglichst intensiv Gebrauch machen, da strukturelle Einschränkungen häufig einfacher zu verstehen sind als solche, die (in der semantischen Analyse) durch beliebige Prädikate ausgedrückt werden.

Die Güte einer abstrakten Syntax kann durch eine Analyse der Compilerphasen, die auf ihr operieren, evaluiert werden. Es muß hierzu jeweils identifiziert werden, welche Informationen benötigt und welche produziert werden. Bei einer guten abstrakten Syntax sind die benötigten Kontextinformationen jdedes Knotens in einem einfachen rekursiven Abstieg leicht zugreifbar.

Aus diesen Gründen ist eine abstrakte Syntax auch besser als eine konkrete für die Entwicklung und Semantikdefinition von Programmiersprachen geeignet. Es wird oft dazu geraten, mit einer abstrakten Syntax zu beginnen, für die erst dann eine konkrete Syntax definiert wird, wenn alle Konzepte der Sprache gut verstanden worden sind. In der Praxis muß jedoch häufig der umgekehrte Weg beschritten werden: Viele Sprachreporte geben nur die konkrete Syntax vor. Für einen Compilerbauer, der bereits Implementierungserfahrung mit den Paradigmen der Sprache besitzt, für die ein Übersetzer entwickelt werden soll, ist das Entwerfen einer abstrakten Syntax meist nicht schwer. So kommt schnell der Wunsch nach Werkzeugunterstützung auf.

In Abschnitt 4.3.1 werden die Aspekte identifiziert, die für eine Werkzeugunterstützung relevant sind. Es werden weiterhin die Anforderungen an eine solche formuliert. Eine Übersicht über existierende Ansätze wird in Abschnitt 4.3.2 gegeben. Zuletzt wird in Abschnitt 4.3.3 begründet, warum in dem hier entwickelten Werkzeug keine Unterstützung für die automatische Konstruktion abstrakter Syntaxen erfolgt.

4.3.1 Ziele

Aus der Definition des Begriffs *abstrakte Syntax* lassen sich drei Aspekte ableiten, die bei eine Automatisierung mit einbezogen werden müssen: 1) die Spezifikation der konkreten Syntax k durch eine Grammatik, 2) die Herleitung einer abstrakten Syntax a, die beispielsweise durch eine Typdefinition gegeben wird, und 3) die Abbildung $\phi : k \longrightarrow a$ (eine totale Funktion).

Die Entwicklung eines Werkzeugs, das diese Transformation eigenständig durchführt, kommt der Festlegung einer begrenzten Klasse von erzeugbaren abstrakten Syntaxen gleich. Anders ausgedrückt: Sei K die Menge aller konkreten Syntaxen. Dann wird die Funktionsweise des Werkzeugs durch die Menge der abstrakten Syntaxen A, die es erzeugen kann, sowie durch eine Funktion $\Phi : K \longrightarrow A$ definiert. (Damit sind mit den obigen Bezeichnungen $a \in A$, $k \in K$ und $\phi \in \Phi$.)

Mit dieser Formulierung der Aufgabe ist klar, in welchen Aspekten sich Lösungen unterscheiden können:

- Ein wichtiger Aspekt ist die Gestalt von A, das heißt, mit welchen Datenstrukturen der Zielsprache die abstrakte Syntax aufgebaut wird. Hier können grob drei Möglichkeiten unterschieden werden.

  Tupel. Unter Tupeln werden hier solche Tupel wie in Oz oder Produkttypen wie in Gofer verstanden, also das Cartesische Produkt von n Typen, versehen mit einer benannten Markierung (*Label* oder *Konstruktor* genannt). Diese wird entweder explizit gespeichert oder durch die Typisierung der Zielsprache verwaltet. Tupel stellen aufgrund ihrer Ähnlichkeit sowohl zu BNF-Produktionen als auch zu Bäumen mit benannten Knoten die einfachste und verbreiteste Form abstrakter Syntax dar. Wichtige Aspekte sind, für welche semantischen Konzepte Konstruktoren definiert werden und welche Teilwerte jeweils gespeichert werden müssen.

  Records. Als Records werden hier Tupel bezeichnet, deren Elemente durch *Feldnamen* angesprochen werden. Es kommt die Notwendigkeit hinzu, Namen für diese Felder zu finden. Angenommen, es bestehe eine Beziehung zwischen bestimmten Produktionen und den Konstruktoren; dann gibt es zwei Möglichkeiten der Benennung: Entweder kann der Benutzer die Feldnamen durch Annotation der Produktionen vergeben oder das Werkzeug erzeugt die Namen aus den in der Produktion verwendeten Grammatiksymbolen. Dabei wird die Eindeutigkeit der Namen durch einen angehängten Index zugesichert, der das Auftreten in der Produktion kennzeichnet. Der Vorteil einer Recorddarstellung ist, daß diese in der Verwendung weniger fehleranfällig sowie beständiger gegen Modifikationen der abstrakten Syntax ist.

  Objekte. Bei einer objektorientierten abstrakten Syntax kommen weitere grundlegend verschiedene Strukturierungsmöglichkeiten hinzu: Zwischen den einzelnen Knotentypen können nun Vererbungsbeziehungen bestehen (ein Knotentyp ist eine Spezialisierung eines anderen) und es können Eigenschaftsklassen (auch: Mixin-Klassen) definiert werden. Dadurch, daß mit den Knotentypen direkt Operationen (Methoden) verbunden werden, ändert sich außerdem die Implementierung der Compilerphasen erheblich: An die Stelle einer Fallunterscheidung bezüglich des Konstruktors eines Tupels oder Records tritt die dynamische Bindung. Eine weitere Implikation ist, daß hier die Wiederverwendung von Eigenschaftsklassen für verbreitete Konzepte oder Paradigmen von Programmiersprachen möglich wird. Aus diesen Gründen ist die Erstellung einer abstrakten Syntax in Objektform ein weit weniger trivialer Prozeß als die Verwendung von Tupeln oder Records, wenn diese Möglichkeiten augeschöpft werden sollen.

- Der andere wichtige Aspekt ist, wie abstrahierend die Funktionen ϕ sind, die das Werkzeug verwendet. Wird Φ bei der Entwicklung des Werkzeugs als konstant festgelegt, werden also abstrakte Syntaxen ausschließlich aus der konkreten Syntax hergeleitet, so können keinerlei Kontextabhängigkeiten (die nicht in die kontextfreie

Grammatik kodiert werden können) einbezogen werden. Dieser Fall wird auch als eine *abstrakte Syntax erster Ordnung* bezeichnet [Bev93]. Ansonsten muß der Benutzer zusätzliche Angaben machen können, die eine Steuerung der erzeugten Funktion ϕ erlauben.

Abgesehen von diesen grundsätzlichen Fragestellungen sind folgende Aspekte für eine Werkzeugunterstützung wichtig:

- Lokale Änderungen der Grammatik sollen nur lokale Änderungen der abstrakten Syntax bewirken. Diese Forderung ist wichtig für in der Entwicklung befindliche Sprachen oder für die Entwicklung eines Compilers durch schrittweise Verfeinerung.

- Das Werkzeug soll sowohl die Definition der abstrakten Syntax a generieren, als auch das zugrundeliegende ϕ auf die Eingabe anwenden. Das Resultat wird als der *abstrakte Syntaxbaum* bezeichnet.

- Der automatische Aufbau einer abstrakten Syntax soll auf Teile der Grammatik eingeschränkt werden können. Bei interaktiven Systemen beispielsweise muß dieser Vorgang auf Teilbäume begrenzt werden können: Jede Verarbeitungseinheit einer textbasierten Benutzungsschnittstelle (wie eine Zeile oder ein Befehl) kann automatisch in eine interne Darstellung gewandelt werden, jedoch muß diese dann sofort verarbeitet werden. wichtiger ist diese Forderung, wenn tief im Baum Seiteneffekte ausgeübt werden müssen, etwa *lexical tie-ins* oder Modifkationen der Symboltabelle.

Als Vorteile der Werkzeugunterstützung erhofft man sich nicht nur eine kompaktere und besser lesbare Spezifikation der Transformation in eine interne Darstellung oder Arbeitsersparnis. Ein viel wichtiger Aspekt ist, daß eine formal beschriebene abstrakte Syntax für die Automatisierung anderer Compilerphasen genutzt werden kann: Für viele Werkzeuge, die Pattern-basierte Baumtransformationen, eindeutige Umbenennung der gebundenen Bezeichner oder automatische Attributierung übernehmen, ist dies eine grundlegende Voraussetzung. Die oben vorgestellten Ersetzungsregeln und Bindungsanalyse stellen hier eher die Ausnahme dar.

4.3.2 Ansätze

Die Erzeugung einer abstrakten Syntax erster Ordnung in Tupel- oder Recordform aus einer konkreten Syntax ist ein sehr simpler Vorgang. Es werden einfach folgende Umsetzungen der Bestandteile einer (E)BNF vorgenommen:

Konstante Terminale werden eliminiert. *Konstant* bedeutet hierbei, daß das Terminal keinen interessanten Wert trägt, wie dies bei Literalen üblicherweise der Fall ist.

Variable Terminale werden direkt auf einen ihnen zugeordneten eindeutigen Typnamen mit einem einzigen Konstruktor abgebildet. *Variabel* bedeutet hierbei, daß das Terminal einen *Intrinsic*-Wert trägt, etwa den Zahlenwert eines Integer-Tokens.

Nonterminale werden ebenso wie variable Terminale auf Typnamen abgebildet. Diese können allerdings mehrfache Konstruktoren enthalten (siehe unten).

Sequenzen werden zunächst in ihre Bestandteile zerlegt, für die jeweils die abstrakte Repräsentation bestimmt wird. In diesem Moment muß eine Fallunterscheidung gemacht werden: Bleibt ein einziges Element übrig, so handelt es sich um eine sogenannte *Kettenregel* und dessen Konstruktoren werden in den aktuell erstellten Typ übernommen. Ansonsten werden die Ergebnisse mit einem neuen Konstruktor (für einen Produkttyp mit entsprechenden Elementen) versehen. Die eindeutige Benennung dieses Konstruktors kann entweder durch eine Benutzerangabe oder durch die Konkatenation des Nonterminals, zu dem die betrachtete Produktion gehört, mit der Nummer der Produktion erfolgen.

Alternativen bestehen aus mehreren Sequenzen, deren abstrakte Darstellungen jeweils ermittelt werden. Die dabei identifizierten Konstruktoren werden in den aktuell erstellten Typ übernommen.

Klammerungen werden in neue Regeln ausgelagert. Der Name des neuen Nonterminals kann entweder durch Benutzerangabe erfolgen oder durch die Generierung eines beliebigen eindeutigen Grammatiksymbols.

Optionen werden dadurch aufgelöst, daß ein Defaultwert verwendet wird, wenn die Option in der Eingabe nicht auftaucht. Dieser kann entweder vom Benutzer spezifiziert oder durch den Generator vorgegeben werden.

Wiederholungen werden in einen primitiven Listentyp der Zielsprache übersetzt.

Die resultierende Typdefinition kann ebenfalls durch eine kontextfreie Grammatik beschrieben werden; diese braucht – da Elemente der durch sie definierten Sprachen ausschließlich als Baum mit benannten Knoten gespeichert werden – jedoch weder eindeutig noch den Forderungen irgendeiner Parsetechnik konform zu sein.

Beispiele für Werkzeuge, die in dieser Art (oder noch einfacher – ohne EBNF oder Elimination von konstanten Terminalen und/oder Kettenregeln) vorgehen, sind *TXL* [CCH95] oder der Ansatz aus [Wad90]. Bei dem Werkzeug *Maptool* [KW95] aus dem *Eli*-System entspricht dies ebenfalls der vorgegebenen Beziehung zwischen der konkreten und der abstrakten Syntax. Der Benutzer kann jedoch durch spezielle Direktiven diesen Prozeß steuern und die abstrakte Syntax damit tatsächlich abstrakter als die konkrete machen: Es können Knoten verschiedener Nonterminale miteinander identifiziert (wodurch beispielsweise Kettenregeln aufgelöst werden können), Grammatiksymbole eliminiert oder für dasselbe Nonterminal – abhängig von dessen Kontext – verschiedene Knotentypen eingesetzt werden.

An den Stellen, an denen sich die konkrete und die abstrakte Syntax (mit obigen Regeln) entsprechen, braucht nur eine der beiden angegeben zu werden, mit der die jeweilige andere dann automatisch vervollständigt wird.

Eine alternative Vorgehensweise bietet das Werkzeug *PCCTS* [Par95]. Hier werden nur eingeschränktere Baumstrukturen unterstützt (aus denen allerdings auch komplizierte aufgebaut werden können), die den *Conses* aus Lisp entsprechen: Jeder Knoten besitzt Slots für einen *first-child*- und einen *next-sibling*-Zeiger. Produktionen können entweder eine automatische Knotenkonstruktion vornehmen lassen oder in einer speziellen semantischen Aktion die genaue Gestalt des zu instanziierenden Teilbaums spezifizieren. Wird die automatische Konstruktion durchgeführt, so stehen weitere Operatoren in der EBNF zur Verfügung: Die Einfügung eines Teilbaums in die *sibling*-Liste kann verhindert werden (Elimination konstanter Terminale) oder ein Teilbaum kann zu der neuen Wurzel des aktuell konstruierten Ergebnisses der Produktion erklärt werden (Elimination von Kettenregeln), wodurch er nicht in die *sibling*-Liste übernommen wird, sondern stattdessen über diese gehängt.

Soll eine objektorientierte Darstellung gefunden werden, so kann das oben gegebene Standardschema wie folgt modifiziert werden:

Variable Terminale und Nonterminale werden auf Klassen abgebildet.

Sequenzen werden in ihre Elemente zerlegt, die in Attributen der Klasse gespeichert werden. Erzeugt werden nicht Konstruktoren, sondern Klassen.

Alternativen sammeln nicht einfach verschiedene Konstruktoren für denselben Typ auf, sondern die aus den Sequenzen erzeugten Klassen. Diese werden dann über Vererbung zu alternativen Möglichkeiten für die aktuell erzeugte Klasse erklärt. (Ist die Grammatik eindeutig, so werden auch nur zykelfreie Klassenhierarchien erzeugt.)

Es sei bemerkt, daß in diesem Mechanismus keine Vererbung von Eigenschaftsklassen unterstützt wird. Analog dazu, wie die Produktionsschemata in Abschnitt 5.3.3 eine Verallgemeinerung der EBNF-Operatoren waren, wäre weiterhin denkbar, benutzerdefinierte EBNF-Operatoren durch die Instanziierung generischer Klassen zuzulassen.

Zum Beispiel würde nach dieser Methode die Grammatik

Type : NamedType | ArrayType | RecordType.
NamedType : Identifier.
ArrayType : **ARRAY** Integer **OF** Type.
RecordType : **RECORD** Fields **END**.

in eine Klassenhierarchie übersetzt werden, wie sie in Abbildung 4.1 gezeigt wird. Die abgerundeten Rechtecke stellen dabei Klassen dar, deren Name jeweils über dem waagerechten Strich angegeben wird und dessen Attribute mit Typ darunter erscheinen. Pfeile

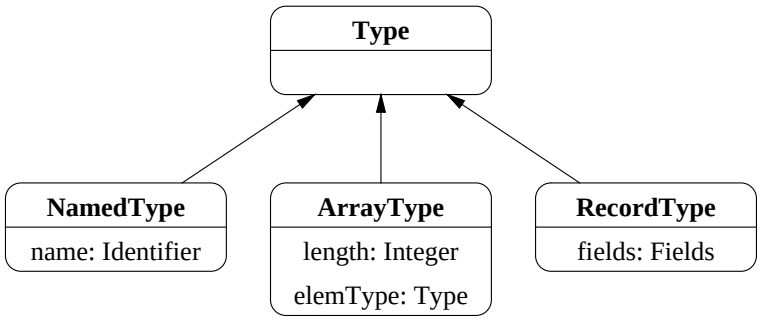

Abbildung 4.1: Beispiel für eine objektorientierte abstrakte Syntax

verbildlichen die „spezialisiert“-Beziehung. In der Abbildung sind für die (wie auch immer generierten) Attributnamen sprechende Bezeichnungen gewählt worden.

Ein entsprechendes Verfahren verwendet beispielsweise *Mango* [Age94]. Dort wird die Situation allerdings dadurch vereinfacht, daß strukturierte Grammatiken (vgl. 3.1.3) verwendet werden: Jede strukturierte Produktion entspricht genau der Anwendung einer der obigen Regeln.

Ein weiteres Werkzeug, das in diese Richtung arbeitet, ist *Ast* [Gro93] aus dem *Cocktail*-Werkzeugkasten. Damit lassen sich abstrakte Syntaxen spezifizieren, für die das Werkzeug dann automatisch Typdefinitionen sowie Baumdurchlaufs- und Ein-/Ausgabefunktionen generiert (den Aufbau des abstrakten Syntaxbaums muß der Benutzer jedoch in den semantischen Aktionen von Hand vornehmen). Es handelt sich dabei zwar nicht um objektorientierte abstrakte Syntaxen, aber um Records, bei denen Vererbungsbeziehungen zwischen Knoten sowie Entsprechungen zu Mixin-Klassen definiert werden können.

4.3.3 Entwurfsentscheidungen

Zusammenfassend läßt sich sagen, daß im Bereich der abstrakten Syntaxen in Tupelform durchaus Möglichkeiten existieren, Unterschiede zwischen der konkreten und der abstrakten Syntax zu spezifizieren – wenn auch nur wenige Werkzeuge dies unterstützen. Das Ziel, eine wirklich abstraktere Repräsentation zu erzeugen, könnte folglich erfüllt werden, wird jedoch oft durch eine weitere (nichttriviale) Notation erkauft. Bedenkt man, wie leicht lesbar und kompakt sich eine Recordkonstruktion in Oz notieren läßt (für die auch keine Typdefinition nötig ist), so wird klar, daß eine Werkzeugunterstützung hier nicht nötig ist: Verwendet man statt eines dedizierten Werkzeugs einfach semantische Aktionen in Oz, so hat man eine sehr mächtige Notation – ohne eine weitere eigene definieren zu müssen.

Für den Aufbau objektorientierter abstrakter Syntaxen gibt es sehr viel weniger Ansätze in der Literatur. Das größte Potential im Vergleich zu den Tupeln besteht hier in der Wiederverwendung von Eigenschaftsklassen, wie dies im OCC-System [Kna96] für die imperativen und objektorientierten Paradigmen im großen Stil exerziert wird. Welche Eigenschaftsklassen jedoch für andere Paradigmen benötigt würden, ist nicht untersucht.

Aus diesen Gründen wurde in der vorliegenden Arbeit von einer Werkzeugunterstützung im Bereich der abstrakten Syntax abgesehen. Dies schränkt die Möglichkeiten der anderen geplanten Teilwerkzeuge nicht ein, da im Gegensatz zu den meisten anderen Werkzeugen die hier definierten Ersetzungsregeln und die Bindungsanalyse nicht von der Existenz einer abstrakten Syntax abhängen.

4.4 Automatisches Unparsing

Unparsing bezeichnet den Vorgang, aus einem abstrakten Syntaxbaum eine Zeichenkette in konkreter Syntax zu erzeugen, die beim erneuten Parsen den betreffenden Syntaxbaum liefert. Dies ist dann relevant, wenn ein Übersetzer implementiert wird, der Programme in einer Sprache in eine Programme in derselben Sprache überführt. Beispiele hierfür sind Optimierungen oder Simplifikationen auf Quelltextebene. Ist die Sprache über Reduktion in eine Kernsprache definiert (wofür die oben vorgestellten Ersetzungsregeln verwendet werden können), so ist die Ausgabe dieser Phase ein Hilfsmittel für den Programmierer, wenn er die Semantik der Sprache besser verstehen will.

Typischerweise besteht die Lösung dieser Aufgabe lediglich aus einem rekursiven Durchlaufen des abstrakten Syntaxbaums, bei dem für jeden Knoten eine Zeichenkette erzeugt wird. Diese wird aus den Informationen des Knotens selbst sowie aus den für die Nachfolgerknoten erzeugten Zeichenketten konstruiert. Ein solches Programm von Hand zu schreiben stellt eine lästige und fehleranfällige Aufgabe dar, da zugesichert werden muß, daß die ausgegebene Zeichenkette eine legale Symbolfolge der konkreten Syntax ist (die zudem noch ein dem Syntaxbaum äquivalentes Programm darstellt). Aus diesem Grund ist eine Werkzeugunterstützung wünschenswert.

In Abschnitt 4.4.1 werden Anforderungen an diesen Prozeß definiert. Daraufhin werden in Abschnitt 4.4.2 existierende Ansätze vorgestellt. Abschnitt 4.4.3 erklärt, warum eine Automatisierung in dem vorliegenden Werkzeug nicht implementiert wurde.

4.4.1 Ziele

Da die von dieser Phase produzierte Ausgabe in dem Kontext, in dem in dieser Arbeit ein Werkzeug entwickelt werden soll, hauptsächlich für den Benutzer des generierten Compilers gedacht ist, werden folgende Forderungen aufgestellt:

- Die Ausgabe sollte lesbar sein. Insbesondere bedeutet dies, daß sie die in der Sprache üblichen Formatierungskonventionen respektiert. Diese sind bei vollkompositionalen Sprachen komplizierter als bei anderen Sprachen: Da hier sehr komplexe geschachtelte Ausdrücke auftreten können, reicht es nicht, pro „Anweisung“ eine Zeile vorzusehen. Es müssen auch innerhalb von Ausdrücken erlaubte Stellen für Zeilenumbrüche sowie zugehörige Einrückungen definiert werden können.

- Ist die Sprache durch Reduktion in eine Kernsprache definiert und ist diese durch Ersetzungsregeln implementiert, so kommt nur ein kleiner Teil der Produktionen der konkreten Syntax für die Ausgabe in Frage. Entsprechend sollte die Erzeugung der Unparsing-Funktion auf die Knotentypen der Kernsprache eingeschränkt werden können.

- Die Unparsing-Spezifikationen sollen von der abstrakten Syntax so unabhängig wie möglich sein, damit Änderungen an dieser sich nur lokal auf die zusätzlichen Unparsing-Deklarationen auswirken.

4.4.2 Ansätze

Da der Baumdurchlauf bei Unparsing die steuernde Rolle spielt, ist es offensichtlich, daß die abstrakte Syntax dem Werkzeug bekannt sein muß. Die bei der Abbildung der konkreten auf die abstrakte Syntax zugrundeliegende Funktion (oben ϕ genannt) muß gewissermaßen umgekehrt werden (was die erlaubten abstrakten Syntaxen auf solche erster Ordnung einschränkt).

Es gibt Werkzeuge, die aus der konkreten Syntax nicht nur automatisch eine abstrakte Syntax erzeugen, sondern auch eine Unparsing-Funktion. Diese sollen im folgenden kurz vorgestellt werden.

Das Werkzeug *Kimwitu* [vEB93] bietet eine spezielle Schreibweise für das Spezifizieren des Unparsing. Dabei werden Regeln durch Pattern-Matching ausgewählt (der Benutzer muß also den rekursiven Abstieg selbst vornehmen), worauf die rechte Seite die zu erzeugende Zeichenkette durch eine Reihe von *unparsing items* angibt. Diese werden einzeln unparst (beispielsweise ein Attribut oder eine konstante Zeichenkette) und konkateniert. Auch in der Sprache C geschriebene Aktionen sind möglich; in diese können wiederum weitere *unparsing items* eingebettet werden.

Ähnlich läuft die Spezifikation des Unparsing in *SSL* ab, der Spezifikationssprache des *Cornell Synthesizer Generator* [RT88]. Hier können jedoch noch Stellen angegeben werden, an denen Zeilenumbrüche zugelassen sind, aber nicht unbedingt erfolgen müssen. Die tatsächliche Formatierung stellt sich dann zur Laufzeit heraus und kann von Größen wie beispielsweise der Fensterbreite abhängen.

Eine andere Idee liegt dem Werkzeug *ParsesraP* [Bea90] zugrunde: Dort wird die Eingabegrammatik gleichzeitig als Ausgabegrammatik verwendet. Für jede Produktion kann neben der semantischen Aktion für das Einlesen auch eine Aktion für die Ausgabe hinzugefügt werden. Dieses Werkzeug braucht die abstrakte Syntax nicht zu kennen – die Unparsing-Regeln werden in den semantischen Ausgabeaktionen durch Nennung ihres Nonterminals aufgerufen. Da dabei nicht zwischen mehreren Produktionen eines Nonterminals unterschieden werden kann, wird defaultmäßig die erste ausgewählt. Ein weiterer Nachteil ist, daß Formatierungskonventionen direkt in die Grammatik eingearbeitet werden müssen – jede Stelle, an der Leerzeichen stehen dürfen, muß in der Grammatik formuliert werden.

Auch in *TXL* [CCH95] wird die Eingabegrammatik gleichzeitig zur Ausgabegrammatik. Da hier die Struktur der abstrakten Syntax jedoch bekannt ist und da verschiedene Produktionen für dasselbe Nonterminal unterschieden werden, reicht es hier aus, die Grammatik um einige Formatierungshinweise zu ergänzen. Hierfür gibt es die Deklarationen `[SP]` für ein Leerzeichen, `[NL]` für einen Zeilenumbruch, `[IN]` für eine Erhöhung der Einrückungstiefe und `[EX]` für dessen Verminderung. Das Werkzeug bietet zudem einen vorgefertigten Satz von Formatierungsregeln, die laut Aussage der Autoren für die meisten Pascal- oder C-ähnlichen Sprachen sinnvolle Ergebnisse liefern. Sie können bei Bedarf jedoch auch abgeschaltet werden.

4.4.3 Entwurfsentscheidungen

Da in dem hier entwickelten Werkzeug keine automatische abstrakte Syntax entwickelt wurde, ist die Voraussetzung für eine Unterstützung des Unparsing nicht gegeben. Allerdings ist es durch die virtuellen Zeichenketten von Oz sehr bequem, die Ausgabe aufzubauen – in den meisten anderen Sprachen ist dies umständlicher.

Um Formatierungsinformationen in den String zu kodieren, ist folgendes flexible, aber einfache System denkbar. Man würde Steuerungen für die Ausgabeformatierung als Tupel der Form `format(`*atom*`)` in den virtuellen String einfügen, die in einem Nachbearbeitungsschritt durch Indentierungen und Zeilenumbrüche ersetzt würden. Dabei wird der String durchlaufen und intern eine Liste mitgeführt, die die letzten Indentierungstiefen speichert. Folgende Steuerungsatome wären wünschenswert:

`break` erzeugt einen Zeilenumbruch und rückt entsprechend der zuoberst auf dem Stack gegebenen Indentierungstiefe ein.

`glue` kennzeichnet eine Stelle, an der ein Zeilenumbruch erlaubt ist, aber nicht zwangsweise erfolgen muß. Abhängig davon, wieviel Platz auf der aktuellen Zeile noch ist, wird diese Steuerungsinformation ignoriert oder als `break` behandelt.

`indent` erhöht die zuoberst auf dem Stack gegebene Einrückungstiefe um eine Konstante n (beispielsweise 3).

`exdent` erniedrigt die zuoberst auf dem Stack gegebene Einrückungstiefe um eine Konstante n.

`push` legt die aktuelle Spalte auf den Stack.

`pop` entfernt das oberste Element des Stacks.

Diese Formatierungsdirektiven sind eine Erweiterung derer, die *TXL* bietet. Durch optionale Zeilenumbrüche wie in *SSL* und den Einrückungsstack werden sie vollkompositionalen Sprachen jedoch mehr gerecht. Eine Funktion, die diese Formatierungen unterstützt, sollte in späteren Versionen des Werkzeugs zum Standardumfang gehören.

4.5 Modularisierung von Spezifikationen

Die Eingabe für ein Werkzeug, das ein Front-End generieren soll, kann gegebenenfalls sehr groß werden. Es ist daher naheliegend, Spezifikationen modular gestalten zu wollen. Abschnitt 4.5.1 untersucht, welche Teile der Spezifikation modularisiert werden sollten und wie. Existierende Ansätze werden dann in Abschnitt 4.5.2 vorgestellt und bewertet.

4.5.1 Ziele

In diesem Abschnitt werden nicht die generellen Vorteile einer Modularisierung von Spezifikationen untersucht; es werden lediglich die Ziele erläutert, die sich aus der Gestalt des in dieser Arbeit entwickelten Werkzeugs ergeben.

Durch den klassenbasierten Ansatz des Werkzeugs ist es leicht, die lexikalische Analyse von den restlichen Phasen zu trennen, wie dies auch in Abschnitt 1.4 als Anforderung formuliert wurde. Da jedoch die Ersetzungsregeln und die Bindungsanalyse verschränkt mit dem Parsen durchgeführt werden, scheint es besser, diese Teilspezifikationen nicht zu trennen. Es ist somit zu erwarten, daß der größte Teil einer Front-End-Spezifikation von den syntaktischen Regeln mit ihren Annotationen beansprucht wird. Von besonderem Interessen sind also Modularisierungen kontextfreier Grammatiken.

Eine Frage, die bei Modularisierung immer auftaucht, ist die der getrennten Übersetzbarkeit. Dies ist je nach Parsealgorithmus prinzipiell denkbar, sofern der Kopplungsgrad der Teilparser sehr gering ist und derselbe Tokenstrom zugrundeliegt. So könnte beispielsweise eine (abgeschlossene) Grammatik für Ausdrücke geschrieben werden und von einer anderen Grammatik verwendet werden, sofern diese nur vollständige Ausdrücke und keine Teilausdrücke benötigt. Parsetechniken, für die eine entsprechende inkrementelle Parsergenerierung möglich wäre, sind solche, bei denen jedes Nonterminal in eine eigene Funktion übersetzt wird – etwa in *PCCTS* [PDC91] für LL oder der Ansatz in [BP95] für LR. Auch bei *Combinator Parsing* stellt dies kein Problem dar. Der Aspekt der Modularisierung ist auf dieser Ebene in der Literatur jedoch kaum wiedergefunden worden. Da somit nur Möglichkeiten existieren, die stark von der Parsetechnik abhängen, wird getrennte Übersetzbarkeit nicht gefordert.

4.5.2 Ansätze

PCCTS [PDC91] ermöglicht es, die Produktionen über mehrere Dateien zu verteilen, die bei der Übersetzung einfach konkateniert werden. Eine wirkliche logische Unabhängigkeit zwischen den Teilspezifikationen liegt nicht vor; es existieren keine kontrollierbaren Schnittstellen und die Kopplung kann beliebig hoch sein.

Ein Beispiel dafür, wie unter Modularisierung die Übersichtlichkeit leiden kann, ist das Werkzeug *Eli* [Com96a], [Com96b]. Dort *müssen* viele Direktiven in getrennte Dateien

geschrieben werden, selbst dann, wenn sie logisch zusammenhängen. Von einer solchen Lösung ist abzusehen.

Ein interessanterer Ansatz ist der aus [CMA94], bei dem Grammatiken inkrementell definiert werden können, indem eine existierende durch Hinzufügungen von Nonterminalen, Hinzufügen von Alternativen zu existierenden Nonterminalen und Ersetzungen von Produktionen modifiziert wird. Die Autoren möchten hiermit zwar vor allem Sprachvarianten, Teilsprachen und erweiterte Sprachen unterstützen, eine Anwendung dieses Systems zwecks Modularisierung einer einzigen Grammatik ist aber auch denkbar.

4.6 Zusammenfassung

In diesem Kapitel sind mehrere Möglichkeiten untersucht worden, das Werkzeug zu erweitern, das in den Kapiteln 2 und 3 entwickelt wurde. Betrachtet wurden dabei die automatische Durchführung von Simplifikationen, Bindungsanalyse, Konstruktion abstrakter Syntaxen und Unparsing. Weiterhin wurden Möglichkeiten der Modularisierung bedacht.

An Erweiterungen wurden letztendlich nur die ersten beiden in das Werkzeug übernommen. Es wurden Ersetzungsregeln und eine Spezifikation für Bindungsanalyse entwickelt, die sich von den Ansätzen aus der Literatur unterschieden, um dem intendierten Einsatzgebiet des Werkzeugs – der Übersetzung multiparadigmatischer und vollkompositionaler Sprachen – gerecht zu werden.

Kapitel 5

Definition des Werkzeugs

In diesem Kapitel wird eine detaillierte Beschreibung des im Laufe der vorhergehenden Kapitel entworfenen Front-End-Generators gegeben.

Abschnitt 5.1 gibt eine Übersicht dafüber, wie die Teilwerkzeuge zu einem ganzen zusammengefügt werden. Durch ein Beispiel einer vollständigen Spezifikation wird ein Eindruck davon vermittelt, wie das Ergebnis aussieht. Danach wird jede Komponente im Detail erläutert: Abschnitte 5.2 bis 5.5 beschreiben respektive den Scannergenerator, den Parsergenerator, die Durchführung der Ersetzungen und die Bindungsanalyse. Die Bedeutung der Sprachkonstrukte wird jeweils anhand ihrer konkreten Syntax erläutert und es werden die vordefinierten Klassen beschrieben, die die Operationalisierung der Spezifikation übernehmen. Dort, wo es zum Verständnis notwendig erschien, wird auch auf verwendete Algorithmen eingegangen. Häufig werden Beispiele aus der Oz-Syntax aufgeführt.

Zuletzt werden in Abschnitt 5.6 einige Hinweise zur Implementierung gegeben und es wird erläutert, wie das Werkzeug unter Verwendung seiner selbst implementiert wurde. Interessant ist dabei, wie das erste lauffähige System erstellt wurde.

5.1 Übersicht über das Gesamtsystem

Dieser Abschnitt beschreibt, wie die Teilwerkzeuge interagieren und wie ihre Eingaben in einer einzigen Spezifikationssprache vereint werden, die zudem in Oz eingebettet ist.

Abschnitt 5.1.1 gibt eine Übersicht darüber, welche Phasen eines Front-Ends unterstützt werden und wie die korrespondierenden Teilwerkzeuge aufeinander aufbauen. Daraufhin gibt Abschnitt 5.1.2 ein Beispiel für eine vollständige Spezifikation, das diese Interaktionen verdeutlicht. In Abschnit 5.1.3 wird auf Ebene der konkreten Syntax gezeigt, wie der Grammatik-Rahmen, der Front-End-Spezifikationen aufnimmt, in die Sprache Oz eingebettet ist.

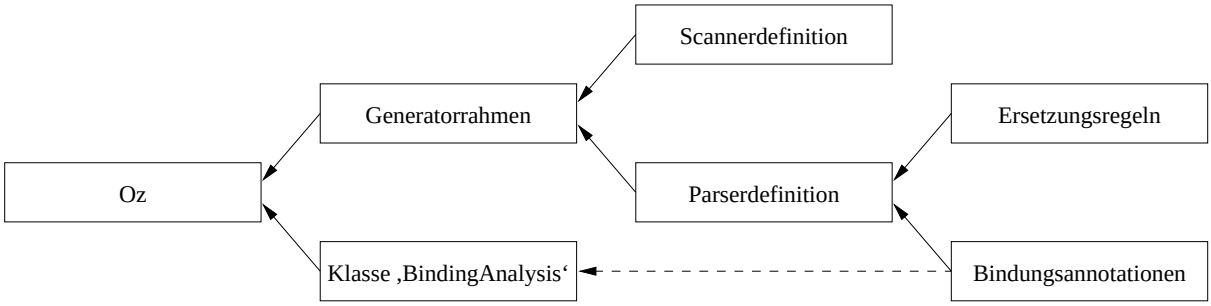

Abbildung 5.1: Abhängigkeiten zwischen den Teilwerkzeugen

5.1.1 Abhängigkeiten zwischen den Teilwerkzeugen

Abbildung 5.1 zeigt, wie die einzelnen Komponenten des Werkzeugs voneinander abhängen. Ein Pfeil verdeutlicht eine *verwendet*-Relation.

Der in Abschnitt 5.1.3 erläuterte *Generatorrahmen* ist in Oz eingebettet und wird in eine Oz-Klassendefinition übersetzt. In ihm lassen sich Scanner und/oder Parser definieren; diese Definitionen sind aber unabhängig voneinander. Die Ersetzungsregeln und Variabenannotationen für die Bindungsanalyse jedoch bauen auf einem Parser auf. Die in Oz geschriebene vordefinierte Klasse ‚BindingAnalysis‘ wird von den Bindungsannotationen referenziert, kann jedoch auch ohne diese verwendet werden.

5.1.2 Vollständiges Beispiel

Das folgende Beispiel gibt eine vollständige Front-End-Spezifikation für eine kleine Testsprache. Zusätzlich zur lexikalischen und syntaktischen Analyse enthält das Beispiel Angaben zu den Sichtbarkeitsregelungen der Sprache, so daß die gebundenen Bezeichner automatisch umbenannt werden.

Die Sprache ist lexikalisch skopiert. Über **`decl`** wird ein Bezeichner gebunden, der in seinem Sichtbarkeitsbereich über **`use`** referenziert werden kann. Bezeichner sind immer in dem ganzen Block zwischen den Schlüsselwörtern **`block`** und **`end`** sichtbar, in dem sie definiert werden. Für Label, die hinter dem Schlüsselwort **`label`** verwendet werden können, existiert nur ein Sichtbarkeitsbereich, der das gesamte Programm umfaßt.

Bezeichnertoken werden intern durch Atome repräsentiert, die den Namen der Bezeichner entsprechen. Daher sind die Operationen, die in ‚IdentifierBindings‘ für die Bindungsanalyse definiert werden, sehr simpel.

```
\switch +synverbose
declare
class IdentifierBindings from BindingAnalysis
   attr count: 0
   meth anonymize(X ?Y)          Y = _   end
```

```
   meth nameToken(X A)            X = A   end
   meth tokenToAtom(X ?A)         A = X   end
   meth makeTokenFromAtom(A ?X)   X = A   end
   meth generate(X ?A)
      count <- @count + 1
      A = {String.toAtom &X|{Int.toString @count}}
   end
end
grammar Example from UrObject LexBaseClass SynBaseClass
   attr default labels
   meth init
      LexBaseClass, init
      SynBaseClass, init
      default <- {New IdentifierBindings init}
      labels <- {New IdentifierBindings init}
   end

   lex <block> <<lexYield(´block´)>> end
   lex <end>   <<lexYield(´end´)>>   end
   lex <decl>  <<lexYield(´decl´)>>  end
   lex <use>   <<lexYield(´use´)>>   end
   lex <label> <<lexYield(´label´)>> end

   lex <[A-Za-z][A-Za-z0-9_-]*>
      <<lexYield(ident {String.toAtom @lexeme})>>
   end

   lex <[ \t\n]+> skip end
   lex <.>
      <<synError("illegal character in input")>>
   end

   token ´block´ ´end´ ´decl´ ´use´ ´label´ ident

   syn program(?B)
      scope(labels) B=Block($) end
   end
   syn Block($)
      ´block´
         scope Ss=( Statement($) )+ end
      ´end´
      => block(Ss)
   end
   syn Statement($)
         Block($)
```

```
      [] ´decl´ ident(I): binder           => decl(I)
      [] ´use´ ident(I): reference         => use(I)
      [] ´label´ ident(I): binder(labels)  => label(I)
   end

   meth start(FileName ?Program ?Status)
      Buffer = {New LexBuffer fromFile(FileName)}
   in
      LexBaseClass, lexSwitchToBuffer(Buffer)
      SynBaseClass, synParse(program(?Program) ?Status)
      {Buffer close}
   end
end
```

In der Grammatik werden zunächst Attribute definiert, die die Informationen für die Bindungsanalyse von Bezeichnern (`default`) und Labels (`labels`) aufnehmen, sowie die Methode `init`, die sie initialisiert. Darauf folgen die Definitionen der lexikalischen Struktur. Für die Parserdefinition werden erst die Tokentypen deklariert, dann die Syntaxregeln angegeben. Die für die Bindungsanalyse notwendigen Informationen werden direkt in den Syntaxregeln angegeben. Die letzte Methode startet die Analyse einer Datei und liefert den abstrakten Syntaxbaum zurück.

5.1.3 Konkrete Syntax von Grammatikdefinitionen

Die Notation, die in diesem Abschnitt für kontextfreie Grammatiken verwendet wird, wird in Anhang A genauer beschrieben.

Der Front-End-Generator wird durch Angabe einer Oz-Datei gestartet, die auch Grammatikdefinitionen enthält. Die Ausgabe des Generators besteht in einer neu angelegten Datei, die denselben Inhalt wie die Eingabedatei hat mit der Ausnahme, daß alle Grammatikdefinitionen durch Klassendefinitionen ersetzt worden sind.

Eine Grammatikdefinition wird also überall zugelassen, wo eine benannte Klassendefinition stehen darf:

⟨*expression*⟩ += ⟨*grammar*⟩

Eine Grammatikdefinition ähnelt einer Klassendefinition, wird jedoch durch das Schlüsselwort **`grammar`** anstelle von **`class`** eingeleitet und muß immer durch eine Variable benannt werden. Deren Name wird während der Übersetzung für die Benennung generierter Dateien verwendet.

⟨*grammar*⟩ ::= **`grammar`** ⟨*variable*⟩ { ⟨*grammar descriptor*⟩ } **`end`**

Als *Grammatikdeskriptoren* werden alle Klassendeskriptoren (**from**, **attr**, **feat** und **meth**) zugelassen. Später werden dieser Regel weitere Alternativen hinzugefügt, in denen die lexikalische Struktur und die kontextfreie Grammatik spezifiziert werden.

⟨*grammar descriptor*⟩ ::= ⟨*class descriptor*⟩

5.2 Der Scannergenerator

Der Scannergenerator ist ein Teilwerkzeug des Front-End-Generators. Wenn eine Grammatik lexikalische Definitionen enthält, wird er automatisch aufgerufen, um diese zu analysieren und entsprechende Deskriptoren für die generierte Klasse zu erzeugen.

Die Beschreibung des Scannergenerators ist in drei Teile gegliedert. Der erste (in Abschnitt 5.2.1) beschreibt anhand der dafür verwendeten konkreten Syntax, wie der dem Scanner zugrundeliegende Automat spezifiziert wird. Die Laufzeitfunktionalität des Scanners und dessen Schnittstellen aus Sicht des Benutzers werden durch die vordefinierten Klassen geliefert, die in Abschnitt 5.2.2 erläutert werden. Der dritte Teil betrifft die Steuerung des Scannergenerators selbst, die über die in Abschnitt 5.2.3 aufgeführten Compilerdirektiven realisiert wird.

5.2.1 Scannerspezifikation

Scanner werden in zusätzlichen Grammatikdeskriptoren spezifiziert. Diese werden um lexikalische Definitionen und Modeklauseln erweitert.

⟨*grammar descriptor*⟩ += ⟨*lexical abbreviation*⟩
| ⟨*lexical rule*⟩
| ⟨*mode clause*⟩

Reguläre Ausdrücke können benannt werden. Sie können dann durch Angabe ihres Namens abkürzend zur Konstruktion neuer regulärer Ausdrücke verwendet werden.

⟨*lexical abbreviation*⟩ ::= **lex** ⟨*label*⟩ **=** ⟨*regex*⟩ **end**

Beispiele:

Die folgenden Abkürzungen sind praktisch, um Oz-Token zu spezifizieren:

```
lex lower = <[a-z]> end
lex upper = <[A-Z]> end
lex digit = <[0-9]> end
lex alphaNum = <{lower}|{upper}|{digit}|_> end
lex atom = <{lower}{alphaNum}*> end
```

Eine lexikalische Regel hat Ähnlichkeit mit einer Methodendefinition. Sie wird jedoch durch das Schlüsselwort **lex** anstelle von **meth** eingeleitet und ihr Kopf besteht aus einem regulären Ausdruck. Der Methodenrumpf wird ausgeführt, wenn beim Scannen der reguläre Ausdruck ausgewählt wird.

⟨*lexical rule*⟩ ::= **lex** ⟨*regex*⟩ [⟨*expression*⟩ **in**] ⟨*expression*⟩ **end**

Beispiele:

Die folgende Regel erkennt (mit obigen Definitionen) ein Oz-Atom und liefert ein Token mit dem Tokentyp `'ATOM'`*. Der Wert des Tokens ist das Lexem als Atom:*

```
lex <{atom}>
   <<lexYield('ATOM' {String.toAtom @lexeme})>>
end
```

Es ist möglich, alle Ein-Zeichen-Tokens durch eine einzige Regel zu matchen. Hierfür gibt es zwei Methoden. Die erste wandelt das gematchte Zeichen einfach in ein Atom, das daraufhin als Tokentyp zurückgegeben wird:

```
lex <[{}()[\]|#:=.^@$!~_,]>
   <<lexYield({String.toAtom @lexeme})>>
end
```

Bei der zweiten Möglichkeit wird ausgenutzt, daß die ganzen Zahlen 1–255 im Parser vordefiniert sind. Wird der Scanner mit einem generierten Parser zusammen verwendet, so hat diese Spezifikation dieselbe Semantik wie die vorherige, ist aber effizienter:

```
lex <[{}()[\]|#:=.^@$!~_,]>
   <<lexYield(@lexeme.1)>>
end
```

Eine *Modeklausel* führt einen neuen lexikalischen Modus ein. Die enthaltenen lexikalischen Regeln gelten in dem Modus sowie in allen von ihm abgeleiteten Modi. Es können Definitionen anderer Modi geerbt werden, indem diese in einer **from**-Klausel aufgeführt werden. Eingeschachtelte Modi erben implizit die lexikalischen Regeln aller sie umgebenden Modi.

⟨*mode clause*⟩ ::= **mode** ⟨*variable*⟩ { ⟨*mode descriptor*⟩ } **end**
⟨*mode descriptor*⟩ ::= ⟨*inherited modes*⟩
| ⟨*mode clause*⟩
| ⟨*lexical rule*⟩
⟨*inherited modes*⟩ ::= **from** { ⟨*variable*⟩ }

Beispiel:

Über lexikalische Modi werden in diesem Beispiel geschachtelte Kommentare erkannt. Hierzu wird ein Attribut verwendet, das die aktuelle Schachtelungstiefe speichert. Die Regeln des lexikalischen Modus' `COMMENT` *versuchen aus Effizienzgründen, immer möglichst viel Text zu matchen:*

```
attr CommentDepth
lex <"/*">
   CommentDepth <- 1
   <<lexSetMode(COMMENT)>>
end
mode COMMENT from INITIAL
   lex <"/*">
      CommentDepth <- @CommentDepth + 1
   end
   lex <"*/">
      CommentDepth <- @CommentDepth - 1
      case @CommentDepth == 0 then
         <<lexSetMode(INITIAL)>>
      else skip
      end
   end
   lex <[^*/]+> skip end
   lex <"*"+[^*/]> skip end
   lex <"/"+[^*/]> skip end
   lex <<EOF>>
      <<reportError("unterminated comment")>>
   end
end
```

Die Syntax der regulären Ausdrücke wird hier nicht genau erläutert. Sie entspricht der, die *flex* [Pax95] verwendet, mit folgenden Ausnahmen:

- Die Modi, in denen ein Ausdruck gilt, werden nicht in spitzen Klammern dem regulären Ausdruck vorgestellt, sondern über die **`mode`**-Klauseln spezifiziert.
- Die Syntax der Namen von lexikalischen Abkürzungen, notiert in geschweiften Klammern, wird auf die Syntax von Oz-Atomen erweitert.

Reguläre Ausdrücke werden – um Verwechselungen mit den Vergleichoperatoren von Oz zu vermeiden – nur erkannt, wenn ihnen das Schlüsselwort **`lex`** vorausging.

5.2.2 Vordefinierte Klassen

Zur Generierung des Automaten wird der Scannergenerator *flex* eingesetzt. Diese Tatsache braucht dem Benutzer aber nicht bekannt zu sein, weil die Grundfunktionalität des Scanners durch einige vordefinierte Klassen gegeben ist, die die Schnittstelle zu dem *flex*-Scanner kapseln.

Die Klasse ‚LexBaseClass'

Die Mixin-Klasse ‚LexBaseClass' implementiert die Methoden zur Interaktion mit dem Scanner und den Tabellenscanner selbst. Soll ein aus einer Grammatikdefinition generierter Scanner zur Ausführung gebracht werden, so muß von dieser Klasse geerbt werden.

‚LexBaseClass' benötigt einige Features und Methoden seitens der erbenden Klasse. Für die Modi von Methodenparametern wird dieselbe Notation verwendet wie bei der Beschreibung der *Oz Standard Modules* [HMSW96]).

lexer [Feature]
Dieses Feature enthält die generierten Tabellen für den Scanner.

lexExecuteAction(+*I*) [Methode]
Diese Methode wird aufgerufen, wenn ein Match ausgewählt wurde. Der Parameter ist die Nummer des regulären Ausdrucks; die Methode muß die entsprechende semantische Aktion ausführen. Die Methode wird vom Generator erstellt; auch die Numerierung der regulären Ausdrücke erfolgt automatisch.

‚LexBaseClass' definiert folgende Funktionalität:

lexeme [Attribut]
Enthält das zuletzt gematchte Lexem als String.

length [Attribut]
Enthält die Länge des zuletzt gematchten Lexems als Integer.

lexYield(+*A* *X* ⇐ Unit) [Methode]
Hängt ein Token der Tokenklasse A mit dem Wert X an den Tokenstrom an.

lexSwitchToBuffer(+*Buffer*) [Methode]
Lenkt den Eingabestrom auf den übergebenen *Buffer* um. Dieser muß eine Instanz der unten erklärten Klasse ‚LexBuffer' sein.

lexCurrentBuffer(?*Buffer*) [Methode]
Liefert den aktuell aktiven Eingabebuffer (eine Instanz der Klasse ‚LexBuffer').

lexSetMode(+*I*) [Methode]
Schaltet auf den Automaten um, der dem lexikalischen Modus I entspricht.

lexCurrentMode(?*I*) [Methode]
Liefert die ganze Zahl, die für den aktiven lexikalischen Modus steht.

lexInput(?*C*) [Methode]
Fordert das nächste Zeichen des Eingabestroms an und entfernt es aus der Eingabe. Hierüber können handgeschriebene Miniscanner in den Scanner eingebettet werden.

lexUnput(+*C*) [Methode]
Fügt das übergebene Zeichen vorne an den Eingabestrom an. Damit kann das Folgeverhalten des Scanners beeinflußt werden.

lexAppendMatch() [Methode]
Bewirkt, daß bei dem nächsten Match nicht dessen Lexem in ‚lexeme' und ‚length' gespeichert, sondern zusätzlich das aktuelle davorgehängt wird. (Diese Methode entspricht der *flex*-Funktion *yymore*.)

lexShortenMatch(+*I*) [Methode]
Kürzt das aktuelle Lexem am Ende um I Zeichen, die wieder in den Eingabestrom zurückkehren.

lexRejectMatch() [Methode]
Bewirkt, daß der ausgewählte Match abgelehnt wird. Daraufhin wird der nächstbeste ausgewählt.

lexGetToken(?*A* ?*X*) [Methode]
Diese Methode ist für den Aufruf von außerhalb vorgesehen. Sie liefert das Token (mit Typ A und Wert X), das am Anfang des Tokenstroms steht.

lexPreAction() [Methode]
Tut nichts; die Methode kann vom Benutzer überladen werden. Sie wird bei jedem Match aufgerufen, bevor die zugehörige semantische Aktion ausgeführt wird. In dieser Aktion können beispielsweise die Quelltextkoordinaten aktualisiert werden oder Debug-Informationen ausgegeben werden.

close() [Methode]
Sollte immer nach Abschluß eines Scanvorgangs aufgerufen werden.

Die Klasse ‚LexLineNo'

Die Mixin-Klasse ‚LexLineNo' bietet eine Möglichkeit, Informationen über die aktuelle Position im Quelltext zu aktualisieren:

lexLine [Attribut]
Enthält die aktuelle Zeilennummer als Integer ≥ 1.

lexColumn [Attribut]
Enthält die aktuelle Spaltennummer als Integer ≥ 0.

lexPreAction() [Methode]
Überlädt die gleichnamige Methode aus ‚LexBaseClass' und aktualisiert die obigen beiden Attribute bei jedem Match. Das bedeutet, daß die Koordinaten immer auf das Zeichen nach dem aktuellen Lexem deuten.

Die Klasse ‚LexBuffer'

Die Klasse ‚LexBuffer' repräsentiert einen Eingabepuffer für einen Scanner. Ihre Instanzen können der Methode ‚lexSwitchToBuffer' der Mixin-Klasse ‚LexBaseClass' übergeben werden. Sie bietet folgende Funktionalität:

init() [Methode]
Muß immer aufgerufen werden (wird von ‚fromFile' und ‚fromVirtualString' automatisch erledigt).

fromFile(+*V*) [Methode]
Initialisiert den Eingabebuffer mit dem Inhalt der Datei mit Namen *V*. Der Buffer gilt defaultmäßig als interaktiv.

fromVirtualString(+*V*) [Methode]
Initialisiert den Eingabebuffer mit dem Wert des virtuellen Strings *V*. Der Buffer gilt nicht als interaktiv.

setInteractive(+*B*) [Methode]
Setzt den Eingabemodus des aktuellen Buffers auf interaktiv oder nicht, je nach Wert der Boole'schen Variable *B*. Interaktiv bedeutet, daß die Zeichen einzeln angefordert werden und nicht *en bloc*, also eine Mindestanzahl von Zeichen vorliegen muß. Bezieht sich die Eingabedatei beispielsweise auf die Tastatur, so sollen Benutzereingaben sofort bearbeitet werden können. Nicht-interaktive Buffer können schneller bearbeitet werden.

getInteractive(?*B*) [Methode]
Liefert ein Flag, ob der Buffer aktuell als interaktiv gilt oder nicht.

setBOL(+*B*) [Methode]
Setzt den Wert eines Flags, das angibt, ob der Eingabezeiger aktuell an einem Zeilenbeginn (**B**eginning **O**f **L**ine) steht oder nicht, sprich: ob der ^-Operator gematcht werden kann oder nicht.

getBOL(?*B*) [Methode]
Liefert den Wert des Flags, das anzeigt, ob der Eingabezeiger an einem Zeilenbeginn steht.

close() [Methode]
Schließt die Datei, aus der gelesen wird. Muß immer aufgerufen werden, wenn ein Buffer nicht mehr benötigt wird.

5.2.3 Compilerdirektiven

Tabelle 5.1 gibt eine Übersicht über die unterstützten Compilerdirektiven, die die Scannergenerierung beeinflussen. Syntaktisch entsprechen sie den Oz-Compilerdirektiven, werden aber vom Front-End-Generator verarbeitet und tauchen in der erzeugten Oz-Datei nicht mehr auf. Die meisten der Direktiven sind Optionen von *flex* Version 2.5.2 nachempfunden, wo eine genaue Beschreibung gegeben wird. Insbesondere die Optionen, die die Tabellengenerierung steuern (lexC...), werden hier nicht näher beschrieben und nur der Vollständigkeit halber aufgeführt.

Switchname	Default	Bedeutung
lexignorecase	off	Keine Unterscheidung zwischen Groß-/Kleinschreibung
lexbestfit	on	Verwendung der Best-Fit- statt der First-Fit-Regel
lexusereject	off	Unterstützung von ‚lexReject' bei der Tabellengenerierung
lexCa	off	Entspricht ‚flex -Ca'
lexCe	on	Entspricht ‚flex -Ce'
lexCf	off	Entspricht ‚flex -Cf'
lexCF	off	Entspricht ‚flex -CF'
lexCm	on	Entspricht ‚flex -Cm'
lexbackup	off	Ausgabe der Zustaende, die Backing-Up benötigen
lexperfreport	off	Ausgabe eines Performance-Reports des Scanners
lexstatistics	off	Ausgabe von Statistiken über den Scanner
lexnowarn	off	Unterdrückung von Warnungen

Tabelle 5.1: Compilerdirektiven zur Steuerung der Scannergenerierung

5.3 Der Parsergenerator

In diesem Abschnitt wird der Parsergenerator vorgestellt. Dessen Konzepte werden anhand seiner Spezifikationssprache erläutert und die vordefinierte Funktionalität wird vorgestellt. Ebenso wie der Scannergenerator wird er automatisch aufgerufen, wenn eine Grammatik Syntaxregeln definiert.

Tokentypen müssen deklariert werden; die notwendigen Spezifikationen werden in Abschnitt 5.3.1 vorgestellt. Daraufhin werden in Abschnitt 5.3.2 Syntax und Semantik von Syntaxregeln definiert. Dieser bereits voll funktionsfähige Parsergenerator wird dann in Abschnitt 5.3.3 durch die Einführung von Produktionsschemata sehr viel ausdrucksstärker

gemacht. Zuletzt werden in Abschnitt 5.3.4 die vordefinierten Klassen und Produktionsschemata vorgestellt.

Die konkrete Syntax ist in diesem Abschnitt zugunsten der Verständlichkeit vereinfacht worden. Die in dem Werkzeug verwendete Syntax läßt zusätzliche notationelle Abkürzungen zu, die aber die Mächtigkeit des Werkzeugs nicht verändern.

5.3.1 Tokendeklarationen

Die Atome der Länge 1 sind als Terminale vordefiniert, die anderen Tokentypen müssen jedoch erst deklariert werden, bevor sie in Syntaxregeln als Grammatiksymbole verwendet werden können. Dies erfolgt über einen weiteren Grammatikdeskriptor.

⟨*grammar descriptor*⟩ += ⟨*token clause*⟩
⟨*token clause*⟩ ::= **token** ⟨*token declaration*⟩ { ⟨*token declaration*⟩ }

Tokentypen werden genau wie im Scanner durch Atome repräsentiert. Zusätzlich können jedem Tokentyp noch höchstens eine Assoziativität und eine Präzedenz zugeordnet werden. Dies wird durch einen auf einen Doppelpunkt folgenden Term getan:

⟨*token declaration*⟩ ::= ⟨*atom*⟩ [**:** ⟨*term*⟩]

Der Term muß ein Tupel mit einem der Labels `leftAssoc`, `rightAssoc` oder `nonAssoc` sein, die die Assoziativität des Tokens angeben. Als einziges Feature des Tupels ist eine Präzedenz als ganze Zahl > 0 zugelassen. Die absoluten Werte der Präzedenzen sind beliebig, nur deren Ordnung zählt: Je größer der Wert ist, desto stärker bindet der Operator. Diese Informationen werden dazu verwendet, Uneindeutigkeiten in der Grammatik aufzulösen, wie dies in Abschnitt 3.2.1 beschrieben wurde.

Beispiel:

Als Beispiel seien hier die für das Parsen von Oz notwendigen Angaben von Präzedenzen und Assoziativitäten aufgeführt:

```
token
   'declare': rightAssoc(1)
   '=': rightAssoc(2) 'FDCOMPARE': rightAssoc(2)
   '<-': rightAssoc(2)
   'orelse': rightAssoc(3)
   'andthen': rightAssoc(4)
   'COMPARE': leftAssoc(5)
   'FDIN': leftAssoc(6)
   '|': rightAssoc(7)
```

```
'#': rightAssoc(8)
'ADD': leftAssoc(9)
'FDMUL': leftAssoc(10) 'OTHERMUL': leftAssoc(10)
',': rightAssoc(11)
'~': leftAssoc(12)
'.': leftAssoc(13) '^': leftAssoc(13)
'@': leftAssoc(14)
```

5.3.2 Syntaxregeln

In diesem Abschnitt wird beschrieben, wie Syntaxregeln definiert werden. Nachdem einige Grundprinzipien anhand der abstrakten Syntax erläutert worden sind, wird auf die Herleitung der Attributarten der Parameter von Nonterminalen eingegangen. Im Anschluß daran werden Bemerkungen zu der nichttrivialen Umsetzung in einen LR-Parser gegeben und dieser mit LL-Parsern verglichen.

Konkrete Syntax

Auch Syntaxregeln werden als Grammatikdeskriptoren angegeben.

⟨*grammar descriptor*⟩ += ⟨*syn clause*⟩

Eine Syntaxregel besteht – ähnlich wie eine Oz-Methode – aus einem Kopf und einem Rumpf. Der Kopf gibt das Nonterminal an, als Atom oder Variable, sowie dessen Parameter. Atome gelten als Startsymbole. Pro Nonterminal darf nur eine Syntaxregel definiert werden. Der Rumpf wird durch eine EBNF-Alternation gegeben.

⟨*syn clause*⟩ ::= **syn** ⟨*syn head*⟩ ⟨*syn alt*⟩ **end**
⟨*syn head*⟩ ::= ⟨*atom label*⟩ ⟨*syn formals*⟩
| ⟨*variable label*⟩ ⟨*syn formals*⟩
⟨*syn formals*⟩ ::= **(** { ⟨*syn formal*⟩ } **)**

EBNF-Ausdrücke können semantische Werte liefern; ansonsten haben sie höchstens Seiteneffekte auf die Variablen ihrer Umgebung. Im folgenden wird jeweils hervorgehoben, welche Konstrukte Werte liefern und welche nicht sowie wo Werte erwartet werden und wo keine stehen dürfen.

Parameter werden durch Variablen bezeichnet. Für höchstens einen Parameter darf anstelle einer Variablen ein *nesting marker* (**$**) eingesetzt werden. In diesem Fall muß der Rumpf der Syntaxregel einen Wert liefern, der bei Applikationen des Nonterminals (siehe unten) mit dem aktuellen Parameter unifiziert wird.

⟨*syn formal*⟩ ::= ⟨*variable*⟩
| **$**

Eine EBNF-Alternation gibt mehrere EBNF-Sequenzen an, getrennt durch den Auswahl-Operator **[]**. Entweder muß jede Sequenz einen Wert liefern oder keine; entsprechendes gilt dann für die Alternation.

⟨*syn alt*⟩ ::= ⟨*syn seq*⟩ { **[]** ⟨*syn seq*⟩ }

Am Anfang einer Sequenz können lokale Variablen vereinbart werden. Diese sind nur innerhalb der Sequenz sichtbar. Die Sequenz selbst besteht aus einer Folge von EBNF-Faktoren, optional gefolgt von einer semantischen Aktion; diese wird in den folgenden Ausführungen wie ein EBNF-Faktor behandelt. Für eine leere Folge kann auch **true** geschrieben werden.

Wird als semantische Aktion ein Oz-Term angegeben, so liefert diese dessen Wert, ansonsten keinen. Kein EBNF-Faktor einer Sequenz mit Ausnahme des letzten darf einen Wert liefern; die Sequenz liefert den Wert ihres letzten Faktors oder keinen.

⟨*syn seq*⟩ ::= [{ ⟨*variable*⟩ } **in**] { ⟨*syn factor*⟩ } [⟨*syn action*⟩]
| **true** [⟨*syn action*⟩]
⟨*syn action*⟩ ::= **=>** [⟨*expression*⟩ **in**] ⟨*expression*⟩
| **=>** [⟨*expression*⟩ **in**] [⟨*expression*⟩] ⟨*term*⟩

Ein EBNF-Faktor ist entweder eine Applikation oder eine Zuweisung. Eine Applikation wird durch ein Terminal oder Nonterminal notiert, gefolgt von eventuellen aktuellen Parametern in Klammern. Terminale dürfen entweder keinen Parameter oder genau einen Variablenparameter haben; eine Variable wird mit dem Tokenwert unifiziert. Bei einem Nonterminal muß die Anzahl der aktuellen Parameter mit der Anzahl der formalen Parmeter in dessen Definition übereinstimmen. Variablen, die ohne Fluchtsymbol (**!**) als Parameter angegeben werden, werden in der umgebenden Sequenz implizit als lokale Variable deklariert (Variablen als aktuelle Parameter stehen also – in Oz-Terminologie – an *Patternposition*).

Höchstens ein aktueller Parameter darf ein *nesting marker* sein. Ist dies der Fall, so liefert die Applikation den Wert des Parameters, ansonsten keinen.

⟨*syn factor*⟩ ::= ⟨*syn application*⟩
| ⟨*syn assignment*⟩
⟨*syn application*⟩ ::= ⟨*atom label*⟩ ⟨*syn actuals*⟩
⟨*syn actuals*⟩ ::= **(** { ⟨*term*⟩ } **)**

An dieser Stelle sollen zwei vordefinierte Grammatiksymbole erwähnt werden, die eine Spezialbehandlung erfahren:

`prec` Durch Einfügen einer Applikation des vordefinierten Grammatiksymbols `prec` werden einer Produktion eine Präzedenz und eine Assoziativität zugeordnet. Der einzige aktuelle Parameter muß ein Terminalsymbol sein, dessen Präzedenz und Assoziativität übernommen werden. Bei Produktionen ohne Applikation von `prec` werden die Daten des letzten verwendeten Terminals in der Produktion genommen. Diese Informationen werden dazu verwendet, Uneindeutigkeiten in der Grammatik aufzulösen, wie dies in Abschnitt 3.2.1 beschrieben wurde.

`error` Durch eine Applikation des vordefinierten Terminals `error` wird ein *restart point* für die Error-Recovery definiert. Deren Verarbeitung erfolgt wie bei *Bison* [DS95]. Das Terminal besitzt keinen Tokenwert.

Eine Zuweisung setzt eine Variable mit dem Wert eines EBNF-Faktors gleich. Dieser muß einen Wert liefern. Die Variable muß innerhalb der Syntaxregel deklariert worden sein. Wird sie ohne Fluchtsymbol (`!`) notiert, so wird sie implizit deklariert. Eine Zuweisung liefert keinen Wert.

⟨*syn assignment*⟩ ::= ⟨*escaped variable*⟩ `=` ⟨*syn factor*⟩

Beispiele:

An dieser Stelle sollte noch einmal das Beispiel aus 5.1.2 gelesen werden. Dort werden viele dieser Konstrukte verwendet.

Die folgenden Syntaxregeln setzen die Syntaxregeln für Typdefinitionen um, die in Abschnitt 4.3.2 angegeben wurden. Die Parameter liefern den objektorientierten Syntaxbaum:

```
syn Type($)
      NamedType($)
   [] ArrayType($)
   [] RecordType($)
end
syn NamedType($)
   Identifier(I) => {New NamedType init(I)}
end
syn ArrayType($)
   'ARRAY' Integer(I) 'OF' Type(T) => {New ArrayType init(I T)}
end
syn RecordType($)
   'RECORD' Fields(Fs) 'END' => {New RecordType init(Fs)}
end
```

Herleitung der Attributarten

Bei Parametern von Grammatiksymbolen werden zwei Arten von Attributen unterschieden, nämlich *synthetisierte* und *vererbte* Attribute. Bei einer Applikation kann man vererbte Attribute wie Eingabeparameter, synthetisierte wie Ausgabeparameter betrachten. Im Gegensatz zu Oz, wo dank der Unifikation Ein- und Ausgabeparameter nicht unterschieden zu werden brauchen, müssen die beiden Attributarten aufgrund von Einschränkungen, die durch den Parsealgorithmus bedingt sind, sehr verschieden behandelt werden. Um dem Benutzer diese Aufgabe abzunehmen, wurde ein System entwickelt, das die Attributarten aus der Verwendung der Parametervariablen ableitet. Dieses wird im vorliegenden Abschnitt vorgestellt.

Vorab muß jedoch noch ein Begriff eingeführt werden.

Definition: Sei eine Sequenz mit EBNF-Faktoren $0, \ldots, n$ gegeben. Sei i der Index des ersten Faktors (Applikation, Zuweisung oder semantische Aktion), in dem eine lokale Variable V der Sequenz verwendet wird. Dann gilt V in allen Faktoren mit Index j, $j \geq i$, als **initialisiert**, in allen anderen als **uninitialisiert**. □

Nun können die Regeln vorgestellt werden, die der Herleitung der Attributarten zugrunde liegen:

- Bei dem optionalen Parameter eines Terminals handelt es sich immer um ein synthetisiertes Attribut.
- Wird in einer Applikation eines Grammatiksymbols B eine bisher unitialisierte lokale Variable oder ein *nesting marker* als i-ter aktueller Parameter angegeben, so ist der i-te formale Parameter von B ein synthetisiertes Attribut. Die uninitialisierte Variable darf in den anderen aktuellen Parametern der Applikation nicht vorkommen.
- Wird in einer Applikation eines Grammatiksymbols B eine bereits initialisierte lokale Variable oder ein komplexer Term als i-ter aktueller Parameter angegeben, so ist der i-te formale Parameter von B ein vererbtes Attribut. In dem aktuellen Parameter darf keine uninitialisierte Variable vorkommen.
- Wird in der Syntaxregel eines Nonterminals A eine von deren formalen Parametervariablen in einer Applikation eines Grammatiksymbols B als Parameter angegeben, so sind die korrespondierenden formalen Parameter von A und B Attribute der gleichen Art, also entweder beide synthetisiert oder beide vererbt.

Es sei bemerkt, daß aus der Verwendung einer formalen Parametervariablen in einer semantischen Aktion nichts über dessen Attributart gefolgert werden kann: In Oz wird zwischen Zugriff auf und Zuweisung an Variablen nicht unterschieden, da beides über Unifikation geschieht.

Werden widersprüchliche Attributarten für denselben formalen Parameter eines Nonterminals gefolgert, so liegt ein Fehler vor; kann keine Art ermittelt werden, so wird ein synthetisiertes Attribut angenommen.

Beispiel:

Dieses Beispiel implementiert die Deklarationen aus Abschnitt 3.3, bei denen ein vererbtes Attribut nötig ist, um den Typ durchzureichen. Da T in der Produktion von ‚Declaration' zweimal in derselben Sequenz verwendet wird, ist das Argument von ‚DeclaredVariables' ein vererbtes Attribut. Entsprechend folgt dies auch für ‚DeclaredVariable':

```
syn Declaration
   Type(?T) DeclaredVariables(T) ';'
end
syn DeclaredVariables(T)
      DeclaredVariable(T)
   [] DeclaredVariables ',' DeclaredVariable(T)
end
syn DeclaredVariable(T)
   Variable(V) => <<enterDeclaration(V T)>>
end
```

Realisierung von Attributen und lokalen Variablen in LR-Parsern

Sowohl die Definition lokaler Variablen in EBNF-Syntaxregeln als auch vererbte Attribute sind in LR-Parsern schwierig zu implementieren. Dies ist auch der Grund, weshalb viele Compilerbauer LL-Parser vorziehen. In diesem Abschnitt wird erläutert, wie sich diese Features in dem entwickelten System mit wenigen Einschränkungen umsetzen lassen.

Da bei LL-Parsern eine eins-zu-eins-Beziehung zwischen Nonterminalen und Funktionen im generierten Programm hergestellt werden kann, können lokale Variablen bei den Eintritt in eine solche Funktion deklariert werden. Bei LR-Parsern ist dies nicht der Fall, da nicht eindeutig feststeht, welchem Nonterminal die geparste Eingabe entspricht. Die einzige Möglichkeit ist daher, lokale Variablen auf dem Parsestack zu speichern.

Dank der Unterscheidung zwischen initialisierten und uninitialisierten lokalen Variablen ist es möglich, für jede lokale Variable eine feste Stelle in jeder Produktion zu finden, an der sie alloziert werden kann. Der Zeitpunkt ihrer Initialisierung entspricht damit ihrer Allokation auf dem Parsestack. Relativ zu den darauffolgenden EBNF-Faktoren ist diese Position immer konstant und zur Übersetzzeit bekannt; also kann effizient auf die Variablen zugegriffen werden. Daß diese Variable nicht lokal in einer Funktion deklariert wurde, sondern im Parsestack steht, ist für den Benutzer transparent, weil es sich um logische Variablen handelt.

Auch bei der Vererbung von Attributen tritt das Problem auf, daß Nonterminalen nicht eindeutig Funktionen zugeordnet werden können, denen man als Argumente die Werte der vererbten Attribute übergeben könnte. Der Zugriff auf vererbte Attribute wird daher ähnlich realisiert wie der auf lokale Variablen, indem ihre Werte direkt aus dem Parsestack geholt werden. Dabei tauchen allerdings zwei Probleme auf:

- Die relative Position im Parsestack von als vererbtes Attribut übergebenen lokalen Variablen muß nicht eindeutig sein. Dies zeigt das folgende Beispiel:

```
syn N()
      a(V) M(V)
   [] a(V) b() M(V)
end
syn M(W) ... end
```

  Dabei sei `a` ein Terminalsymbol; dessen Applikation synthetisiert den Wert der Variablen `V` (genaugenommen steht `V` ab diesem Zeitpunkt für den Tokenwert, der im Parsestack gespeichert ist). Damit wird für den Parameter `W` von `M` ein vererbtes Attribut hergeleitet (in beiden Sequenzen der Syntaxregel von `N`). Dessen relative Position im Parsestack ist in den beiden Sequenzen (von `M` aus gesehen) jedoch eine andere.

- Steht an der Stelle eines vererbten Attributs als aktueller Parameter ein komplexer Term, so hat dessen Wert noch überhaupt keine Position im Parsestack, da der Term noch nicht ausgewertet worden ist; zum Beispiel:

```
syn R()
   a(V) S({F V})
end
syn S(W) ... end
```

Die Lösung besteht darin, alle Nonterminale zu ermitteln, die mit einem dieser Fälle angewendet werden. Direkt vor den Applikationen jedes dieser Nonterminale wird ein zusätzlicher Slot auf dem Parsestack mit den Werten ihrer vererbten Attribute angelegt. Damit werden sowohl komplexe Terme ausgewertet als auch konstante relative Positionen für alle vererbten Variablen zugesichert. In den obigen Beispielen sähe dies folgendermaßen aus:

```
syn N()
      a(V) Gen1(V V1) M(V1)
   [] a(V) b() Gen2(V V1) M(V1)
end
syn Gen1(X Y)
   => Y = X
```

```
end
syn Gen2(X Y)
   => Y = X
end
syn R()
   a(V) Gen3(V V1) S(V1)
end
syn Gen3(X Y)
   => Y = {F X}
end
```

Die Nonterminale „`Gen...`“ seien dabei von allen anderen Grammatiksymbolen verschieden; sie heißen *Marker-Nonterminale*. Ihr Parameter `X` ist jeweils ein vererbtes und `Y` ein synthetisiertes Attribut. Da garantiert ist, daß diese Nonterminale nur genau einmal in der Grammatik appliziert werden, ist die relative Position aller vererbten Attribute im Parsestack nach dieser Transformation konstant. Die Einschränkung ist jedoch, daß hierbei eventuell Parsekonflikte in die Grammatik eingeführt werden können. In diesem Fall muß der Benutzer seine Grammatik umformulieren.

Bei der Expansion der unten vorgestellten Produktionsschemata kann der Fall hinzutreten, daß in Sequenzen Alternativen eingeschachtelt werden; außerdem können dadurch semantische Aktionen an beliebigen Stellen in einer Sequenz stehen (sogenannte *mid-rule* im Gegensatz zu *end-rule actions*). LR-Parsetabellen können nur aus strikter BNF erzeugt werden, daher müssen eingeschachtelte Alternativen und *mid-rule actions* in eigene Regeln ausgelagert werden. Die Schwierigkeit ist, daß diese auf lokale Variablen der umgebenden Sequenzen zugreifen können. Die Definitionen der *initialisierten lokalen Variablen* und des *EBNF-Faktors* greifen hier nicht mehr. Die Lösung ist einfach: Bei der Auslagerung werden die referenzierten lokalen Variablen der Parameterliste hinzugefügt. Die Herleitung der Attributarten und der Mechanismus der vererbten Attribute erledigen dann den Rest.

Vergleich mit LL-Parsern

Der durch die Transformation eingeschränkte Parsealgorithmus hat zwar nicht mehr die volle Mächtigkeit von LR, wenn diese Features verwendet werden, ist aber immer noch mindestens so mächtig wie LL. Das rührt daher, daß LR genauso mächtig wird wie LL, wenn am Anfang *jeder* BNF-Produktion ein Marker-Nonterminal eingefügt wird. Der einzige Unterschied in der Semantik besteht darin, daß im Zweifelsfall die Werte vererbter Attribute immer ausgewertet werden, wenn ein Nonterminal folgen könnte, das sie benötigt. Daher sollten bei der Berechnung der Werte für vererbte Attribute weder Seiteneffekte produziert noch nichtmonotone Funktionen verwendet werden, wenn die Spezifikation unabhängig davon, ob ein LL- oder ein LR-Parser verwendet wird, dieselbe Semantik haben soll.

5.3.3 Produktionsschemata

In diesem Abschnitt werden die oben definierten Syntaxregeln um *Produktionsschemata* erweitert. Diese bieten dem Benutzer die Möglichkeit, eigene EBNF-Operatoren zu definieren.

In einem ersten Schritt wird hier erläutert, wie Produktionsschemata spezifiziert werden. Daraufhin wird ihre Expansion besprochen und es werden einige Beispiele gegeben.

Definition

Produktionsschemata können an zwei Stellen definiert werden: auf der globalen Ebene einer Datei und innerhalb einer Grammatik. Die globalen Definitionen werden in die Liste der vordefinierten Produktionsschemata aufgenommen, die in jeder folgenden Grammatik gültig sind. Lokale Produktionsschemata gelten nur innerhalb der Grammatik, in der sie definiert werden.

⟨*top level expression*⟩ += ⟨*prod clause*⟩
⟨*grammar descriptor*⟩ += ⟨*prod clause*⟩

Wie die meisten anderen Grammatikdeskriptoren besteht auch ein Produktionsschema aus einem Kopf und einem Rumpf. Der Rumpf gibt den EBNF-Ausdruck an, der bei Anwendung eines Produktionsschemas eingesetzt werden soll. Dieser kann optionale lokale Syntaxregeln referenzieren.

⟨*prod clause*⟩ ::= **prod** ⟨*prod head*⟩ [⟨*local rules*⟩ **in**] ⟨*syn alt*⟩ **end**
⟨*local rules*⟩ ::= ⟨*syn clause*⟩ { ⟨*syn clause*⟩ }

Der Kopf eines Produktionsschemas liefert neben den Namen seiner Argumente noch eine eindeutige Identifikation des Schemas. Diese setzt sich aus folgenden Bestandteilen zusammen:

1. der Tatsache, ob das Schema, wenn es an der Stelle eines EBNF-Faktors verwendet wird, einen Wert liefert (notiert durch „`V=`...“, wobei `V` die Variable bezeichnet, deren Wert geliefert wird),
2. dem Namen, den das Schema gegebenenfalls trägt, notiert vor einem Doppelpunkt,
3. dem verwendeten Klammernpaar (runde, eckige oder geschweifte Klammern),
4. der Anzahl der Argumente, die durch `//` getrennt werden, und
5. dem verwendeten Postfixoperator, sofern einer angegeben wurde.

Beispielsweise ist `[ X ]` eine verbreitete EBNF-Notation für eine Option und `{ X // Y }+` könnte für eine separierte Liste mit mindestens einem Element stehen. Dieses Konstrukt könnte einen Wert ergeben, etwa eine Oz-Liste der von den `X` gelieferten Werte, was durch `Z={ X // Y }+` notiert wird.

⟨*prod head*⟩ ::= ⟨*template definition*⟩
| ⟨*variable*⟩ **=** ⟨*template definition*⟩
⟨*template definition*⟩ ::= ⟨*prod formal list*⟩
| ⟨*atom*⟩ **:** ⟨*prod formal list*⟩
⟨*prod formal list*⟩ ::= **(** ⟨*prod formals*⟩ **)** [⟨*prod postfix*⟩]
| **[** ⟨*prod formals*⟩ **]** [⟨*prod postfix*⟩]
| **{** ⟨*prod formals*⟩ **}** [⟨*prod postfix*⟩]
⟨*prod formals*⟩ ::= [⟨*variable*⟩ { **//** ⟨*variable*⟩ }]
⟨*prod postfix*⟩ ::= **+**
| *****

Expansion

Die Instanziierung eines Produktionsschematas ist überall möglich, wo ein EBNF-Faktor stehen darf:

⟨*syn factor*⟩ += ⟨*template instantiation*⟩

Die Anwendung eines Produktionsschematas gleicht ihrer Definition, nur, daß anstelle der formalen Parametervariablen aktuelle EBNF-Ausdrücke zugelassen werden.

⟨*template instantiation*⟩ ::= ⟨*prod actual list*⟩
| ⟨*atom*⟩ **:** ⟨*prod actual list*⟩
⟨*prod actual list*⟩ ::= **(** ⟨*prod actuals*⟩ **)** [⟨*prod postfix*⟩]
| **[** ⟨*prod actuals*⟩ **]** [⟨*prod postfix*⟩]
| **{** ⟨*prod actuals*⟩ **}** [⟨*prod postfix*⟩]
⟨*prod actuals*⟩ ::= [⟨*syn alt*⟩ { **//** ⟨*syn alt*⟩ }]

Bei der Expansion von Produktionsschemata muß *Kapern* von Variablen vermieden werden, also das Entstehen eines Konflikts zwischen gleichlautenden lokalen Variablen des Produktionsschemas und freien Variablen der Argumente. Die Expansion wird in folgenden Teilschritten durchgeführt:

- Die lokalen Variablen des Produktionsschemas werden eindeutig umbenannt, sowohl in dem EBNF-Ausdruck als auch in den lokalen Regeln. Zwischen freien Variablen des Schemas und anderen Variablen kann kein *variable capturing* auftreten, wenn vorausgesetzt wird, daß im gesamten abstrakten Syntaxbaum alle Variablen eindeutig umbenannt worden sind.

- Die lokalen Regeln werden eindeutig umbenannt, um Verwechslungen mit existierenden Grammatiksymbolen zu vermeiden.
- Die aktuellen EBNF-Ausdrücke werden für die Parametervariablen des Produktionsschemas eingesetzt. Die formalen Parametervariablen dürfen als Grammatiksymbol in Applikationen verwendet werden und entweder kein oder ein Argument haben. Haben sie ein Argument, so muß der aktuelle EBNF-Ausdruck einen Wert liefern, der mit dem Argument unifiziert wird.
- Die lokalen Regeln werden über die in den aktuellen Parametern verwendeten lokalen Variablen der aufrufenden Syntaxregel quantifiziert, indem ihnen Parameter hinzugefügt werden. Deren Attributarten werden mit dem oben definierten Verfahren automatisch ermittelt.
- Die lokalen Regeln werden in die Tabelle der Grammatiksymbole eingetragen.
- Der Schemaaufruf wird durch den EBNF-Ausdruck des Produktionsschematas ersetzt.

Um die Termination der Expansion von Produktionsschemata zuzusichern, dürfen die in der Definition eines Schemas verwendeten EBNF-Ausdrücke nur auf vorhergehende Produktionsschemata zurückgreifen. Nachdem alle Produktionsschemata expandiert worden sind, werden einige Simplifikationen der Grammatik vorgenommen; beispielsweise werden Alternativen und Sequenzen abgeflacht und aufeinanderfolgende semantische Aktionen zusammengefaßt. Danach werden alle Konstrukte in eigene Regeln ausgelagert, sofern dies nötig ist, um eine strikte BNF zu erhalten.

Durch die lokalen Regeln können Zugriffe der aktuellen Ausdrücke auf lokale Variablen des Aufruforts zu nichtlokalen Zugriffen werden. Man sieht, wie leicht dies durch den Mechanismus der automatischen Attributartherleitung gehandhabt wird.

Beispiele:

Das einfachste Produktionsschema ist das Klammerungskonstrukt:

```
prod ( X )
   X()
end
```

Beispielsweise würde die Sequenz

```
T in ( 'INT' => T=intTp [] 'REAL' => T=realTp )
```

zu `T in Gen(T)` *expandiert, wobei* `Gen` *ein neues Nonterminal mit folgender Regel ist:*

```
syn Gen(T)
      'INT' => T=intTp
   [] 'REAL' => T=realTp
end
```

Das Klammerungskonstrukt kann auch einen Wert liefern:

```
prod V=( X )
   X(V)
end
```

Das folgende Konstrukt implementiert den EBNF-Operator „Option“:

```
prod [ X ]
   X [] true
end
```

Das folgende Beispiel soll verdeutlichen, wie Produktionsschemata dazu verwendet werden können, für ein Konstrukt effiziente Formulierungen anzugeben, die je nach Parsealgorithmus anders aussehen. Durch Produktionsschemata ist dieses Implementierungsdetail dem Benutzer jedoch verborgen.

In dem Beispiel wird die optionale Wiederholung umgesetzt, die als Wert eine Liste der von dem wiederholten Element gelieferten Werte zurückgibt. Wiederholungen müssen in LL-Grammatiken in BNF durch rechtsrekursive Regeln formuliert werden:

```
prod V={ X }
   syn N($)
         true => nil
      [] X(A) N(As) => A|As
   end
in
   N(V)
end
```

Jedes Element wird, nachdem es erkannt worden ist, vor die Ergebnisliste gehängt. Diese Implementierung könnte auch für einen LR-Parser verwendet werden; dort werden aber linksrekursive Regeln vorgezogen, da bei diesen der Speicherverbrauch des Parsestacks konstant ist und nicht linear in der Anzahl der Elemente der Wiederholung. Um die Listenkonstruktion trotzdem effizient zu halten, werden Differenzlisten verwendet.

```
prod V={ X }
   syn N(Hd Tl)
         true => Hd=Tl
      [] N(Hd Tl0) X(Elem) => Tl0=Elem|Tl
   end
in
   N(Hd Tl) => Tl=nil V=Hd
end
```

5.3.4 Vordefinierte Klassen und Produktionsschemata

Der Front-End-Generator generiert aus einer Grammatik eine Klasse einige Features und Methoden für diese, in denen die Parsetabellen und das Verhalten bei Reduktionen (also die semantischen Aktionen) gespeichert werden. Die vordefinierte Klasse ‚SynBaseClass' operiert auf diesen Daten.

Die Klasse ‚SynBaseClass'

Die Mixin-Klasse ‚SynBaseClass' implementiert den Tabellenparser und stellt Methoden zur Verfügung, über die dessen Verhalten gesteuert werden kann. Wenn von ihr geerbt wird, können die Syntaxregeln einer Grammatik zur Analyse einer Eingabe verwendet werden.

Die Klasse ‚SynBaseClass' benötigt folgende Methoden von abgeleiteten Klassen:

lexYield(+*A X* ⇐ Unit) [Methode]
Diese Methode soll ein Token der Tokenklasse A mit dem Wert X an den Tokenstrom anhängen. Sie wird dazu verwendet, ein internes Terminalsymbol zusammen mit den vererbten Attributen des Startsymbols zu erzeugen.

lexGetToken(?*A* ?*X*) [Methode]
Der Parser fordert über diese Methode ein neues Terminalsymbol an. Sie muß mindestens einen Tokentyp A zurückgeben und kann diesem auch einen Tokenwert in X zuordnen.

synExecuteAction(*I Xs* ?*Ys* ?*Z*) [Methode]
Diese Methode wird vom Parser aufgerufen, wenn eine Reduktion durchgeführt wird. Sie erhält die Nummer I der BNF-Regel und den aktuellen Parsestack Xs. Sie muß den Parsestack Ys nach der Reduktion und den erzeugten semantischen Wert Z zurückgeben. Da hier implizit die Darstellung der Attribute und lokalen Variablen auf dem Parsestack eingeht, braucht der Benutzer sich um diese Methode nicht zu kümmern: Sie wird automatisch vom Parsergenerator erzeugt.

Hinzu kommen noch Features für die Parsetabellen, die vom Parsergenerator automatisch erzeugt werden. Diese beginnen alle mit ‚syn...'.

‚SynBaseClass' besitzt folgende vordefinierte Funktionalität:

synLookaheadSymbol [Attribut]
In diesem Attribut wird der Tokentyp des momentan betrachteten Lookahead-Tokens gespeichert.

synLookaheadValue [Attribut]
In diesem Attribut wird der Tokenwert des momentan betrachteten Lookahead-Tokens gespeichert.

synNoLookahead [Feature]
Dieser Wert wird in ‚synLookaheadSymbol' gespeichert, wenn momentan kein Lookahead-Token vorliegt. Dies ist der Fall, wenn das Token verbraucht wurde, aber noch kein neues angefordert wurde. In Fehlersituationen kann das Lookahead-Token über diesen Wert gelöscht werden.

synParse(*T ?B*) [Methode]
Diese Methode startet den Parsevorgang. Das Label des Tupels T gibt das Startsymbol an, seine Features die Parameter des entsprechenden Nonterminals. Die Werte der vererbten Attribute des Startsymbols werden aus diesem Tupel genommen und die synthetisierten Attribute nach Abschluß des Parsevorgangs mit den übrigen Teilbäumen des Tupels unifiziert.

synAccept() [Methode]
Mit dieser Methode wird der Parsevorgang abgebrochen und ein positiver Status gemeldet. Sie kann dann verwendet werden, wenn die restliche Eingabe ignoriert werden soll.

synAbort() [Methode]
Mit dieser Methode wird der Parsevorgang abgebrochen und ein negativer Status gemeldet. Sie kann dann verwendet werden, wenn eine Fehlersituation erkannt wurde, in der keine *error recovery* durchgeführt werden soll.

synRaiseError() [Methode]
Diese Methode versetzt den Parser in denselben Zustand, als sei gerade ein Syntaxfehler erkannt worden. Der Parser fährt im *error-recovery*-Modus fort.

synErrorOK() [Methode]
Hierüber wird dem *error-recovery*-Modus des Parsers mitgeteilt, daß die Fehlerbehandlung abgeschlossen ist und mit dem normalen Parsen fortgefahren werden kann.

synClearLookahead() [Methode]
Diese Methode löscht das aktuelle Lookahead-Token. Dies ist praktisch, wenn das

Lookahead-Token bei einem durch das Nonterminal `error` angegeben *restart-point* ungültig ist.

synError(+V) [Methode]
Der Parser ruft diese Methode auf, wenn eine Fehlermeldung V ausgegeben werden soll. Sie kann überladen werden, um diese Ausgabe anders zu formatieren oder weiterzuverarbeiten.

Vordefinierte Produktionsschemata

Tabelle 5.2 zeigt die vordefinierten Produktionsschemata. Es sind alle EBNF-Konstrukte vordefinert, die in Abschnitt 3.1.3 beschrieben wurden. Für jeden Operator existieren mehrere gleichbedeutende Schreibweisen. Die Operatoren existieren alle auch als Formen, die einen Wert liefern: Die Klammerung liefert den Wert ihres Arguments; die Option ebenfalls beziehungsweise `nil`, wenn sie nicht gewählt wird; die Wiederholungskonstrukte liefern Oz-Listen ihres ersten Arguments.

Konstrukt	Schema	Synonyme
Gruppierung	`( A )`	—
Option	`[ A ]`	—
Obligatorische Wiederholung	`( A )+`	`{ A }+`
Optionale Wiederholung	`( A )*`	`{ A }`, `{ A }*`
Obligatorische separierte Wiederholung	`( A // B )+`	`( A // B )`, `{ A // B }+`, `{ A // B }`
Optionale separierte Wiederholung	`( A // B )*`	`{ A // B }*`

Tabelle 5.2: Vordefinierte Produktionsschemata

5.3.5 Compilerdirektiven

Tabelle 5.3 gibt eine Übersicht über die Compilerdirektiven, die die Ausgabe zusätzlicher Informationen über die Parsergenerierung bewirken. Die Ausgaben erfolgen in Dateien, die durch den Namen der Grammatik mit angehängtem `.simplified` beziehungsweise `.output` gebildet werden.

Switchname	Default	Bedeutung
synoutputsimplified	off	Bewirkt eine Ausgabe der BNF-Form (`.simplified`)
synverbose	off	Bewirkt eine Ausgabe des Automaten (`.output`)

Tabelle 5.3: Compilerdirektiven zur Steuerung der Parsergenerierung

5.4 Ersetzungsregeln

In diesem Abschnitt wird die Erweiterung realisiert, die in Abschnitt 4.1 untersucht wurde. In Abschnitt 5.4.1 wird gezeigt, wie sich Ersetzungsregeln in die existierenden Syntaxregeln einordnen. Abschnitt 5.4.2 erläutert ihre Umsetzung. Mit einigen Beispielen wird die Beschreibung der Ersetzungsregeln in Abschnitt 5.4.3 abgeschlossen.

5.4.1 Konkrete Syntax

Eine Ersetzungsregel ist eine spezielle semantische Aktion.

⟨*syn action*⟩ += **=>** ⟨*rewrite action*⟩

Ihre linke Seite ist die EBNF-Sequenz, an deren Ende sie steht. Die rechte Seite ist durch eine Applikation und eine Folge von Termen gegeben. Die Applikation besagt, mit welchem Nonterminal die Termfolge geparst werden soll. Jeder Term ist entweder ein Tupel – und entspricht dann einer Applikation – oder eine Variable. Diese muß eine lokale Variable der Produktion sein, die als einziger Parameter einer Applikation implizit deklariert wurde; sie steht dann für diese Applikation.

⟨*rewrite action*⟩ ::= **rewrite** ⟨*syn application*⟩ { ⟨*term*⟩ } **end**

5.4.2 Realisierung

Jede Ersetzungsregel wird in eine semantische Aktion transformiert. Dies wird in einer zusätzlichen Phase zur Übersetzzeit erledigt: Mit den Informationen aus den Ersetzungsregeln werden nach der Analyse der Grammatik und ihrer Transformation in BNF temporäre Parsetabellen generiert, die nur zur Auflösung der rechten Seiten verwendet werden. Danach kann mit der Parsergenerierung fortgefahren werden. In diesem Abschnitt wird die zusätzliche Phase informell beschrieben.

Zunächst muß ermittelt werden, welche Nonterminale für das Parsen der rechten Seiten in der Grammatik verwendet werden. Diese sind dann die Startsymbole der temporären Parsetabellen. Weiterhin werden alle Nonterminale N_i bestimmt, die auf einer rechten Seite appliziert werden. Für jedes Nonterminal N_i wird ein frisches Terminalsymbol a_i erzeugt, durch das alle Auftreten von N_i in den rechten Seiten der Ersetzungsregeln ersetzt werden. Weiterhin wird die Grammatik um die Produktionen $N_i \to a_i$ erweitert. Wenn die Grammatik keine Konflikte enthielt, dann gilt das auch für die erweiterte Grammatik, für die nun ein Parser erzeugt wird.

Mit diesem Parser kann jede rechte Seite geparst werden: Sei S die leere semantische Aktion. Bei jeder Reduktion mit einer Produktion i wird die korrespondierende semantische Aktion S_i nicht ausgeführt, sondern an S angehängt. Dabei werden Attributreferenzen

entsprechend angepaßt, damit sie sich nicht auf den Parsestack, sondern auf die vorgegebene Symbolfolge beziehen. Nach dem Parsen jeder rechten Seite enthält S die semantische Aktion, durch die die Rewrite-Aktion ersetzt wird.

Es ist offensichtlich, daß Ersetzungsregeln bei diesem Algorithmus nicht rekursiv definiert werden können. Dieser Fall muß abgefangen und als Fehler gemeldet werden.

5.4.3 Beispiele

In diesem Abschnitt werden einige Ersetzungsregeln aus der Oz-Notation beispielhaft vorgestellt. Die semantischen Aktionen, die einen abstrakten Syntaxbaum aufbauen, wurden aus Gründen der Übersichtlichkeit weggelassen.

Das erste Beispiel ist die Syntax der Methodenapplikation, bei der die Objektangabe fortgelassen werden kann, wenn es sich um `self` handelt:

```
syn Methapply($)
      '<<' Term(X1) Term(X2) '>>' => ...
   [] '<<' Term(X) '>>'
      => rewrite Methapply($)
            '<<' 'self' X '>>'
         end
end
```

Das nächste Beispiel zeigt, wie die Kurzschreibweise für die Deklaration lokaler Variablen in Prozedurrümpfen etc. expandiert werden kann.

```
syn Expression($)
      'local' Expression(E1) 'in' Expression(E2) 'end' => ...
   [] ...
end
syn InExpression($)
      Expression($)
   [] Expression(E1) 'in' Expression(E2)
      => rewrite Expression($)
            'local' E1 'in' E2 'end'
         end
end
```

Diese Ersetzungsregel implementiert das Beispiel aus Abschnitt 4.1. Es ist hier ein wenig komplizierter, da statt eines einfachen `then`-Zweigs auch eine Folge von `of` ... `elseof` ... zugelassen ist.

```
syn CaseExpression($)
   'case' Expression(E) ElseOfList(EOs) ElseCases(Cs) 'end' => ...
end
syn ElseCases($)
      'elsecase' Expression(E) ElseOfList(EOs) ElseCases(Cs)
      => rewrite ElseCases($)
            'else' 'case' E EOs Cs 'end'
         end
   [] 'else' InExpression($)
   [] true => ...
end
```

5.5 Bindungsanalyse

Dieser Abschnitt stellt die Umsetzung der Bindungsanalyse vor, die in Abschnitt 4.2 untersucht wurde. In Abschnitt 5.5.1 wird der deklarative Teil erläutert, nämlich die Grammatikannotationen (anhand ihrer konkreten Syntax). Der operationale Teil in Gestalt der vordefinierten Klasse ‚BindingAnalysis' wird in Abschnitt 5.5.2 beschrieben.

Als lauffähiges Beispiel mag begleitend zu den Erläuterungen die Spezifikation aus Abschnitt 5.1.2 gelesen werden.

5.5.1 Konkrete Syntax

In Abschnitt 4.2 wurde erläutert, wie Sichtbarkeitsbereiche in *Blöcke* gegliedert werden. Ein EBNF-Faktor kann einen solchen Block darstellen. Dabei kann optional die Variablensorte hinter dem Schlüsselwort **scope** mit angegeben werden, für die dieser Block gilt, und ein Term hinter **end**, der mit einer Liste von Paaren unifiziert wird, die die Umbenennungen des Blockes enthält.

⟨*syn factor*⟩ += ⟨*scope start*⟩ ⟨*syn alt*⟩ ⟨*scope end*⟩
⟨*scope start*⟩ += **scope**
| **scope(** ⟨*variable sort*⟩ **)**
⟨*scope end*⟩ += **end**
| **end(** ⟨*term*⟩ **)**
⟨*variable sort*⟩ ::= ⟨*atom*⟩

Die Semantik dieser Annotationen ist durch folgende Regeln definiert. Abkürzende Schreibweisen werden um Defaultwerte vervollständigt und daraufhin auf die Klasse ‚BindingAnalysis' zurückgeführt, die im nächsten Abschnitt vorgestellt wird.

```
scope          ⇒  scope(default)
end            ⇒  end(Unit)
scope(S) ...   ⇒  <<openScope(S)>> ...
end(T)         ⇒  <<closeScope(S T)>>
```

Annotationen der Variablenauftreten erfolgen hinter einem Doppelpunkt auf eine Applikation. Diese muß eine Terminalapplikation mit einer Variable als Parameter sein. Die Variable nimmt den von der Bindungsanalyse modifizierten Tokenwert auf.

⟨*syn application*⟩ += ⟨*atom label*⟩ ⟨*syn actuals*⟩ **:** ⟨*occurrence kind*⟩

Es wird zwischen bindenden und referenzierenden Auftreten unterschieden. Die Annotation **fresh** besagt, daß an dieser Stelle ein frischer Bezeichner des angegebenen Tokentyps erzeugt werden soll. Der zusätzlich zur Variablensorte übergebene Term wird an die Funktion weitergereicht, die das Lexem generiert; er kann dort beispielsweise in der Konstruktion eines *print name* verwendet werden.

⟨*occurrence kind*⟩ ::= **binder**
| **binder(** ⟨*variable sort*⟩ **)**
| **reference**
| **reference(** ⟨*variable sort*⟩ **)**
| **fresh**
| **fresh(** ⟨*term*⟩ **)**
| **fresh(** ⟨*variable sort*⟩ ⟨*term*⟩ **)**

Analog zu den Blöcken wird auch die Semantik der Variablenauftreten dadurch definiert, daß semantische Aktionen eingeführt werden, die Methoden der Klasse ‚BindingAnalysis' aufrufen:

```
id(V): binder          ⇒  id(V): binder(default)
id(V): reference       ⇒  id(V): reference(default)
id(V): fresh           ⇒  id(V): fresh(default Unit)
id(V): fresh(T)        ⇒  id(V): fresh(default T)
id(V): binder(S)       ⇒  V=( id(X) => {@S mkBinder(id X $)} )
id(V): reference(S)    ⇒  V=( id(X) => {@S mkReference(id X $)} )
id(V): fresh(S T)      ⇒  V=( => {@S mkFresh(id T $)} )
```

Beispiele hierzu findet man in Abschnitt 5.1.2.

5.5.2 Die Klasse ‚BindingAnalysis'

Die Klasse ‚BindingAnalysis' stellt den operationalen Teil der Bindungsanalyse dar, der bereits in Abschnitt 4.2 motiviert wurde. Ihre Ziele können dort nachgelesen werden.

Um bei der Durchführung der Bindungsanalyse von der Repräsentation der Variablentoken unabhängig zu bleiben, werden in abgeleiteten Klassen benutzerdefinierte Methoden benötigt, die die notwendigen Operationen auf den Variablentoken implementieren:

anonymize(+*X* ?*Y*) [Methode]
Über diese Methode wird ein Token X kopiert und dabei das Attribut für den Namen durch eine undeterminierte Variable ersetzt; die anderen Intrinsics bleiben erhalten. Die Kopie wird in Y zurückgegeben.

nameToken(+*X* +*A*) [Methode]
Hierüber soll ein bereits anonymisiertes Token X mit dem Namen A versehen werden.

generate(+*X* ?*A*) [Methode]
Diese Methode wird aufgerufen, wenn ein neuer Variablenbezeichner A benötigt wird. Die Methode muß bei jedem Aufruf ein anderes Ergebnis liefern. Der eingegebene Term X stammt aus den Grammatikannotationen und kann zur Vergabe eines *print name* verwendet werden.

tokenToAtom(+*X* ?*A*) [Methode]
Um Variablentoken intern auseinanderhalten zu können, werden sie durch Atome repräsentiert. Diese Methode übernimmt die Konversion.

makeTokenFromAtom(+*A* ?*X*) [Methode]
Wenn ein frischer Variablenname A über ‚generate' als Atom erzeugt wurde, wird er über diese Methode in ein Variablentoken X gewandelt. Die anderen Intrinsics als der Name sollen dabei auf Default-Werte gesetzt werden.

Die Klasse ‚BindingAnalysis' definiert folgende Methoden für den Benutzer:

init() [Methoden]
initialisiert die internen Strukturen, die die Umbenennungen speichern.

openScope(+*A*) [Methode]
öffnet einen Scopus der Variablensorte A.

closeScope(+*A* ?*X*) [Methode]
schließt einen Scopus der Variablensorte A. In X wird eine Liste von Paaren zurückgegeben, die Atome auf Atome abbilden. Dies sind die in dem Block durchgeführten Umbenennungen.

mkBinder(+*A* +*X* ?*Y*) [Methode]
deklariert ein Token mit Typ A und Wert X als bindendes Auftreten einer Variable, die umbenannt werden soll. Das umbenannte Variablentoken wird in Y zurückgegeben.

mkReference(+*A* +*X* ?*Y*) [Methode]
erklärt ein Token mit Typ A und Wert X als referenzierendes Auftreten einer Variable. Das umbenannte Variablentoken wird in Y zurückgegeben. Eventuell ist es noch anonym und wird erst später benannt.

mkFresh(+*A* +*X* ?*Y*) [Methode]
fordert die Erzeugung eines Variablentokens mit Tokentyp A an. In X können beliebige Benutzer-Daten aus der Grammatikannotation übergeben werden, die etwa einen Präfix für den *print name* liefern. In Y wird das erzeugte Variablentoken zurückgegeben.

enterSubstitution(+*A1* +*A2*) [Methode]
trägt eine Umbenennung von Atom *A1* nach *A2* in die Umbenennungsfunktion ein. Dies kann dazu verwendet werden, implizit deklarierte Bezeichner (etwa durch eine Import-Direktive) in die Bindungsanalyse aufzunehmen. Die Substitution muß dabei aus einer von ‚closeScope' gelieferten Liste stammen.

5.6 Implementierung

Wie gezeigt, hat die Spezifikationssprache des Werkzeuges selbst eine relativ umfangreiche Eingabesprache. Die Expansion der Produktionsschemata verläßt sich weiterhin auf die eindeutige Umbenennung der Bezeichner. Es ist also naheliegend, das System unter Verwendung seiner selbst zu implementieren. Da es in einem ersten Schritt noch gar nicht existiert, ist ein sogenannter *Bootstrapping-Prozeß* nötig. Dabei wird mit möglichst geringem Aufwand ein lauffähiger Prototyp des Werkzeugs erstellt, der zwar nicht unbedingt die volle Funktionalität bietet, mit der jedoch die endgültige Version übersetzt werden kann.

Im vorliegenden Fall wurde dies dadurch realisiert, daß das Frontend des existierenden DFKI-Oz-Compilers modifiziert wurde. Dieses wurde in C++ unter Verwendung von *flex* und *Bison* geschrieben und erstellt einen abstrakten Syntaxbaum in Tupelform als Tel-Werte. Tel ist die Programmiersprache, in der die restlichen Compilerphasen implementiert sind. Allerdings existierte bereits eine daraus hervorgegangene Version, die stattdessen Oz-Tupel aufbaut.

Letztere wurde um die für die Grammatikspezifikation hinzugekommenen Syntaxregeln erweitert. Dabei wurde allerdings alles fortgelassen, was mit der Bindungsanalyse oder den Ersetzungsregeln zusammenhängt. Das modifizierte Frontend erstellt seine abstrakte Syntax ebenfalls in Tupelform; eine Forderung bei der Implementierung des Werkzeugs war aber, eine objektorientierte abstrakte Syntax zu verwenden. Daher muß die Tupelform nach Abschluß des Parsevorgangs durchlaufen werden, wobei die enthaltenen Grammatikspezifikationen in Objekte umgesetzt werden. Hier kann nun das endgültige Werkzeug aufsetzen und die benötigten Transformationen durchführen, um ein ausführbares Oz-Programm zu erhalten.

Abbildung 5.2 zeigt die Architektur des Werkzeugs. Dabei sind die Phasen durch gestrichelte Rechtecke gekennzeichnet, die bei dem Bootstrapping-Prozeß und bei der Production-Version des Werkzeugs unterschiedlich ablaufen.

In der Abbildung ist auch gezeigt, daß für die Generierung des Scanners auf *flex* zurückgegriffen wird, indem eine Eingabedatei für diesen generiert wird. Der daraus erstellte

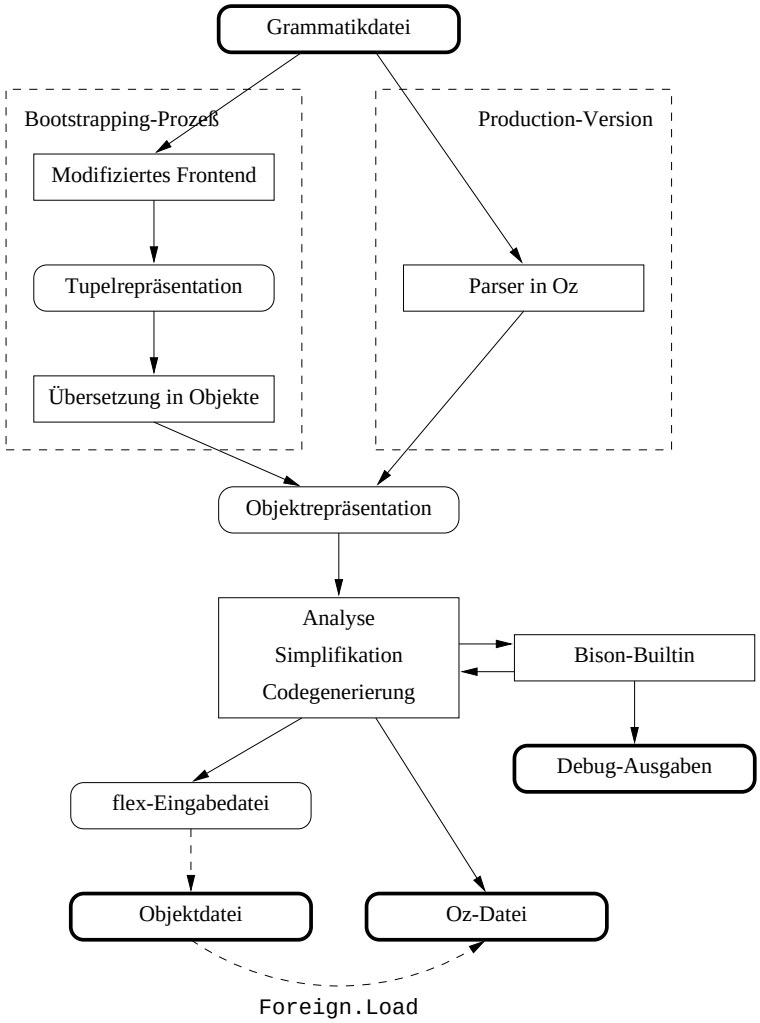

Abbildung 5.2: Die Architektur des Werkzeugs

Scanner wird zu dem Benutzerprogramm über das *foreign-function*-Interface von Oz hinzugebunden. Ähnlich wird für die Generierung der Parsetabellen *Bison* verwendet. Statt über Dateien zu kommunizieren wurde *Bison* aber abgewandelt und als Oz-Builtin zur Verfügung gestellt. Die in BNF transformierte Grammatik wird dem Builtin in Tupeldarstellung übergeben, das seinerseits die daraus generierten Parsetabellen als Tupel zurückliefert. Der Tabellenparser von Bison wurde in Oz übersetzt, so daß das *foreign-function*-Interface hierfür zur Laufzeit nicht notwendig ist.

Kapitel 6

Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefaßt. Zunächst wird in Abschnitt 6.1 das entwickelte Werkzeug bewertet. Es wird überprüft, ob es die in Abschnitt 1.4 gestellten Anforderungen erfüllt. Anhand der gemachten Erfahrungen wird weiterhin beurteilt, welche Aspekte in dem System gut beziehungsweise weniger gut realisiert wurden. Abschnitt 6.2 gibt einen Ausblick, in welchen Bereichen noch Arbeit geleistet werden sollte, um das Werkzeug zu verbessern.

6.1 Bewertung des Systems

Zur Validierung des Systems wurde ein Front-End für seine eigene Spezifikationssprache erstellt. Teilmenge dieser Sprache ist die gesamte Syntax von Oz; damit handelt es sich also um ein nichttriviales Testbeispiel. Basierend auf den dabei gewonnenen Erfahrungen soll in diesem Abschnitt die Qualität des entwickelten Front-End-Generators evaluiert werden.

Im folgenden wird das System gegen die globalen Anforderungen validiert, die in Abschnitt 1.4 formuliert wurden.

Unterstützte Phasen. Der Front-End-Generator unterstützt alle Phasen, die geplant waren, mit Ausnahme des automatischen Aufbaus einer abstrakten Syntax. Für diese sind noch weitere Nachforschungen nötig (vgl. Ausblick).

Unabhängige Verwendbarkeit. Durch den klassenbasierten Ansatz ist die getrennte Verwendbarkeit von Scanner und Parser (mit dessen Erweiterungen) möglich. Auch die freie Kombination von Scannern und Parsern, ohne daß diese mehrfach im Speicher stehen müßten, wird durch Vererbung erlaubt.

Hohe Integration. Der Benutzer muß sich um die Schnittstellen zwischen Scanner und Parser für die meisten Anwendungen nicht sorgen, da die Kopplung vom Werkzeug

übernommen wird – unabhängig davon, ob die lexikalische und die syntaktische Analyse in derselben oder in unterschiedlichen Grammatiken definiert wurden und durch Vererbung gekoppelt werden. Weiterhin werden die mit dem Parsen verzahnt ablaufenden Phasen in einer eigenen konkreten Syntax direkt in der Grammatik spezifiziert. Somit ist die Forderung nach hoher Integration erfüllt.

Einbettung in Oz. Überall, wo semantische Berechnungen nötig sind, wird Oz-Syntax verwendet, daher kann immer die volle Mächtigkeit dieser Sprache ausgenutzt werden. Weiterhin gliedern sich Grammatikdefinitionen nach ihrer Transformation in Klassen gut in die Sprache ein. Damit kann die Einbettung in Oz als erfolgreich bewertet werden.

Information Hiding. Da der Benutzer eigene Methodendefinitionen in die Grammatikklasse einbringen, diese von beliebigen Klassen erben lassen oder sie spezialisieren kann, sind für benutzerdefinierte Operationen alle Möglichkeiten des Information Hiding von Oz verfügbar. Seitens der Parsetabellen jedoch sind in dieser Beziehung noch Verbesserungen möglich: Um die vordefinierte Parserklasse von der generierten Klasse abkoppeln zu können, mußten die Features für die Speicherung der Parsetabellen sowie die generierten Methoden öffentlich sein und sind demzufolge von außen zugreifbar. Wenn Systemvariablen verwendet würden, könnte dies vermieden werden.

Interaktive Systeme. Bei jedem Teilwerkzeug wurde darauf geachtet, daß es die Implementierbarkeit interaktiver Systeme nicht einschränkt. Der Parser ist deterministisch und die Ersetzungsregeln werden verschränkt mit dem Parsen ausgeführt, sobald sie anwendbar sind. Auch die Bindungsanalyse braucht semantische Aktionen nicht zu verzögern, da mit undeterminierten Variablen als Platzhalter für die Namen von Bezeichnern weiterverfahren wird.

Nun werden die Fortschritte, die gegenüber existierenden Werkzeugen gemacht wurden, zusammengefaßt.

- Es wurden Produktionsschemata entwickelt, die in dieser Form in der Literatur nicht gefunden wurden. Ein wichtiges Merkmal ist ihre (bei *Combinator Parsing* nicht gegebene) statische Expansion in BNF. Diese hat mehrere Vorteile. Zum einen kann die tatsächlich erkannte Sprache damit formal beschrieben werden. Zum anderen bedeutet dies, daß die Verwendung von Produktionsschemata keinen Effizienzverlust bewirkt, sondern daß im Gegenteil für einfache EBNF-Operatoren komplexe und effiziente Implementierungen bereitgestellt werden können.

- Die automatische Herleitung der Attributarten, ob also ein Parameter einer Produktion ein synthetisiertes oder ein vererbtes Attribut darstellt, erlaubt sehr mächtige Erweiterungen des LR-Parseverfahrens: Es wurde die Definition und Verwendung lokaler Variablen in Regeln möglich und es wurde ein Schritt in Richtung der Unterstützung von L- statt nur S-attributierten Grammatiken gemacht. Damit wurde

die Unabhängigkeit von der Parsetechnik deutlich verbessert. Etwas Äquivalentes wurde in keinem existierenden Werkzeug gefunden; somit stellt dies eine wirkliche Neuerung dar.

- Es wurde ein neues System zur Spezifikation von Ersetzungsregeln entwickelt. Dieses erfüllte alle gewünschten Eigenschaften, ist aber leider nicht so lesbar geworden wie erhofft.

- Ein existierendes System zur Spezifikation von Bezeichnerbindungen wurde erweitert und in einen deklarativen und einen operationalen Teil faktorisiert, wodurch es deutlich mächtiger wurde.

Trotz der Mächtigkeit des Systems ist die Effizienz sehr zufriedenstellend: Die Generierung eines Front-Ends aus einer Beschreibung ist besonders deswegen sehr schnell, weil für die Tabellenerzeugung auf die Werkzeuge *flex* und *bison* zurückgegriffen wurde. Da auch zur Laufzeit ein *flex*-Scanner verwendet wird, ist die lexikalische Analyse nicht bedeutend langsamer als in der Programmiersprache C, da die Ausführung der semantischen Aktionen nur einen kleinen Teil von ihr ausmachen. Am langsamsten ist zur Laufzeit die syntaktische Analyse, da der Tabellenparser von *Bison* nach Oz übersetzt worden ist. Auch der Speicherverbrauch ist nur linear größer als in C, da Oz auch für Methoden, in die die `goto`s aus der C-Version übersetzt werden mußten, Tail-Calls unterstützt.

6.2 Ausblick

Es steht fest, daß das Werkzeug sich nun in der Praxis bewähren muß. Aber bereits die Erfahrungen aus der Entwicklung und den Testbeispielen lassen den Wunsch nach einigen zukünftigen Untersuchungen und Erweiterungen aufkommen.

- Um noch weiter von der verwendeten Parsetechnik abstrahieren zu können und vielleicht in einer späteren Version verschiedene Parsetechniken zur Auswahl anbieten zu können, müssen weitere automatische Transformationen der Grammatik implementiert werden. Diese sollten eine unabhängig von jeder Parsetechnik spezifizierte Grammatik für das jeweils gewünschte Verfahren aufbereiten.

- Das vielversprechende Konzept dynamischer Operatortabellen, das in Abschnitt 3.2.1 vorgestellt wurde, sollte implementiert werden.

- Zugunsten der besseren Wartbarkeit und der größeren Ähnlichkeit zu Oz-Methoden sollte es auch für die Parameterlisten von Nonterminalen möglich sein, statt nur Tupelsyntax auch die Recordschreibweise zuzulassen.

- Es wäre sinnvoll, im Standardumfang eine von der Klasse „BindingAnalysis“ abgeleitete Klasse anzubieten, die eine vollwertige Symboltabellenverwaltung übernimmt.

- Bei der Bindungsanalyse gibt es noch keine konkrete Syntax, mit der man deklarieren könnte, für welche Tokentypen sie durchgeführt werden soll. Die benötigten Klassenattribute muß der Benutzer bisher noch explizit erklären und initialisieren. Dies sollte automatisiert werden.

- Um die Umbenennung von Variablennamen bei Im- und Export zu vermeiden, sollten bei der Bindungsanalyse bestimmte Variablenauftreten (beispielsweise die aus dem globalen Sichtbarkeitsbereich) mit der Eigenschaft markiert werden können, daß sie nicht umbenannt werden sollen. Dies würde jedoch bedeuten, daß die endgültigen Namen erst nach Abschluß des gesamten Parsevorgangs determiniert werden könnten. Es ist denkbar, eine alternative Klasse zu „BindingAnalysis“ anzubieten, die dieses unterstützt.

- Es müssen noch Untersuchungen gemacht werden, wie ein automatischer Aufbau objektorientierter abstrakter Syntax realisiert werden könnte, der nicht unter den Nachteilen leidet, die in Abschnitt 4.3.3 erläutert wurden.

- In den Standardumfang des Front-End-Generators sollte eine Funktion übernommen werden, die einen aus (manuellem) Unparsing gewonnenen virtuellen String ausgibt. Dabei sollten die Steuertupel für die Formatierung, wie sie in Abschnitt 4.4.3 beschrieben wurden, korrekt umgesetzt werden.

- Es sollten weitere Untersuchungen über die Möglichkeit der Modularisierung von Spezifikationen gemacht werden. Es sollte von einer Grammatik geerbt werden können und diese durch Erweiterungen, Streichungen und Modifikationen geändert werden können. Da jedoch bei vollkompositionalen Sprachen eine Aufteilung der Grammatik in wenig gekoppelte oder gar hierarchische Teilaufgaben nur sehr begrenzt möglich ist, hat dieses Ziel eine geringe Priorität.

Anhang A

Die verwendete Grammatik-Notation

In diesem Anhang wird die Notation für kontextfreie Grammatiken erläutert, mit der in Kapitel 5 die konkrete Syntax der Spezifikationssprache definiert wird.

Es handelt sich um eine erweiterte Backus-Naur-Form, die sich aus folgenden Bestandteilen zusammensetzt:

- Terminale und Nonterminale werden in spitze Klammern $\langle \ldots \rangle$ eingeschlossen;
- die linke Seite wird durch ::= oder += von der rechten getrennt, wobei durch += Produktionen zu einem bestehenden Nonterminal hinzugefügt werden;
- der senkrechte Strich trennt Alternativen;
- die geschweiften Klammern notieren eine 0 bis n-fache Wiederholung ihres Arguments;
- die eckigen Klammern markieren eine 0 bis 1-malige Wiederholung;
- Literale Zeichenketten werden im `Typewriter`-Zeichensatz gesetzt.

Das besondere an dieser Notation ist, daß Grammatiken inkrementell definiert werden können: Eine bestehende Grammatik kann dadurch erweitert werden, daß ihren Nonterminalen neue Produktionen über += hinzugefügt werden.

Elemente aus der Oz-Syntax

Da die Spezifikationssprache des Werkzeugs in Oz eingebettet ist, gelten dieselben lexikalischen Konventionen wie für Oz. Ein zusätzliches Terminal ist $\langle regex \rangle$, das für einen regulären Ausdruck steht. Dieser wird in *flex*-Syntax notiert und durch spitze Klammern (< und >) eingeschlossen.

Die folgenden Terminale aus der Oz-Syntax werden verwendet:

⟨*atom*⟩ steht für ein Oz-Atom (ob gequotet oder nicht, wird nicht unterschieden).

⟨*variable*⟩ ist eine Oz-Variable.

⟨*atom label*⟩ ist ein Oz-Atom, das direkt von einer öffnenden runden Klammer gefolgt wird.

⟨*variable label*⟩ ist eine Oz-Variable, die direkt von einer öffnenden runden Klammer gefolgt wird.

Weiterhin werden einige Nonterminale aus der Oz-Syntax referenziert:

⟨*label*⟩ steht für eine Oz-Variable oder ein Oz-Atom.

⟨*escaped variable*⟩ steht für eine Oz-Variable, die optional mit einem Ausrufezeichen präfixiert wird, das eine implizite Deklaration der Variablen verhindert.

⟨*top level expression*⟩ ist ein Ausdruck auf oberster Ebene des Programmtextes.

⟨*expression*⟩ ist ein Oz-Ausdruck.

⟨*term*⟩ steht für einen Oz-Term.

⟨*class descriptor*⟩ steht für einen erlaubten Klassendeskriptor, also für eine der **`from`**-, **`feat`**-, **`attr`**- oder **`meth`**-Klauseln.

Literaturverzeichnis

[Aas92] Annika Aasa. *User Defined Syntax.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, 1992.

[Ada83] Ada Joint Program Office, U. S. Government. *Reference Manual for the Ada Language*, ANSI/MIL-STD 1815a, 1983.

[Age94] Ole Agesen. Mango: A parser generator for Self. SMLITR 94–27, Computer Science Department, Stanford University/Sun Microsystems Laboratories, Inc., June 1994.

[AMT92] Andrew W. Appel, James S. Mattson, and David R. Tarditi. *A Lexial Analyzer Generator for Standard ML.* Princeton University, Version 1.4, October 1992.

[And95] Anders Andersson. *SAGA User Manual.* Programming Systems Group, Swedish Institute of Computer Science, June 1995.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Aug93] Mikhail Auguston. *Rigal Programming System Language Description.* Department of Mathematics and Computer Science, University of Latvia, July 1993.

[Ave95] Jürgen Avenhaus. *Reduktionssysteme.* Springer-Lehrbuch, Berlin, Heidelberg, 1995.

[BB92] P. T. Breuer and J. P. Bowen. A prettier compiler-compiler: Generating higher order parsers in C. PRG-TR 20–92, Programming Research Group, Oxford University Computing Laboratory, 1992.

[Bea90] Steven J. Beaty. ParsesraP: Using one grammar to specify both input and output. *ACM SIGPLAN Notices*, 30(2), February 1990.

[Bev93] Stephen J. Bevan. Abstract syntax: Is there such a thing as the right one?, January 1993.

[BP95] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast YACC-compatible parsers (for very little effort). TR 95–09, Department of Computer Science of the University of Arizona, September 1995.

[CCH95] James R. Cordy, Ian H. Carmichael, and Russell Halliday. *The TXL Programming Language*. Software Technology Laboratory, Department of Computing and Information Science, Queen's University, Kingston, Canada, April 1995. Commercial Distribution by Legasys Corp.

[CMA94] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, February 1994.

[Coë93] Alain Coëtmeur. *Bison++*. R&D Department, Informatique-CDC, France, March 1993. Based on [DS95, Version 1.21].

[Com96a] Compiler Tools Group, Department of Electrical and Computer Engineering, University of Colorado. *Eli – Lexical Analysis*, Revision 2.8, 1996.

[Com96b] Compiler Tools Group, Department of Electrical and Computer Engineering, University of Colorado. *Eli – Syntactic Analysis*, Revision 1.10, 1996.

[Dor96] Chris Dornan. *lx – A Lex for Haskell Programmers*. ftp://ftp.cs.bris.ac.uk/users/dornan/lx.tar.gz, July 1996.

[DS95] Charles Donelly and Richard Stallman. *Bison – The YACC-compatible Parser Generator (Reference Manual)*. Free Software Foundation, Version 1.25, November 1995. On-Line Info File.

[DSH95] Charles Donelly, Richard Stallman, and Wilfred J. Hansen. *Bison – The YACC-compatible Parser Generator (Reference Manual)*. Andrew Consortium, Carnegie Mellon University, Version A2.6, June 1995. Modified from [DS95, Version 1.24].

[EKVW94] Reinhard Eppler, Peter Knauber, Stefan Vorwieger, and Hans-Wilm Wippermann. The Refus programming language. Interner Bericht 253/94, AG Programmiersprachen und Compiler, Fachbereich Informatik, Universität Kaiserslautern, September 1994.

[Fai87] J. Fairbairn. Making form follow function: An exercise in functional programming style. *Software – Practice and Experience*, 17(6):379–386, 1987.

[GM96] Andy Gill and Simon Marlow. *Happy, the Parser Generator System for Haskell*, Version 0.9, February 1996. ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/happy/.

[Gro87] Josef Grosch. Efficient generation of table-driven scanners. Compiler Generation Report No. 2, GMD-Forschungsstelle an der Universität Karlsruhe, May 1987.

[Gro88a] Josef Grosch. Lalr – a generator for efficient parsers. Compiler Generation Report No. 8, GMD-Forschungsstelle an der Universität Karlsruhe, October 1988.

[Gro88b] Josef Grosch. Selected examples of scanner specifications. Compiler Generation Report No. 7, GMD-Forschungsstelle an der Universität Karlsruhe, March 1988.

[Gro89] Josef Grosch. Efficient and comfortable error recovery in recursive descent parsers. Compiler Generation Report No. 19, GMD-Forschungsstelle an der Universität Karlsruhe, December 1989.

[Gro91a] Josef Grosch. Transformation of attributed trees using pattern matching. Compiler Generation Report No. 27, GMD-Forschungsstelle an der Universität Karlsruhe, August 1991.

[Gro91b] Josef Grosch. Transformation of attributed trees using pattern matching. Compiler Generation Report No. 26, GMD-Forschungsstelle an der Universität Karlsruhe, November 1991.

[Gro92] Josef Grosch. Rex – a scanner generator. Compiler Generation Report No. 5, GMD-Forschungsstelle an der Universität Karlsruhe, July 1992.

[Gro93] Josef Grosch. Ast – a generator for abstract syntax trees. Compiler Generation Report No. 15, GMD-Forschungsstelle an der Universität Karlsruhe, August 1993.

[GS88] Christian Genillard and A. Strohmeier. GRAMOL: A grammar description language for lexical and syntactic parsers. *ACM SIGPLAN Notices*, 23(10):103–122, 1988.

[GV92] Josef Grosch and Bertram Vielsack. The parser generators lalr and ell. Compiler Generation Report No. 8, GMD-Forschungsstelle an der Universität Karlsruhe, July 1992.

[GW85] G. Goos and W. M. Waite. *Compiler Construction.* Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1985.

[HDB92] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Syntactic abstraction in Scheme. Technical Report #355, Computer Science Department, Indiana University, June 1992.

[Hen95] Martin Henz. The Oz Notation. Oz Documentation Series, Programming Systems Lab, DFKI, May 1995.

[Hil94] Steve Hill. Continuation passing combinators for parsing precedence grammars. Technical Report 24–94, University of Kent, Computing Laboratory, Canterbury, UK, November 1994.

[HMSW96] Martin Henz, Martin Müller, Christian Schulte, and Jörg Würtz. The Oz Standard Modules. Oz Documentation Series, Programming Systems Lab, DFKI, May 1996.

[HPJW92] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell, a non-strict purely funktional programming languge (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hud96] Scott E. Hudson. *CUP User's Manual*. Graphics Visualization and Usability Center, Georgia Institute of Technology, version 0.9e, March 1996.

[Jan94] Sverker Janson. *AKL – A Multiparadigm Programming Language*. PhD thesis, Swedish Institute of Computer Science, 1994.

[Joh75] S. C. Johnson. Yacc – yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[JPB94] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. A multiparadigm approach to compiler construction. *ACM SIGPLAN Notices*, 29(9):29–37, September 1994.

[JW75] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, New York, 1975.

[Kle56] S. C. Kleene. Representation of events in nerve nets. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–40. Princeton University Press, 1956.

[Klu91] Michael Klug. VisiCola, a model and a language for visibility control in programming languages. *ACM SIGPLAN Notices*, 26(2):51–63, February 1991.

[Kna96] Peter Knauber. Das OCC-System. AG Programmiersprachen und Compiler, Fachbereich Informatik, Universität Kaiserslautern, 1996.

[Knu91] Donald Ervin Knuth. *The TEXbook*, volume A of *Computers & Typesetting*. Addison-Wesley, May 1991.

[KR78] Brian Wilson Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall Software Series, New Jersey, 1978.

[Küh94] Guido Kühn. Definition und Implementierung eines Scannergenerators. Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1994.

[KVE94] Peter Knauber, Stefan Vorwieger, and Reinhard Eppler. Terminologie des Übersetzerbaus. Interner Bericht 255/94, AG Programmiersprachen und Compiler, Fachbereich Informatik, Universität Kaiserslautern, October 1994.

[KW95] Basim M. Kadhim and William M. Waite. Maptool – mapping between concrete and abstract syntaxes. CS-CU 765-95, Department of Computer Science, University of Colorado, February 1995.

[Les75] M. E. Lesk. LEX – a lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, 1975.

[MAS96] MASTER Information Systems Corporation, 3596 Pimlico Drive, Pleasanton, CA 94588. *The MUSKOX Software Engineering Tool*, Version 2.0, July 1996. Product Description and Release Notes, http://misc-sun.mastersys.com.

[May81] Brian H. Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3), 1981.

[MW91] H. Mössenböck and N. Wirth. The programming language Oberon-2. Report 160, Institut für Computersysteme, ETH Zürich, May 1991.

[N+63] P. Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[Par77] David Lorge Parnas. Use of abstract interfaces in the development of software for embedded computer systems. NRL Report 8047, Naval Research Laboratory, Washington D. C., 1977.

[Par95] Terence John Parr. Language translation using PCCTS and C++ (a reference guide). ftp://ftp.parr-research.com/pub/pccts/Book/reference.ps, June 1995.

[Pax95] Vern Paxson. *flex – Fast Lexical Analyzer Generator*, Version 2.5.2, April 1995. Unix On-Line Manual Page.

[PDC91] T. J. Parr, H. G. Dietz, and W. E. Cohen. *PCCTS Reference Manual*. School of Electrical Engineering, Purdue University, Version 1.0, August 1991.

[PJ88] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, New-York, 1988.

[PQ95] Terence J. Parr and Russell W. Quong. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34, July 1995.

[PVGK93] Kjell Post, Allen Van Gelder, and James Kerr. Deterministic parsing of languages with dynamic operators. In Dale Miller, editor, *Logic Programming – Proceedings of the 1993 International Symposium*, pages 456–472, Vancouver, Canada, 1993. The MIT Press.

[PW80] F. C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis. *Artificial Intelligence*, 13:231–278, 1980.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, Third edition, 1988. First edition, Cornell University, August 1985; Second edition, Cornell University, June 1987.

[Sch] F. W. Schröer. The Gentle compiler construction system. http://www.first.gmd.de/gentle/.

[Sch89] F. W. Schröer. Gentle. In W. M. Waite, J. Grosch, and F. W. Schröer, editors, *Three Compiler Specifications.* GMD – German National Research Center for Information Technology, 1989. GMD Studien Nr. 166.

[Smo94] Gert Smolka. The definition of Kernel Oz. Oz Documentation Series, Programming Systems Lab, DFKI, November 1994.

[Str87] Bjarne Stroustup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1987.

[TA91] David R. Tarditi and Andrew W. Appel. *ML-Yacc Users's Manual.* School of Computer Science, Carnegie Mellon University and Department of Computer Science, Princeton University, Version 2.1, March 1991.

[vEB93] Peter van Eijk and Axel Belinfante. *The Term Processor Kimwitu – Manual and Cookbook.* University of Twente, Eschente, The Netherlands, Version 7 for Kimwitu Version 3.8, March 1993.

[Wad90] Vance E. Waddle. Production trees: A compact representation of parsed programs. *ACM Transactions on Programming Languages and Systems*, 12(1):61–83, January 1990.

[War77] David H. D. Warren. Implementing Prolog – compiling predicate logic programs. Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[War80] David H. D. Warren. Logic programming and compiler writing. *Software – Practice and Experience*, 10:97–125, 1980.

[WEKV93] Hans-Wilm Wippermann, Reinhard Eppler, Peter Knauber, and Stefan Vorwieger. W-Lisp Sprachbeschreibung. Interner Bericht 237/93, Fachbereich Informatik, Universität Kaiserslautern, Dezember 1993.

[Wil79] R. Wilhelm. Attributierte Grammatiken. *Informatik-Spektrum*, 2(3):123–130, 1979.

[WKE96] Hans-Wilm Wippermann, Peter Knauber, and Reinhard Eppler. W-Lisp-Sprachreport Version 3. Version 2 wird in [WEKV93] beschrieben, 1996.