Objekt-orientierter Compilerentwurf

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Roland Loser

1997

Leiter der Arbeit:
Prof. O. Nierstrasz
Institut für angewandte Mathematik und Informatik

Zusammenfassung

Bei Entwicklung von Compilerern und Interpretern für Programmiersprachen kommen häufig Werkzeugen wie Lex und Yacc zur Anwendung. Wenn gleichzeitig objekt-orientierte Methoden in der Entwicklung zum Einsatz kommen, kann dies zu Paradigmen-Konflikte führen. In dieser Arbeit wird ein objekt-orientierter Compilerentwurf in C++ vorgestellt, welcher auf Lex und Yacc verzichtet, und den objekt-orientierten Ansatz auf alle Komponenten eines Compilers anwendet.

Inhaltsverzeichnis

Kapitel 1

Einführung

Seit der Einführung von Programmiersprachen in den späten 1950er Jahren spielen Compiler eine zentrale Rolle in der Software-Entwicklung. In den 1960er und 70er Jahren wurde viel in die Forschung über höhere Programmiersprachen und die entsprechenden Compiler investiert. Dementsprechend weit entwickelt und standardisiert sind die Grundkomponenten eines Ein-Pass-Compilers, als dies sind, lexikalische Analyse, syntaktische Analyse, semantische Analyse, Optimierung und Codeerzeugung. Vorallem für die ersten beiden Komponenten sind gute Werkzeuge wie **lex & yacc**[LMB92] vorhanden, die sich bereits über Jahre bewährt haben, und für fast alle Probleme welche im Zusammenhang mit Grammatiken für Programmiersprachen auftreten gute Lösungen bieten.

Lex und Yacc sind typische Unix-Werkzeuge, welche mit der Programmiersprache **C** erstellt wurden. Diese Werkzeuge erzeugen Scanner und Parser als **C** Quelltextdateien, welche dann mit der eigentlichen Applikation compiliert und gelinkt werden müssen. Die Programmiersprache **C** wurde in den letzten Jahren von der objekt-orientierten Erweiterung **C++** abgelöst. So ist es ganz natürlich, dass neben Lex und Yacc auch C++ in der Compilerentwicklung zum Einsatz kommt. Wenn diese gleichzeitig zur Anwendung gelangen, kann es jedoch zu Paradigmen-Konflikten kommen, weil sich das objekt-orientierte Paradigma[1] von C++ zum Beispiel nicht mit der Kommunikation von Lex und Yacc über globale Variabeln verträgt.

Es gibt neuere Werkzeuge die solche Konflikte beheben wollen, wie zum Beispiel Bison++ welches ein Nachfolger von Yacc ist, und im Gegensatz zu Yacc nicht C sondern C++ Quelltext erzeugt. Dabei werden die Möglichkeiten von C++ aber einzig zur Datenkapselung verwendet, das ganze restliche Verhalten orientiert sich immer noch am alten C-Stil.

[Mey94] beschreibt seine Beweggründe, warum er eine eigene objekt-orientierte Scanner- und Parserlibrary in Eiffel implementiert hat, wie folgt:

[1] Es ist interessant die Bedeutung des Wortes Paradigma zu betrachten. Paradigma bedeutete ursprünglich ein veranschaulichendes Beispiel, insbesondere ein Beispielsatz, welcher alle gebeugten Formen eines Wortes zeigt.Thomas Kuhn erweiterte den Begriff in seinem Buch *The Structure of Scientific Revolutions*[Kuh70] auf die Definition: ”Ein Paradigma ist eine Menge von Theorien, Standards und Methoden welche zusammen einen Weg zur Organisation von Wissen darstellten - das heisst, eine Art die Welt zu sehen”.

- Der Wunsch die Aufgabe der Syntaxanalyse mit dem Rest eines objekt-orientierten Systems auf einfache und geeignete Weise zu verknüpfen.
- Der Wunsch objekt-orienterte Prinzipien möglichst vollständig auf ein System anzuwenden, um die Vorzüge[2] dieser Methode nicht durch einzelne Komponenten zu gefährden.

Aus den oben genannten Gründen erscheint es sinnvoll, einen objekt-orientierten Compilerentwurf zu betrachten, welcher auf Werkzeuge wie Lex und Yacc verzichtet und somit den Pardigmenkonflikt eliminiert und gleichzeitig von den Vorteilen des objekt-orientierten Ansatzes profitiert.

1.1 Ziele der Arbeit

Ziel der Arbeit ist es durch die Implementation eines Frameworks[3] für den Compilerbau, die objekt-orientierten Methoden auf alle Bereiche eines Compilers anzuwenden. Das Framework soll anhand von Beispielimplementationen getestet werden und die Erkenntnisse mit den Aussagen, welche im Buch *Object-oriented Compiler Construction*[Hol95] gemacht werden, verglichen werden. Dabei wird bewusst auf Werkzeuge wie Lex und Yacc verzichtet werden, da sie nicht dem objekt-orientierten Paradigma entsprechen.

Für den Implementationsteil wird die Programmiersprache C++ gewählt, da sich mit ihr alle objekt-orientierten Konzepte umsetzen lassen und da sie auf fast allen Plattformen frei verfügbar ist.

1.2 Überblick über die Arbeit

Das Framework, das in dieser Arbeit entwickelt wird, besteht grob gesehen aus 4 Komponenten.

1. Klassen welche die Grammatik einer Sprache repräsentieren. Diese werden in Kapitel 2 beschrieben.
2. Klassen für die Konstruktion eines Scanners zur lexikalischen Analyse. Diese werden in Kapitel 3 beschrieben.
3. Klassen für die Konstruktion eines Parsers für die syntaktische Analyse. Diese werden in Kapitel 4 beschrieben. Kapitel 5 und 6 befassen sich mit Aspekten dieses Problems, namentlich der Fehlerbehandlung und der dynamischen Konfliklösung von shift/reduce-Konflikten.

[2] im speziellen Wiederverwendung und Ausbaufähigkeit

[3] [Joh97] definiert den Frameworkbegriff wie folgt : ”ein Framework ist ein wiederverwendbares Design für alles oder einen Teil eines Systems, welches durch eine Menge abstrakter Klassen und die Art wie ihre Instanzen interagieren, repräsentiert wird. ”

4. Klassen für die Konstruktion eines abstrakten Syntaxbaumes. Diese werden in Kapitel 7 beschrieben.

Kapitel 8 beschreibt zudem einen Objektspeichermechanismus, welcher es ermöglicht, eine einmal erzeugte Objekthierarchie des Frameworks abzuspeichern und wieder zu laden.

Kapitel 9 vergleicht die Arbeit mit dem Buch von Holmes[Hol95] und weiteren Arbeiten auf diesem Gebiet.

Kapitel 10 zieht die Schlussfolgerungen und gibt Anregungen für weitere Arbeiten.

Im Anhang wird zudem die Implementation eines Parsergenerators für das Framework vorgestellt. Dieses Beispiel soll gleichermassen die Anwendung und den Einsatz des Frameworks demonstrieren.

Die meisten Kapitel beginnen mit einem Theorieteil, welcher die anschliessende Implementation im Framework motivieren soll.

Kapitel 2

Grammatik

Programmiersprachen werden in der Regel durch kontextfreie Grammatiken beschrieben. Auf die Theorie der formalen Grammatiken und deren Einteilung nach Chomsky[Cho56] wird im ersten Abschnitt dieses Kapitels eingegangen. Gleichzeitig werden die einzelnen Elemente einer Grammatik betrachtet und so die Grundlagen für den zweiten Abschnitt dieses Kapitels gelegt. Dieser befasst sich damit, wie die Grammatik im Framework dargestellt wird.

2.1 Theorie

Um über Grammatiken reden zu können, müssen zuerst einige grundlegende Definitionen aufgestellt werden.

2.1.1 Definitionen

Ein *Alphabet* ist eine endliche Menge von Symbolen. Ein *Wort* über einem Alphabet ist ein endliche Sequenz von Symbolen des Alphabets. Sei Σ ein Alphabet , dann bezeichnet Σ^* die Menge aller Wörter, welche über Σ gebildet werden können, inklusive des leeren Wortes ϵ. Die Menge Σ^+ wird als $\Sigma^* - \{\epsilon\}$ definiert. Wenn $A \subseteq \Sigma^*$ und $B \subseteq \Sigma^*$ Mengen von Wörtern über Σ sind, so wird die Konkatenation der beiden Mengen als $AB := \{ab | a \in A \wedge b \in B\}$ beschrieben.

Eine *Sprache* ist eine beliebige Menge von Wörtern über einem Alphabet.

2.1.2 Chomsky-Grammatiken

Eine formale Chomsky-Grammatik ist:

- Ein Quadrupel $\mathbf{G} = (\mathbf{V}, \Sigma, \mathbf{S}, \mathbf{P})$ wobei
  - $V = N \cup \Sigma$. V wird als Vokabular der Grammatik bezeichnet, welches aus den Nichtterminalen $N = V - \Sigma$ und dem Alphabet Σ besteht. Die Elemente von Σ werden auch als Terminale bezeichnet.

  - $S \in N$. S ist ein ausgezeichnetes Nichtterminal und stellt das Startsymbol der Grammatik dar.
  - $P \subseteq V^+ \times V^*$. $(u, v) \in P, u \in V^+, v \in V^*$ sind die Produktionen der Grammatik.

Durch eine formale Grammatik G wird eine formale Sprache $L(G)$ definiert:

- $L(G) := \{w \in \Sigma^* | S \rightarrow^*_G w\}$ die Menge von Wörtern, die vom Startsymbol abgeleitet werden können.
  - $w \in L(G)$ wird auch als Satz der Sprache bezeichnet
  - Wenn $S \stackrel{*}{\rightarrow} v$ mit $v \in V^*$ dann ist v eine Satzform

Die formalen Grammatiken werden in verschiedene Typen unterteilt:

- **Typ-0, Satzgliederungsgrammatik:** ist äquivalent mit der beschriebenen formalen Grammatik.
- **Typ-1, kontextsensitiv:** G wie Typ-0 Grammatik mit der Einschränkung dass $\forall(u, v) \in P$ gilt $|u| \leq |v|$.
- **Typ-2, kontexfrei:** G formale Grammatik mit $\forall(u, v) \in P$, mit $u \in N$,$v \in V^*$.
- **Typ-3, einseitig linear:** G formale Grammatik heisst regulär, wenn
  - **rechtslinear:** $\forall(u, v) \in P$, $u \in N, v \in (\Sigma^* \cup \Sigma^* N)$
  - **linkslinear:** $\forall(u, v) \in P$, $u \in N, v \in (\Sigma^* \cup N\Sigma^* +)$

In Programmiersprachen kommen in der Regel kontextfreie Grammatiken zum Einsatz. Diese werden auch in diesem Framework verwendet. Die folgende Grammatik G, ist ein Beispiel für eine kontextfreie Grammatik:

$$G = (\{E, T, +, -, *, id, number, (,)\}, \{+, -, *, id, number, (,)\}, E, P)$$

Wobei P aus folgenden Produktionen besteht:

$$\begin{aligned}
&E \rightarrow T + E \\
&E \rightarrow T - E \\
&E \rightarrow T \\
&T \rightarrow F * T \\
&T \rightarrow F/T \\
&T \rightarrow F \\
&F \rightarrow id \\
&F \rightarrow number \\
&F \rightarrow (E)
\end{aligned} \tag{2.1}$$

Im folgenden werden die einzelnen Bestandteile einer Grammatik näher betrachtet.

2.1.3 Terminal

Bevor ein Quelltext analysiert werden kann, muss er durch eine lexikalische Analyse in die elementaren Symbole der Sprache, die Terminale, unterteilt werden.

Beschreibung der Terminale

Wenn der Quelltext als Folge von Wörtern $w \in \Sigma^*$ über einem Alphabet Σ betrachtet wird, definiert man für jedes Terminal t der Grammatik einen regulären Ausdruck r_t, welcher die Menge von Wörtern definiert, die als Terminal t interpretiert werden sollen.

- Sei Σ ein Alphabet. Eine **reguläre Menge** ist:
  - ϕ ist ein regulärer Ausdruck zur Bezeichnung der leeren Menge $\emptyset$
  - ϵ ist ein regulärer Ausdruck zur Bezeichnung der Menge $\{\epsilon\}$
  - a ist ein regulärer Ausdruck zur Bezeichnung der Menge $\{a\}$ für alle $a \in \Sigma$.
  - Sind r und s reguläre Ausdrücke zur Bezeichnung der Wortmengen R und S so sind auch $(r|s)$,(rs) und (r^*) reguläre Ausdrücke zur Bezeichnung der Wortmengen $R \cup S$, RS und R^*
- Sei $M(r_t)$ die reguläre Menge welche durch den regulären Ausdruck r_t gebildet wird.

Die regulären Ausdrücke beziehungsweise die regulären Mengen für zwei Terminale (id und $+$) der Grammatik 2.1 werden zum Beispiel folgendermassen aufgebaut:

$$
\begin{aligned}
letter &:= (a|b|\ldots|z|A|B|\ldots|Z) \\
digit &:= (0|1|\ldots|9) \\
M(r_{id}) &:= letter(letter|digit)^* \\
M(r_+) &:= +
\end{aligned}
$$

Als nächstes werden die Terminale nach verschiedenen Aspekten klassifiziert.

Klassifikation

Übliche Programmiersprachen ordnen ihre Terminale in folgende Standardklassen ein:

- Schlüsselwörter (wie if, else,char).
- Bezeichner (wie myNumber, aInteger).
- Operatoren (wie +, -).
- Symbole (wie ();)
- Literale

ohne Attributwert	**mit Attributwert**
$\mathbf{M(r_t) = 1}$	$\mathbf{M(r_t) > 1}$
Operatoren Symbole	Bezeichner Literale

Tabelle 2.1: Aufteilung der Terminalklassen nach Existenz eines Attributwertes Familie

- Stringkonstanten (wie Hello world!).
- Integer-,Floatkonstanten (wie 3.14159).

Eine Aufteilung der Terminale der Grammatik 2.1 nach dem oben genannten Schema ist in Abbildung 2.1 abgebildet.

Dabei fällt auf, dass gewisse Klassen genau durch ein Terminal, andere durch mehrere Terminale repräsentiert werden. Diese Eigenschaft ist mit zwei weiteren Eigenheiten verknüpft:

- **Wertattribut:** Es gibt Terminalklassen welche ein Wertattribut definieren. Die Bezeichnerklasse, als Beispiel, definiert üblicherweise ein Stringattribut, welches den Namen des Bezeichners repräsentiert. Daneben gibt es aber auch Terminalklassen, welche kein Wertattribut benötigen, wie zum Beispiel Schlüsselwörter oder Operatoren.
- **Mächtigkeit der regulären Mengen:** Ob eine Terminalklasse ein Wertattribut benötigt, kann meist über die Analyse der Mächtigkeit der regulären Mengen der Terminale dieser Klasse entschieden werden. Wenn für ein t $M(r_t) > 1$ ist, heisst das, dass mehrere $w \in \Sigma^*$ auf dasselbe Terminal t abgebildet werden. Damit diese Abbildung eindeutig umkehrbar ist, wird das Terminal mit einem Wertattribut erweitert, so das die verschieden $w \in \Sigma^*$ auf unterschiedliche Terminale t_w abgebildet werden.

Wenn die in Abbildung 2.1 identifizierten Terminalklassen, nach den oben beschrieben Eigenschaften weiter unterteilt werden, ergibt dies eine Aufteilung, wie sie in Tabelle 2.1 dargestellt ist.

Ein Beispiel, wie ein Quelltext in eine Folge von Terminalen umgewandelt werden kann, ist hier angegeben.

$$5.65 + A/B \Rightarrow number_{5.65} + id_A/id_B$$

2.1.4 Nichtterminal

Nichtterminale sind syntaktische Variablen, die Mengen von Strings von Grammatiksymbolen bezeichnen. Die Nichtterminale stehen für String-Mengen, welche die Definition

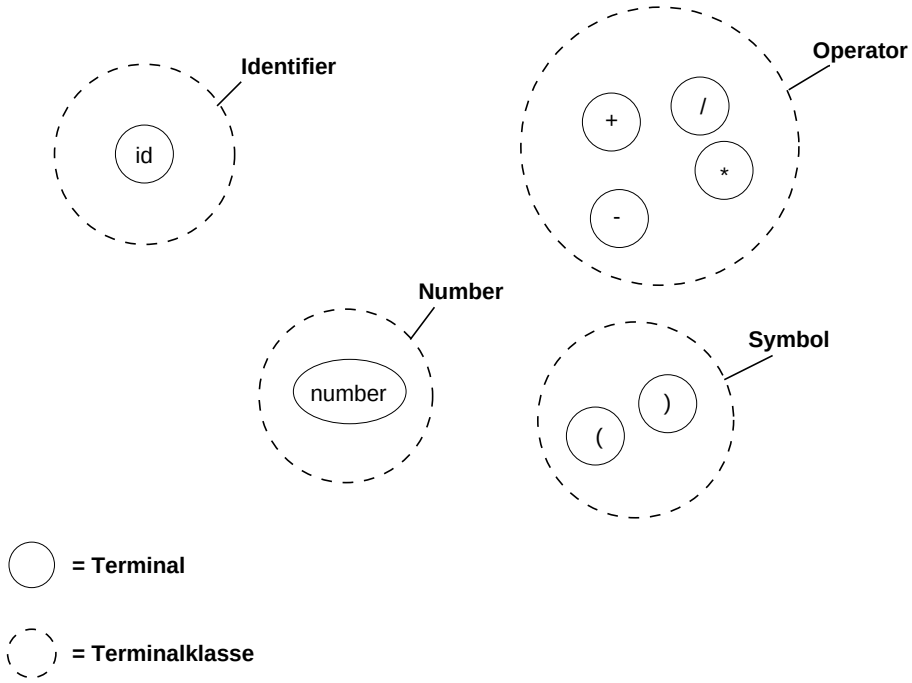

Abbildung 2.1: Terminale und Terminalklassen der Grammatik 2.1

der von der Grammatik erzeugten Sprache vereinfachen. Darüber hinaus prägen sie der Sprache eine hierarchische Struktur auf, die für Syntaxanalyse und Übersetzung wichtig sind. Jedem Nichtterminal sind eine oder mehrere Produktionen zugeordnet, die bestimmen, wie Satzformen zum jeweiligen Nichtterminal reduziert werden. Ziel der Syntaxanalyse ist es, den gegebenen Quelltext darauf zu überprüfen, ob er einen Satz der vorgegebenen Grammatik darstellt. Dies ist der Fall, wenn es gelingt, den Quelltext durch Anwendung von Produktionen auf ein einziges ausgezeichnetes Nichtterminal, das Startsymbol, zu reduzieren. Neben den Produktionen, die einem Nichtterminal zugeordnet sind, gibt es noch weitere syntaktische Informationen, die zu einem Nichtterminal gehören.

FIRST-Menge : Sie beschreibt die Menge aller Terminale, die in einer beliebigen Herleitung des jeweiligen Nichtterminals am Anfang der hergeleiteten Satzform stehen können. Die FIRST-Menge ist auch für Terminale definiert. Sie wird als die Menge, welche genau dieses Terminal enthält, definiert.

$$X \in N \quad FIRST(X) = \{x | X \rightarrow^*_G xw \wedge x \in \Sigma \wedge w \in V^*\}$$

$$x \in \Sigma \quad FIRST(x) = \{x\}$$

Die FIRST-Mengen der Nichtterminale E,F und T der Beispielgrammatik 2.1 sind alle gleich.

$$FIRST(E) = FIRST(T) = FIRST(F) = \{id, number, (\}$$

$$FIRST(id) = \{id\}$$

FOLLOW-Menge : Sie beschreibt die Menge aller Terminale, welche in einer beliebigen Satzform direkt auf das jeweilige Nichtterminal folgen können. Sei $X \in N$ dann ist

$$FOLLOW(X) = \{x | S \overset{*}{\rightarrow}_G uXxw \wedge u, w \in V^* \wedge x \in \Sigma\}$$

Die FOLLOW-Menge von E,T und F sind

$$FOLLOW(E) = \{), \$\}$$

$$FOLLOW(T) = \{+, -,), \$\}$$

$$FOLLOW(F) = \{*, /, +, -,), \$\}$$

wobei das $-Zeichen für das Abschlusszeichen eines Satzes steht. Die FOLLOW-Menge ist nur für Nichtterminale definiert.

Diese beiden Mengen werden bei Berechnung der Tabelle eines LL- beziehungsweise eines LR-Parsers (Kapitel 4) benötigt.

2.1.5 Produktion

Eine kontextfreie Produktion p definiert eine Ableitungsregel, die beschreibt, wie eine bestimmte Satzform der Grammatik auf ein bestimmtes Nichtterminal X reduziert werden kann.

$$p \in P \quad p : X \rightarrow_G w, X \in N \wedge w \in V^*$$

Das Nichtterminal X wird dabei als linke Seite der Produktion bezeichnet und die Satzform w als rechte Seite.

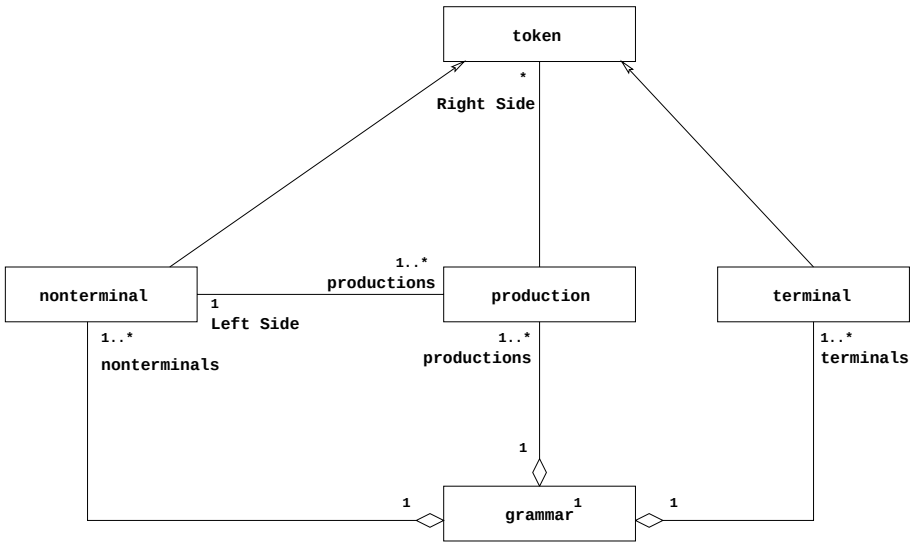

Abbildung 2.2: Klassenhierarchie um die Grammatikrepräsentation im Framework

2.2 Implementation

Zu Beginn wird hier die Implemtentation in einem Überblick beschrieben, bevor die konkreten Klassen vorgestellt werden. Die Konzepte, welche hinter dem in diesem Framework gewählten Ansatz stehen, sind die folgenden:

- Für Terminale, Nichtterminale und Produktionen wird je eine Klasse definiert (terminal, nonterminal, production), wobei terminal als Basisklasse für spezielle Terminalklassen wie *Identifer* oder *Keyword* dient. Zusätzlich wird eine weitere Basisklasse token eingeführt, welche eine Verallgemeinerung der Klassen terminal und nonterminal darstellt.

- Ein Objekt der Klasse grammar stellt eine konkrete kontextfreie Grammatik dar. Für jedes Terminal und jedes Nichtterminal der Grammatik wird ein Objekt der entsprechenden Klasse terminal beziehungsweise nonterminal erzeugt. Für jede Produktion wird ein Produktionsobjekt erstellt, wobei sich dieses wiederum aus Objekten der oben genannten Klassen zusammensetzt. Diese Terminal-, Nichtterminal- und Produktionsobjekte werden dem Grammatikobjekt zugewiesen und repräsentiern auf diese Weise eine Grammatik.

- Die Grammatikklasse definiert Methoden zur Eliminierung von Linksrekursion und für die Berechnung von FIRST- und FOLLOW-Mengen. Bei der Eliminerung von Linksrekursion[1] werden Produktionsobjekte umgeformt, sowie neue Nichterminalobjekte eingeführt. Bei der Berechnung der FIRST- und FOLLOW-Mengen, wer-

[1] Der Algorithmus ist auf Seite 23 beschrieben

den die entsprechenden Mengen, welche als Attribute der Klassen nonterminal auftreten, berechnet.

- Ein, in dieser Weise erzeugtes, Grammatikobjekt wird verwendet, um den Scanner[2] beziehungsweise den Parser[3] zu initalisieren.

- Terminalobjekte werden nicht nur zur Repräsentation der Grammatik verwendet. Sie werden auch vom Scanner erzeugt, jedesmal wenn dieser ein Terminal erkennt. Die Terminalobjekte, welche am Anfang für die Bildung der Grammatik benötigt werden, spielen auch bei der Erzeugung von konkreten Terminalobjekten im Scanner eine Rolle. Die zu Beginn erstellten Terminalobjekte dienen dem Scanner als sogenannte Prototypinstanzen. Jedesmal wenn der Scanner ein Terminal erkennt, wählt er die entsprechende Prototypinstanz, kopiert diese und initialisiert sie mit der Position des erkannten Terminals im Quelltext und dem erkannten Quelltextstring. Diese Art der Objekterzeugung wird in [GHJV95] als *Prototyp-Pattern* beschrieben. Der genaue Grund für diese Wahl wird in Kapitel 3 erläutert.

Im folgenden werden die oben beschrieben Klassen näher erläutert.

2.2.1 Klasse token

Die Klasse token ist eine abstrakte Basisklasse für die im folgenden beschriebenen Klassen terminal und nonterminal. Der Hauptgrund für die Einführung dieser Klasse ist, dass die rechte Seite einer Produktion als Liste von Objekten der Klasse token beschrieben werden kann.

Im folgenden sind die Attribute und Methoden kurz beschrieben:

- **Attribute :**
  - typedef set<terminal*,less<terminal*>> terminalSet. Definition einer Terminalmenge.
  - String name. Jedes Terminal beziehungsweise Nichtterminal hat einen Namen. Der Name eines Terminals darf aber nicht mit dem Wert eines Terminals verwechselt werden.
  - terminalSet* FIRST. Die FIRST-Menge wird nur alloziert, wenn sie auch wirklich benötigt wird.

- **Methoden :**
  - virtual terminalSet calculateFIRST() = 0. Eine abstrakte Methode, welche von der Terminalklasse und der Nonterminalklasse überschrieben wird.

[2] siehe Kapitel 3 auf Seite 25
[3] siehe Kapitel 4 auf Seite 34

2.2.2 Klasse terminal

Die Klasse terminal ist die abstrakte Basisklasse für alle konkreten Terminalklassen. Ein Teil der im folgenden besprochenen Implementation ist in Abbildung 2.3 dargestellt.

- **Attribute:**
  - int x,y. Beschreibt die Zeile und die Spalte, in welcher das Terminal im Quelltext auftritt.
  - int precedence. Attribute für Priorität und Assoziativität. Diese werden benötigt, um Konflikte, welche bei der Syntaxanalyse auftreten, zu lösen[4].
  - enum Associativity {NON,LEFT,RIGHT}. Typ der Assoziativität eines Terminals.
  - Associativity associativity.
- **Methoden:**
  - virtual terminal* clone(String scText,int x,int y) const = 0. Diese Methode ist Bestandteil des Prototyp-Patterns und muss von jeder konkreten Terminalklasse implementiert werden.
  - virtual stateMachine* createStateMachine() const = 0. In Kapitel 3 wird erläutert, wie ein Terminal von einem deterministischen endlichen Automaten erkannt wird. Jede Terminalklasse muss diese Methode implementieren, welche einen entsprechenden endlichen Automaten erzeugt.
- **Konkrete Terminalklassen:** Zusätzlich zu dieser Basisklasse müssen je nach Sprache weitere Terminalklassen definiert werden. Dabei muss jedoch nicht zwingend die Aufteilung gewählt werden, wie sie im theoretischen Teil besprochen wurde. Vorallem die Terminalklassen, welche kein Wertattribut besitzen, können häufig über ein und dieselbe Klasse repräsentiert werden. Das Framework definiert einige der oben erwähnten Standardklassen. Dazu gehören keywordtok für Schlüsselwörter, stringtok für Stringkonstanten, identtok für Bezeichner und numbertok für Floatkonstanten. Wie oben erwähnt, kann die Klasse keywordtok auch für Operatoren und Symbole verwendet werden, da diese ebenfalls kein Wertattribut benötigen. Zusätzlich dazu werden für spezielle Terminale weitere Terminalklassen definiert. Dazu gehören epsilontok für das ϵ-Terminal, errortok für ein Error-Terminal, welches vom Scanner erzeugt wird, wenn er ein ungülties Wort liest und ein endtoken , welches ebenfalls vom Scanner generiert wird, wenn die Eingabe fertig gelesen wurde.

2.2.3 Klasse nonterminal

Die Nichtterminale werden in der Klasse nonterminal implementiert. Im folgenden werden die Attribute und Methoden dieser Klasse beschrieben.

[4] siehe Kapitel 5 auf Seite 47

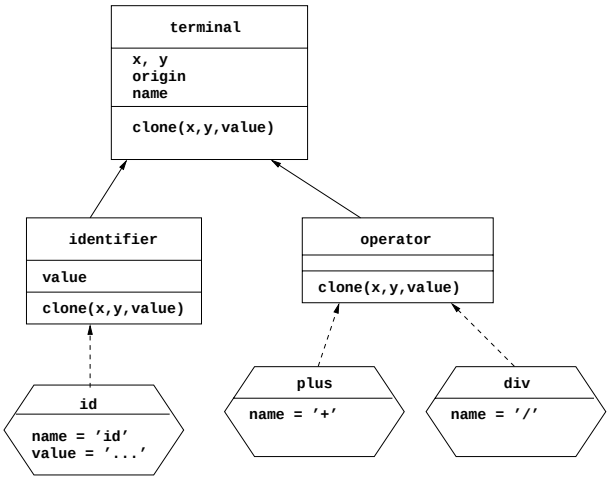

Abbildung 2.3: Die Terminal Klassenhierarchie für einen Teil der Grammatik 2.1

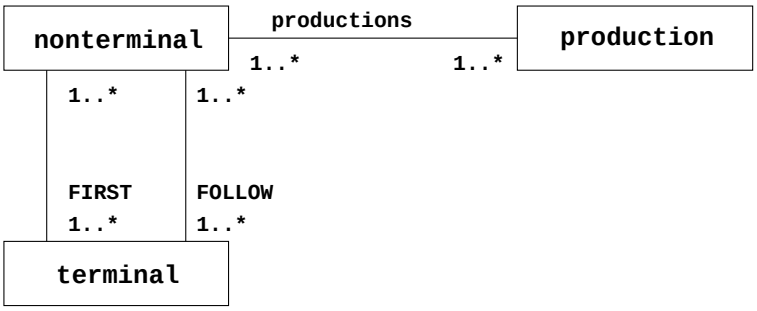

Abbildung 2.4: Die Klasse nonterminal und ihre Relationen

- **Attribute:**
  - list<production*>* productions. Repräsentiert die Liste der Produktionen, welche mit diesem Nichtterminal assoziert sind, das heisst auf deren linken Seite das entsprechende Nichtterminal erscheint.
  - terminalSet* FOLLOW. Die FOLLOW-Menge dieses Nichtterminals
- **Methoden:**
  - terminalSet* calculateFIRST(). Implementiert die Methode zur Berechnung der FIRST-Menge dieses Nichtterminals. Diese Berechnung erfolgt inkrementell, das heisst, dass diese Methode unter Umständen mehrmals aufgerufen werden muss, bis die FIRST-Menge entgültig berechnet ist. Die Berechnung wird vom Grammatikobjekt aus gesteuert.
  - int addFollowTerminal(terminal* t). Die Berechnung der FOLLOW-Menge geschieht im Grammatikobjekt. Diese Methode dient nur dazu, der Menge ein Element hinzuzufügen. Der Rückgabewert der Methode signalisiert, ob das Element bereits vorhanden war.

2.2.4 **Klasse** production

Die Produktionen werden in einer Klasse production implementiert. Eine production wird von list<token*> abgeleitet, welche die rechte Seite der Produktion darstellt. Ein Objekt dieser Klasse enthält auch die Informationen darüber, wie ein Knoten im abstrakten Syntaxbaum erstellt werden muss, wenn die entsprechende Produktion reduziert wird. Die entsprechenden Attribute und Methoden werden hier auch aufgeführt, die eigentliche Erklärung folgt bei der Besprechung des abstrakten Syntaxbaums in Kapitel 7.

- **Attribute:**
  - nonterminal * Nonterminal. Die linke Seite der Produktion.
  - grammar* Grammar. Ein Verweis auf die zugrundeliegende Grammatik.
  - NodeCreationFnct NCF. Ein Zeiger auf eine statische Methode in einer Klasse, welche einen Knotes den abstrakten Syntaxbaums darstellt. Diese Methode wird aufgerufen, um einen entsprechenden Knoten zu erzeugen.
  - String createName. Jede dieser Methoden hat einen Namen, über den in einem Verzeichnis die entsprechende reale Methode ermittelt werden kann.
  - list<int>* childNodes. Die Parameterliste dieser Methode. Sie definiert, welche Elemente der rechten Seite der Produktion dem neu erzeugten Syntaxbaumknoten als Kinder übergeben werden sollen.
- **Methoden:**
  - set <terminal*>* calculateFIRST(). Berechnet die FIRST-Menge der rechten Seite der Produktion. Diese Methode wird vom nonterminal-Objekt der linken Seite benötigt, um seine eigene FIRST-Menge zu berechnen.

– node* createNode(). Wird vom Parser aufgerufen, wenn die Produktion reduziert wird, um einen neuen Knoten im Syntaxbaum zu erzeugen[5].

2.2.5 Klasse grammar

Die Grammatik wird nun in einer Klasse grammar implementiert (Abbildung 2.2).

- **Attribute :**
  - list<production*> productions. Die Produktionen der Grammatik.
  - map<String,terminal*> terminals. Die Terminale der Grammatik.
  - map<String,nonterminal*> nonTerminals. Die Nichtterminale der Grammatik.
  - nonterminal* startSymbol. Das Startsymbol der Grammatik.
  - registry* Registry. Verzeichnis, in welchem, über den Methodennamen, die Erzeugungsmethode eines bestimmten Syntaxbaumknotens gesucht werden kann[6]
- **Methoden :**
  - calculateFIRST(). Methode zur Berechnung der FIRST-Mengen der Nichtterminale.
  - calculateFOLLOW(). Methode zur Berechnung der FOLLOW-Mengen der Nichtterminale.
  - addTerminal(terminal* t). Fügt der Grammatik ein Terminal hinzu.
  - addNonterminal(nonterminal* n). Fügt der Grammatik ein Nichtterminal hinzu.
  - terminal* findTerminal(const String& name). Liefert das entsprechende Terminal zurück.
  - nonterminal* findNonTerminal(const String& name). Liefert das entsprechende Nichtterminal zurück.
  - NodeCreationFnct getMethod(const String& ncf) const. Liefert den entsprechenden Methodenzeiger aus der Registry.
  - createProduction(String& prodString). Vereinfacht die Konstruktion von Produktionsobjekten. Erwartet als Eingabe einen String wie zum Beispiel 'E -> E + E' und konstruiert daraus ein Objekt der Klasse production. Dieses Objekt wird in der Liste productions eingetragen.
  - elimDirectLeftRecursion(). Eliminiert direkte Linksrekursion aus der Grammatik.
  - elimLeftRecursion(). Eliminiert Linksrekursion aus der Grammatik.

[5] siehe Abschnitt 4.4 auf Seite 46
[6] Siehe Kapitel 7 auf Seite 62

Berechnung der FIRST-Mengen

Im folgenden sind konstruktive Anweisungen angegeben, wie die FIRST-Menge eines Nichtterminals berechnet werden kann. Da die Nichtterminale in den meisten Fällen gegenseitig voneinader abhängen, müssen diese Anweisungen iterativ wiederholt werden, bis sich die entsprechende FIRST-Menge nicht mehr verändert.

Sei $X \in N$ mit $X \to w_1, X \to w_2, \ldots, X \to w_n$ mit $w_i \in V^*$ die Menge aller Produktionen welche X als linke Seite haben.

$$\text{so ist } FIRST(X) = \bigcup_{1 \leq i \leq n} FIRST(w_i)$$

Nun muss angegeben werden, wie $FIRST(w_i)$ berechnet werden soll. Falls $w = \epsilon$ dann ist $FIRST(w) = \{\epsilon\}$. Sonst sei $w = X_1 X_2 \ldots X_m$ mit $X_j \in V$. Dann werden zwei Fälle unterschieden:

- **1. Fall:**
  $\forall j, 1 \leq j \leq m, \epsilon \in FIRST(X_j) \Rightarrow FIRST(w) = (\bigcup_{1 \leq j \leq m} FIRST(X_j)) \bigcup \{\epsilon\}$
- **2. Fall:**
  $k := min\{l | \epsilon \notin FIRST(X_l)\} \Rightarrow FIRST(w) = \bigcup_{1 \leq l \leq m} FIRST(X_l)$

Die Klassen grammar, production und nonterminal implementieren alle eine Methode calculateFIRST(). Jede dieser Methoden übernimmt ihren Teil an der oben beschriebenen Berechnung. Anhand einer einfachen Grammatik soll dies illustriert werden.

$$A \to AB$$
$$A \to \epsilon$$
$$A \to a$$
$$B \to BC$$
$$B \to \epsilon$$
$$B \to b$$
$$C \to c$$
$$C \to d$$

Es wird nun die Berechnung, wie sie in Tabelle 2.2 dargestellt ist, beschrieben. Zuerst wird die FIRST-Menge von A berechnet. Schon bei der ersten Produktion ergibt sich ein Problem. Da die Produktion in A links-rekursiv ist, kann die FIRST-Menge dieser Produktion,das heisst $FIRST(AB)$, nicht ermittelt werden. Somit ist $FIRST(A)$, nachdem die restlichen trivialen Produktionen von A verarbeitet worden sind, $\{a, \epsilon\}$. Die Berechnung

Nichtterminal	**1. Durchgang**	**2. Durchgang**	**3. Durchgang**
A	a, ϵ	a,b,ϵ	a,b,c,d,ϵ
B	b, ϵ	b,c,d,ϵ	b,c,d,ϵ
C	c,d	c,d	c,d

Tabelle 2.2: Entwicklung der FIRST-Mengen

von $FIRST(B)$ verhält sich wie bei A, und $FIRST(C)$ ist trivial. Die Mengen sind nach diesem ersten Durchgang aber noch nicht vollständig. Beim zweiten Durchgang kann nun auch die erste Produktion verarbeitet werden. Das hat zur Folge, dass $FIRST(A)$ um das Terminal b erweitert wird, da $FIRST(AB) = \{a, b, \epsilon\}$. Auch $FIRST(B)$ verändert sich, und $FIRST(C)$ bleibt konstant. Da $FIRST(A)$ von $FIRST(AB)$ abhängt und $FIRST(B)$ sich seit der letzten Berechnung von $FIRST(A)$ verändert hat, ist ein dritter Durchgang nötig. Nach diesem Durchgang verändern sich die Mengen nicht mehr.

Dieses Beispiel zeigt, dass im Normalfall mehrere Berechnungsdurchgänge notwendig sind. Konkret wird dies dadurch gelöst, dass so viele Durchgänge gerechnet werden, bis sich keine Menge mehr verändert.

Berechnung der FOLLOW-Mengen

Nachdem die FIRST-Mengen berechnet wurden, können daraus die FOLLOW-Mengen abgeleitet werden. Dies geschieht indem für jedes Nichtterminal diejenigen Produktionen betrachtet werden, auf deren rechten Seite das Nichtterminal vorkommt. Anhand eines Ausschnitts aus einer Grammatik soll die Berechnung von $FOLLOW(C)$ erläutert werden.

$$A \rightarrow abCDE$$

$$A \rightarrow bcCa$$

$FOLLOW(C)$ berechnet sich aus den FIRST-Mengen, der Symbole welche rechts von C stehen.

Dabei werden zwei Fälle unterschieden:

- **1. Fall:** $\epsilon \notin FIRST(DE)$
  $\Rightarrow FOLLOW(C) = FIRST(DE) \cup FIRST(a)$
- **2. Fall:** $\epsilon \in FIRST(DE)$
  $\Rightarrow FOLLOW(C) = (FIRST(DE) \setminus \{\epsilon\}) \cup FIRST(a) \cup FOLLOW(A)$

Falls in einer dieser FIRST-Mengen ϵ vorkommt, wird dieses entfernt. Dafür kommt die FOLLOW-Menge des Nichtterminals auf der linken Seite der Produktion hinzu.

Eliminierung von Linksrekursion

Man spricht von Links-Rekursion in einer Grammatik, wenn es eine oder mehrere Herleitungen der Form $A \Rightarrow^*_G A\beta$ mit $A \in N, \beta \in V^*$ gibt. Dabei unterscheidet man zwischen direkter Rekursion $N \rightarrow N\beta \in P$ und indirekte Rekursion, welche sich über mehrere Produktionen hinzieht, wie zum Beispiel $B \rightarrow_G A\beta$,$A \rightarrow_G B\delta$ woraus folgt dass $B \Rightarrow^*_G B\beta\delta$.

Aus links-rekusiven Grammatiken können keine Top-Down Parser gebildet werden. Es gibt aber Algorithmen, welche die Links-Rekursion aus einer Grammatik entfernen, ohne die von der Grammatik erzeugte Sprache zu verändern. Diese Algorithmen[ASU92] sollen im folgenden betrachtet werden.

Direkte Links-Rekursion lässt sich mit der folgenden Technik eliminieren. Dazu wird ein Nichtterminal, welches links-rekursive Produktionen hat, gewählt und dessen Produktionen nach folgendem Muster geordnet

$$A \rightarrow_G A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$$

wobei α_i und β_j beliebige Satzformen sind und kein β_j mit einem A beginnen darf. Dann ersetzt man die Produktionen durch

$$A \rightarrow_G \beta_1 A' |\beta_2 A' |\ldots|\beta_n A'$$

$$A' \rightarrow_G \alpha_1 A' |\alpha_2 A' |\ldots|\alpha_m A' |\epsilon$$

dadurch ist die Links-Rekursion beseitigt, und das Nichtterminal A erzeugt immer noch dieselbe Menge von Sätzen.

Indirekte Links-Rekursion kann so jedoch noch nicht eliminiert werden. Ein Beispiel für indirekte Rekursion stellt folgende Grammatik dar:

$$S \rightarrow Aa|b$$

$$A \rightarrow Ac|Sd$$

$$S \Rightarrow_G Aa \Rightarrow_G Sda$$

Um solche Grammatiken von direkter und indirekter Rekursion zu befreien wendet man folgenden Algorithmus an:

- Sei $G = (N, T, S, P)$ kontextfreie Grammatik, mit $N = \{A_1, A_2 \ldots A_n\}$

```
for i := 1 to n do
  for j := 1 to i-1 do begin
    ersetze jede Produktion der Form A_i →_G A_j γ durch die
      Produktionen A_i →_G δ_1γ|δ_2γ|...|δ_kγ,
      wobei A_j →_G δ_1|δ_2|...|δ_k alle aktuellen
      A_j-Produktionen sind;
    eliminiere direkte Links-Rekursion in den A_i-Produktionen
  end
```

Diese Algorithmen zur Eliminierung von Links-Rekursion sind in der Klasse grammar implementiert und können dazu verwendet werden, die Grammatik umzuformen. Um einen vollständigen Satz von Umformungsroutinen zu besitzen, müsste zusätzlich noch eine Methode zur Linksfaktorisierung[7] implementiert werden, was im Rahmen dieser Arbeit nicht getan wurde.

[7] Zwei Produktionen wie $A \rightarrow abc$ und $A \rightarrow abd$ haben den sogenannten *Common Left-Factor* ab, dass heisst es kann erst nach Lesen von ab bestimmt werden welche Produktion zur Anwendung kommt. Solche *Common Left-Factors* können durch Linksfaktorisierung eliminiert werden.

Kapitel 3

Scanner

Der Scanner liest den Quelltext und extrahiert daraus, mit Hilfe von regulären Ausdrücken, Terminale, welche er an den Parser weitergibt. In Abschnitt 2.1.3 wurde diesbezüglich schon einiges vorweggenommen. So wurde bereits erwähnt, dass die Terminale durch reguläre Ausdrücke definiert werden und das der Scanner als Ausgabe Terminalobjekte erzeugen soll. Darum wird es in diesem Abschnitt nur noch darum gehen, wie die regulären Ausdrücke im Scanner implementiert werden.

3.1 Definition eines deterministischen endlichen Automaten

Jede Sprache, welche durch eine reguläre Grammatik beziehungsweise einen regulären Ausdruck erzeugt wird, kann durch einen deterministischen endlichen Automaten (DEA) erkannt werden[ASU92]. Ein DEA ist ein 5-Tupel[PP92]:

- $A = (\Sigma, Q, \Delta, q_0, F)$ wobei
  - Σ das Eingabealphabet darstellt.
  - Q eine endliche Menge von Zuständen ist, und $q_0 \in Q$ den Startzustand darstellt.
  - $F \subseteq Q$ eine Menge von Endzuständen ist.
  - und Δ eine partielle Übergangsfunktion $\Delta : Q \times \Sigma \rightarrow Q$ bezeichnet.

Der DEA besitzt zudem eine Eingabe aus welcher er, beginnend im Startzustand q_0, Zeichen für Zeichen liest und dabei, mit Hilfe der Übergangsfunktion Δ, seinen Zustand ändert. Als eine *Konfiguration* wird ein Paar (q, w) bezeichnet, wobei $q \in Q$ den aktuellen Zustand des DEA's und $w \in \Sigma^*$ die noch verbleibende Eingabe repräsentiert. Ein Automat in der Konfiguration (q, ϵ), mit $q \in F$, hält an und signalisiert, dass er die gelesene Eingabe akzeptiert. Falls $q \notin F$ ist, weist er die Eingabe, als nicht zur Sprache gehörend, zurück. Falls er in eine Konfiguration gelangt, aus welcher kein Übergang definiert ist,blockiert er und weist die Eingabe ebenfalls zurück.

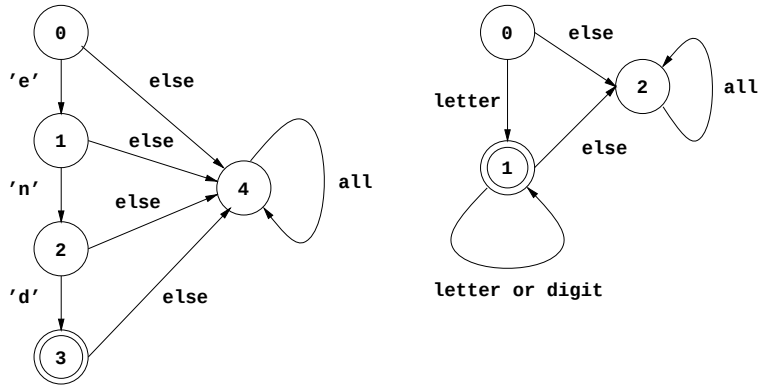

Abbildung 3.1: Zwei erweiterte DEA's. Der Erste erkennt das Schlüsselwort *end*. Akzeptierender Zustand ist 3, Errorzustand ist 4. Der Zweite erkennt Bezeichner. Akzeptierender Zustand 1, Errorzustand ist 2.

3.2 Design des Scanners

Um einen Scanner für die Erkennung der Terminale einer Grammatik G zu bauen, kann nun auf die im Abschnitt 2.1.3 beschriebenen regulären Ausdrücke zurückgegriffen werden. Für jedes Terminal t wird ein regulärer Ausdruck r_t definiert, welcher die Menge der Wörter über einem Alphabet Σ definiert, die auf t abgebildet werden sollen. Für jeden regulären Ausdruck r_t kann anschliessend ein DEA erstellt werden, welcher die durch r_t erzeugte Sprache erkennt.

Die so erstellten DEA's werden zu einem Scanner zusammengefasst. Um ein Terminal zu erkennen, wird die Eingabe von allen DEA's parallel gelesen und anhand ihres Verhaltens entschieden, was für ein Terminal erkannt wurde.

3.3 Implementation

3.3.1 Klasse stateMachine

Ein DEA wird in der Klasse stateMachine implementiert. Beispiele für DEA's sind in Abbildung 3.1 und Tabelle 3.1 dargestellt. Im folgenden ist die Implementation, nach Attributen und verschiedenen Methodentypen gegliedert, beschrieben.

- **Attribute :** Die Zustände werden gegenüber der formalen Definition erweitert. So kann ein Zustand nicht nur ein Endzustand sein sondern auch ein Errorzustand.
  - vector<state> states. Bezeichnet die Zustände des DEA's, wobei ein Zustand state zwei Attribute hat. Das erste Attribut ist err und zeigt an, ob es sich um einen Errorzustand handelt. Das zweite Attribute ist prototyp und zeigt im Fall, dass es sich um einen Endzustand handelt, auf die Prototypinstanz des erkannten Terminals.

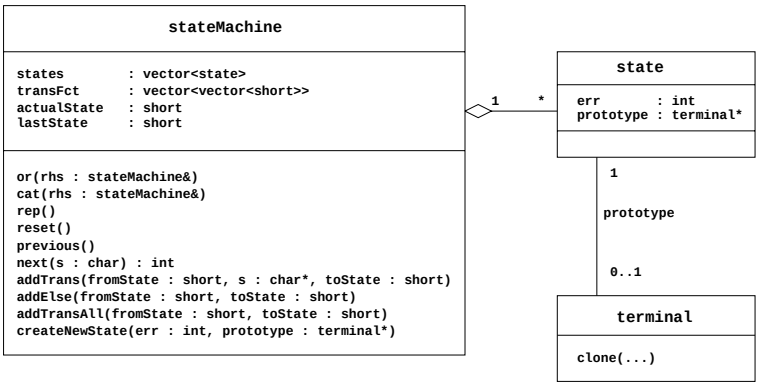

Abbildung 3.2: Klassendiagramm der Klasse stateMachine, welche einen DEA implementiert. Die Klasse terminal ist stark vereinfacht dargestellt.

DEA für Schlüsselwort *end*		DEA für Bezeichner		Eingabesymbol
Zustände		Zustände		
aktuell	letzter	aktuell	letzter	
0	-	0	-	”e”
1	0	1	0	”n”
2	1	1	1	”d”
3	2	1	1	”e”
4	3	1	1	” ”
4	4	2	1	

Tabelle 3.1: Erkennungsablauf für die beiden DEA’s aus Abbildung 3.1. Der Eingabestring ist ”ende ” und wird beiden DEA’s parallel Zeichen für Zeichen übergeben, bis sich beide im Errorzustand befinden. Anschliessend wird der DEA, der als letzter einen akzeptierenden Zustand hatte ausgewählt. In diesem Fall der DEA für Bezeichner.

  - int actState. Aktueller Zustand des DEA's.
  - int lastState. Letzter Zustand des DEA's.
  - vector<vector<short>> transFnct. Implementiert die Übergangsfunktion Δ des DEA's in Form einer Automatentabelle.

- **Konstruktionsmethoden :** Um einen DEA zu erstellen, gibt es verschiedene Konstruktionsmethoden.

  - short createNewState(int err,const terminal* prototype). Erzeugt einen neuen Zustand. Wenn err als 1 übergeben wird, wird ein Errorzustand erzeugt und wenn prototyp ungleich NULL übergeben wird, ein akzeptierender Zustand. Als Rückgabewert wird die Zustandsnummer, welche für die nächsten Methoden benötigt wird, zurückgegeben.
  - addTrans(fromState : short,const char *s,toState : short). Richtet für alle Zeichen in s einen Zustandsübergang von fromState nach toState ein.
  - addTransAll(fromState : short,toState : short) Für alle möglichen Eingabesymbole wird ein Zustandsübergang erstellt.
  - addElse(errorState : short). Diese Methode muss am Schluss der Konstruktion aufgerufen werden, um alle möglichen verbleibenden Zustandsübergänge auf den Zustand errorState abzubilden.

- **Betriebsmethoden :** Um den DEA zu betreiben, werden vier Methoden angeboten.

  - reset(). Versetzt den DEA in seinen Startzustand.
  - int next(char s). Überführt den DEA in den nächsten Zustand aufgrund des aktuellen Zustands und des Eingabesymbols s. Falls der neue Zustand ein Errorzustand ist, wird 0 zurückgegeben, ansonsten 1.
  - previous(). Versetzt den DEA in den letzten Zustand,welcher vor dem aktuellen besucht wurde.
  - const terminal* accepted() const. Falls der DEA in einem akzeptierenden Zustand ist, wird die Prototypinstanz des erkannten Terminals zurückgegeben.

- **Operationen :** Es werden die drei Standardoperationen für reguläre Ausdrücke implementiert. Die or-Methode wird vom Scanner benötigt, um mehrere DEA's zu einem DEA zu verknüpfen.

  - or(const stateMachine& rhs). Äquivalent zu $(r|s)$.
  - cat(const stateMachine& rhs). Analog (rs).
  - rep(). Analog (r^*).

Ein Beispiel für den Erkennungsablauf ist in Tabelle 3.1 gegeben. Im nachfolgenden Quelltextausschnitt wird demonstriert, wie eine stateMachine für Bezeichner konstruiert werden kann.

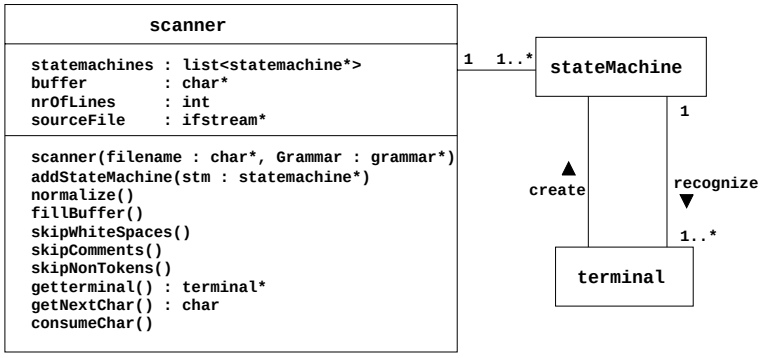

Abbildung 3.3: Die Klasse scanner und ihre Abhängigkeiten.

```
class stmIdentifers : public stateMachine {
  stmIdentifiers(identtok* id, char* firstchar, char* rest);
}

stmIdentifiers::stmIdentifiers(identtok* id,char* firstchar,char* rest)
{
  unsigned short s1,s2,s3;

  s1 = createNewState(0,NULL);
  s2 = createNewState(1,NULL); // error state
  s3 = createNewState(0,id); // accepting state
  addTrans(s1,firstchar,s3);
  addTrans(s3,rest,s3);
  addElse(s2); // map all other transitions to error
}
```

3.3.2 Klasse scanner

Der Scanner wird durch die Klasse scanner (Abbildung 3.3) implementiert.

- **Attribute :**
  - list<stateMachine*> stateMachines. Eine Liste von DEA's, von welchen jeder DEA eines oder mehrere Terminale erkennt.
  - char* buffer. Ein Buffer für die Eingabedatei.
  - ifstream* sourceFile. Der Eingabestream, der mit der Eingabedatei verbunden wird.
  - int nrOfLines. Die aktuelle Zeilennumer in der Eingabedatei.
- **Konstruktionsmethoden :**

  - scanner(char* filename,grammar* Grammar). Über das Grammatikobjekt Grammar wird der Scanner initialisiert. Dieser Vorgang ist im nächsten Abschnitt genauer erläutert.
  - addStateMachine(stateMachine* stm). Fügt der Liste stateMachines einen neuen DEA hinzu.
  - normalize().Verknüpft alle DEA's in der stateMachines Liste zu einem einzigen DEA. Dazu wird die or-Methode von stateMachine verwendet.

- **Betriebsmethoden :** Die nachstenden Methoden sind bis auf getterminal() interne Methode. Sie werden hier nur beschrieben, da sie für getterminal() verwendet werden.

  - fillBuffer(). Füllt buffer mit Zeichen aus der Eingabedatei.
  - char getNextChar(). Holt das aktuelle Zeichen aus dem Buffer..
  - consumeChar(). Konsumiert beziehungsweise löscht das aktuelle Zeichen im Eingabebuffer
  - skipWhiteSpaces(). Funktion, die alle anstehenden Leerzeichen liest. Kann für einen speziellen Scanner überschrieben werden.
  - skipComments(). Funktion, die Kommentare überliest. Muss für speziellen Scanner überschrieben werden.
  - skipNonTokens(). Ruft die beiden letztgenannten Funktionen solange auf, bis der Anfang des nächsten Tokens erkannt wird.
  - terminal* getterminal(). Der Scanner liest die Eingabe solange, bis er ein Terminal erkennt. Das entsprechende Terminalobjekt wird erzeugt und zurückgegeben. Der im folgenden beschriebene Ablauf ist in den Abbildungen 3.4 und 3.5 dargestellt.
    1. *Zurücksetzen der DEA's*. Zuerst werden alle DEA's in der Liste stateMachines über ihre previous Methode in den Anfangszustand versetzt.
    2. *Überlesen von Leerzeichen und Kommentaren*. Anschliessend wird die Methode skipNonTokens() aufgerufen, welche alle Leerzeichen beziehungsweise Kommentare bis zum Anfang des nächsten Terminals liest.
    3. *Solange Eingabezeichen lesen, bis alle DEA's im Errorzustand sind*. Nun wird über getNextChar() das erste Zeichen gelesen. Anhand dieses Zeichens werden die beteiligten DEA's über ihre Methode next(char s) in ihren nächsten Zustand versetzt. Jeder DEA signalisiert über den Rückgabewert dieser Methode, ob er sich nach Lesens dieses Zeichens in einem Errorzustand befindet. Solange nicht alle DEA's in einem Errorzustand sind, wird das gelesene Zeichen konsumiert consumeChar(), und der Vorgang wiederholt.
    4. *Alle DEA's in den letzten Zustand versetzten*. Das letztgelesene Zeichen gehört nicht mehr zu einem gültigen Terminal, darum wird es nicht konsumiert. Die DEA's werden über ihre previous-Methode in den letzten

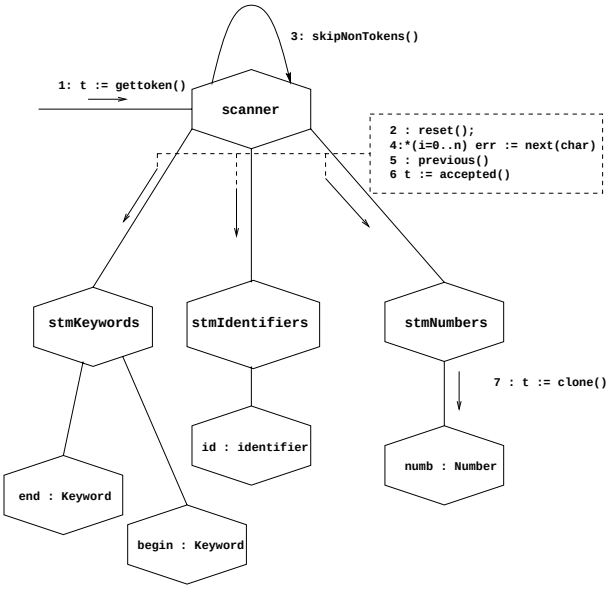

Abbildung 3.4: Objektdiagramm zum Scanner mit mehreren Statemachines

Zustand, welcher vor dem Lesen dieses Zeichens eingenommen wurde, versetzt.

5. *Akzeptierenden DEA auswählen.* Nun wird jeder DEA abgefragt, ob sein aktueller Zustand ein akzeptierender ist. Die dazu aufgerufene Methode accepted() des DEA's, liefert, wenn er akzeptiert hat, die Prototypinstanz des erkannten Terminals. Falls mehr als ein DEA akzeptiert, wird der DEA bevorzugt, welcher in der Liste stateMachines als erster auftritt.
6. *Terminal klonen.* Die clone-Methode der erhaltenen Prototypinstanz wird mit dem gescannten Text sowie den aktuellen Koordinaten im Quelltext als Parameter aufgerufen. Sie liefert ein neues Terminalobjekt vom gleichen Typ zurück. Falls kein DEA in einem akzeptierenden Zustand war oder ein ungültiges Zeichen gelesen wurde, erzeugt der Scanner ein spezielles Terminalobjekt der Klasse errortok. Falls die Eingabe leer ist, beziehungsweise zu Ende gelesen wurde, erzeugt der Scanner ein Terminalobjekt vom Typ endtoken, welches genau diesen Sachverhalt signalisiert.

3.4 Initialisierung des Scanners über ein Grammatikobjekt

Bei der Initialisierung des Scanners müssen die benötigten DEA's erzeugt werden. Damit nicht jedesmal, wenn ein neuer Scanner erstellt werden soll, die Scannerklasse scanner

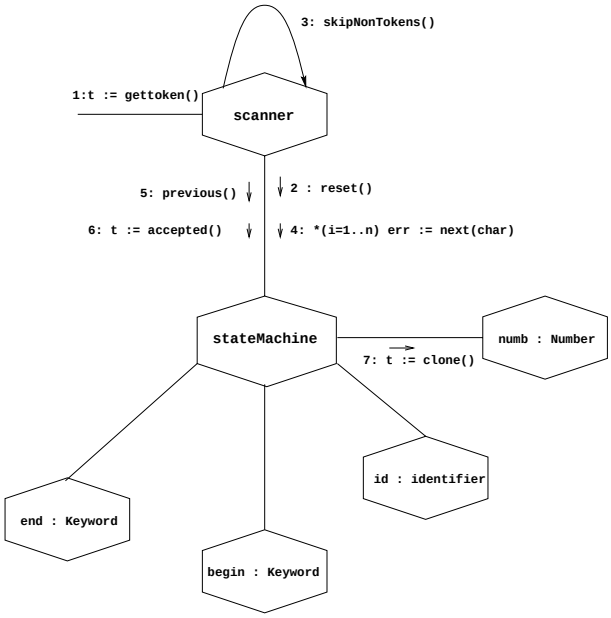

Abbildung 3.5: Objektdiagramm zum Scanner mit einer Statemachine

abgeleitet werden muss, um anschliessend den Konstruktor zu überschreiben, soll der Scanner über ein Objekt der Klasse grammar initialisiert werden können. Da die Grammatikklasse aber nur die Prototypinstanzen der zu erkennenden Terminale, aber nicht die dazu gehörenden DEA's kennt, müssen die beteiligten Klassen erweitert werden.

Dies geschieht, indem jeder Terminalklasse die Fähigkeit vermitteln wird, einen DEA zu erzeugen und zu parametrisieren. Die Klasse terminal definiert dazu eine abstrakte Methode createStateMachine(), welche von den konkreten Terminalklasse überschrieben werden muss. Zwei Beispiele sind im folgenden Quelltextauszug dargestellt.

```
stateMachine* identtok::createStateMachine() {
  return new stmIdentifier(this,this->firstChar,this->rest);
}

stateMachine* keywordtok::createStateMachine() {
  stmKeywords* stm = stmKeywords::Instance(); // singelton-Pattern
  stm->addWord(this);
  return stm;
}
```

Das Beispiel zeigt zwei verschiedene Typen von DEA's. Der erste, stmIdentifier, wurde schon weiter oben beschrieben. Es handelt sich um einen DEA, der genau ein Terminal erkennt. Der zweite, stmKeywords, ist auf Terminale spezalisiert, welche genau durch

ein Wort der Eingabesprache repräsentiert werden. Der DEA bietet eine Methode addWord(terminal*) an, über welche er erweitert werden kann. Schlussendlich wird nicht nur ein, sondern mehrere Terminals erkannt. Damit aber nur eine Instanz dieses DEA's erzeugt wird, wurde in stmKeywords das *Singleton*-Pattern, wie es in [GHJV95] beschrieben wird, implementiert.

Wenn nun der Scanner mit einem Grammatikobjekt initialisiert wird, kann dieser, via die Prototypinstanzen der Terminalobjekte, die nötigen DEA's erzeugen[1].

[1] für ein Beispiel siehe Anhang A auf Seite 94

Kapitel 4

Parser

Ein Parser für eine Grammatik $G = (\Sigma, V, P, S)$ ist ein Programm, welches für ein Wort $w \in \Sigma^*$ entscheiden kann, ob es sich um einen Satz der von G erzeugten Sprache $L(G)$ handelt. Falls es sich um einen Satz handelt, erzeugt der Parser einen Syntaxbaum, welcher die Herleitung des Satzes w repräsentiert.

Man unterscheidet grundsätzlich zwei Ansätze in der Syntaxanalyse. Der erste Ansatz ist die *Top-Down* Methode. Ein Parser arbeitet *top-down*, wenn er den Syntaxbaum, welcher zu einem Satz w gehört, von der Wurzel (dem Startsymbol) zu den Blättern (den Terminalen) hin erstellt. LL-Parser sind ein Beispiel für Parser, welche mit dieser Methode arbeiten. Dabei bedeutet der Kürzel *LL*, dass die Eingabe von links nach rechts gelesen wird, und das für die Herleitung eines Satzes immmer das am weitesten links stehende Nichtterminal abgeleitet wird.

Der zweite Ansatz wird als *Bottom-Up* bezeichnet. Ein bottom-up Parser erkennt die Struktur eines Syntaxbaums indem er sich bei den Blättern beginnend zur Wurzel des Syntaxbaums hocharbeitet. LR-Parser sind ein Beispiel für Bottom-Up Parser. *LR* steht dabei dafür, dass die Eingabe von links nach rechts, und die Herleitung eines Satzes in Form einer Rechtsherleitung erfolgt, was bedeutet, dass in einer Satzform immer das am weitesten rechts stehende Nichtterminal abgeleitet wird.

Beide Ansätze haben Vor- und Nachteile. Der Hauptvorteil eines LL-Parsers ist, das er sehr einfach von Hand implementiert werden kann, zum Beispiel in einem Parser welcher durch rekursiven Abstieg funktioniert. Dabei wird üblicherweise für jedes Nichtterminal der Grammatik eine Funktion geschrieben, welche jedesmal wenn dieses Nichtterminal erkannt werden soll, aufgerufen wird. Die Analyse wird mit dem Aufruf der Funktion, welche für das Startsymbol der Grammatik erstellt wurde, gestartet.

Der Nachteil ist, dass aus Grammatiken, welche Linksrekursion enthalten, keine LL-Parser erstellt werden können. LR-Parser im Gegensatz dazu haben keine Probleme mit Linksrekursion. Dieser Vorteil eines LR-Parsers ist mit dem Nachteil verknüpft, dass ein LR-Parser kaum von Hand erstellt werden kann. LR-Parser benutzen normalerweise eine Parsertabelle, deren Berechnung sehr aufwendig und kompliziert ist. Um diesen Nachteil aufzuheben, wurden Parsetabellengeneratoren entwickelt wie zum Beispiel Yacc.

Da LR-Parser mächtiger sind als LL-Parser, und da die meisten Grammatiken für

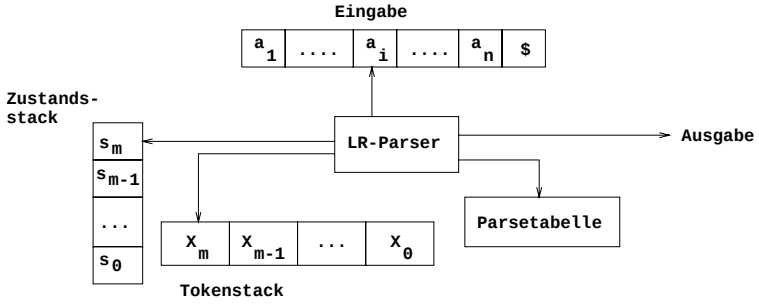

Abbildung 4.1: Der Parser mit seinen Komponenten

Programmiersprachen in LR-Form vorliegen, wurde in diesem Framework ein LR-Parser implementiert. Die in Kapitel 2 vorgestellte Grammatikklasse implementiert einen Algorithmus zur Elimination von Linksrekursion und wäre somit auch durchaus geeignet für einen LL-Parser. Bei der Elimination von Linksrekursion wird jedoch oft die Lesbarkeit einer Grammatik beeinträchtigt, was als weiterer Grund für die Überlegenheit eines LR-Parsers angegeben werden kann.

4.1 LR-Syntaxanalyse

Ein LR-Parser für eine Grammatik G ist ein deterministischer Kellerautomat, welcher versucht, eine Zeichenreihe $w \in V^*$ auf das Startsymbol S zu reduzieren. Ein LR-Parser besteht aus folgenden Komponenten[ASU92] (Abbildung 4.1):

Eingabe : Die Eingabe stellt die Schnittstelle zum Scanner dar. Sie enthält eine Folge von Terminalsymbolen, welche vom Parser darauf geprüft werden, ob sie einen gültigen Satz der Sprache $L(G)$ bilden.

Zustandsstack : Der Zustandsstack ist ein Speicher für Zustände. Das oberste Element repräsentiert den aktuellen Zustand.

Tokenstack : Der Tokenstack ist ein Speicher für Tokens. Er besteht aus Terminalen und Nichtterminalen. Die Eingabe wird akzeptiert, wenn am Schluss der Tokenstack nur noch das Startsymbol enthält.

Parsetabelle : Die Parsetabelle definiert für jeden Zustand und jedes Eingabesymbol die auszuführende Aktion und den Folgezustand.

Die Parsetabelle wird in 3 Untertabellen unterteilt.

Action-Tabelle : Diese Tabelle liefert für einen Zustand s_m und ein Eingabesymbol a_i die Art der auszuführenden Aktion. Dabei werden 4 Arten von Aktionen unterschieden.

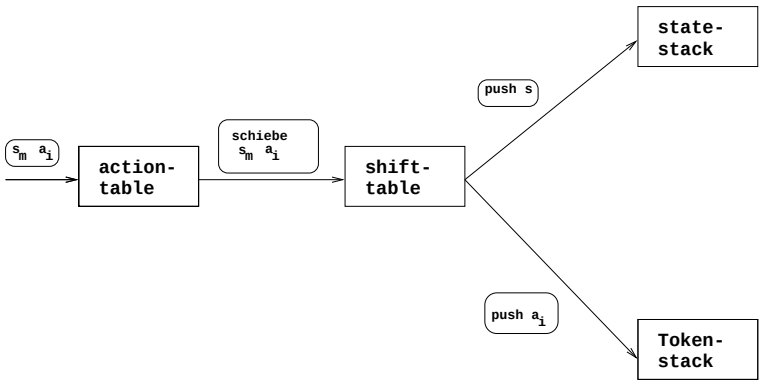

Abbildung 4.2: Ablauf einer shift-Aktion

shift : Die nächste Aktion ist eine shift-Aktion. Die Anweisungen dafür stehen in der Shift-Tabelle.

reduce : Die nächste Aktion ist eine reduce-Aktion. Die entsprechenden Anweisungen stehen in der Reduce-Tabelle.

accept : Wenn die Action-Tabelle diesen Zustand anzeigt, bedeutet dies, dass die Eingabe erfolgreich geparst wurde.

error : Im aktuellen Zustand und für das aktuelle Eingabesymbol gibt es keine gültige Aktion. Das heisst, dass die bis jetzt analysierte Eingabe zusammen mit dem aktuellen Eingabesymbol keinen gültigen Satz der Sprache $L(G)$ mehr bildet. Eine entsprechende Fehlerbehandlungsroutine muss versuchen, den Parser wieder in einen Zustand zu versetzen, in welchem er weiterarbeiten kann[1].

Shift-Tabelle : Wenn die Action-Tabelle für ein Zustands-Eingabesymbol Paar s_m, a_i eine shift-Aktion signalisiert, wird das Eingabesymbol a_i auf den Tokenstack, und der Folgezustand s_{m+1}, welcher aus der Shift-Tabelle ermittelt wird, auf den Zustandsstack geschoben (Abbildung 4.2).

Reduce-Tabelle : Wenn die Action-Tabelle eine reduce-Aktion anzeigt, wird in dieser Tabelle über das Zustands-Eingabesymbol Paar s_m, a_i die entsprechende zu reduzierende Produktion $A \rightarrow X_1 \ldots X_k$ gefunden. Vom Tokenstack und vom Zustandsstack werden die obersten k Elemente gelöscht und das Nichtterminal A neu auf den Tockenstack geschoben. Abbildung 4.3 stellt dies dar.

Jump-Tabelle : Nachdem reduziert wurde, kann über dass Nichtterminal A, welches nun zuoberst auf dem Tokenstack liegt und dem aktuellen Zustand s_{m-k} der Folgezustand s herausgefunden werden, der dann auf den Zustandsstack geschoben wird (Abbildung 4.3).

[1] Fehlerbehandlung Abschnitt 6

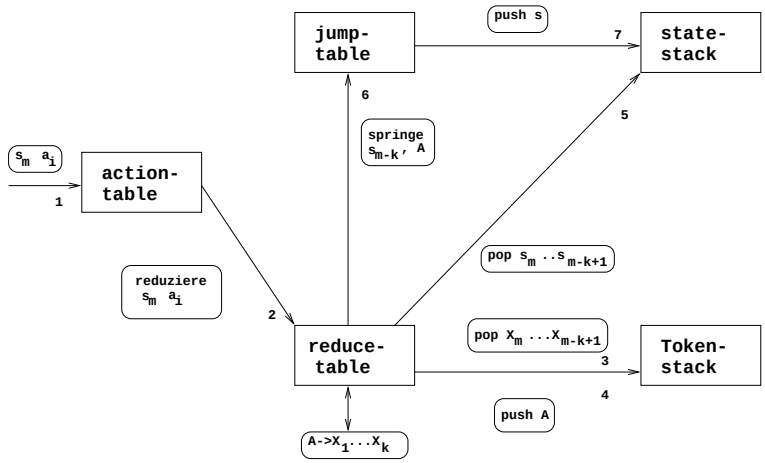

Abbildung 4.3: Ablauf einer reduce-Aktion

4.2 LR-Parsetabellenberechnung

Es gibt grundsätzlich drei bekannte Verfahren zur Berechnung von Parsetabellen für LR-Parser. Die Verfahren unterscheiden sich in der Leistungsfähigkeit und der Grösse der erzeugten Parsetabelle.

- **SLR(1)** einfacher (**s**imple LR). Ist das einfachste aber auch schwächste dieser drei Verfahren.
- **LR(1)** kanonischer **LR**. Der allgemeinste und gleichzeitig auch mächtigste der drei Algorithmen. Sein Nachteil ist, dass die generierten Parsetabellen deutlich mehr Zustände aufweisen als die durch einen SLR(1)- beziehungsweise einen LALR(1)-Algorithmus berechneten Tabellen.
- **LALR(1)** vorausschauender (**l**look**a**head) **LR**. Der LALR(1) stellt eine praktische Zwischenlösung zwischen dem LR(1) und dem SLR(1) dar. Der LALR(1) ist fast so mächtig wie der LR(1), hat dabei aber für diesselbe Grammatik nur soviele Zustände wie der SLR(1).

Das LALR(1)-Verfahren ist für den praktischen Einsatz aus oben genannten Gründen die beste Wahl, und wurde darum in diesem Framework implementiert. Im folgenden soll der Algorithmus beschrieben werden.

4.2.1 Definitionen

- **LR(0)-Element**. Ein LR(0)-Element einer Grammatik G ist eine Produktion von G mit einem Punkt an irgendeiner Position auf der rechten Seite. Die Produktion $A \rightarrow XYZ$ bringt zum Beispiel vier Elemente hervor:

  - $A \to \cdot XYZ$
  - $A \to X \cdot YZ$
  - $A \to XY \cdot Z$
  - $A \to XYZ \cdot$

  Diese LR(0)-Elemente stellen Situationen dar, die während der Syntaxanalyse auftreten können, wobei man sich den Punkt $\cdot$ als Berabeitungsmarke des Parsers vorstellen kann.

- **LR(1)-Element**. Ein LR(1)-Element besteht aus einem LR(0)-Element und einem lookahead Symbol der Länge 1. Ein Beispiel für ein LR(1)-Element ist:

  - $A \to X \cdot YZ, a$

  Die LR(1)-Elemente sind eine Erweiterung der LR(0)-Elemente. Das lookahead Symbol kommt dann zum tragen, wenn eine Situation der Form $A \to \gamma \cdot, a$ vorliegt. Das Lesezeichen ist am Ende der rechten Seite, was bedeutet, dass die Produktion reduziert werden kann. Im Falle eines LR(0)-Elementes wird die Reduktion vorgenommen wenn in der Eingabe ein Element aus $FOLLOW(A)$ folgt. Im Falle des oben genannten LR(1)-Elementes jedoch nur wenn das Eingabesymbol gleich dem lookahead Symbol a ist.

- **erweiterte Grammatik**. Um den Algorithmus ausführen zu können, muss eine gegebene Grammatik $G = (N, T, S, P)$ wie folgt erweitert werden $G' = (N \cup \{S'\}, T, S', P \cup \{S' \to S\})$.

- $\mathbf{CLOSURE(I)}$: sei I eine Menge von LR(1)-Elementen dann ist $CLOSURE(I)$ eine Funktion auf I wie folgt definiert:
  für jedes Element $A \to \alpha \cdot B\beta, a \in I$ und jede Produktion $B \to \gamma \in G'$ und jedes Terminal $b \in FIRST(\beta a)$ mit $B \to \cdot\gamma, b \notin I$ füge $B \to \cdot\gamma, b$ zu I hinzu. Dieser Vorgang wird wiederholt, bis kein Element mehr zu I hinzugefügt werden kann.

- $\mathbf{GOTO(I, X)}$: sei I eine Menge von LR(1)-Elementen und $X \in V$:
  sei $J := \{A \to \alpha X \cdot \beta, a | A \to \alpha \cdot X\beta, a \in I\}$ dann ist $GOTO(I, X) := CLOSURE(J)$.

4.2.2 Berechnung der Zustände

Die Zustände des Parsers werden beginnend mit $C := \{CLOSURE(\{S\text{`} \to \cdot S, \$\})\}$ berechnet. Für jede Menge von Elementen $I \in C$ und jedes $X \in V$ mit $GOTO(I, X) \neq \emptyset$ und $GOTO(I, X) \notin C$ wird $GOTO(I, X)$ zu C hinzugefügt. Dieser Vorgang wird solange wiederholt, bis keine Mengen mehr zu C hinzukommen. Die resultierenden Mengen repräsentieren die Zustände eines kanonischen LR-Parsers und können bei einer üblichen Programmiersprache tausend oder mehr Elemente umfassen. Die LALR(1)-Methode unterscheidet sich dahingehend, als dass sie Mengen, welche sich nur in den lookahead Symbolen ihrer LR(1)-Elemente unterscheiden, zusammenfasst (Abbildung 4.4).

Action			
State	c	d	$
0	shift	shift	
1			accept
2	shift	shift	
3	shift	shift	
4	reduce	reduce	reduce
5			reduce
6	reduce	reduce	reduce

Tabelle 4.1: Die Action-Tabelle

	Shift			Reduce			Jump	
State	c	d	$	c	d	$	S	C
0	3	4					1	2
1								
2	3	4						5
3	3	4						6
4				$C \to d$	$C \to d$	$C \to d$		
5						$S \to CC$		
6				$C \to cC$	$C \to cC$	$C \to cC$		

Tabelle 4.2: Shift-, Reduce- und Jump-Tabelle

Anhand der berechneten Mengen, kann mit dem in [ASU92] beschriebenen Algorithmus die Parsetabelle berechnet werden. Im nachfolgenden Beispiel ist die Parsetabelle für eine Grammatik dargestellt.

4.2.3 Beispiel

$$\begin{aligned} &S \to CC \\ &C \to cC \\ &C \to d \end{aligned} \tag{4.1}$$

Wenn der LALR(1)-Algorithmus auf diese Grammatik anwendet wird, erhält man die Tabellen 4.1 und 4.2. In Tabelle 4.3 ist dargestellt, wie sich die entsprechenden Stacks beim Parsen der Eingabe $ccdcd$ verändern.

4.3 Implementation

Die Klassenhierarchie wurde so gestaltet, dass möglichst alle Komponenten, welche den Parser ausmachen, austauschbar sind. So wurden zum Beispiel die Parsetabellenberech-

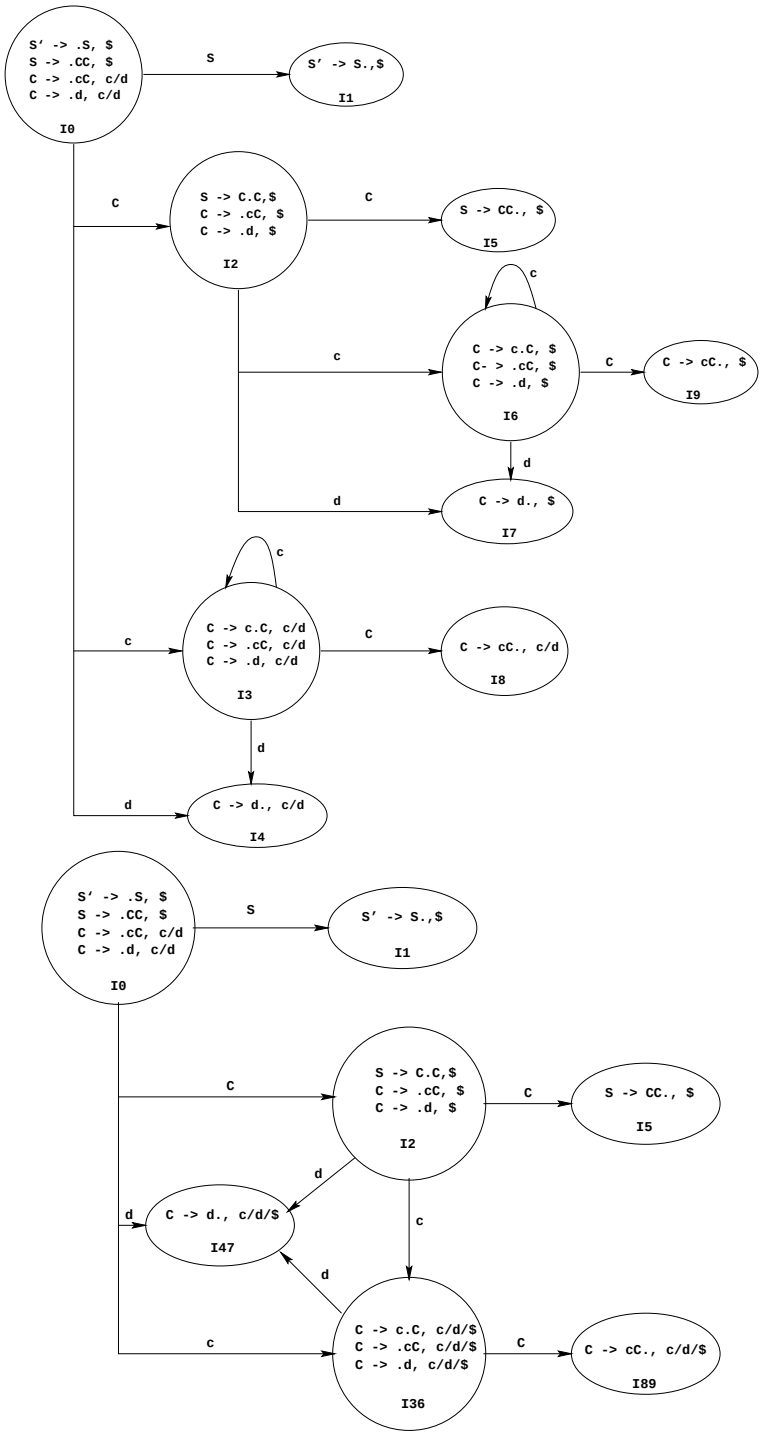

Abbildung 4.4: Sprunggraphen für die Grammatik 4.1,berechnet mit dem LR(1)- beziehungsweises den LALR(1)-Algorithmus. Der zweite Sprunggraph entsteht durch Zusammenfassen von Zuständen mit den gleichen LR(0)-Elementen ($I3 + I6$,$I4 + I7$ und $I8 + I9$)

Zustandsstack	Tokenstack	Eingabe
0		c c d c d $
0 3	c	c d c d $
0 3 3	c c	d c d $
0 3 3 4	c c d	c d $
0 3 3 6	c c C	c d $
0 3 6	c C	c d $
0 2	C	c d $
0 2 3	C c	d $
0 2 3 4	C c d	$
0 2 3 6	C c C	$
0 2 5	C C	$
0 1	S	$

Tabelle 4.3: Parsing der Eingabe $ccdcd$ nach Grammatik

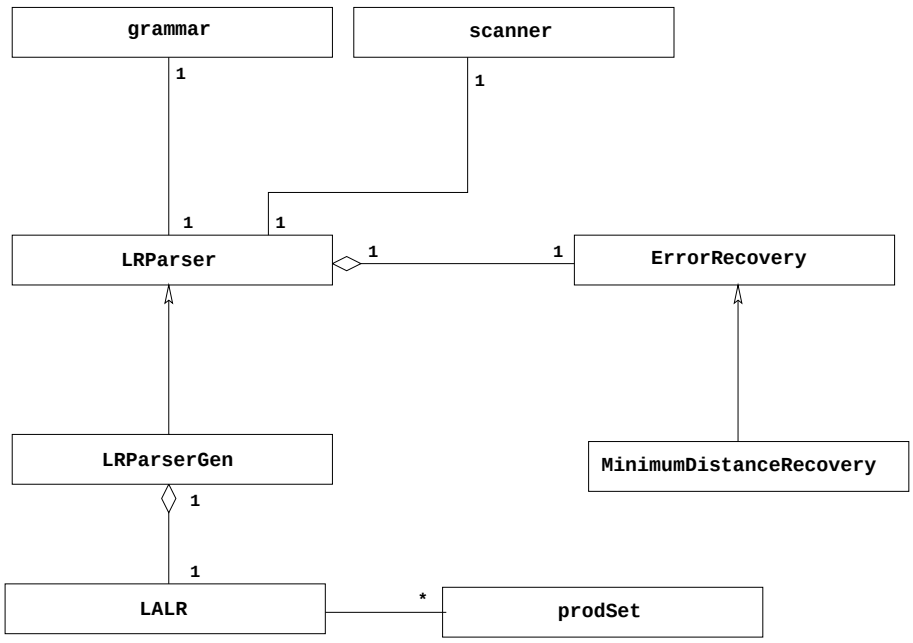

Abbildung 4.5: Klassenhierarchie um den Parser

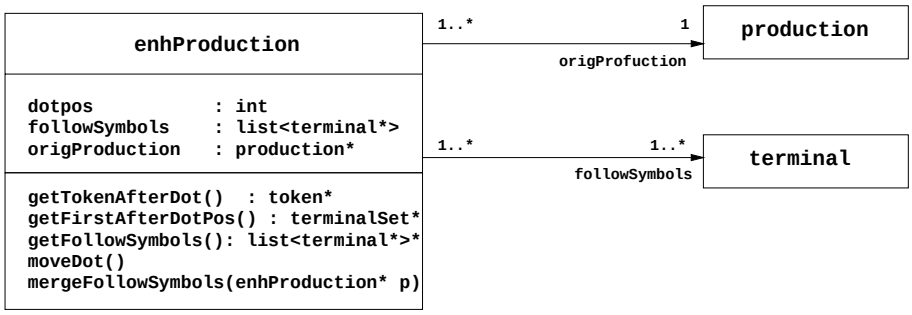

Abbildung 4.6: Klassendiagramm der erweiterten Produktion

nung und die Fehlerbehandlung in eigenen Klassen implementiert, damit diese bei Bedarf ersetzt werden können. Die im folgenden beschriebenen Klassen sind in Abbildung 4.5 in ihrem Zusammenspiel dargestellt.

4.3.1 Klasse enhProduction

Die LR(1)-Elemente werden als erweiterte Produktionen in der Klasse enhProduction (Abbildung 4.6) implementiert, welche folgende Attribute und Methoden definiert:

- **Attribute:**
  - production* origProduction. Ein Zeiger auf die zu Grunde liegende Produktion.
  - int dotpos. Die Position der Bearbeitungsmarke.
  - list<terminal*> followSymbols. Die Menge der lookahead Symbole.
- **Methoden:**
  - moveDot(). Bewegt, wenn möglich, die Bearbeitungsmarke ein Zeichen nach rechts.
  - termialSet* getFIRSTAfterDotPos() const. Berechnet die FIRST-Menge der Zeichen nach der Bearbeitungsmarke.
  - mergeFollowSymbols(const enhProduction* p). Nimmt die lookahead Symbole von p zu den eigenen lookahead Symbolen hinzu.
  - operator==(const enhProduction &p). Zwei Objekte der Klasse enhProduction werden als gleich definiert, wenn von der gleichen Produktion origProduction abstammen und die gleiche Position der Bearbeitungsmarke dotpos aufweisen.

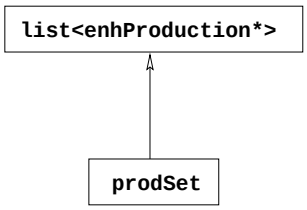

Abbildung 4.7: Die Klasse prodSet implementiert die LR(1)-Elementmengen

4.3.2 Klasse prodSet

Eine Menge von LR(1)-Elementen wird als Klasse prodSet implementiert. Die Klasse wird von list<enhProduction*> abgeleitet (Abbildung 4.7).

- **Attribute:**
  - Grammar* grammar. Ein Zeiger auf die zu Grunde liegende Grammatik. Wird benötigt um closure() zu berechnen.
- **Methoden:**
  - closure(). Berechnet die CLOSURE-Menge wie sie im theoretischen Teil beschrieben wurde.
  - add(enhProduction* p). Fügt der Menge ein LR(1)-Element hinzu.

4.3.3 Klasse LRParser

Die Klasse LRParser implementiert einen LR-Parser wie er im theoretischen Teil beschrieben wurde. Die Klasse weiss nichts über den zur Verwendung kommenden Algorithmus zur Parsetabellenberechnung, sondern definiert nur die eigentliche Steuerroutine des LR-Parsers. Durch den Objektspeichermechanismus[2] wird es ermöglicht, diese Klasse in einem Compiler einzusetzten, wenn die Parsetabelle bereits berechnet wurde.

- **Definitionen:** Im folgenden werden die Zeilenelemente der verschiedenen Untertabellen der Parsetabelle definiert. Ein solches Zeilenelement ist eine map [3], welche einen Terminal auf einen Folgezustand (Shift-Tabelle) eine Aktion (Action-Tabelle) oder eine Produktion (Reduce-Tabelle) abbildet oder ein Nichtterminal auf einen Folgezustand (Jump-Tabelle) abbildet.
  - typedef map<terminal*,int,less<terminal*>> shiftFunc. Definiert eine Zeile in der Shift-Tabelle.
  - typedef map<terminal*,char,less<terminal*>> actionFunc. Definiert eine Zeile in der Action-Tabelle.

[2] siehe Kapitel 8

[3] eine assoziative Containerklasse aus der **STL**-Library

  - typedef map<nonterminal*,int,less<nonterminal*>> jumpFunc. Definiert eine Zeile in der Jump-Tabelle.
  - typedef map<terminal*,production*,less<terminal*>> reduceFunc. Definiert eine Zeile in der Reduce-Tabelle.

- **Attribute:**
  - vector<actionFunc*> action. Die Action-Tabelle. Wenn s ein Zustand ist und t ein Zeiger auf ein Terminalobjekt, dann findet man über (*action[s])[t] die auszuführende Aktion, falls eine existiert.
  - vector<shiftFunc*> shift. Die Shift-Tabelle. Wenn s ein Zustand ist und t ein Zeiger auf ein Terminalobjekt, dann findet man über (*shift[s])[t] den Folgezustand.
  - vector<jumpFunc*> jump. Die Jump-Tabelle. Wenn s ein Zustand ist und n ein Zeiger auf ein Nichtterminalobjekt, dann findet man über (*jump[s])[n] den Folgezustand.
  - vector<reduceFunc*> reduce. Die Reduce-Tabelle. Wenn s ein Zustand ist und t ein Zeiger auf ein Terminalobjekt, dann findet man über (*reduce[s])[t] die einen Zieger auf das zu reduzierende Produktionsobjekt.
  - int state. Der aktuelle Zustand
  - vector<int> stateStack. Der Zustandsstack
  - vector<node*> tokenStack. Der Tokenstack
  - vector<terminal*> terminalBuffer. Ein Buffer welcher für die Fehlerbehandlung genutzt werden kann. Solange er nicht leer ist, holt getterminal() die Elemente nicht vom Scanner sondern aus diesem Buffer.
  - scanner* Scanner. Der benutzte Scanner.
  - grammar* Grammar. Die Grammatik.
  - errorRecovery* ErrorRecovery. Verweis auf eine Fehlerbehandlungsobjekt. Die Klasse ErrorRecovery wird als friend deklariert. Eine konkrete Fehlerbehandlungsroutine[4] muss in einer Klasse welche von ErrorRecovery abgeleitet wird, definiert werden.

- **Methoden:**
  - terminal* getterminal(). Interne Methode, welche das nächste Element holt.
  - char solveConflict(int state,terminal* actTerminal). Methode wird aufgerufen, wenn ein Konflikt auftritt[5].
  - node* start(). Startet den Parser. Gibt bei erfolgreicher Analyse die Wurzel des erzeugten Syntaxbaums zurück.
  - setErrorRecovery(ErrorRecovery* E). Ordnet dem Parser ein Fehlerbehandlungsobjekt zu.

[4] Abschnitt 6
[5] siehe Abschnitt 5

4.3.4 Klasse LRParserGen

LRParserGen ist eine abgeleitete Klasse von LRParser, welche den LALR(1)-Algorithmus verwendet, um die Parsetabelle zu berechnen. Durch diese Veerbung wird gewährleistet, dass die eigentliche Parserklasse LRParser unabhängig[6] vom Parsetabellenberechnungsalgorithmus bleibt.

LRParserGen unterscheidet sich von LRParser nur dadurch, dass im Konstruktor eine Instanz der Klasse LALR aufgerufen wird, welche die Berechnung der Parsetabelle vornimmt.

4.3.5 Klasse LALR

Die Parsetabellenberechnung wird in einer eigenen Klasse LALR gekapselt. Um die Berechnung zu starten, wird der Konstruktor der Klasse aufgerufen. Die Klasse hat, da sie als friend der Parserklasse definiert ist, vollen Zugriff auf dessen Parsetabelle.

- **Attribute:**
  - list<prodSet*> LR1Set. Die Menge der Mengen der LR(1)-Elemente. Repräsentiert die Zustände des zu erzeugenden Parsers.
  - int changed. Die Parsetabellenberechnung erfolgt in mehreren Durchgängen. Changed zeigt an, wenn sich in den lookahead Symbolen nichts mehr verändert.
- **Methoden:** LALR definiert ausser dem Konstruktor keine öffentlichen Methoden. Die hier aufgeführten Methoden sind alle für den internen Gebrauch der Klasse.
  - LALR(LRParserGen* parser). Mit parser wird der zu berechnende Parser übergeben, welcher von der Klasse LRParserGen sein muss. Diese Klasse stellt alle Methoden zur Verfügung, welche es erlauben, die Parsetabelle zu manipulieren. Mit dem Konstruktoraufruf wird auch die Parsetabellenberechnung gestartet.
  - prodSet* GOTO(prodSet* I,token* t). Die GOTO-Methode, wie sie im theoretischen Teil besprochen wurde.
  - PROPAGATE(prodSet* I, token * t ,prodSet* J). Nach dem ersten Berechnungsdurchgang ist die Menge der Menge der LR(0)-Elemente, sowie ein Teil der lookahead Symbole berechnet. In den weiteren Durchgängen werden nur noch die lookahead Symbole propagiert. Zu diesem Zweck wird anstatt der GOTO-Methode nur noch diese Methode aufgerufen.
  - addToLR1Set(prodSet* I). Fügt LR1Set die Menge I von LR(1)-Elementen hinzu, falls diese nicht schon vorhanden ist. Wenn Sie vorhanden ist, werden nur die lookahead Symbole vereinigt.

[6] unabhängig auf zwei Arten. 1. kann zum Beispiel eine weitere Klasse abgeleitet werden, welche einen SLR(1)-Algorithmus verwendet. 2. weiss die Klasse LRParser nichts von einer Parsetabellenberechnung, und so muss, wenn die Tabelle einmal berechnet und gespeichert ist, dieser Algorithmus im Compiler auch nicht mehr vorhanden sein.

4.4 Schnittstelle zum abstrakten Syntaxbaum

Jedesmal, wenn eine Produktion reduziert wird, wird ein neuer Knoten im Syntaxbaum erzeugt. Dies geschieht indem gleichviele Symbole, wie auf der rechten Seite der Produktion stehen, vom Tokenstack genommen (reduziert) werden, und mit Hilfe dieser Symbole ein neuer Syntaxbaumknoten erstellt wird. Wie bereits in Abschnitt 2.1.5 bei der Beschreibung der Klasse production erwähnt, besitzt jedes Objekt dieser Klasse einen Zeiger auf eine statische Methode einer Syntaxbaumknotenklasse. Indem diese Methode mit den oben erwähnten Symbolen als Parameter aufgerufen wird, wird ein neues Objekt dieser Syntaxbaumknotenklasse erzeugt. Bei den übergebenen Symbolen kann es sich um andere Syntaxbaumknotenobjekte oder um Terminalobjekte handeln. Diese werden je nachdem als Kinderknoten oder als Blattknoten an den neu erstellten Knoten gehängt, und der so erstellte Knoten zurück auf den Tokenstack gelegt.

Am Schluss der Syntaxanalyse liegt auf dem Tokenstack der oberste Knoten des Syntaxbaums. Die hier erwähnten Syntaxbaumklassen werden in Abschnitt 7 beschrieben.

Kapitel 5

Dynamische Konfliktlösung

Wenn eine kontextfreie Grammatik mehrdeutig ist, das heisst, wenn ein Satz der Sprache auf verschiedene Weise hergeleitet werden kann, dann erfüllt diese Grammatik weder das LL- noch das LR-Kriterium. Es gibt verschieden Ansätze, um Mehrdeutigkeiten aus einer Grammatik zu eliminieren. Neben der Möglichkeit, die Grammatik umzuschreiben, bieten die meisten Parserwerkzeuge spezielle Methoden an, welche, indem sie die Terminale der Grammatik mit Prioritäts- und Assoziativitätsattributen erweitern, die Mehrdeutigkeiten während der Parsetabellenberechnung auflösen können.

In diesem Kapitel soll neben diesen Methoden ein neuer Ansatz vorgestellt werden, der gewisse Mehrdeutigkeiten erst während der Syntaxanalyse auflöst. Zuerst werden die verschiedenen Arten von Mehrdeutigkeiten besprochen.

5.1 Beispiele für mehrdeutige Grammatiken

Mehrdeutigen Grammatiken, wie die folgende, werden oft benutzt um arithmetische Ausdrücke zu beschreiben.

E → E + E
E → E * E
E → numb

Diese Grammatik ist nicht nur mehrdeutig, sondern sie sagt auch nichts über die Operatorpriorität der beiden Operatoren $+$ und $*$ aus. So können aus der Eingabe $3 + 4 * 5$ mehrere Syntaxbäume abgeleitet werden (Abbildung 5.1).

Ein anderes, häufig auftretendes Problem, ist das sogenannte *hängende* else.

S → if E then S
S → if E then S else S
S → other

Bei einer Eingabe der Form if A then if B then C else D kann nicht entschieden werden, ob das letzte else zum ersten oder zum zweiten if gehört.

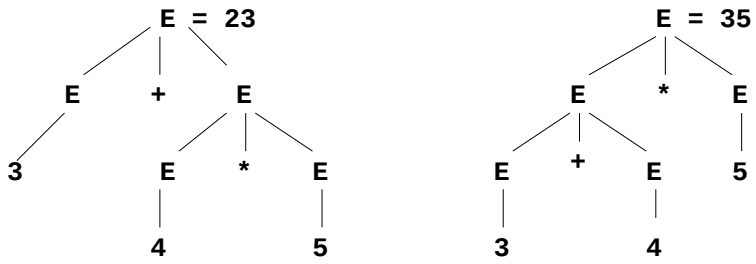

Abbildung 5.1: Zwei Ableitungen der gleichen Eingabe

Die beiden oben beschriebenen Grammatiken produzieren bei der Parsetabellenberechnung mit einem LALR(1)-Algorithmus sogenannte **shift/reduce**-Konflikte. Während der Berechnung gibt es eine Menge von LR(1)-Elementen, welche die beiden folgenden LR(1)-Element enthält.

$S \rightarrow \text{if } E \text{ then } S \cdot$
$S \rightarrow \text{if } E \text{ then } S \cdot \text{ else } S$

Das heisst, wenn als nächstes das Terminal else betrachtet wird, kann auf Grund des ersten LR(1)-Elementes[1] die Produktion $S \rightarrow \text{if } E \text{ then } S$ reduziert werden. Auf Grund des zweiten Elementes könnte else aber auch geschoben werden. Dass heisst der Algorithmus kann sich hier nicht entscheiden, ob er die Shift- oder die Reduce-Aktion in die Parsetabelle eintragen soll. Man spricht in diesem Fall von einem **shift/reduce**-Konflikt.

Andere Mehrdeutigkeiten drücken sich bei der Berechnung der Parsetabelle durch sogenannte **reduce/reduce**-Konflikte aus. Das heisst der Algorithmus kommt in eine Konfiguration, in welcher er zwei oder mehrere verschieden Produktionen reduzieren könnte. Ein Beispiel für eine Grammatik, die einen solchen Konflikt erzeugt, ist hier angegeben.

$S \rightarrow A \mid B$
$A \rightarrow x$
$B \rightarrow x$

Im folgenden Abschnitt werden bekannte Strategien zur Elimination solcher Konflikte besprochen.

5.2 Bekannte Lösungen

Für die im vorherigen Abschnitt beschriebenen Konflikte gibt es drei grundsätzliche Lösungen:

[1] die LR(1)-Elemente sind ohne die lookahead Symbole dargestellt. Das Terminal else ist Element beider lookahead Mengen

1. **Umformung der Grammatik**. Durch eine Grammatikumformung können Konflikte oft eliminiert werden. Eine äquivalente Grammatik zu der Grammatik für arithmetische Ausdrücke ohne Konflikte sieht wie folgt aus:

   E → T + E
   E → T
   T → F * T
   T → F
   F → numb

   Durch die Einführung der neuen Nichtterminale T und F und der entsprechenden Produktionen werden die Konflikte eliminiert und die natürliche Operatorpriorität (* vor +) garantiert. Diese Lösung hat den Vorteil, dass keine neuen Konzepte eingeführt werden müssen. Der Nachteil besteht darin, dass die entsprechende Parsetabelle grösser wird, und während der Syntaxanalyse viele Reduktionen der Form $E \rightarrow T$ und $T \rightarrow F$ gemacht werden. Bei **reduce/reduce**-Konflikten müssen häufig grundsätzliche Änderungen in der Grammatik vorgenommen werden, um den Konflikt zu lösen.

2. **Standardverhalten :** Yacc definiert für shift/reduce- und reduce/reduce-Konflikte ein Standardverhalten. Im ersten Fall zieht er die shift-Aktion der reduce-Aktion vor, im zweiten Fall bevorzugt er die Produktion welche als Erste in der Definition erscheint. Durch dieses Verhalten wird das Problem des hängenden else gelöst, denn wenn standardmässig geschoben wird, bedeutet dies, dass das letzte else immer dem letzten if zugeschrieben wird, was im Normalfall auch der Intention der Programmiersprache entspricht.

3. **Prioritäts und Assoziativitätsattributen für Terminale**. Dieser Ansatz beruht auf der Operator-Precedence Syntaxanalyse [Flo63]. Die Idee besteht darin, dass den Terminalen Prioritäts- und Assoziativitätsattribute hinzugefügt werden können. Wenn bei der Berechnung der Parsetabelle ein shift/reduce-Konflikt auftritt, werden die beiden beteiligten Elemente auf ihre Prioritäts- und Assozitivitätsattribute hin untersucht. Die beiden Elemente bestehen aus dem zu schiebenden Terminal und der zu reduzierenden Produktion. Je nachdem, welches Element die höhere Priorität hat, wird geschoben oder reduziert. Dabei wird die Priorität der Produktion anhand der Priorität des am weitesten rechts stehenden Terminal der Produktion bestimmt. Falls sich bei der Analyse der Prioritäten keine Unterschiede ergeben, wird die Assoziativiät der zu reduzierenden Produktion betrachtet. In Tabelle 5.1 sind Priorität und Assozitaivität für die beiden Operatorterminale $+$ und $*$ definiert. In Tabelle 5.2 ist dargestellt, wie zwei Konflikte gelöst werden können.

In den meisten Fällen kommt man durch Anwendung einer dieser vorgestellten Lösungen zum Ziel. Es gibt aber auch Fälle, in denen keines der oben beschrieben Verfahren weiterhilft. Ein solcher Fall soll im folgenden beschrieben werden.

Terminal	Priorität	Assoziativität
$+$	1	links
$*$	2	links

Tabelle 5.1: natürliche Precedence und Assoziativität der Operatoren

Produktion	Priorität	Terminal	Lösung (Grund)
$E \rightarrow E + E$	1	$*$	schiebe (Priorität)
$E \rightarrow E + E$	1	$+$	reduziere (Assoziativität)

Tabelle 5.2: S/R-Konflikte zwischen einer zu reduzierenden Produktion und einem zu schiebenden Terminal

5.3 Dynamische Konfliktlösung während der Analyse

5.3.1 Motivation

An einem Beispiel soll gezeigt werden, dass die im letzten Abschnitt vorgestellten Lösungen nicht immer zum Ziel führen.

Die Sprache **PICT**[PT95] definiert ein spezielles Terminal *symbolischer Bezeichner*. Wie normale Bezeichner können auch diese aus einer vorgegebenen Menge von Zeichen geformt werden. Diese Zeichenmenge besteht aus den folgenden Zeichen:

```
!,?,#,~,*,%, /,\,+,-,<,<,=,&,@,|,$
```

Die symbolischen Bezeichner werden in PICT als Operatoren verwendet. Das Spezielle an diesen symbolischen Bezeichnern ist, dass sie je nach Zeichenzusammensetzung unterschiedliche Prioritäten und Assoziativitäten haben. Zwei Beispiele für solche symbolische Identifer sind in Tabelle 5.3 wiedergegeben.

In der Sprache *PICT* gibt es genau eine Produktion, in der die symbolischen Bezeichner vorkommen.

$$\mathsf{InfVal} \rightarrow \mathsf{InfVal\ symbolicId\ InfVal}$$

Diese Produktion erinnert stark an die Grammatik für arithmetische Ausdrücke, wobei InfVal für ein Ausdruck E steht und symbolicId einen beliebigen Operator bezeichnet. Da die Priorität und die Assoziativität des Terminals symbolicId erst berechnet werden

symb. Bezeichner	Precedence	Assoziativität
*$$	5	rechts
&&&	2	links

Tabelle 5.3: Symbolische Bezeichner der Sprache *PICT*

Action				
State	numb	$+$	$*$	$
0	shift			
1		reduce	reduce	reduce
2		shift	shift	accept
3	shift			
4	shift			
5		**conflict**	**conflict**	reduce
6		**conflict**	**conflict**	reduce

Tabelle 5.4: Die Action-Tabelle für die einfache Ausdrucksgrammatik, mit den Konflikteinträgen **conflict**

kann, nachdem es vom Scanner gelesen wurde[2], ist es nicht möglich, die durch die oben genannte Produktion enstandenen Konflikte bei der Parsetabellenberechnung zu lösen.

Das beschriebene Problem kann somit mit Yacc nicht befriedigend gelöst werden. Yacc kann in diesem Fall nur auf sein Standardverhalten zurückgreifen. Der von Yacc berechnete Parser produziert nun unter Umständen einen Syntaxbaum, welcher nicht korrekt ist, und in einer Nachbearbeitung, unter Berücksichtigung der unterdessen bekannten Prioritäten und Assozitivitäten, umgeformt werden muss.

5.3.2 Dynamische Lösung

Damit das oben beschriebene Problem während der Syntaxanalyse gelöst werden kann, muss ein neuer Ansatz gewählt werden. Offensichtlich ist, dass der Konflikt erst während der eigentlichen Syntaxanalyse gelöst werden kann. Die Idee der dynamischen Lösung besteht darin, das gleiche Entscheidungsverfahren welches Yacc während der Parsetabellenberechnung anwendet, auf den Zeitpunkt der eigentlichen Syntaxanalyse zu verschieben. Um dies tun zu können, müssen folgende Voraussetzungen erfüllt werden.

- Die Aktionstabelle der Parsertabelle muss um einen Eintragstyp conflict erweitert werden (Abbildung 5.4). Die beiden, während der Parsetabellenberechnung ermittelten, Aktionen müssen vermerkt werden, da die Entscheidung für eine der beiden erst während der Analyse fällt. Das heisst, die Shift- und Reduce-Aktionen müssen in verschiedenen Tabellen eingetragen werden (Abbildung 5.5).
- Der Parser selbst muss eine Konfliktlösungsmethode anbieten, welche während der Analyse aufgerufen werden kann, wenn der Parser auf einen Konflikteintrag in der Aktionstabelle stösst (Abbildung 5.2).
- Die vom Scanner erzeugten Terminalobjekte müssen auf ihre Priorität und Assoziativität befragt werden können.

[2] Die Attribute berechnen sich ja aus dem Namen des symbolischen Bezeichners

	Shift				Reduce				Jump
State	numb	$+$	$-$	\$	numb	$+$	$-$	\$	E
0	1								2
1					$E \rightarrow numb$	$E \rightarrow numb$	$E \rightarrow numb$		
2		3	4						
3	1								5
4	1								6
5		**3**	**4**		$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E}$	$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{E}$	$E \rightarrow E + E$		
6		**3**	**4**		$\mathbf{E} \rightarrow \mathbf{E} * \mathbf{E}$	$\mathbf{E} \rightarrow \mathbf{E} * \mathbf{E}$	$E \rightarrow E * E$		

Tabelle 5.5: Shift-, Reduce- und Jump-Tabelle für die Grammatik für arithmetische Ausdrücke. Die Einträge in der Shift- und Reduce-Tabelle welche miteinander kollidieren sind **fett** markiert.

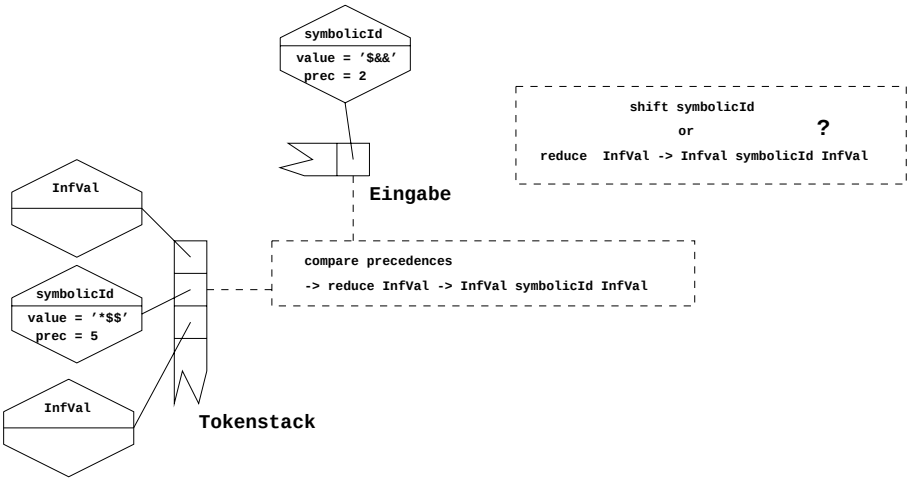

Abbildung 5.2: Dynamische Konfliktlösung

5.4 Implementation

In der Beschreibung der Terminalklassen wurde bereits darauf hingewiesen, dass die Basisklasse terminal Prioritäts- und Assoziativitätsattribute implementiert, welche von jeder konkreten Terminalklasse geerbt werden. Diese garantieren, dass der Parser die oben beschriebenen Abfragen vornehmen kann. Die Attribute werden standardmässig auf Priorität 0 und keine Assoziativität gesetzt. Diese Attribute können entweder bei der Erstellung der Grammatik gesetzt werden, oder vom Terminalobjekt bei seiner Erzeugung zum Beispiel anhand des gescannten Textes berechnet werden, wie dies bei den symbolischen Bezeichnern der Fall wäre.

Die Parserklasse definiert die Methode solveConflict(int state,terminal* actTerminal), welche aufgerufen wird, wenn der Parser auf einen Konflikteintrag in seiner Parsetabelle stösst. Der Methode wird der aktuelle Zustand state und das Terminal actTerminal, welches den Konflikt auslöst, übergeben. Die Methode bestimmt zuerst die Produktion, welche reduziert werden könnte. Anschliessend sucht sie auf dem Tockenstack das erste Terminalobjekt. Wenn Sie ein Terminal innerhalb der ersten l (wenn l die Länge der Produktion bezeichnet) Elemente des Stacks findet, ist dies das Terminal, welches am weitesten rechts in der Produktion steht. Wenn eines gefunden wurde (firstTerminalInProd), kann die Analyse der Prioritäten beginnen.

```
char LRParser::solveConflict(int state,terminal* actTerminal) {
  production* p = getReduce(actTerminal,state);
  terminal* firstTerminalInProd =
          searchFirstTerminalOnStack(p->length());
  if (firstTerminalInProd) { // terminal exists
    int precForShift = actTerminal->getPrecedence();
    int precForReduce = firstTerminalInProd->getPrecedence();
    if (precForShift > precForReduce)
      return 's'; // shift
    else if (precForShift == precForReduce)
      return 'r'; // reduce
    else if (firstTerminalInProd->rightAssociativ())
      return 's'; // shift
    else return 'r'; // reduce
  } else return 's';
}
```

Das Entscheidungsverfahren gleicht dem von Yacc, mit dem Unterschied, dass es nicht statisch sondern dynamisch zur Anwendung kommt.

Kapitel 6

Fehlerbehandlung

6.1 Allgemeine Betrachtungen

Bei der Erstellung von Programmen für eine Sprache entstehen immer wieder Fehler. Ziel eines Compilers muss es sein, nicht nur anzuzeigen, dass ein Fehler aufgetreten ist, sondern den aufgetretenen Fehler möglichst genau zu beschreiben und zu lokalisieren, und die Analyse, wenn möglich, weiterzuführen.

Ein Programm kann auf verschiedenen Ebenen Fehler enthalten. [ASU92] unterscheidet beispielsweise

- lexikalische Fehler wie falsch geschriebene Bezeichner, Schlüsselwörter oder Operatoren,
- syntaktische Fehler wie einen falsch geklammerten arithmetischen Ausdruck,
- semantische Fehler wie die Anwendung eines Operators auf inkompatible Operanden
- logische Fehler wie ein nicht-terminierender rekursiver Aufruf.

Die Fehler die hier betrachtet und behandelt werden sollen sind entweder lexikalischer oder syntaktischer Natur.

[ASU92] definiert die Ziele einer Fehlerbehandlung wie folgt

- Sie sollte verständlich und präzise das Vorhandensein von Fehlern melden.
- Sie sollte sich von einem Fehler schnell genug erholen, um auch Folgefehler entdecken zu können.
- Sie sollte die Verarbeitung korrekter Programme nicht unverhältnismässig verlangsamen.

Um zu verstehen wie Syntaxfehler aussehen können, welche von einer Fehlerbehandlung verarbeitet werden müssen, sollen die folgenden Beispiele betrachtet werden.

- `if (x == 5) a = 4 else a = 5;`. Fehlendes Semikolon zwischen dem Ausdruck und dem Schlüsselwort `else`. Der Fehler ist eindeutig und lokalisierbar.
- `a = (x + x b b)*c`. Der Fehler ist lokalisierbar aber nicht eindeutig.
- `a = (x * x + b))`. Der Fehler ist weder eindeutig noch lokalisierbar.

Diese Beispiele zeigen, dass der Begriff des Syntaxfehlers viele Variationen hat. Ebenso wird ersichtlich, dass es nicht das Ziel sein kann, einen Fehler so korrigieren zu wollen, dass das korrigierte Programm der ursprünglichen Intention des Programmierers entspricht, da dies in vielen Fällen unmöglich ist.

Syntaxanalysemethoden, wie die LL- und die LR-Methoden, entdecken einen Syntaxfehler zum frühestmöglichen Zeitpunkt. Sie besitzen die sogenannte *viable-prefix-Eigenschaft*, was bedeutet, dass sie das Auftreten eines Fehlers bereits im Moment erkennen, in dem das gelesene Präfix der Eingabe nicht mehr ein Präfix eines Wortes der Sprache sein kann. Dieser Moment der Fehlererkennung wird auch als *parser defined Error* bezeichnet, um zu betonen, dass der eigentlich Fehler unter Umständen schon früher eingetreten sein kann.

Aufbauend auf diesen Überlegungen definiert man 3 grundsätzliche Strategien in der Fehlerbehandlung[ASU92].

- **Panische Recovery:** Diese Strategie ist die einfachste und eignet sich für die meisten Sytnaxanlysemethoden. Wenn der Parser einen Fehler entdeckt, überliest er soviel von der Eingabe bis er auf ein Symbol trifft, welches Element einer Menge sogenannter *Stabilisierungssymbole* ist. Solche Symbole sind üblicherweise Begrenzer wie zum Beispiel ein Semikolon oder das Schlüsselwort `end`. Der Nachteil dieser Methode ist, dass sie einen Teil der Eingabe überliest, ohne ihn auf weitere Fehler zu prüfen.
- **konstrukt-orientierte Recovery:** Diese Strategien versuchen einen Präfix der anstehenden Eingabe so zu korrigeren, dass dieser mit der bereits gelesenen gültigen Eingabe zusammen einen korrekten Präfix der Sprache bildet. Diese Strategien zeigen vorallem in Situationen, in welchen der eigentliche Fehler vor dem *parser definied Error* Auftritt, Schwächen.
- **Fehlerproduktionen:** Wenn klar ist, welche Fehler gewöhnlich auftreten, kann die Grammatik mit sogenannten Fehlerproduktionen erweitert werden, welche diese Fehler simulieren. Der Vorteil dieser Methode ist, dass wenn die Fehlerproduktionen gut gewählt werden, auch gute Fehlermeldungen und Korrekturen zu erwarten sind. Der Nachteil liegt darin, dass die Grammatik erweitert werden muss, und ein Entwickler fundierte Kenntnise über die Grammatik und die häufigen Fehler haben muss.

Die Fehlerbehandlung, welche für dieses Framework gewählt wird, soll folgende Eigenschaften haben:

- Sie muss allgemein sein. Da das Framework nicht auf eine bestimmte Sprache fixiert ist, muss die Fehlerbehandlung möglichst allgemein sein.
- Die Fehlerbehandlung soll ein Standardverhalten implementieren. Das heisst, die Fehlerbehandlung soll funktionieren ohne das der Anwender etwas dafür tun muss.

Aufgrund dieser Überlegungen wurde eine konstrukt-orientierten Recovery implementiert. Die Methode wird in einem Artikel von [Dai] beschrieben und nennt sich *Minimum Distance Recovery*. Sie wird im Rest dieses Abschnittes erläutert.

6.2 Definitionen

Sei Σ ein endliches Alphabet von Symbolen. a und b bezeichnen Symbole aus Σ, u und v Sätze über Σ, und ϵ sei der leere Satz. Sei Δ die Menge der edit-operationen $\{(a,b)|a,b \in \Sigma \cup \{\epsilon\}, (a,b) \neq (\epsilon,\epsilon)\}$.

DEFINITION 1 *Für Sätze u,v aus Σ^*, $u \rightarrow v$ via $a \rightarrow b$ wenn $(a,b) \in \Delta$ und es gibt Sätze w,x in Σ^*, so dass $u = wax$ und $v = wbx$.*

DEFINITION 2 *$u \rightarrow v$ via E wenn E eine endliche Sequenz von edit-operationen $e_1 \ldots e_n$ und es gibt Sätze $w_1, \ldots, w_{n-1}$ in Σ^* so dass $u \rightarrow w_1$ via e_1.*

DEFINITION 3 *Die minimale Distanz $d(u,v)$ zweier Strings u,v ist gegeben durch $d(u,v) = min\{n|u \rightarrow v$ via E für eine Sequenz von edit-operationen $E = e_1 \ldots e_n\}$.*

DEFINITION 4 *Die Menge der Perfixe $Pre(L)$ einer Sprache L über Σ^* ist gegeben durch $Pre(L) = \{u \in \Sigma^*|uv \in L$ für ein $v \in \Sigma^*\}$.*

DEFINITION 5 *Der Fehler in einem String u, der $u \notin L$, ist nach einem String x beim Symbol a wenn $u = xay$, $x \in Pre(L)$ und $xa \notin Pre(L)$.*

DEFINITION 6 *Die Menge der möglichen Nachfolgestrings $Cont(u)$ für einen gültigen Präfix u sind definiert als $Cont(u) = \{w \in \Sigma^*|uw \in L\}$.*

6.3 Beschreibung der Methode

Ziel ist es, eine praktikable Methode für die Fehlerbehebung anzugeben, welche die Metrik der minimalen Distanz verwendet, um eine Reparatur zu finden. Idealerweise würde die Reparatur in Form eines Nachfolgestrings erfolgen, dessen minimale Distanz von der wirklichen verbleibenden Eingabe minimal über der Menge aller Nachfolgestrings wäre. Dies ist aber meist nicht durchführbar. Um eine praktikable Methode anzugeben, muss die Menge der möglichen Nachfolgestrings beschränkt werden. Dies wird erreicht, indem nur Präfixe von Nachfolgestrings bis zu einer gewissen Länge σ betrachtet werden. Diese Präfixe werden mit einer festen Anzahl τ von verbleibenden Eingabesymbolen verglichen.

Aus diesem Vergleich wird der beste Präfix ausgesucht, der dann einen Präfix der verbleibenden Eingabe ersetzt. Die Methode garantiert nicht, dass eine Reparatur gefunden wird, die zu einem gültigen Satz der Sprache führt, sondern nur, dass eine gewisse Anzahl von verbleibenden Eingabesymbolen vom Parser akzeptiert werden.

Zwei Probleme, die gelöst werden müssen sind, welcher Präfix gewählt und wieviel vom verbleibenden Eingabestring ersetzt werden soll. Es ist wünschenswert, die bestmögliche Übereinstimmung zwischen dem Nachfolgestringpräfix und dem verbleibenden Eingabestring zu finden. Das heisst aber nicht, dass zwingend der ganze verbleibende Eingabestring der Länge τ ersetzt werden soll. Die beste Übereinstimmung wird gefunden, indem der Nachfolgestringpräfix mit der kleinsten minimalen Distanz von einem Präfix der verbleibenden Eingabe der Länge τ gefunden wird.

Die Methode kann nun wie folgt beschrieben werden.

1. Der Eingabestring sei dargestellt als uv, wobei der Fehler direkt nach u auftritt und sei $v = v_1 \dots v_n$ für $v_i \in \Sigma, i = 1 \dots n$.
2. Sei $R = \{w | uw \in L, |w| < \sigma\} \cup \{w | uw \in Pre(L), |w| = \sigma\}$, wobei σ eine konstante Zahl ist.
3. Wähle x in R und $m, 1 \leq m \leq \tau$, so dass $d(v_1 \dots v_m, x) \leq d(v_1 \dots v_j, y)$ für alle $j, 1 \leq j \leq \tau$, und alle y in R, wobei τ eine konstante Zahl ist.
4. Der Reparaturstring ist $uxv_{m+1} \dots v_n$.

Die Methode welche in [WF74] beschrieben wird, wird benutzt um die minimale Distanz $d(u, v)$ zwischen zwei Strings $u = u_1 \dots u_m$ und $v = v_1 \dots v_n$ zu berechnen. Dabei entsteht eine $m \times n$ Matrix M, in der $M[i, j]$ die minimale Distanz zwischen $u_1 \dots u_i$ und $v_1 \dots v_j$ bezeichnet. Der Algorithmus wie er in [WF74] vorgeschlagen wird, berechnet die kleinsten Kosten. Er wird dahingehend verändert, dass alle edit-operationen einen Kostenwert von 1 erhalten. Somit ergibt sich eine Metrik wie sie in Definition 3 beschrieben ist.

6.4 Beispiel

Im diesem Beispiel, ausgehend von Grammatik 2.1, soll der folgende fehlerhaften Eingabestring korrigiert werden.

$$id * (\ id \ id \ id \ ) + id + id$$

Sei $\tau = 6$ und $\sigma = 3$, so sind die nächsten τ Eingabesymbole nach auftreten des Fehlers $id \ id \ ) \ + \ id \ +$. Zwei der möglichen Nachfolgestrings sind $+ \ id \ +$ und $+ \ id \ )$. Tabelle 6.1 zeigt wie die Minimum Distance Matrizen dieser beiden Strings bezüglich der noch verbleibenden Eingabe aussehen.

Obwohl die minimale Distanz zwischen $id \ id \ ) \ + \ id \ +$ und $\ + \ id \ +$ drei ist, weniger als die Distanz von vier zwischen $id \ id \ ) \ + \ id \ +$ und $\ + \ id \ )$, ist der letztere String

M_1	ϵ	id	id	)	+	id	+
ϵ	0	1	2	3	4	5	6
+	1	1	2	3	3	4	5
id	2	1	1	2	3	3	4
+	3	2	2	2	2	3	3

M_2	ϵ	id	id	)	+	id	+
ϵ	0	1	2	3	4	5	6
+	1	1	2	3	3	4	5
id	2	1	1	2	3	3	4
)	3	2	2	1	2	3	4

Tabelle 6.1: Minimum distance Matrizen

die bessere Reparatur, weil er eine Distanz von eins vom Präfix $id\ id\ )$ des verbleibenden Eingabestrings hat, was durch den kleinsten Eintrag in der letzten Zeile der beiden Matrizen signalisiert wird. Der String $+\ id\ )$ wird darum gewählt, um den Präfix $id\ id\ )$ der verbleibenden Eingabe zu ersetzen, was zu folgendem reparierten Eingabestring führt:

$$id\ *(id\ +\ id\ )\ +\ id\ +$$

Die Methode garantiert nicht, dass nach dem eingefügten Reparaturstring nicht wieder ein Fehler auftritt. Indem vorallem σ erhöht wird, kann das Ergebnis jedoch positiv beeinflusst werden. Die Menge der zu vergleichenden Nachfolgestringpräfixe wächst jedoch im schlimmsten Fall exponentiell bezüglich σ, was sich natürlich nachteilig auf die Geschwindigkeit der Fehlerbehebung auswirkt.

6.5 Implementierung

Bei der Implementierung stellen sich vorallem zwei Probleme :

1. Erzeugung der Nachfolgestringpräfixe.
   Sobald ein Fehler auftritt, wird die Fehlerbehandlungsroutine aufgerufen. Die Routine merkt sich den aktuellen Zustand des Parsers und versetzt den Parser zurück in den letzten Zustand bevor der Fehler aufgetreten ist. Anschliessend werden mit Hilfe der Parsetabelle rekursiv alle möglichen Nachfolgestringpräfixe der Länge σ erzeugt.

2. Bewertung der Nachfolgestringpräfixe.
   Die Berechnung der minimalen Distanz Matrix erfolgt wie schon erwähnt, nach einem Algorithmus von [WF74].Bis jetzt wurden für jede Edit-operation Kosten von 1 verrechnet. Dies ist aber in der Praxis vielfach keine gute Lösung. So ist zuim Beispiel die Wahrscheinlichkeit in einem Pascal-Programm grösser, dass einmal ein Symbol vergessen wird (ein Semikolon **;** oder ein **begin**), als dass zwei Symbole miteinander vertauscht werden. Darum ist es wünschenswert, dass das Einfügen eines Symbols weniger kosten sollte, als das Ersetzen eines Symbols mit einem anderen. Es wird darum eine Methode einführt, welche die Kosten einer Edit-Operation berechnet. Diese Methode kann dann je nach Belieben überschrieben werden, und

so, an die eigenen Bedürfnisse beziehungsweise an die der Sprache, angepasst werden.

Die Implementation wird in zwei Klassen vollzogen.

6.5.1 Klasse ErrorRecovery

Diese Klasse implementiert noch keine Fehlerbehandlung, sondern dient als abstrakte Basisklasse für konkrete Fehlerbehandlungsroutinen. Da sie als friend der Klasse LRParser definiert wird, hat sie Zugriff auf die internen Methoden dieser Klasse.

- **Attribute:**
  - LRParser* parser. Dieses Attribut ist private deklariert und verweist auf den Parser, zu dem ein Objekt einer konkreten Fehlerbehandlungsklasse die Fehlerbehandlung übernimmt.
- **Methoden:** Die nachfolgenden Methoden, definieren einerseits Zugriffsmethoden auf die Parsetabelle, den Tokenbuffer und den Scanner des Parsers aber auch eine Standardmethode welche vom Parser beim auftreten eines Fehlers aufgerufen wird.
  - char getAction(int s, terminal* t). Holt aus der Action-Tabelle von parser für den Zustand s und das Terminal t die auszuführende Aktion.
  - int getShift(int s, terminal* t). Dasselbe für die Shift-Tabelle.
  - production* getReduce(int s, terminal* t). Dasselbe für die Reduce-Tabelle.
  - int getJump(int s, nonterminal* t). Dasselbe für die Jump-Tabelle.
  - terminal* getterminal(). Ruft die gleichnamige Methode von parser auf.
  - void addToBuf(terminal* t). Fügt dem Terminalbuffer von parser ein Terminal hinzu.
  - ErrorRecovery(LRParser* p). Der Konstruktor verbindet die Fehlerbehandlung mit dem Parser.
  - virtual void recover(terminal* actTerminal,vector<int> stack) = 0. Definiert eine abstrakte Methode, welche von jeder konkreten Klasse überschrieben werden muss. Diese Methode dient dem Parser als Startmethode für die Fehlerbehandlung. Als Parameter werden das Terminal, welches den Fehler ausgelöst hat, sowie der aktuelle Zustandsstack übergeben. Der Zustandsstack wird als Wertparameter übergeben, weil er bei der Berechnung von Reparaturstrings verändert wird.

6.5.2 Klasse MinimumDistanceRecovery

Die Klasse wird von ErrorRecovery abgeleitet und implementiert, die, in diesem Abschnitt vorgestellte, Fehlerbehandlungsmethode. Die Klasse hat selber keinen direkten Zugriff auf den Parser, für welchen sie die Fehlerbehandlung übernimmt. Alle Zugriffe erfolgen über Methoden der Basisklasse ErrorRecovery (Abbildung 6.1).

- **Attribute:**
  - int sigma. Die Länge σ der betrachteten Reparaturstrings.
  - int tau. Die Länge τ des zu vergleichenden Eingabestrings.
  - int RepairDistance. Die aktuelle minimale Distanz des gefundenen Reparaturstrings
  - int RepairLength. Die aktuelle Länge der Reparatur.
  - vector<terminal*> repairStack. Enthält die Eingabe der Länge τ, die nach dem Auftreten des fehlerhaften Terminals folgt und korrigiert werden muss.
  - vector<terminal*> repairString. Der aktuell beste Reparaturstring.
  - enum editOperation {INSERT,DELETE,REPLACE }. Die drei möglichen Edit-Operationen.
- **Methoden:** Bis auf den Konstruktor und die recover-Methode handelt es sich um im folgenden um interne Methoden der Klasse.
  - int getCost(editOperation e,terminal* t1,terminal* t2). Eine Kostenfunktion, welche für die entsprechende Edit-Operation und die beteiligten Terminale die Kosten berechnet. Indem diese Funktion überschrieben wird, kann das Verhalten der Fehlerbehandlung angepasst werden.
  - minDist(vector<terminal*> &A,vector<terminal*> & B,int & dist,int & length). Berechnet die Distanzmatrix für die Terminalstrings A und B. Die minimale Distanz und die entsprechende Reparaturstring-Länge werden über die Parameter dist und length zurückgegeben.
  - GenerateRepair(vector<int> & Stack,vector<terminal*> & Continuation,terminal* spez). Die eigentliche Hauptroutine. Sie generiert rekursiv alle möglichen Reparaturstrings der Länge σ und vergleicht sie mit der tatsächlichen Eingabe der Länge τ, um den besten Reparaturstring zu finden.
  - MinimumDistanceRecovery(LRParser* p). Verknüpft den Parser mit der Fehlerbehandlung und initalisiert die Attribute sigma auf 4 und tau auf 8.
  - void recover(terminal* actTerminal,vector<int> stack). Die Standardaufrufmethode für den Parser, wie sie von der Basisklasse ErrorRecovery definiert wird. Sie füllt den repairStack mit den nächsten τ Symbolen der Eingabe[1] und startet die Fehlerbehandlung. Der gefundene Reparaturstring wird anschliessend über die Methode addToBuf(terminal* t) in den Terminalbuffer des Parsers geschrieben, von wo er die nächsten Terminals lesen wird.

[1] Die nächsten τ Terminale welche nach dem Terminal, welches den Fehler auslöst, folgen.

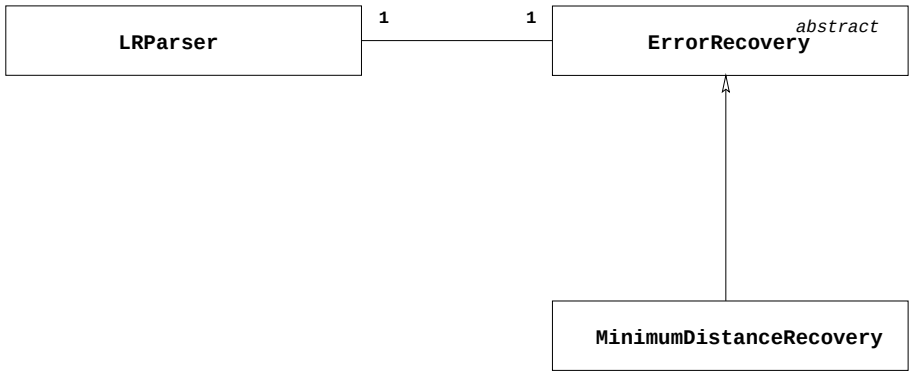

Abbildung 6.1: Eine konkrete Fehlerbehebung kann nur via das Interface der abstrakten Klasse ErrorRecovery auf den Zustand des Parsers zugreifen.

6.5.3 Erweiterungen

Eine konkrete Fehlerbehandlung orientiert sich an den Anforderungen, welche an den entsprechenden Compiler gestellt werden, sowie an den Voraussetzungen welche eine bestimmte Programmiersprache mitbringt. Die hier vorgestellte Fehlerbehandlung besticht vorallem durch ihre allgemeine Einsetzbarkeit. Es ist jedoch durchaus denkbar eine andere konkrete Fehlerbehandlung im Framework zu implementieren.

Dadurch dass die Basisklasse ErrorRecovery vollen Zugriff auf die Parsetabelle, sowie auf den Zustands- und Tokenstack des Parsers hat, kann fast jede andere Fehlerbehandlung implementiert werden.

Kapitel 7

Abstrakter Syntaxbaum

Die Bedeutung eines Programmkonstrukts einer Programmiersprache wird oft durch die Bedeutung der einzelnen Komponenten dieses Konstrukts beschrieben. Ein Baum verkörpert die Beziehung zwischen Konstrukten und ihren Komponenten, und darum wird ein Grossteil der Analyse, welche ein Compiler auf einem Program ausführt, in Form von Berechnungen über diesem Baum ausgedrückt. Obwohl die Baumstruktur in Relation zur Satzstruktur des Programms steht, sind die beiden im allgemeinen nicht identisch[KW95].

Beide, die Satzstruktur und die Baumstruktur, können durch kontextfreie Grammatiken definiert werden. Die Grammatik, welche die Satzstruktur beschreibt, wird als *konkrete* Syntax bezeichnet, während die Grammatik, welche die Baumstruktur beschreibt, als *abstrakte* Syntax bezeichnet wird.

Die konkrete Syntax kann als Eingabe für einen Parsergenerator dienen, welcher einen entsprechenden Parser erzeugt, während die abstrakte Syntax beschreibt, wie ein Syntaxbaum gebildet wird. Die abstrakte Syntax eliminiert den syntaktischen Overhead[1] der konkreten Syntax. Ein konkrete Grammatik für eine Liste von Terminals a kann wie folgt aussehen

$$
\begin{aligned}
P &\rightarrow aP' \\
P &\rightarrow \epsilon \\
P' &\rightarrow, aP' \\
P' &\rightarrow \epsilon
\end{aligned}
$$

wogegen ein abstrakter Syntaxbaum für die Liste aa wie in Abbildung 7.1 rechts aussieht.

Das in dieser Arbeit erstellt Framework soll den Compilerbauer bei der Erstellung des abstraken Syntaxbaums unterstützen. Syntaxbaumknoten können als Objekte bestimmter Syntaxbaumklassen betrachtet werden, welche Attribute und semantische Aktionen für diesen bestimmten Knoten implementieren. Da die Aktionen, welche die konkreten Syntaxbaumklassen implementieren, sehr sprachspezifisch sind, kann das Framework nur eine

[1] zum Beispiel Satzzeichen oder auch Schlüsselwörter

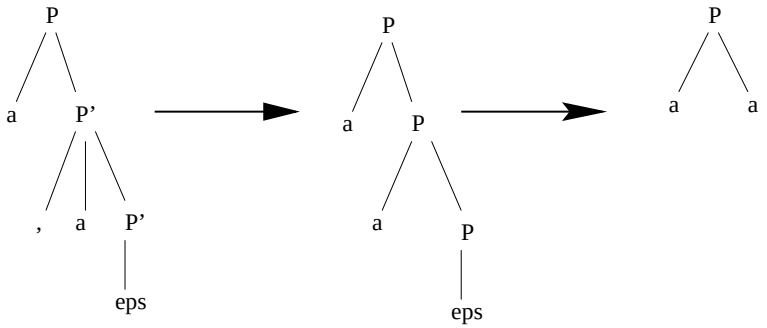

Abbildung 7.1: Vereinfachung von konkreter zu abstrakter Syntax

Basisklasse definieren, welche das grundsätzliche Verhalten einens Syntaxbaumknotens definiert.

Bei dieser Basisklasse handelt es sich um die Klasse node, deren Implementation im folgenden aus verschiedenen Gesichtspunkten betrachtet werden soll.

7.1 Erzeugung eines Syntaxbaumknotens

Jeder Syntaxbaumknoten hat genau einen Vaterknoten und einen oder mehrere Kinderknoten. Ausnahmen davon sind der Wurzelknoten, welcher keinen Vater besitzt und die Blattknoten, welche keine Kinderknoten haben. Dieser Sachverhalt ist in Abbildung 7.2 dargestellt. Damit wird beschrieben, wie die Syntaxbaumknoten einen Syntaxbaum bilden. Die Syntaxbaumklasse node wird in der Klassenhierarchie oberhalb der Klasse token eingefügt, damit die Terminale als Blätter im Syntaxbaum verwendet werden können.

Dieser hier vorgestellte Ansatz kann mit dem *Composite*-Pattern, wie es in [GHJV95] beschrieben wird, verglichen werden. Er unterscheidet sich dahingehend, als dass nicht zwischen *Composite*-Elementen mit Kindern und *Leaf*-Elementen ohne Kinder unterschieden wird.

Jeder Syntaxbaumknoten kann verschiedene Erzeugungsmethoden definieren.

- Erzeugungsmethoden sind statische Methoden einer Syntaxbaumklasse und besitzen alle das gleiche Format. Als Eingabe erwarten sie eine Liste mit den Kinderknoten und als Ausgabe produzieren Sie einen neuen Syntaxbaumknoten dieser Klasse.
- Erzeugungsmethoden definieren zudem die semantischen Aktionen, welche bei der Erzeugung eines Syntaxbaumknotens dieser Klasse ausgeführt werden sollen.

Diese Erzeugermethoden werden als Methodenzeiger den zugehörigen Produktionen zugewiesen. In Abbildung 7.3 ist dargestellt, wie die Erzeugung eines neuen Syntaxbaumknotens vor sich geht.

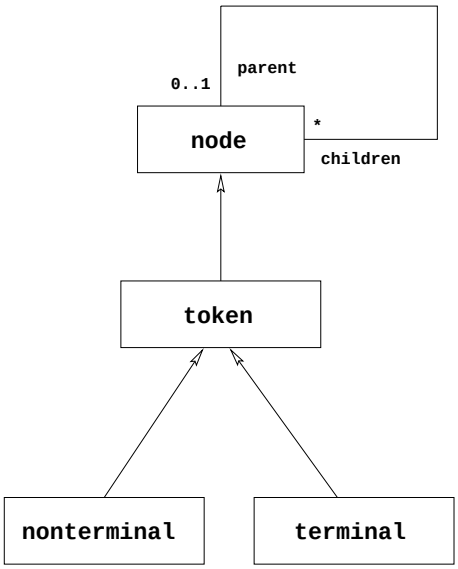

Abbildung 7.2: Die Syntaxbaumknotenklasse *node* mit der Vater und Kinderbeziehung

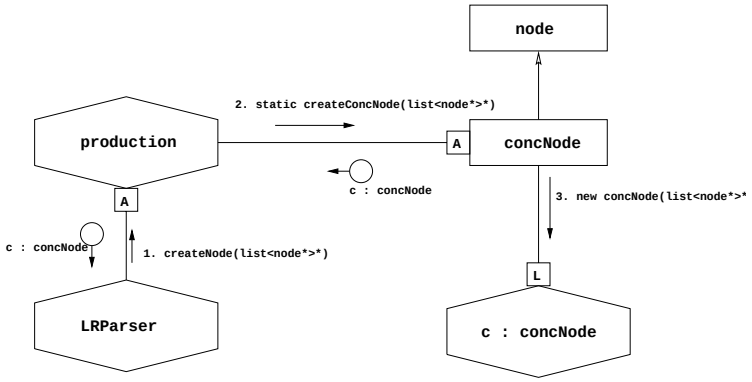

Abbildung 7.3: Verlauf einer Syntaxbaumknotenerzeugung

1. Der Parser nimmt die Elemente vom Tokenstack, die durch die Produktion reduziert werden. Er stellt sie zu einer Liste zusammen und übergibt sie der Methode createNode der entsprechenden Produktion.

2. Diese aktiviert die Methode auf welche der Methodenzeiger zeigt, in diesem Beispiel createConcNode, und reicht die Liste mit den vorher erstellten Kinderknoten weiter.

3. Die angesprochene Methode ist eine statische Methode einer Klasse, welche einen konkreten Syntaxbaumknoten implementiert. Die Methode ruft nun den Konstruktor ihrer Klasse auf, um eine neue Instanz zu erzeugen. Dieser ruft seinerseits den Konstruktor der node-Klasse auf und übergibt ihr die Liste mit den Kinderknoten. Der node-Konstruktor ist verantwortlich dafür, dass die Kinderknoten korrekt mit den neuen Knoten verknüpft werde. Die createConcNode-Methode führt anschliessend, wenn gewünscht, noch Aktionen oder Überprüfungen aus, bevor sie den neuen Knoten, als Rückgabewert der Produktion, dem Parser zurückgibt. Der Parser legt den Knoten auf den Tokenstack.

Diese concNode-Klasse ist nur ein Beispiel für eine wirkliche Syntaxbaumknotenklasse. Eine solche Syntaxbaumknotenklasse kann auch mehrere Methoden, wie sie createConcNode darstellt, anbieten. Das hängt davon ab, ob für einen Knoten verschiedene Verhalten implementiert werden sollen.

7.2 Methodenregistry

Das Konzept, wie ein Syntaxbaumknoten erzeugt wird, soll im folgenden verallgemeinert werden. Das Ziel ist es, die beiden Bereiche Syntaxanalyse und Syntaxbaumerzeugung beziehungsweise Attributwertberechnung möglichst voneinander zu trennen. Ein Parser kann aufgrund der syntaktischen Regeln einer Grammatik automatisch erzeugt werden. Das heisst, der Parser bleibt bis auf die Einträge in der Parsetabelle immer gleich, egal was für eine Sprache analysiert werden soll. Dies gilt mehr oder weniger für das ganze Frontend eines Compilers, da es nur nach lexikalischen und syntaktischen Regeln den Quelltext analysiert. Mit dem Übergang zum Backend eines Compilers beziehungsweise bei der Erzeugung des Syntaxbaums, kommen sprachspezifische Klassen in Form von Syntaxbaumklassen zum tragen, welche ein bestimmtes Element der Sprache implementieren. Dieses Interface zwischen Front- und Backend soll nun nicht einfach über einen Methodenzeiger, sondern assoziativ über einen Methodenname implementiert werden. Das heisst, die Produktion kennt die Methode, welche sie aufrufen muss um einen Syntaxbaumknoten zu erzeugen, nur beim Namen. Wenn der Ablauf in diesem Sinne verändert wird, ergibt sich daraus ein neuer Ablauf, wie er in Abbildung 7.4 dargestellt ist.

7.2.1 Klasse registry

Es wird eine neue Klasse registry definiert, in welcher die Erzeugermethoden der Syntaxbaumklassen via ihren Namen eingetragen werden können. Ein registry-Objekt wird mit

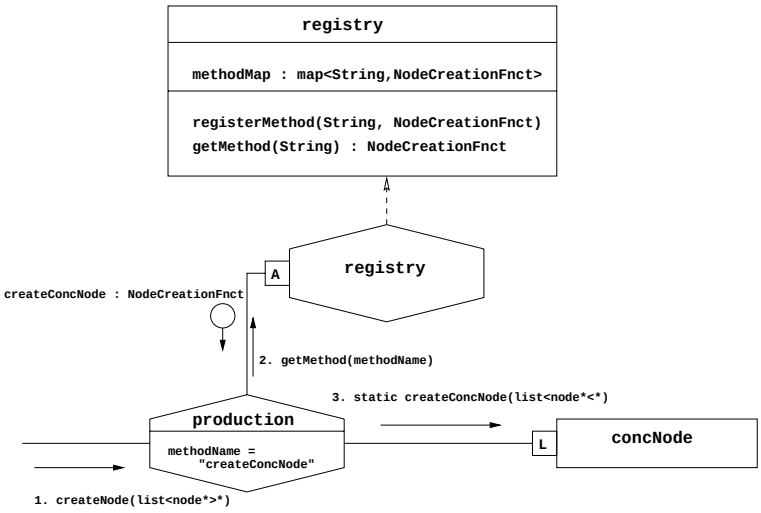

Abbildung 7.4: Ablauf der Syntaxbaumknotenerzeugung mit einer Methodenregistry

einem grammar-Objekt verbunden. Ein Produktionsobjekt kann dann via Grammatikobjekt in der Registry über einen Methoden-Namen, den entsprechenden Methodenzeiger finden. Im folgenden sind die Attribute und Methoden der Klasse beschrieben.

- **Attribute:**
  - typedef node* (*NodeCreationFnct)(vector<node*>* subtree). Definiton eines Methodenzeigers.
  - map<String,NodeCreationFnct> methodMap. Der assoziative Container, welche die Abbildung vom Namen einer Methode auf den entsprechenden Methodenzeiger vollzieht.
- **Methoden:**
  - registerMethod(const String& name, NodeCreationFnct ncf). Methode zum Eintragen eines Methodenzeigers.
  - NodeCreationFnct getMethod(const String& name). Methode zum Abfragen eines Methodenzeigers.

Für jeden Compiler oder Parser, welcher einmal implementiert werden soll, muss eine eigene Registry-Klasse von dieser Klasse registry abgeleitet werden. In dieser Klasse wird der Konstruktor überschreiben, in welchem alle Eintragungen der Methoden vorgenommen werden. Eine Instanz dieser Registry wird dem Grammatikobjekt übergeben, welche es den Produktionen wiederum erlaubt, die Registry zu befragen[2].

[2] Ein Beispiel findet sich in Abschnitt A.1

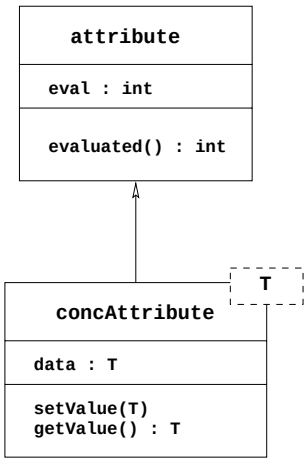

Abbildung 7.5: Attributklassen

7.3 Attribute

Jeder Syntaxbaumknoten kann mehrere Attribute definieren. Man unterscheidet dabei zwei Typen von Attributen.

- **berechnete Attribute**. Berechnete Attribute sind Attribute, die aus Kindknoten berechnet werden. Sie werden im Syntaxbaum von unten nach oben weitergegeben. In einem Bottom-Up Parser können sie direkt während der Syntaxanalyse berechnet werden.
- **vererbte Attribute**. Vererbte Attribute werden im Syntaxbaum von oben nach unten weitergegeben. Diese Attribute können erst berechnet werden, wenn die entsprechenden Attribute des Vaterknotens berechnet sind. Sie können während einer Bottom-Up Syntaxanalyse nicht direkt berechnet werden, sondern müssen wenn der Syntaxbaum einmal erstellt ist, in zusätzlichen Traversierungen ermittelt werden.

Um ein Attribut zu berechnen, müssen oft die Werte von anderen Attributen zur Verfügung stehen. Um zu entscheiden, ob ein gegebenes Attribut berechnet werden kann, müssen die Attribute, welche die Werte für die Berechnung liefern sollen, danach befragt werden können, ob ihre Werte schon berechnet wurden.

7.3.1 Implementation

Die Attribute sind in zwei Klassen implementiert (Abbildung 7.5). Die erste ist die Basisklasse attribute, welche nur den oben beschrieben Abfragemechanismus implementiert.

- **Attribute:**
  - int eval. Zeigt an, ob das Attribut bereits berechnet wurde.

- **Methoden:**
  - int evaluatet(). Methode für die Abfrage von eval.

Die zweite Klasse ist eine Template-Klasse concAttribute<T>, abgeleitet von attribute, welche mit dem Datentyp T des Attributes parametrisiert wird.

- **Attribute:**
  - T data. Der Wert des Attributs vom Typ T.
- **Methoden:**
  - setValue(T d). Methode zum Setzen des Wertes. Wenn der Wert gesetzt wird, wird automatisch das eval-Flag gesetzt.
  - T getValue(). Methode zur Abfrage des Wertes.

In der Attributwertberechnung wird sich die Frage stellen, wie Attribute aus fremden Syntaxbaumklassen angesprochen werden sollen. Es gibt zwei offensichtliche Lösungen.

1. Die Attribute werden öffentlich in der Syntaxbaumklasse deklariert, und können direkt angesprochen werden.
2. Die Attribute werden nicht öffentlich deklariert. Für jedes Attribut einer Syntaxbaumklasse werden eigene Zugriffsmethoden geschrieben.

Beide Lösungen sind nicht wirklich befriedigend. Die erste widerspricht dem Bestreben nach Datenkapselung, die zweite Lösung kapselt zwar die Daten, hat aber den Nachteil, dass für jedes Attribut immer wieder die gleichen Zugriffsmethoden implementiert werden müssen.

Das Framework bietet eine dritte Lösung an, welche in der Mitte der beiden anderen liegt. Dazu wird der *node*-Klasse eine sogenannte *Attributeregistry* hinzugefügt. Die Idee dahinter ist, dass ein Syntaxbaumknoten alle Attribute mit einem Attributnamen in seiner Attributregistry einträgt. Ein anderer Syntaxbaumknoten kann dann via den Attributnamen ein gewünschtes Attribut ansprechen. Dies geschieht, indem die Attributregistry anhand des Namens einen Zeiger auf das entsprechende Attribut zurückliefert. Wahrheitshalber muss hier zugegeben werden, dass die Datenkapselung so natürlich auch wieder verletzt wird. Dafür hat diese Lösung den Vorteil, dass ein Attribut dynamisch während der Laufzeit, aufgrund seines Namens, abfragt werden kann. Das kann zum Beispiel hilfreich sein, wenn man eine Sprache für semantische Aktionen einführen möchte, welche es notwendig macht, dass ein Attribut dynamisch ansprechbar ist.

Dieser ganze Attributmechanismus muss von einem Anwender nicht zwingend genutzt werden, da dieses Konzept nicht von der *node*-Klasse vererbt werden kann, weil die Attribute ja erst in den konkreten Syntaxbaumklassen definiert werden.

7.4 Attributwertberechnung

An einem kleinen Beispiel[PP92] soll betrachtet werden , wie die Attributwertberechnung durch das Framework unterstützt wird. Dazu wird eine Grammatik zur Berechnung von binären Festkomazahlen eingeführt.

$$
\begin{aligned}
N &\rightarrow S.S \\
S &\rightarrow SB \\
S &\rightarrow B \\
B &\rightarrow 0 \\
B &\rightarrow 1
\end{aligned}
$$

Das Startsymbol N repräsentiert die ganze binäre Zahl. N besitzt ein berechnetes Attribut v, welches den Wert der binären Zahl darstellt. Um dies darzustellen wird eine neue Notation eingeführt.

$$N \uparrow v$$

Der aufwärtsgerichtete Pfeil ($\uparrow$) bedeutet, dass das Attribut v ein berechnetes Attribut ist, das heisst, dass sich sein Wert aus den Attributen der Kinder von N berechnen lässt.

Das Nichtterminal B repräsentiert ein binäres Symbol (0 oder 1). Es hat ebenfalls ein Attribut v, welches den Wert bezeichnet. Der Wert, den ein einzelnes binäres Symbol zum Gesamtwert der binären Zahl beisteuert, hängt von seiner Position in der Zahl ab. Der Wert v von B wird mit einer Potenz von 2 skaliert, abhängig davon, wie weit entfernt sich das Symbol vom binären Punkt befindet. Dieser Skalierfaktor f kann nicht aus den Informationen des binären Symbols berechnet werden, sondern muss von den Eltern, in diesem Fall dem Nichtterminal S, vererbt werden. Das wird durch den abwärtsgerichteten Pfeil ($\downarrow$) signalisiert.

$$B \uparrow v \downarrow f$$

Das Nichtterminal S repräsentiert einen String von binären Symbolen. Es hat wie alle anderen Nichtterminale einen Attribut v, welches den Wert darstellt. Dazu kommt der Skalierfaktor f und eine Längenangabe l.

$$S \downarrow f \uparrow v \uparrow l$$

Im folgenden ist die oben aufgeführte Grammatik, mit den Attributen und den entsprechenden Berechnungsanweisungen erweitert, dargestellt.

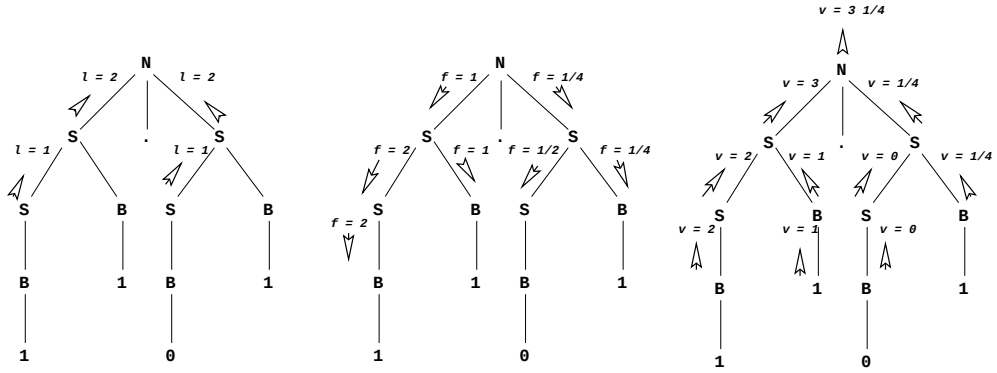

Abbildung 7.6: Der Fluss der Attribute bei einer Eingabe von 11.01

$$\begin{aligned}
\mathbf{N} \uparrow v \rightarrow \mathbf{S} \downarrow f_1 \uparrow v_1 \uparrow l_1 \ . \ \mathbf{S} \downarrow f_2 \uparrow v_2 \uparrow l_2 \qquad & [v = v_1 + v_2; f_1 = 1; f_2 = 2^{-l_2}] \\
\mathbf{S} \uparrow v \uparrow l \rightarrow \mathbf{S} \downarrow f_1 \uparrow v_1 \uparrow l_1 \ \ \mathbf{B} \downarrow f_2 \uparrow v_2 \qquad & [v = v_1 + v_2; f_1 = 2 * f; f_2 = f; \\
& l = l_1 + 1] \\
\mathbf{S} \uparrow v \uparrow l \rightarrow \mathbf{B} \downarrow f_1 \uparrow v_1 \qquad & [v = v_1; f_1 = f; l = 1] \\
\mathbf{B} \downarrow f \uparrow v \rightarrow 0 \qquad & [v = 0] \\
\mathbf{B} \downarrow f \uparrow v \rightarrow 1 \qquad & [v = f]
\end{aligned}$$

In Abbildung 7.6 ist dargestellt, wie sich der Fluss der Attributwerte bei einer Eingabe von 11.01 gestaltet. Da ein Bottom-Up Parser verwendet wird, wird der Syntaxbaum von unten gebildet. Das einzige Attribut, dass zu diesem Zeitpunkt berechnet werden kann, ist das Längenattribut l. Das Wertattribut v hängt vom Skalierfaktor f ab, welcher wiederum erst bei Erreichen des Startsymbols, das heisst, wenn die ganze Eingabe geparst ist, initialisiert wird. In einem zweiten Lauf kann dann von oben nach unten der Skalierfaktor f durchgereicht werden, bevor im dritten Lauf von unten nach oben der Wert v berechnet werden kann.

Dem Benutzer soll möglichst erspart werden, dass er alle Attributabhängigkeiten bis ins Detail kennen muss. Am Beispiel der folgenden Produktion aus der erweiterten Grammatik soll das Vorgehen erläutern werden.

$$\mathbf{N} \uparrow v \rightarrow \mathbf{S} \downarrow f_1 \uparrow v_1 \uparrow l_1 \ . \ \mathbf{S} \downarrow f_2 \uparrow v_2 \uparrow l_2 \qquad [v = v_1 + v_2; f_1 = 1; f_2 = 2^{-l_2}]$$

Die Berechnung kann in zwei Teile getrennt werden. Der erste Teil beinhaltet die Berechnung berechneter Attribute von N.

$$v = v_1 + v_2$$

Der zweite Teil umfasst alle Berechnungen die vererbte Attribute von Kindknoten betreffen

$$f_1 = 1; f_2 = 2^{-l_2}$$

Die berechneten Attribute eines Knotens können normalerweise erst berechnet werden, nachdem die Attribute der Kindknoten berechnet worden sind. Damit diese Kinder ihre Attribute berechnen können, muss ihnen die Attribute, welche dieser Knoten vererbt, weitergeben werden. Eine Pseudocoderoutine, die dieses Konzept ausdrückt, könnte so aussehen.

```
void node::evaluate() {
 calculateInheritedAttributes();
 for all children c do c.evaluate();
 calculateSynthetiziedAttributes();
}


class N : public node {
 ...
protected :
 concAttribute<float>* v;
 ...
public :
 void evaluateInheritedAttributes();
 void evaluateSynthesizedAttributes();
 ...
}

void N::evaluateInheritedAttributes() {
 concAttribute<float>* f1 = childNode(0)->getAttribute('f');
 concAttribute<float>* f2 = childNode(1)->getAttribute('f');
 concAttribute<int>* l2 = childNode(1)->getAttribute('l');
 f1->setValue(1);
 if (l2->evaluated())
  f2->setValue(power(2,-l2->getValue()));
}

void N::evaluateSynthesizedAttributes() {
 concAttribute<float>* v1 = childNode(0)->getAttribute('v');
 concAttribute<float>* v2 = childNode(1)->getAttribute('v');
 if (v1->evaluated() && v2->evaluated())
  v->setValue(v1->getValue() + v2->getValue());
}
```

In N::evaluateInheritedAttributes() werden die Skalierfaktorattribute f der beiden S Kinderknoten, die als childNode(0) und childNode(1) angesprochen werden, initialisiert, wobei f von childNode(1) vom Längenattribut l desselben Knotens abhängt. In N::evaluateSynthesizedAttributes() wird das Wertattribut v berechnet, was allerdings erst geschehen kann, wenn die entsprechenden Wertattribute v der beiden Kinderknoten berechnet wurden. node::evaluate() definiert das grundsätzliche Verhalten der Attributberechnung. Die beiden anderen Methoden, müssen für jede neue Syntaxbaumknotenklasse neu definiert werden.

7.5 Syntaxbaumtraversierung

Wenn ein Syntaxbaum einmal erzeugt wurde, möchte man auf ihm verschiedene Operationen (Codeerzeugung, Optimierung) ausführen oder Informationen (Crossreferenzen, Softwaremetriken) extrahieren. Dazu wäre es praktisch, wenn der Syntaxbaum auf verschiedene Arten traversierbar wäre. Dabei gibt es zwei sinnvolle Möglichkeiten

1. Traversierung in **Preorder**. Zuerst wird der Elternknoten besucht, anschliessend die Kinderknoten.
2. Traversierung in **Postorder**. Zuerst werden die Kinderknoten, dann der Elternknoten besucht.

Um von ausserhalb der Syntaxbaumklasse die Traversierung steuern zu können, wird ein Postorder- und einen Preorder-Iterator auf der Syntaxbaumklasse definiert. Wenn der entsprechende Iterator einmal initialisiert wurde, das heisst, auf den gewünschten Knoten im Baum zeigt, kann er durch Vorwärtsbewegung (*increment*) zum nächsten Knoten in der entsprechenden Ordnung bewegt werden. Diese Iteratoren ahmen in ihrem Verhalten die Iteratoren der *Standard Template Library* (**STL**) von C++ nach. Wenn die Variable Parsetree die Wurzel eines Syntaxbaums bezeichnet und für jede Syntaxbaumklasse eine Methode display() definiert wurde, welche den Namen der Syntaxbaumklasse ausgibt, dann könnte eine Methode, welche den Syntaxbaum grafisch darstellt, wie folgt aussehen.

```
node* Parsetree;
node::postIterator i;
⋮
Parsetree = Parser->start();
for (i = Parsetree->postBegin(); i != Parsetree->postEnd(); i++)
  (*i)->display();
```

Der Syntaxbaum wird dann in Postorder (Bottom-Up) ausgegeben. Um das gleiche Resultat nur in Preorder zu erreichen, müssen die entsprechenden Iteratoren und Initalisierungsmethoden (postBegin()) geändert werden.

7.6 Implementation der Klasse node

Die vollständige Definition der Klasse node präsentiert sich nun wie folgt:

- **Attribute:**
  - node* parentNode. Die Verknüpfung zum Vaterknoten. Falls es sich um den Wurzelknoten des Baumes handelt, ist parentNode 0.
  - vector<node*> childNodes. Der Vektor mit den Kinderknoten.
  - map<String,attribute*,less<String> > *attrRegistry. Die Attributregistry. In ihr können die Attribute eines Knotens mit ihrem Attributnamen eingetragen werden. Die Registry wird erst alloziert wenn mit registerAttr(const String& name, attribute* attr) das erste Attribut eingetragen wird. Das heisst, dass die Registry keinen[3] Speicherplatz verbraucht, solange sie nicht benötigt wird.
  - typdef vector<node*>::iterator childIterator. Definiert einen Iteratortyp für die Kinderknoten.
- **Methoden:**
  - int NumberOfChildren() const. Gibt die Anzahl Kinder eines Knotens zurück.
  - int isLeaf() const. Gibt an ob es sich bei diesem Knoten um einen Blattknoten handelt[4].
  - virtual int isTerminal() const. Gibt an ob es sich bei diesem Knoten um einen Terminalknoten handelt. Wird von terminal überschrieben.
  - int isComposite() const. Gibt an ob es sich bei diesem Knoten um einen inneren Knoten handelt[5]
  - attribute* getAttribute(const String& name) const. Gibt das Attribut mit dem entsprechenden Attributnamen zurück.
  - postIterator postBegin(). Gibt einen Iterator auf das erste Element in Postorder zurück.
  - preIterator preBegin(). Gibt einen Iterator auf das erste Element in Preorder zurück.

[3] beziehungsweise nur den Speicherplatz für die Zeigervariable
[4] Anzahl Kinder $= 0$
[5] Anzahl Kinder > 0

  - postIterator postEnd(). Gibt einen Iterator auf das Ende des Baumes in Postorder zurück.
  - preIterator preEnd(). Gibt einen Iterator auf das Ende des Baumes in Preorder zurück.

Die Klasse node definiert zudem interne Iteratorklassen.

7.6.1 Klasse Iterator

Iterator ist die abstrakte Basisklasse für konkrete Iteratoren.

- **Attribute:**
  - node* actNode. Der aktuelle Knoten auf den der Iterator zeigt.
  - int depth. Die Tiefe des Baumes an der Iteratorposition.
- **Methoden:**
  - int operator==(const Iterator& it) const. Vergleicht zwei Iteratoren, ob sie auf die gleiche Position im Baum zeigen.
  - Iterator& operator=(const Iterator& it). Zuweisungsoperator.
  - node* operator*(). Dereferenziert den Iterator und gibt einen Zeiger auf den aktuellen Knoten im Baum zurück.
  - int getDepth(). Gibt depth zurück.
  - virtual Iterator& operator++() = 0. Abstrakte inkrement Methode. Muss von den konkreten Iteratoren implementiert werden.

7.6.2 Klasse postIterator

postIterator ist eine abgeleitete Klasse von Iterator, welche den Baum in Postorder traversiert.

- **Methoden:**
  - postIterator(node* n). Initialisiert den Iterator auf einen bestimmten Knoten.
  - postIterator& operator++(). Der Iterator wird um eine Position in Postorder nach vorne geschoben.

7.6.3 Klasse preIterator

preIterator ist eine abgeleitete Klasse von Iterator, welche den Baum in Preorder traversiert.

- **Methoden:**

- preIterator(node* n). Initialisiert den Iterator auf einen bestimmten Knoten.
- preIterator& operator++(). Der Iterator wird um eine Position in Preorder nach vorne geschoben.

Kapitel 8

Objektspeicherung

8.1 Motivation

Im vorgestellten Framework gibt es Klassen, welche Methode definieren, die sehr rechenintensiv sind. Die folgenden drei Beispiele sollen dies illustrieren.

- Ein Objekt der Klasse grammar, welches die FIRST- und FOLLOW-Mengen seiner Nichtterminale berechnet.
- Ein Scanner, welcher aus verschiedenen DEA's aufgebaut wird, kann über die normalize Methode die einzelnen DEA's zu einem DEA verschmelzen.
- Ein Parser, der seine Parsetablle mit dem LALR(1)-Algorithmus berechnen lässt.

Man kann einem Anwender nicht zuzumuten, dass bei jeder Initialisierung eines Compilers diese Aktionen immer wieder ausführt werden. Es gibt grundsätzlich drei Möglichkeiten dies zu Umgehen.

1. **Quelltextgenerierung**. Ein Ansatz, welcher von den meisten Parserwerkzeugen gewählt wird. Nachdem die Parsetabelle berechnet wurde, wird ein **C**-Quelltext generiert, welcher die erzeugte Parsetabelle enthält. Die eigentliche Anwendung wird mit diesem Quelltext compiliert und gelinkt.
2. **Erweiterung einzelner Klassen**. Klassen, welche einmalige rechenintensive Aktion ausführen, werden mit der Fähigkeit ausgestattet, den berechneten Zustand abzuspeichern und wieder laden zu können.
3. **Objektspeichermechanismus**. Es wird ein genereller Objektspeichermechanismus eingeführt, welcher es erlaubt, den Zustand des ganzen Frameworks abzuspeichern.

In diesem Framework wurde aus folgenden Gründen ein Objektspeichermechanismus implementiert.

- **Allgemeinheit**. Die einzelnen Klassen müssen nicht mit spezifischen Methoden erweitert werden, die es ihnen erlauben, einmal berechnete Objektzustände abzuspeichern.
- **Erweiterbarkeit**. Nachdem das Konzept einmal im Framework eingeführt wurde, kann es auf jede beliebige neue Klasse angewendet werden.
- **Dynamik**. Das Framework kann dynamisch initialisiert und verändert werden.

Für den Objektspeichermechanismus gibt es viele interessante Anwendungen.

- Mit Hilfe des Frameworks kann ein Parsergenerator erstellt werde, welcher nicht wie übliche Parsergeneratoren ein **C-** oder **C++**-File mit dem berechneten Parser erstellt, sondern direkt Parser- und Scannerobjekte des Frameworks berechnet, und diese über den Speichermechanismus abspeichert. Die so berechneten Objekte können dann von einem eigentlichen Compiler geladen werden.
- Die Syntaxbaumknotenklassen können eine execute-Methode implementieren, welche, wenn auf dem Wurzelknoten aufgerufen, das durch den Syntaxbaum dargestellte Programm ausführt. Damit der Syntaxbaum nicht jedesmal neu vom Parser erstellt werden muss, kann dieser abgespeichert und in einer virtuellen Machine, welche nur die Syntaxbaumknotenklassen kennen muss, ausgeführt werden.

Der angesprochene Parsergenerator wurde mit diesem Framework erstellt, und wird im Anhang A vorgestellt.

8.2 Implementation

Es gibt verschiedene Ansätze für die Objektspeicherung. [Loo95] unterscheidet die folgenden Konzepte.

1. **File System Persistence**. Typischerweise sind die Transformationen zwischen Objekten und einer *flate-file* Repräsentation nicht trivial. Die Transformationslogik kann zum Beispiel ziemlich komplizierten Code enthalten, um die Referenzen unter den Objekten zu traversieren. Diese Referenzen werden durch Speicheradressen repräsentiert und müssen vom Speichermechanismus in eine andere Form gebracht werden, da diese, wenn die Objekte wieder geladen werden, normalerweise nicht mehr gültig sind. Weitere Komplikationen können entstehen, wenn Veränderungen an der Klassenstruktur vorgenommen werden und eine bestehende File-Repräsentation ungültig gemacht wird.
2. **Relational Database Persistence**. Dieser Methode kann mit dem Filesystem Ansatz verglichen werden, nur das diese Merthode unter Umständen noch mehr Transformationen erfordert. Der Ansatz erscheint verlockend, wenn im Umfeld der Anwendung bereits RDBMS verwendet werden. Ein RDBMS auferlegt dem Benutzer

jedoch zusätzliche Zwänge bezüglich der Datenrepräsentation, welche in der Transformationslogik berücksichtigt werden müssen. Eine Verbesserung wird mit erweiterten RDBMS erreicht, welche das Modell so erweitern, dass gewisse Aspekte von Objekten besser repräsentiert werden können. Dieser Ansatz hat den Vorteil, dass er einige der Transformationsverantwortungen in der DBMS selber versteckt.

3. **Object Database**. Objekte können in einer Objektdatenbank gespeichert werden, ohne dass spezieller Transformationscode geschrieben werden muss. Objektdatenbanken vereinigen den Begriff des *in-memory* und des *persistent* Speichermodells in einem konsistenten single-level Speicher. Weil die ODBMS und die objektorientierte Programmiersprache das gleiche Typsystem und Objektmodell verwenden, muss der Programmierer keine speziellen Transformationen implementieren.

Die Implementation für den Objektspeichermechanismus welcher in diesem Framework benutzt wurde, stammt aus [CN93] und entspricht im grossen und ganzen dem *File System Persistence* Ansatz. Es wäre unter Unständen besser gewesen eine Objektdatenbank zu verwenden, aber da für diese Arbeit kein nichtkomerzielles ODBMS zur Verfügung stand und auch nicht die ganze Funktionalität eines ODBMS benötigt wird, wurde der im folgenden vorgestellte Ansatz gewählt.

Der Ansatz benutzt für die Speicherung der Objekte ein einfaches Textfile. Ein weiterer Hauptpunkt besteht darin, dass der ganze Speichermechanismus weitmöglichst von den eigentlichen Frameworkklassen, im folgenden als PD[1]-Klassen bezeichnet (analog zu [CN93]), getrennt wird. Das hat den Vorteil, dass der Speichermechanismus einfach ausgetauscht werden kann und die PD-Klassen nicht mit zusätzlicher Funktionalität erweitert werden müssen.

Im folgenden werden die Hauptklassen, die zu diesem Speichermechanismus gehören, betrachtet (Abbildung 8.3).

PDCObject : Ist die Basisklasse für alle Klassen, welche speicherbar sein sollen. Sie führt ein Attribut objectID ein. Das heisst, jedes Objekt, das von einer Unterklasse von PDCObject erzeugt wird, erhält zur Identifikation eine Nummer.

ObjectTable : Ist eine Tabelle, welche alle Objekte enthält, die gespeichert werden sollen. Ein Eintrag besteht aus einer Nummer, der objectID, und einem Zeiger auf das entsprechende Objekt.

ObjectTagFormat : Ist die Basisklasse für alle Formatklassen. Für jede PD-Klasse muss eine Formatklasse definiert werden, welche weiss, wie sie den Zustand und die Assoziationen der entsprechenden PD-Klasse speichern und wieder herstellen kann.

ClassMap : Jeder PD-Klasse wird ein eindeutiger Code zugewiesen (eine konstante Zahl). Die ClassMap weiss zu welcher PD-Klasse welche Formatklasse gehört. Die ClassMap erzeugt am Anfang für jede PD-Klasse eine Instanz der entsprechenden Formatklasse, welche dann via den Klassencode abgefragt werden kann.

[1] Problem Domain Class

Darüber hinaus weiss die ClassMap, wie eine Instanz einer PD-Klasse erzeugt wird.

In der Objektspeicherung werden drei Hauptaktionen unterschieden.

Registrierung: Damit die Objekte gespeichert werden können müssen sie zuerst in einer Objekttabelle registriert sein (Abbildung 8.4). Wenn ein PD-Objekt die Anweisung bekommt, sich zu registrieren, trägt es sich in der Objekttabelle ein und gibt die Registrierungsanweisung an alle mit ihm verknüpften Objekte weiter. Wenn wir eine Objekthierarchie mit einem Root-Objekt haben, reicht es aus, diesem die Registrieranweisung zu geben. Die restlichen Objekte werden dann automatisch, wie oben beschrieben erreicht.

Speicherung: Wenn einmal alle Objekte registriert wurden, reicht ein einfacher Speicherbefehl an die Objekttabelle aus, um den aktuellen Zustand des Frameworks zu speichern (Abbildung 8.5). Der grundsätzliche Vorgang bei der Speicherung ist, dass ein PD-Objekt die ClassMap nach seiner Formatklasse befragt. Wenn es diese einmal hat, gibt es ihr den Auftrag sich,das PD-Objekt, zu speichern. Zuerst wird die Objekttabelle gespeichert, anschliessend alle registrierten Objekte.

Wiederherstellung: Der Wiederherstellungsprozess ist um einiges komplizierter (Abbildung 8.6). Zuerst muss eine Objekttabelle und ein Klassenverzeichnis erzeugt werden. Anschliessend wird der Objekttabelle der Auftrag erteilt, den Zustand des Frameworks anhand einer Datei wieder herzustellen. Zuerst wird die Objekttabelle beziehungsweise deren Einträge eingelesen. Jedesmal, wenn ein Eintrag, der ja aus einem Klassencode und einem Objektidentifer besteht, eingelesen wird, wird dem Klassenverzeichnis der Auftrag erteilt, ein Objekt dieser Klasse mit dem entsprechenden Objektidentifer zu erzeugen. Wenn so alle Einträge gelesen wurden, sind alle Objekte wieder erzeugt. Da die Objekte jetzt noch leer sind, müssen ihre Zustände rekonstruiert werden. Dies geschieht indem die Objekttabelle jedem Objekt den Auftrag gibt, sich wieder herzustellen. Das Objekt fordert vom Klassenverzeichnis die entsprechende Formatklasse an und gibt dieser den Befehl, seinen Zustand aus der entsprechenden Datei einzulesen und wiederherzustellen.

8.3 Beispiel

An einem kurzen Beispiel soll gezeigt werden, wie eine Repräsentation eines Objektes in der Objektspeicherdatei aussieht. Das Objekt, welches hier benutzt wird, ist eine Instanz der Klasse nonterminal. Der Name des Nichtterminals ist *TokenDecl* und seine Objektnummer ist 735. Die FIRST-Menge des Nichtterminals besteht aus den Objekten mit den Nummern 124,118,112,106,100 und 94. Die FIRST-Menge ist ein Beispiel dafür, wie Assoziationen gespeichert werden. In einem Nichtterminal wird die FIRST-Menge als Liste von Zeigern auf Terminale verwaltet. Wenn das Nichtterminal gespeichert wird, wird anstatt des Zeigers der Objektidentifier des entsprechenden Terminals benutzt. Wenn das Nichtterminal wieder hergestellt wird, kann anhand des gelesenen Objektidentifiers in der

Objekttabelle, ein Zeiger auf das entsprechende Terminalobjekt geholt werden. Auf diese Weise werden die Assoziationen wieder hergestellt.

```
 @@ObjectDefn#
@@className#nonterminal
@@objectID#@@id#735
@@Name#TokenDecl
@@origin_id#@@id#735
@@first_set#@@id#124
@@first_set#@@id#118
@@first_set#@@id#112
@@first_set#@@id#106
@@first_set#@@id#100
@@first_set#@@id#94
@@ObjectDefn#
```

Abbildung 8.1: Repräsentation eines Nichtterminal-Objektes

```
Compiler_Domain_Class_Map::Compiler_Domain_Class_Map() {
  formats.push_back(new Object_Table_Format());
  formats.push_back(new StateMachine_Format());
  formats.push_back(new Scanner_Format());
  formats.push_back(new Stringtok_Format());
⋮
};
```

Abbildung 8.2: Ein Ausschnitt aus dem Konstruktor von ClassMap des Frameworks

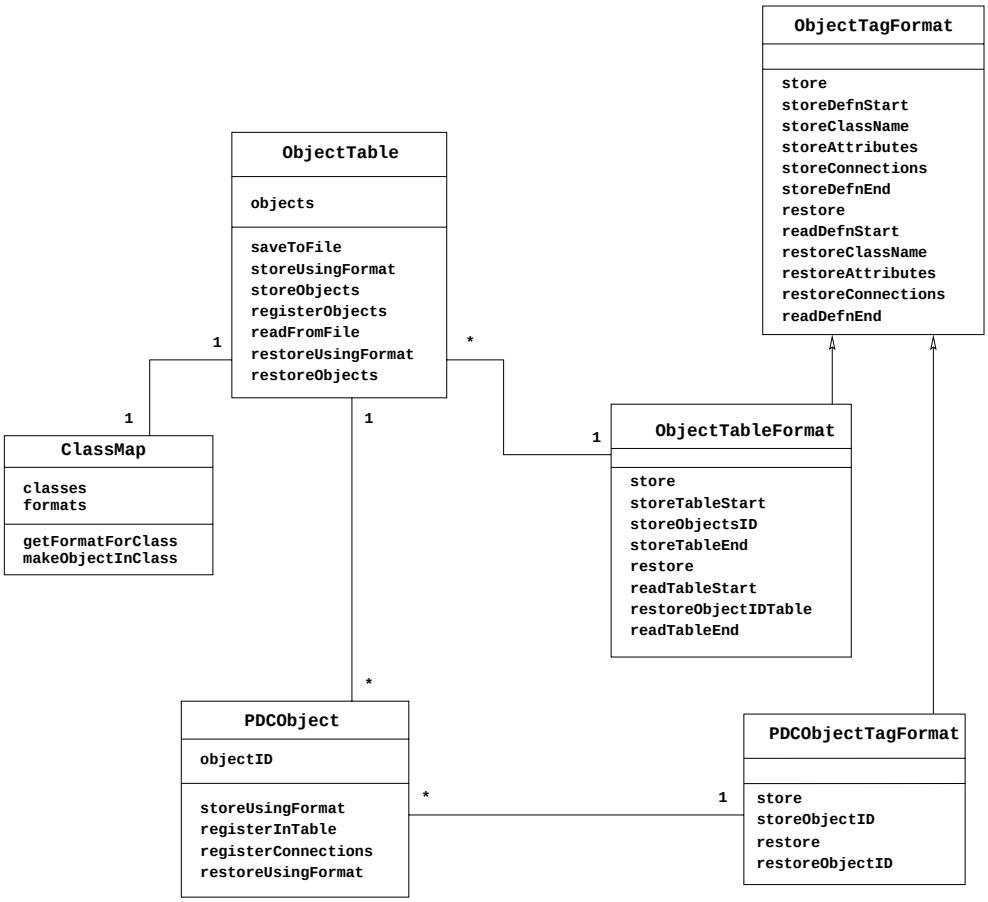

Abbildung 8.3: Klassenhierarchie der Objektspeicherung

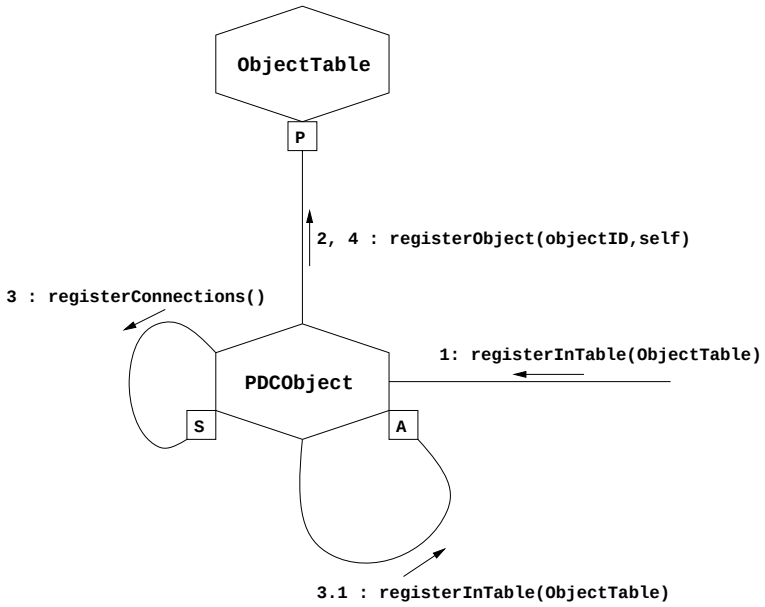

Abbildung 8.4: Registrierung eines Objektes in der Objekttabelle

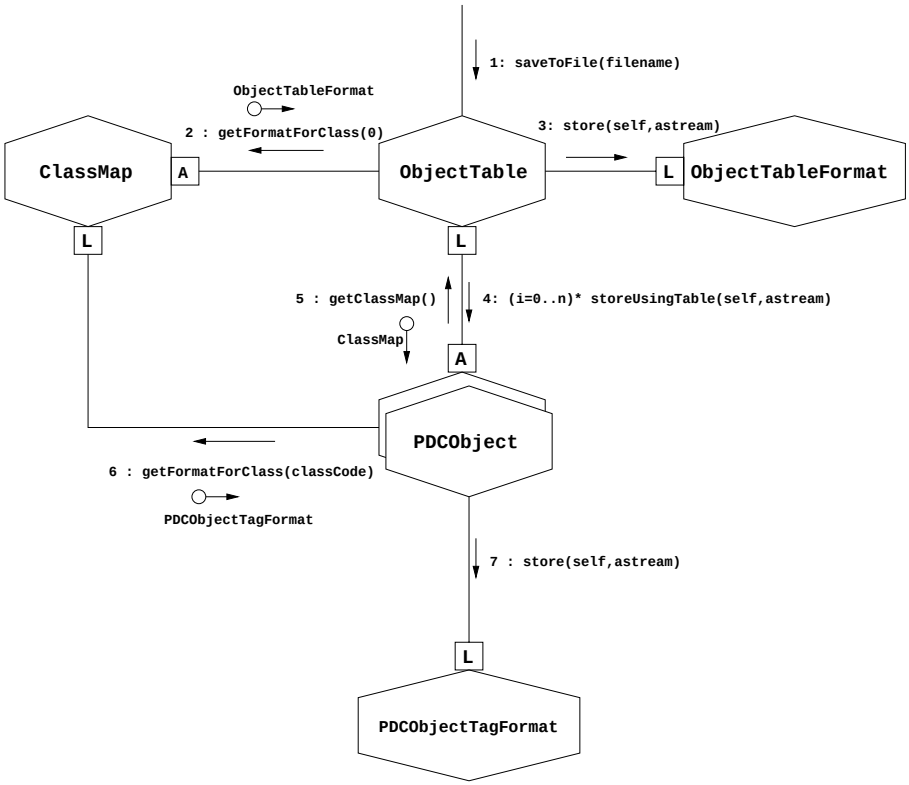

Abbildung 8.5: Speicherung der Objekte

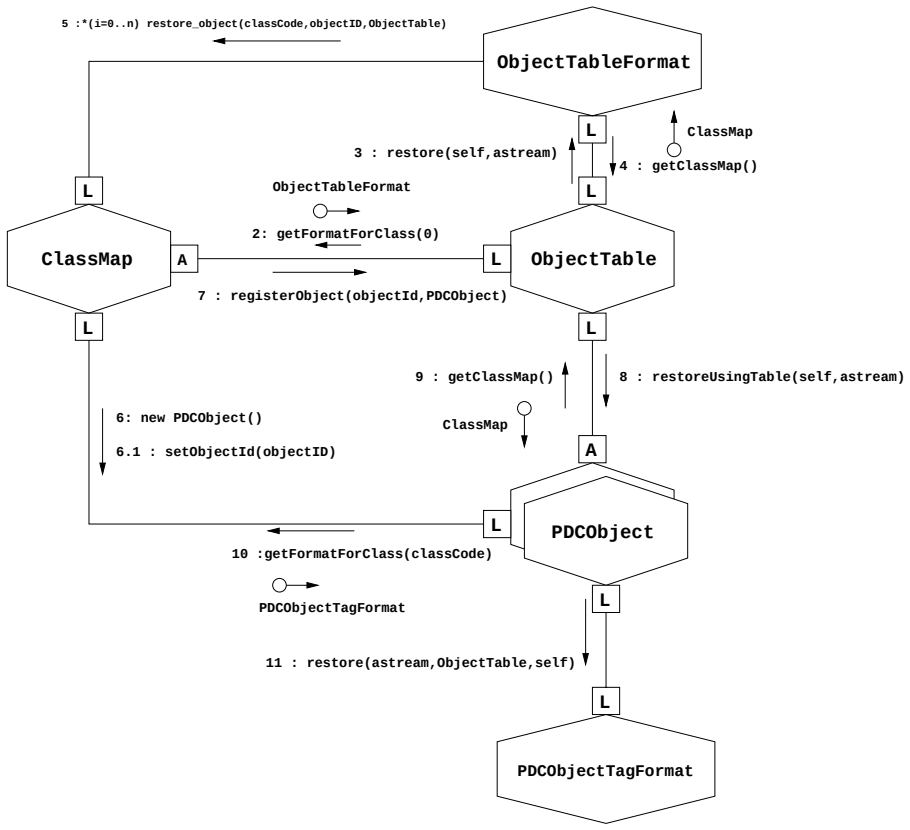

Abbildung 8.6: Wiederherstellung der Objekte

Kapitel 9

Diskussion

In diesem Kapitel sollen die Ansätze dieser Arbeit mit anderen Arbeiten auf diesem Gebiet verglichen werden. Dabei wird ein Hauptgewicht auf den Vergleich mit dem Buch *Object-oriented compiler construction* von Jim Holmes[Hol95] gelegt. In einem späteren Abschnitt werden auch noch weitere Arbeiten auf diesem Gebiet betrachtet.

9.1 Der Ansatz von Jim Holmes

Holmes stellt in seinem Buch die Implementation eines Pascal-Compilers vor. Seine Arbeit kann darum nur beschränkt mit dieser Arbeit verglichen werden, da Holmes eine konkrete Sprache implementiert, wogegen diese Arbeit ein möglichst allgemeines Framework zu beschreiben versucht.

Als Implementationssprache verwendet Holmes **C++**. Scanner und Parser werden mit den Werkzeugen Lex und Yacc erstellt. Der Hauptteil seiner Arbeit liegt in der Entwicklung der abstrakten Syntaxbaumklassen. Wie auch andere Autoren [Mal94] sieht er das Hauptanwendungsgebiet objekt-orientierter Methoden im Compilerbau in der Entwicklung einer geeigneten Klassenhierarchie zur Repräsentation des abstrakten Syntaxbaums (ASB).

Er entwickelt für jeden Knotentyp des ASB eine eigene C++-Klasse (Abbildung 9.2). Der dabei gewählter Ansatz kann mit dem *Interpeter-Pattern* [GHJV95] (Abbildung 9.1) verglichen werden. Das Pattern wird dabei nicht nur für die Interpretation, sondern auch für die Optimierung und die Codeerzeugung verwendet. Dies geschieht, indem jeder Klasse neben einer execute()- auch eine emitt()- und eine optimize()-Operation hinzugefügt wird.

Als Basisklasse für alle ASB-Klassen dient die Klasse PTreeNodeCls. Von dieser Klasse[1] werden dann die konkreten ASB-Klassen abgeleitet. Dabei gibt es Klassen wie StatementCls, deren Instanzen als Kinder von StatementSeqCls auftreten. Da die Anzahl dieser Kinder variabel ist, wird StatementCls zusätzlich von einer Klasse LstSeqBldrCls abgeleitet, welche eine Listenstruktur implementiert. StatementSeqCls

[1] Diese Basisklasse wird erstaunlicherweise nicht abstrakt definiert. Wenn eine abgeleitete Klasse zum Beispiel die Methode optimize() nicht definiert, wird dieser Fehler zur Übersetzungszeit nicht erkannt.

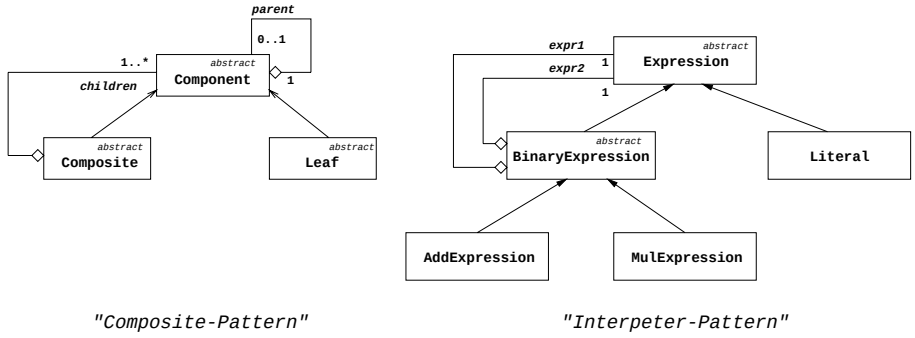

Abbildung 9.1: Composite- und Interpeter-Pattern nach [GHJV95]

muss sich dann nur noch den Anfang head und tail seiner Liste von Kindern merken. Andere Klassen wie IfStmtCls haben eine konstante Anzahl von Kindern. Eine ähnliche Aufteilung in Knoten mit einer konstanten Anzahl von Kindern und in Knoten mit einer variablen Anzahl von Kindern nimmt auch [Mal94] vor (Abbildung 9.3).

Das Interpeter-Pattern ist mit dem Ansatz von Jim Holmes durchaus vereinbar, da er eine konkrete Sprache implementiert und nicht vorsieht, dieses Design für eine andere Sprache wiederzuverwenden. Hier liegt auch der Hauptnachteil des Interpeter-Patterns. Die Verknüpfung zwischen Kinder- und Elternknoten wird in den konkreten ASB-Klassen explizit definiert, das heisst, jede Klasse gibt an wieviele Kinderknoten sie erwartet und von welchem Typ diese sind. Dadurch muss eine Traversierung des ABS für jede Klasse neu implementiert werden.

Das Composite-Pattern (Abbildung 9.1), welches in dieser Arbeit gewählt wurde, definiert die Kinder-Eltern-Beziehung schon in den abstrakten Basisklassen[2]. Somit kann die Traversierung direkt in der Basisklasse implementiert werden. Der Nachteil dieses Designs ist, dass wenn ein Kindknoten als Instanz seiner konkreten Klasse angesprochen werden soll, ein Downcast gemacht werden muss.

Holmes verwendet für die lexikalische und syntaktische Analyse Lex und Yacc. Er begründet den Einsatz dieser Werkzeuge vorallem damit, dass diese Werkzeuge einfach sind und sich vielfach bewährt haben. Eine weitere Begründung welche Holmes für den Gebrauch von Yacc anführt, ist die, dass sich die LALR(1)-Methode, wie sie Yacc anwendet, besonderns gut für eine objekt-orientierte Implementation eines Compilers eigne. Diese Aussage lässt er aber komplett unmotiviert. Man könnte seine Aussgage auf zwei Arten interpretieren.

1. Ein Parser welcher objekt-orientiert implementiert wird, sollte vorzugsweise die LALR(1)-Methode wählen, weil sich diese besser mit dem objekt-orientierten Ansatz verträgt als zum Beispiel die LL(1) Methode.

[2] Die Kinder eines Knoten werden als Liste von Zeigern auf die abstrakte Klasse Component implementiert

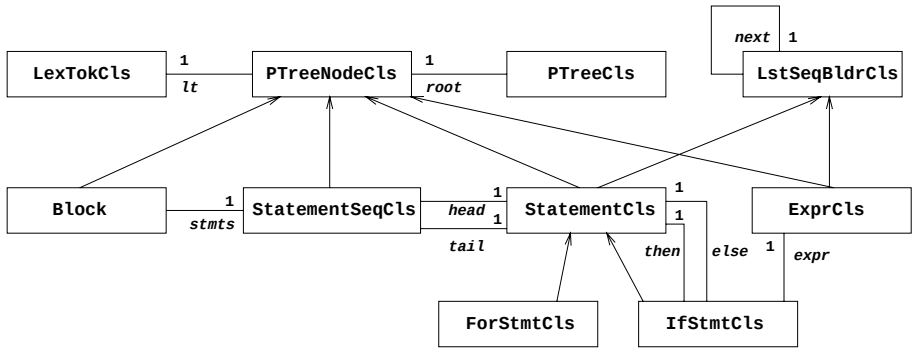

Abbildung 9.2: Ein Ausschnitt aus der ASB-Klassenhierarchie aus [Hol95]

2. LALR(1)-Parser sind besser gegeignet die Erzeugung eines objekt-orientiert implementierten ASB's zu unterstützen.

Beide Interpretationen werden von Terence Parr in seinem Buch über Pccts[Par96] wiederlegt. Pccts kann sogenannte *pred-LL(k)* Grammatiken verarbeiten und erzeugt als Ausgabe eine C++-Parserklasse in welcher für jede Produktion der Grammatik eine Methode implementiert wird. Indem Pccts semantische und syntaktische Prädikate in der Grammatik erlaubt, ist Pccts mächtiger als Yacc und nicht anfällig auf Konflikte erzeugt durch in einer Produktion plazierte Aktionen[3]. Die Parserklasse LRParser welche in dieser Arbeit beschrieben wurde, implementiert zwar ebenfalls ein LALR(1)-Parser, der Entscheid für diese Methode wurde jedoch nur aufgrund der Tatsache, dass mehr Grammatiken die LALR(1)-Eigenschaft als die LL(1)-Eigenschaft haben, gefällt. Ansonstens muss gesagt werden, dass der Ansatz, der von Parr gewählt wurde, objekt-orientierter erscheint als der tabellenbasierte Ansatz des LRParser.

Holmes versucht in seiner Arbeit, den von Yacc erzeugten Parser in einer Klasse zu kapseln. Dies aufgrund der Überlegung, dass ein Compiler manchmal aus mehreren Parsern[4] bestehen kann. Da Yacc und Lex globale Variabeln und Funktionen benötigen, um miteinander zu kommunizieren, ist dies nicht ganz trivial. Holmes bleibt den Beweis, dass sein Ansatz funktioniert, jedoch schuldig, da sein implementierter Pascalcompiler nur einen Parser benötigt. Wenn die Schnittstelle zwischen Scanner und Parser und die Schnittstelle zwischen Parser und Backend betrachtet werden, erscheint seine Lösung zumindest zweifelhaft. Die Scannerdefinition für Lex sieht eine globale Pointervariable auf eine Instanz von LexTokCls (Abbildung 9.2) vor, welche benötigt wird, um die vom Scanner erzeugten Instanzen dieser Klasse an den Parser zu übergeben. Abgesehen davon, dass ein objekt-orientiertes System keine globalen Variabeln verwenden sollte, bedingt diese Art der Kommunikation auch, dass das so übergebene Element sofort im ASB verarbeitet werden muss, da es sonst vom nächsten lexikalischen Element überschrieben wird. Ein ähnliches Vorgehen wählt er bei der Kommunikation zwischen Parser und

[3] Eine Aktion kann zum Beispiel die Erzeugung eines ASB-Knotens umfassen

[4] Zum Beispiel ein Parser für den Präprozessor und einen für die eigentliche Sprache

Backend. Der Synatxbaum wird vom Parser über eine globale Pointervariable der Klasse PTreeCls an das Backend übergeben.

Diese Kommunikation über globale Variabeln ist ein Beispiel für den Paradigmen-Konflikt, welcher in einem solchen System auftreten kann.

Im Gegensatz dazu wird im in dieser Arbeit vorgestellten Framework die Kommunikation zwischen Scanner und Parser über Terminalobjekte geführt, welche die gesamte lexikalische Information kapseln und je nach Terminaltyp auch schon gewisse Attributwertberechnungen vornehmen können. Die Terminalobjekte werden vom Parser auch direkt im Syntaxbaum verarbeitet, und müssen nicht wie bei Holmes in einer abstrakten Syntaxbaumklasse gekapselt werden.

Weiter beschreibt Holmes in seiner Arbeit die Generierung von Assemblercode über die emitt-Methode seiner Parsebaumklassen und geht auf einige Optimierungsmethoden wie den River-Algorithmus und den Greedy-Algorithmus[SU70] ein. Diese Teile seines Buches sind jedoch vorallem Compilertheorie und bringen keine spezifisch objektorientierten Methoden hervor.

Das Buch von Holmes muss als ein Versuch der Kombination zweier Vorlesungen über C++ und Compilerbau betrachtet werden. Da beide Gebiete von Grund auf behandelt werden, darf man nicht erwarten, dass wirklich neue Erkenntnise gefunden werden, was wahrscheinlich auch nicht die Absicht des Autors war. Die Stärken des Buchs liegen eindeutig bei der Beschreibung der Evaluation der abstrakten Syntaxbaumklassen.

9.2 Andere verwandte Arbeiten

Die meisten Arbeiten auf diesem Gebiet befassen sich mit der Repräsentation des ASB durch Objekte. Einige Arbeiten befassen sich aber auch mit der objekt-orientierten Umsetzung von Scanner und Parser oder der Codegenerierung und Optimierung. In den folgenden drei Abschnitten werden einige dieser Arbeiten angesprochen.

9.2.1 Scanner und Parser

In [Par96] wird der Parsergenerator ANTLR als Teil von Pccts beschrieben. ANTLR generiert aus einer sogenannten *pred-LL(k)* Grammatik eine C++-Parserklasse, in welcher die einzelnen Produktionen der Grammatik als Methoden der Klasse implementiert sind. Zusätzlich erzeugt der Scannergenerator DLG eine Scannerklasse als deterministischen endlichen Automaten. Je ein Objekt jeder dieser beiden Klassen bilden zusammen das Frontend. ANTLR und DLG sind in C geschrieben und unterstützen erst ab Version 1.20 C++-Output.

[Mal94] betrachtet die Compilation aus der Sicht des ASB. Der ASB wird in seinem Ansatz von einem Parserobjekt erzeugt, das jedoch nicht weiter definiert wird.

Yacc++ [Zin94] ist eine objektorientierte Neuimplementation von Lex und Yacc. Yacc++ ist ein Generator welcher aus einer Grammatik je eine C++-Klasse für Scanner und Parser erzeugt. Dabei kann der Anwender unter drei verschiedenen Parserklassen auswählen, von welcher sein Parser abgeleitet werden soll. Zur Auswahl stehen eine Par-

serklasse mit einer schnellen Tabelle, mit einer lesbaren Tabelle und mit einer kleinen Tabelle. Der Scanner kann mit verschiedenen Inputobjekten verknüpft werden, je nachdem, woher die zu analysierende Eingabe kommt.

Flex++, Bison++ sind Erweiterungen von Lex und Yacc, welche C++-Klassen erzeugen. Der von diesen Werkzeugen erzeugte Quelltext kann aber seine C-Vergangenheit nicht verleugnen.

In SMALLTALK [GR83] wird eine Scanner- und Parserklasse definiert, welche vom System selber verwendet wird, um den Quelltext zu übersetzen. Dabei wird die Parserklasse von der Scannerklasse abgeleitet. Ralph Johnson[JF88] betrachtet dies als einen typischen Fall für die missbräuchliche Verwendung von Veerbung. Besser wäre es, den Scanner als Komponente des Parsers zu betrachten.

Meyer[Mey94] beschreibt die Scanner- und Parserbibliothek von EIFFEL. Die Scannerbibliothek enthält verschieden Automatenklassen, eine Tokenklasse und Klassen zur Berechnung und Betreibung von Scannern. Die Parserbibliothek bietet Klassen an, welche die Erstellung eines Parsers , welcher den rekursiven Abstieg implementiert, ermöglichen. Die Scannerbibliotheken benutzt den Speichermechanismus der Eiffel-Bibliothek, um einmal berechnete Scanner zu speichern und wieder zu laden.

9.2.2 Abstrakter Syntaxbaum

Der ASB wird von vielen Autoren als zentraler Punkt in einem objekt-orientierten Compiler betrachtet. Im ASB können einige objekt-orientierte Konzepte umgesetzt werden.

1. Vererbung. Jeder Knoten eines ASB ist ein Objekt einer bestimmten ASB-Klasse. Eine entsprechende Klassenhierarchie ist in Abbildung 9.3 dargestellt.

2. Polymorphisums. In der Basisklasse des ASB können Operationen definiert werden, welche von jeder konkreten ASB-Klasse implementiert werden müssen. So kann auf einem ASB eine Operation ausgeführt werden.

3. Datenkapselung. Jede ASB-Klasse definiert ihre eigenen Attribute. Die Informationen werden dezentral in den Knoten des Syntaxbaums verwaltet.

In der Arbeit von [HS94] wird an einem Beispiel erläutert, wie Yacc mit C++ eingesetzt werden kann. Dabei wird auch eine ASB-Klassenhierarchie entwickelt, die mit dem *Composite-Pattern*[GHJV95] verglichen werden kann.

In der Zusammenfassung des *OO Compilation Workshops* anlässlich von OOPSLA 94[Cla94] wird die Traversierung und die Ausführung von Operationen auf abstrakten Syntaxbäumen diskutiert. Mit dem gleichen Thema beschäftigt sich [Rob94]. Auch in der USENET Diskussionsgruppe comp.compiler konnte in den Jahren 1994 und 1995 Diskussionen über die verschiedenen Konzepte verfolgt werden. Dabei werden drei verschiedene Konzepte unterschieden[Cla94].

1. Rekursion. Die einzelen ASB-Klassen definieren die Traversierung rekursiv. Meist wird die Rekursion für jede Operation, welche auf dem ASB ausgeführt wird, einzeln implementiert.

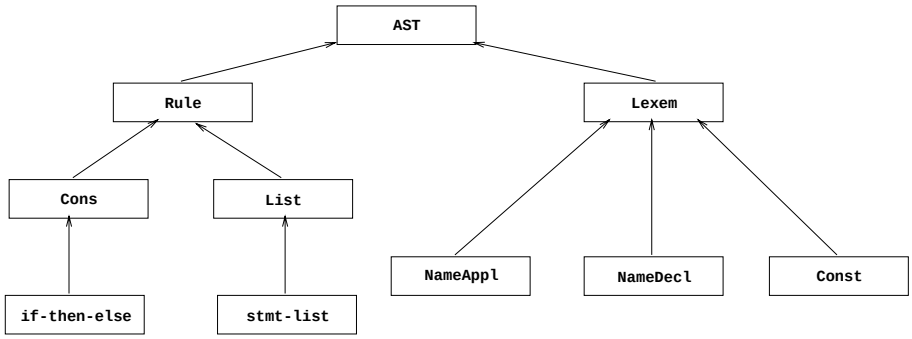

Abbildung 9.3: Die ASB-Klassenhierarchie wie sie von [Mal94] vorgeschlagen wird.

2. Iteratoren. Auf dem ASB werden verschiedene Iteratoren definiert (Preorder, Postoder). Die Traversierung kann so von den eigentlichen Operationen getrennt werden.

3. Vistors. Entsprechen dem *Visitor-Pattern* wie es in [GHJV95] beschrieben ist. Dieses Pattern kommt dann zum Einsatz, wenn die Operationen nicht in den ASB-Klassen selber sondern ausserhalb definiert werden sollen.

Die Entscheidung wo die Operationen auf dem ASB definiert werden sollen, basiert vorallem auf der Überlegung, ob vermehrt die Struktur des ASB's verändert wird (spricht für die Implementation der Operationen in den einzelnen ASB-Klassen), oder ob immer wieder neue Operationen auf dem ASB definiert werden (spricht für das Vistor-Konzept).

9.2.3 Codegenerierung und Optimierung

Hier muss vorallem das RTL-System [Joh94] erwähnt werden. RTL ist ein Framework für Code Optimierung und Generierung, basierend auf einer *Register Transfer Language*. Das RTL System ist ein Toolkit für den Bau von Codeoptimierern, gleichzeitig definiert es eine Zwischensprache für die Programmdarstellung und ist eine grosse wiederverwendbare Klassenbibliothek, welche in Smalltalk implementiert ist.

Kapitel 10

Schluss

In dieser Arbeit wurde der Entwurf und die Implementation eines Compilerframeworks vorgestellt. Der Fokus wurde dabei auf das Frontend gelegt, das heisst lexikalische und syntaktische Analyse sowie die Syntaxbaumerzeugung. Spezielle Lösungen wurde dabei in der Behandlung von Shift/Reduce-Konflikten sowie in der Fehlerbehandlung gefunden.

Durch den Einsatz objekt-orientierter Prinzipien wurde eine Framework geschaffen, welches auf verschieden Arten einsetz- und erweiterbar ist. Dabei wurde vorallem versucht die Komponenten so zu entwerfen, dass auch andere, als die im Framework konkret implementierten Lösungen, gewählt werden können.

10.1 Erkenntnise

In diesem Abschnitt werden die bei der Erstellung des Frameworks gemachten Erfahrungen und Erkenntnisse nocheinmal zusammengefasst.

- Eine objekt-orientierte Implementation von Scanner und Parser fokusiert vorallem auf Datenkapselung, so dass gleichzeitig mehrere Parser- beziehungsweise Scannerinstanzen ohne Konflikte betrieben werden können. Darüber hinaus vereinfacht sie den Austausch von einzelnen Komponenten, namentlich die Fehlerbehandlung und den Algorithmus zur Parsetabellenberechnung.

- Dadurch das die lexikalischen Elemente als Objekte im Scanner erzeugt werden, sind die lexikalischen Informationen direkt in diesen Objekten gekapselt und müssen nicht über globale Elemente ausgetauscht werden. Die Terminalobjekte können schon erste Attributwertberechnungen vornehmen und werden vom Parser im abstrakten Syntaxbaum als Blätter verwertet.

- Die Zusammenfassung aller lexikalischen und syntaktischen Elemente in einem Grammatikobjekt, erlaubt es, Umformungen und Berechnugen auf der Grammatik zu vollziehen, unabhängig von der gewählten Syntaxanalysemethode. Scanner- und Parserobjekte können zudem über ein Grammatikobjekt initialisiert werden.

- Die LALR(1)-Methode, welche in diesem Framework für die Syntaxanalyse gewählt wurde, eignet sich für die meisten Programmiersprachen und ist der LL(1)-Methode überlegen. Der tabellenbasierte Ansatz hat den Vorteil, dass die Tabelle berechnet werden kann, ohne den eigentlichen Parser zu verändern. Parser, welche durch rekusiven Abstieg funktionieren, müssen von Hand erstellt werden, oder von einem Generator als Quelltext generiert werden.

- Dynamische Konfliktlösung verschiebt den Zeitpunkt der Lösung von shift/reduce-Konflikten auf die Syntaxanalyse. So können Konflikte gelöst werden, welche durch Opertoren enstehen, deren Priorität und Assoziativität erst in der lexikalischen Analyse bestimmt werden kann. Herkömmliche Methoden, wie sie Yacc implementiert, können derartige Konflikte nicht lösen.

- Die in diesem Framework implementierte konstrukt-orientierte Fehlerbehandlung nach [Dai] erfordert vom Benutzer keine Änderungen an der Grammatik. Sie ist ideal für den Einsatz in einem Framework, da sie ohne Änderung sofort eingesetzt werden kann.

- Das *Composite-Pattern*, welches im abstrakten Syntaxbaum zum Einsatz kommt, ermöglicht die Implementation von Traversierungen (Rekursion, Iteratoren) direkt in den Basisklassen. Zudem erleichtert es die Konstruktion eines ASB, da auch die Verknüpfung von Eltern- und Kinderknoten in den Basisklassen implementiert werden kann.

- Durch die Objektspeicherung wird die Compilierung, wie sie bei Parsergeneratoren die Quelltext generieren, umgangen. Dadurch kann vermieden werden, dass ein Compiler bei einer kleinen Änderung der Syntax, welche den ASB nicht betrifft, neu compiliert werden muss. Da die Objektspeicherung in einer Textdatei erfolgt, wird zudem Plattform-Unabhängigkeit erreicht.

10.2 Zukünftige Arbeiten

Das Framework kann auf vielen Ebenen verbessert und ausgebaut werden.

- Der Scanner erwartet den zu analysierenden Quelltext in einer Datei. Durch Einführung von Inputklassen, wie sie von [Zin94] vorgeschlagen werden, könnte dieser Aspekt des Frameworks sinnvoll erweitert werden.

- Die DEA's, welche zu einem Scanner zusammengebaut werden, müssen von Hand erstellt werden. Besser wäre es wenn sie über reguläre Ausdrücke automatisch erstellt werden könnten.

- Auf die Terminalklassen könnte das *Flyweight-Pattern*[GHJV95] angewandt werden, um den Namen des Terminals speicherschonender zu implementieren.

- Scanner und Parser könnten über ein Pipeobjekt miteinander verbunden werden, wie es in [Par96] vorgeschlagen wird. Neben der Möglichkeit, dass die Pipe gleichzeitig ein Buffer darstellt der dem Parser einen Lookahead ermöglicht, kann sie auch vervielfältigt werden, um zum Beispiel mehrere Parser mit dem gleichen Scanner zu verbinden.

- Das dem abstrakten Syntaxbaum zugrunde liegende *Composite-Pattern* wurde nicht vollständig implementiert. Die Unterscheidung in *Composite*- und *Leaf*-Knoten wurde nicht explizit gemacht, das heisst, die Terminalobjekte, welche als Blätter im ASB erscheinen, können theoretisch auch Kinderknoten haben.

Anhang A

Beispiele

Das folgende Beispiel soll den richtigen Einsatz des in dieser Arbeit vorgestellten Frameworks demonstrieren. Anhand eines Beispiels wird ein Einblick in die Funktionsweise und das Zusammenspiel der verschiedenen Klassen gegeben. Am Schluss dieses Kapitels sollte ein potentieller Benutzer selber in der Lage sein, eine Anwendung zu schreiben.

A.1 Parsergenerator

Ein Parsergenerator erleichter den Umgang und den Einstieg in das Framework enorm. Hinzu kommt, dass ein Parsergenerator eine Anwendung des Frameworks darstellt und somit gleichzeitig ein Beispiel für dessen Einsatz ist.

Herkömmliche Scanner- beziehungsweise Parsergeneratoren produzieren aus einer Sprachdefinitionsdatei eine Quelltextdatei in einer gewissen Sprache[1], welche dann mit der eigentlichen Anwendung kompiliert und gelinkt werden muss. Der Ansatz, welcher hier gewählt wird, unterscheidet sich grundsätzlich von der oben beschriebenen Vorgehensweise.

Der hier vorgestellte Parsergenerator liest auch eine Sprachdefinitionsdatei, bei deren Interpretation werden dann aber direkt Objekte aus Klassen des Frameworks erzeugt und berechnet. Wenn die Berechnung beendet ist, besteht eine komplette Objekthierarchie, welche einen Parser der gewünschten Sprache repräsentiert. Diese so erzeugten Objekte werden dann über den Objektspeichermechanismus in einer Datei gesichert. Der eigentliche Parser kann die entsprechende Datei einlesen, die Objekte wieder herstellen und einen Quelltext der definierten Sprache analysieren.

In einem ersten Abschnitt wird die Sprache vorgestellt, welche für die Beschreibung einer Grammatik benutzt wird.

A.1.1 Definition der Parsergeneratorsprache

Da sich die Parsergeneratorsprache nicht von einer Sprache unterscheidet, kann sie mit sich selbst definiert werden. Diese Definition sieht wie folgt aus.

[1] meist in C bzw. C++

```
%%
Id "$abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
"_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
String;
Number;
Keyword "Keyword";
Keyword "String";
Keyword "Char";
Keyword "Number";
Keyword "Id";
Keyword "SymbolicId";
Keyword "->";
Keyword "%%";
Keyword ";";
Keyword ",";
Keyword "(";
Keyword ")";
Keyword "precedence";
Keyword "left";
Keyword "right";
Keyword "none";
Terminal;
%%
S -> '%%' TokenDecls '%%' Productions '%%' Synchronisation '%%';
createGenGrammar($2,$4,$6);
TokenDecls -> TokenDecl ';' TokenDecls; createTokenDecls($1,$3);
TokenDecls -> ; createTokenDecls();
TokenDecl -> 'Keyword' 'string' Associativ ; createKeywordDecl($2,$3);
TokenDecl -> 'Keyword' 'string' 'precedence' 'number' Associativ ;
createKeywordDeclPrece($2,$4,$5);
Associativ -> 'left' ; createAssocLeft();
Associativ -> 'right' ; createAssocRight();
Associativ -> 'none' ; createAssocNone();
Associativ -> ; createAssocNone();
TokenDecl -> 'Id' 'string' 'string' ; createIdentDecl($2,$3);
TokenDecl -> 'Id' ; createIdentDecl();
TokenDecl -> 'Number' ; createNumberDecl();
TokenDecl -> 'String' ; createStringDecl();
TokenDecl -> 'Char' ; createCharDecl();
TokenDecl -> 'SymbolicId' ; createSymbolicDecl();
Productions -> Production ';' Productions ; createProductions($1,$3);
Productions -> ; createProductions();
Production -> 'id' '->' ListOfTokens ';' ParseTreeCrt ;
createProduction($1,$3,$5);
ParseTreeCrt -> 'id' '(' ListOfId ')' ; createParseTreeCrt($1,$3);
ParseTreeCrt -> 'id' '(' ')' ; createParseTreeCrt($1);
ListOfId -> 'id' ; createListOfId($1);
ListOfId -> ListOfId ',' id ; createListOfId($1,$3);
ListOfTokens -> 'terminal' ListOfTokens ;
createListOfTokensfTerminal($1,$2);
ListOfTokens -> 'id' ListOfTokens ; createListOfTokensfNonTerm($1,$2);
```

```
ListOfTokens -> ; createListOfTokensfTerminal();
%%
```

Im späteren Verlauf dieses Beispiels wird die genaue Definition und Bedeutung dieser Sprache ersichtlich. Die Definitionsdatei ist primär in zwei Abschnitte eingeteilt. Im ersten werden die Terminale und im zweiten die Produktionen der Grammatik definiert. Die Abschnitte werden durch das Symbol `%%` voneinander getrennt.

A.1.2 Grammatik

Um einen ersten Parsergenerator zu erhalten, muss die oben beschrieben Sprache von Hand implementiert werden. Das heisst, es muss ein Grammatikobjekt mit den entsprechenden Terminalen, Nichterminalen und Produktionen erzeugt werden. Dazu wird von der Grammatikklasse grammar des Frameworks eine Grammatikklasse genGrammar abgeleitet. In dieser Klasse werden dann alle nötigen Objekte der Grammatik erzeugt. Im folgenden Quelltextauszug ist dies dargestellt:

```
genGrammar::genGrammar() {
  addTerminal(
   new identtok("$abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ",
   "_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"));
  addTerminal(new stringtok());
  addTerminal(new numbertok());
  addTerminal(new keywordtok("Keyword"));
  addTerminal(new keywordtok("String"));
  addTerminal(new keywordtok("Id"));
  addTerminal(new keywordtok("Terminal"));
  addTerminal(new keywordtok("->"));
  addTerminal(new keywordtok("precedence"));
  addTerminal(new terminaltok());
  addTerminal(new endtoken());
  addTerminal(new epsilontoken());
⋮
  nonterminal* Ss = new nonterminal("S'");
  addNonterminal(Ss);
  addNonterminal(new nonterminal("S"));
  addNonterminal(new nonterminal("TokenDecls"));
⋮
  setStartSymbol(Ss);
  createProduction("S' -> S ");
  createProduction("S -> %% TokenDecls %% Productions %% Synchronisation
%% ",    "createGenGrammar($2,$4,$6)");
  createProduction("TokenDecls -> TokenDecl ; TokenDecls ",
   "createTokenDecls($1,$3)");
  createProduction("TokenDecls -> epsilon ",    "createTokenDecls()");
⋮
  Registry = new genRegistry();
```

```
  calculateFIRST();
}
```

Dies ist nur ein Teil des Originalquelltexts, der fehlende Teil kann analog der vorher vorgestellen Sprachdefinition ergänzt werden.

Terminale

Zuerst werden die benötigten Terminale erzeugt und direkt über die Methode addTerminal() in den assoziativen Terminalcontainer der Grammatik eingetragen. Von den Terminalklassen identtok, stringtok und numbertok wird genau eine Instanz erzeugt, von der Terminalklasse keywordtok mehrere Instanzen, für jedes Schlüsselwort und jedes Symbol der Sprache eine. Zusätzlich zu den eben erwähnten Terminalen muss jede Grammatik zwei ausgezeichnte Terminale definieren, nämlich ein Epsilon-Terminal epsilontok , sowie ein Terminal endtoken , welches vom Scanner produziert wird, um zu signalisieren, dass das Ende der Eingabe erreicht wurde.

Nichtterminale

Als nächstes werden die Nichtterminale erzeugt. Dies geschieht, indem für jedes Nichtterminal eine Instanz der Klasse nonterminal generiert und mit dem entsprechenden Namen initialisiert wird. Die Nichtterminale werden gleich wie die Terminale in einen assoziativen Container des Grammatikobjekts eingetragen (addNonterminl(nonterminal* n)). Da die LALR(1)-Methode benutzt wird, um die Parsetabelle zu berechnen, muss die Eingangs vorgestellte Grammatik um die Produktion `S' -> S` erweitert werden[2]. Das Nichtterminal S' wird durch das nonterminal Ss repräsentiert, welches über die Methode setStartSymbol() als Startsymbol ausgezeichnet wird.

Produktionen

Nachdem alle Terminale und Nichtterminale definiert wurden, wird dasselbe mit den Produktionen getan. Die Klasse grammar bietet dafür die Methode createProduction(const String& prod, const String& NodeCrtFnct) an. Sie erwartet zwei Stringparameter, welche die Produktion sowie die mit der Produktion assozierte Syntaxbaumknoten-Erzeugungsfunktion definiert. Der String, der die eigentliche Produktion definiert, wird zerlegt und die einzelnen Elemente in den beschriebenen assoziativen Containern nachgeschlagen. Die so erhaltenen Zeiger auf Terminalobjekte und Nichtterminalobjekte werden dann zu einem Produktionsobjekt zusammengesetzt[3]. Das Produktionsobjekt wird in einer Liste productions der Gramatik eingetragen. Der zweite String definiert einerseits den Namen der Funktion, welche bei der Erzeugung eines Parsebaumknotens aufgerufen werden soll, andererseits auch die Parameter, welche übergeben werden. Wenn die Parameterliste zum Beispiel die Einträge $1,$3 und hat, heisst das, dass der Funktion das erste

[2] für eine Erklärung siehe [ASU92, 100-104]

[3] siehe auch Abschnitt 2.2

und das dritte Element der rechten Produktionsseite, übergeben wird. So können Symbole oder auch Schlüsselwörter, welche nur für die Syntaxanalyse benötigt werden, von der Weiterverarbeitung im Parsebaum ausgeschlossen werden. Der Name wird als String und die Parameterliste als Liste von Integern der entsprechenden Produktion übergeben. Diese kann dann, wenn sie einen Parsebaumknoten erzeugen muss, die entsprechende Methode in der Methodenregistry Registry der Grammatik nachschlagen.

Registry

Diese Registry muss ebenfalls erzeugt und in der Grammatik eingetragen werden. Wie schon in Abschnitt 7.2 erkläutert, muss dazu eine eigene Registry-Klasse implementiert werden. Ein Auszug aus dem Quelltext für die Klasse genRegistry soll die Vorgehensweise erläutern:

```
typedef node* (*NodeCreationFnct)(vector<node*>* subtree);

class genRegistry : public registry {
  genRegistry();
}

genRegistry::genRegistry() {
  registerMethod( "createKeywordDecl",
    (NodeCreationFnct)TokenDecl::createKeywordDecl);
  registerMethod( "createKeywordDeclPrece",
    (NodeCreationFnct)TokenDecl::createKeywordDeclPrece);
  registerMethod( "createStringDecl",
    (NodeCreationFnct)TokenDecl::createStringDecl);
  registerMethod( "createNumberDecl",
    (NodeCreationFnct)TokenDecl::createNumberDecl);
⋮
}
```

Im Konstruktor der Klasse genRegistry werden über die Methode registerMethod(String,NodeCreationFnct) die statischen Methoden eingetragen, welche die Erzeugung und Initalisierung eines bestimmten Parsebaumknotens ermöglichen.Über den Namen, welcher als String der Methode übergeben wird, kann später der entsprechende Methodenzeiger referenziert werden.

Am Schluss wird die Methode calculateFIRST() aufgerufen, welche die FIRST-Mengen der Nichtterminale und Produktionen berechnet.

Das Scanner- und das Parserobjekt können nun direkt über dieses Grammatikobjekt initialisiert werden. Als nächstes müssen die Klassen des abstrakten Syntaxbaums definiert werden

A.1.3 Abstrakte Syntaxbaumklassen

Für jedes Nichtterminal wird eine äquivalente ASB-Klasse definiert. Anhand der Klasse TokenDecl soll die Vorgehensweise betrachtet werden.

Klasse TokenDecl

Das entsprechende Nichtterminal TokenDecl hat acht verschiedene Produktionen. Als erstes soll berechnet werden, wieviele Erzeugermethoden für diese ASB-Klasse definiert werden müssen. Dabei können verschiedene Taktiken anwandt werden. Einerseits kann eine einzige Methode definiert werden, welcher alle Elemente der rechten Seite mitgeben werden. Die Methode muss dann selbst herausfinden, von welcher Produktion sie aufgerufen wurde. Ein mögliche Implemetation dieser Taktik ist unten angeführt. Dazu werden zwei der oben erwähnten acht Produktionen ausgewählt. Bei der Reduktion dieser Produktionen wird jeweils die gleiche Erzeugermthode createTokenDecl der Parsebaumklasse TokenDecl aufgerufen. Damit die beiden Produktionen in der ASB-Klasse unterschieden werden, muss das Element der rechten Produktionsseite mitgeben werden. Die Methode prüft dann dieses Element und erzeugt entsprechend eine Instanz der Klasse stringtok beziehungsweise der Klasse numbertok.

```
TokenDecl -> 'String' ; createTokenDecl($1);
TokenDecl -> 'Number' ; createTokenDecl($1);
⋮

TokenDecl* TokenDecl::createTokenDecl(vector<node*>* subtree) {
  decl = new TokenDecl(subtree);
  terminaltok* art = (terminaltok*) decl->getChildNode(0);
  if (art->name = 'String')
    decl->t = new stringtok();
  else if (art->name = 'Number')
    decl->t = new numbertok();
}
```

Wir erachten diese Taktik als falsch, da sie die Information, welche bei der Syntaxanalyse gewonnen wird, wieder aufgibt und die Erzeugermethode diese Arbeit wiederholen muss. Im folgenden wird die ganze Parsebaumklasse TokenDecl betrachtet wie sie effektiv implementiert wurde.

```
class TokenDecl : public node {

protected:
  terminal* t;

  TokenDecl(vector<node*>* subtree) : node(subtree) { };
  ~TokenDecl() { delete t; }
```

```
public:
  static TokenDecl* createKeywordDecl (vector<node*>* subtree);
  static TokenDecl* createKeywordDeclPrece (vector<node*>* subtree);
  static TokenDecl* createStringDecl (vector<node*>* subtree);
  static TokenDecl* createNumberDecl (vector<node*>* subtree);
  static TokenDecl* createIdentDecl (vector<node*>* subtree);
  static TokenDecl* createSymbolicDecl (vector<node*>* subtree);
  static TokenDecl* createCharDecl (vector<node*>* subtree);
  static TokenDecl* createTerminalDecl (vector<node*>* subtree);
  terminal* getTerminal () { return t; }

};
```

Die Klasse hat ein Attribut t, einen Zeiger auf ein terminal, welches je nach aufgerufener Methode erzeugt wird. Für dieses Attribut wird eine Zugriffsmethode getTerminal() definiert. Dabei fällt auf, dass der in Abschnitt 7.3 beschriebene Attributmechanismus nicht angewendet wird. Das kommt daher, dass der ganze Syntaxbaum, mit den Attributen, direkt während der Syntaxanalyse erstellt und berechnet werden kann. Das bedeutet, dass davon ausgegangen werden kann, dass jedes Attribut, auf das zugegriffen werden soll, bereits berechnet ist. Dadurch wird die Attributverwaltung überflüssig. Die ABS-Klasse definiert einen Konstruktor, welcher in dieser Art in jeder ABS-Klasse vorkommt. Dieser Konstruktor wird nicht direkt, sondern über eine der statischen Erzeugermethoden, welche die gleichen Parameter erwarten, aufgerufen. Der Konstruktor wiederum ruft den Konstruktor der Basisklasse node auf. Dieser ist darum bemüht die Kinderknoten, welche im Parameter subtree übergeben werden, richtig mit dem neuen Syntaxbaumknoten zu verknüpfen. Eine dieser statischen Erzeugermethoden ist im folgenden dargestellt.

```
TokenDecl* TokenDecl::createKeywordDeclPrece(PnodeVector subtree) {

  TokenDecl* decl = new TokenDecl(subtree);

  stringtok* name = dynamic_cast<stringtok*>(decl->getChildNode(0));
  numbertok* precedence =
dynamic_cast<numbertok>(decl->getChildNode(1));
  Assoc* associativity = dynamic_cast<Assoc>(decl->getChildNode(2));

  decl->t = new keywordtok(name->getData());
  decl->t->setPrecedence((int)precedence->getData());

  if (associativity->isLeft())
    decl->t->setLeftAssociativ();
  else if (associativity->isRight())
    decl->t->setRightAssociativ();
  else
    decl->t->setNonAssociativ();

  return decl;
```

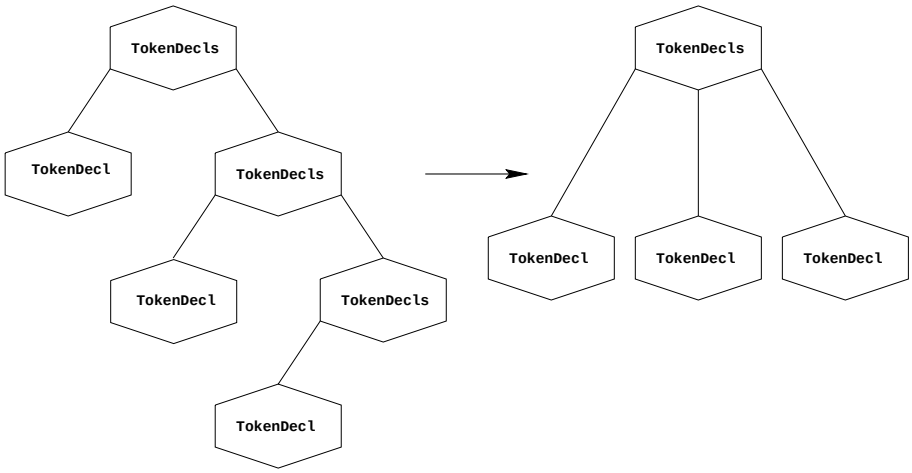

Abbildung A.1: Vereinfachung der Baumstruktur

```
};
```

Diese Methode wird von der Produktion, welche ein Keyword mit Priorität und Assoziativität definiert, aufgerufen. Ziel der Methode ist es, eine Instanz der Klasse keywordtok zu erzeugen und diese mit Namen, Priorität und Assoziativität zu initalisieren.

Dieses Beispiel steht stellvertretend für alle Erzeugermethoden dieser Parsebaumklasse, welche sich nicht gross unterscheiden.

Klasse TokenDecls

Diese einzelnen Terminaldefinitionen werden anschliessend zu einer Liste von Terminaldefinitionen TokenDecls zusammengefasst. Wenn streng nach den entsprechenden Produktionen vorgegangen wird, wird ein daraus entstehender ASB wie derjenige in Abbildung A.1 auf der linken Seite aussehen. Da solche Listenkonstrukte ziemlich häufig vorkommen, sieht die Basisklasse node dafür eine spezielle Umformungsmethode collapseNode(int) vor. Diese Methode hängt alle Blattknoten eines bestimmten Kinderknotens an den aufrufenden Knoten. Die dazwischen liegenden Nichtblattknoten werden gelöscht.

Die Methoden, welche diese Klasse TokenDecls beschreiben, sehen dann wie folgt aus:

```
TokenDecls::TokenDecls(vector<node*>* subtree) : node(subtree) {
  if (this->NumberOfChildren() == 2)
    this->collapseNode(1);
};

TokenDecls* createTokenDecls (vector<node*>* subtree) {
```

```
  return new TokenDecls(subtree);
}

vector<terminal*>* TokenDecls::getTerminalList() {
  vector<terminal*>* termList = new vector<terminal*>;
  for (childIterator i = childNodes.begin(); i != childNodes.end(); ++i)
    termList->push_back(ptr_cast(TokenDecl,(*i))->getTerminal());

  return termList;
};
```

Da die Methode collapseNode(int) laufend im Konstrutor der Klasse ausgeführt wird, ist garantiert, dass eine solche Konfiguration wie sie in Abbildung A.1 links zu sehen ist, gar nie eintritt. Die Klasse bietet eine Methdoe getTerminalList() an, welche es erlaubt, die Terminalobjekte, welche durch TokenDecls repräsentiert werden, abzufragen.

Klasse Production

Als nächstes soll die ASB-Klasse Production betrachtet werden. Diese ASB-Klasse merkt sich die Namen der Nichtterminale und Terminale, welche an dieser Produktion beteiligt sind. Das eigentliche Produktionsobjekt der Klasse production kann erst im Wurzelknoten erzeugt werden, wenn auf die entsprechenden Terminal- und Nichtterminalinstanzen zugegriffen werden kann.

Analog der Liste von Terminaldefinitionen TokenDecls wird auch die Liste der Produktionen Productions organisiert.

Klasse genGrammar

Die Wurzel dieses ASB stellt die Grammatik dar. Diese ASB-Klasse wird durch die Klasse genGrammar dargestellt. Der Wurzelknoten hat zwei Kinderknoten, die Terminaldefinitionen TokenDecls, und die Produktionen Productions. Anhand dieser Informationen muss er die nötigen Objekte erzeugen und in die richtige Liste oder den richtigen Container eintragen.

Die Terminaldefinition sind schon so vorbereitet, dass der entsprechenden Kindknoten TokenDecls nur nach der Liste der erzeugten Terminalobjekte befragt werden muss (getTerminalList()), um diese in den Container des Grammatikobjekts Grammar einzutragen (addTerminal(terminal*)). Zusätzlich werden die beiden Standardterminale endtoken und epsilontok erzeugt, welche in der Sprachdefinition nicht explizit aufgeführt werden. Ebenso wird das spezielle Nichtterminal erzeugt, welches als Startsymbol in der erweiterten Grammatik benötigt wird.

Etwas komplizierter gestaltet sich die Erzeugung der Nichtterminale. Da diese in der Sprachdefinition nicht separat definiert werden, müssen sie während der Erzeugung der Produktionsobjekte laufend generiert werden. Im folgenden ist die ganze oben beschriebe Methode abgebildet:

```
genGrammar::genGrammar(vector<node*>* subtree) : node(subtree) {
  nonterminal* start = new nonterminal("START");
  Grammar = new grammar();
  vector<terminal*>* termList;
  termList = ((TokenDecls*) getChildNode(0))->getTerminalList();
  vector<Production*>* prodList ;
  prodList = ptr_cast(Productions,getChildNode(1))->getProdList();
  vector<terminal*>::iterator i;
  vector<Production*>::iterator j;
  vector<String>::iterator k;
  for (i = termList->begin(); i != termList->end(); i++)
    Grammar->addTerminal(*i);
  Grammar->addTerminal(new endtoken());
  Grammar->addTerminal(new epsilontoken());
  Grammar->addNonterminal(start);
  Grammar->setStartSymbol(start);
  for (j = prodList->begin(); j != prodList->end(); j++) {
    String nonterm = (*j)->getNonTerminal();
    String createMethod = (*j)->getCreateMethod();
    list<int>* paramList = (*j)->getParamList();
    nonterminal* nt = Grammar->findNonTerminal(nonterm);
    if  (!nt) {
      nt = new nonterminal(nonterm);
      Grammar->addNonterminal(nt);
      production* sp = new production(start,Grammar);
      sp->push_back(nt);
      start->addProduction(sp);
      Grammar->addProduction(sp);
    };
    vector<String>* tokenList = (*j)->getRightSide();
    vector<String>::iterator k;
    production* p = new production(nt,Grammar,createMethod,paramList);
    for (k = tokenList->begin(); k != tokenList->end(); k++) {
      terminal* t = Grammar->findTerminal(*k);
      if  (t)
        p->push_back(t);
      else {
        nt = Grammar->findNonTerminal(*k);
        if  (nt)
          p->push_back(nt);
        else {
          nt = new nonterminal(*k);
          Grammar->addNonterminal(nt);
          p->push_back(nt);
        }
      }
    };
    Grammar->addProduction(p);
    p->getNonTerminal()->addProduction(p);
  };
  delete prodList;
  delete termList;
  Grammar->calculateFIRST();
```

```
};
```

Am Schluss wird die Methode calculateFIRST() der Grammatik aufgerufen, um die FIRST-Mengen der Nichtterminale und Produktionen zu berechnen. Das Grammatikobjekt Grammar ist somit komplett und kann nun dazu verwendet werden ein Parser- und ein Scannerobjekt zu initialisieren.

A.1.4 Hauptprogramm

Das Hauptprogramm des Parsergenerators besteht aus drei Teilen.

1. Dem eigentlichen Parsergenerator. Dazu gehören eine Grammatik myGrammar, ein Scanner myScanner, ein Parser myParser und eine Registry myRegistry. Diese Instanzen wurden von Klassen abgeleitet die in diesem Beispiel vorgestellt wurden (generatorGrammar, genRegistry), oder von Standardklassen des Frameworks (scanner, LRParser). Scanner und Parser werden mit Hilfe von myGrammar initialisiert, der Name, der zu interpretierenden Sprachdefinitionsdatei wird dem Scanner von den Argumenten die dem Generator übergeben werden mitgeliefert.

2. Dem zu erzeugenden Parser. Wenn die start-Methode von myParser aktiviert wird, konstruiert er anhand der Sprachdefinition einen ASB, dessen Wurzelknoten wir als Ausgabe der start-Methode erhalten (ParseTree). Mit Hilfe dieses Grammatikobjekts kann der gewünschten Parser (newParser) und Scanner (newScanner) berechnet werden.

3. Dem Objektspeichermechanismus. Um die neu berechnten Objekte abspeichern zu können, müssen zuerst alle für den Objektspechermechanismus notwendigen Objekte erzeugt werden. Dazu gehören eine Objekttabelle my_object_table und ein Klassenverzeichnis myMap [4]. Die beiden Objekte werden miteinander verbunden, indem der Objekttabelle das Klassenverzeichnis zugewiesen wird (set_class_map(ClassMap*)). Anschliessend wird dem Parserobjekt der Auftrag erteilt, sich in der Objekttabelle zu registrieren. Da das Parserobjekt diesen Auftrag an seine mit im in Relation stehenden Objekte weitergibt, reicht dieser Anstoss aus, um alle Objekte in der Tabelle zu registrieren. Über die Methode set_root_ID(int) wird der Objekttabelle mitgeteilt, dass das Parserobjekt das Hauptobjekt ist. Über die Methode get_root_object() kann beim Laden der Objekthierarchie das Parserobjekt wieder identifiziert werden. Mit dem Aufruf der Speichermethode save_to_file(String) wird der Objekttabelle der Auftrag erteilt, sich und die registrierten Objekte in einer Datei mit dem Namen *definition* zu sichern.

Das Hauptprogramm des Parsergenerators ist im folgenden abgebildet.

[4] die entsprechende Klasse wurde in Abbildung 8.2 vorgestellt.

```
#include "grammar.h"
#include "scanner.h"
#include "token.h"
#include "LRParserGen.h"
#include "stateMachine.h"
#include "formats.h"
#include "genRegistry.h"
#include "grammarClasses.h"
#include "error.h"
#include "generatorGrammar.h"

main(int argc,char* argv[]) {
  Object_Table* my_object_table = new Object_Table();
  Compiler_Domain_Class_Map* myMap = new Compiler_Domain_Class_Map();
  genRegistry* myRegistry = new genRegistry();
  generatorGrammar* myGrammar = new generatorGrammar();
  scanner* myScanner = new scanner(argv[1],1,myGrammar);
  LRParserGen* myParser = new LRParserGen(myGrammar,myScanner);
  myGrammar->setRegistry(myRegistry);
  node* ParseTree = Parser->start();
  delete myParser;
  delete myScanner;
  delete myGrammar;
  grammar* ParsedGrammar =
dynamic_cast<genGrammar>(ParseTree)->getGrammar();
  scanner* newScanner = new scanner(1,ParsedGrammar);
  LRParser newParser(newScanner,ParsedGrammar);
  my_object_table->set_class_map(myMap);
  newParser.register_in_table(my_object_table);
  my_object_table->put_object_to_back(ParsedGrammar->get_object_ID());
  my_object_table->set_root_ID(newParser.get_object_ID());
  my_object_table->save_to_file("definition");
  delete ParseTree;
  delete newScanner;
  delete my_object_table;
  delete myMap;
};
```

A.1.5 Interpeter

Nachdem dargestellt wurde, wie ein Parser berechnet wird, soll im nachfolgenden Beispiel der Einsatz eines auf diese Art berechneten Parsers dargestellt werden. Der nachfolgende Quelltext stammt von einem Interpreter für arithmetische Ausdrücke. Neben den Standardklassen werden Klassen benötigt, welche das Verhalten dieses Interpreters implementieren. Das sind die Klassen exprRegistry, welche die Schnittstelle zwischen Parser und ASB herstellt, sowie die ASB-Klassen, welche in der Datei expr.h definiert sind. Die Basisklasse node für alle ASB-Klassen, definiert eine Methode execute(), welche in diesem Beispiel auch von den konkreten ASB-Klassen implementiert wird. Darum kann der ASB, wenn er einmal erzeugt ist, ausgeführt werden.

```
#include "grammar.h"
#include "scanner.h"
#include "classes.h"
#include "token.h"
#include "LRParser.h"
#include "stateMachine.h"
#include "formats.h"
#include "exprRegistry.h"
#include "expr.h"
#include "error.h"

main(int argc,char* argv[]) {
  Object_Table* an_object_table = new Object_Table();
  Compiler_Domain_Class_Map* myMap = new Compiler_Domain_Class_Map();
  exprRegistry* myRegistry = new exprRegistry();
  Object_Table* my_object_table = new Object_Table();
  an_object_table->set_class_map(myMap);
  an_object_table->read_from_file("ExprGrammar");
  LRParser* Parser = (LRParser*)an_object_table->get_root_object();
  scanner* myScanner = Parser->getScanner();
  grammar* myGrammar = Parser->getGrammar();
  myGrammar->setRegistry(myRegistry);
  myScanner->setFileName(argv[1]);
  node* ParseTree = Parser->start();
  delete Parser;
  delete myScanner;
  delete myGrammar;
  if (ParseTree) {
    ParseTree->execute();
    delete ParseTree;
  };
};
```

Anhang A

Auflistung der Headerdateien

Tabelle A.1: Klassen nach Headerdatei sortiert

Headerdatei	Klassen	Bemerkungen
`terminal.h`	terminal stringtok identtok numbertok keywordtok epsilontok endtoken errortok chartok	
`token.h`	token	
`node.h`	node	
`grammarClasses.h`	TokenDecl Assoc TokenDecls ListOfTokens ListOfId Synchronisation ParseTreeCrt Production Productions genGrammar	
`nonterminal.h`	nonterminal	
`production.h`	production	
`formats.h`	Scanner_Format Parser_Format	

Fortsetzung auf der nächsten Seite

Fortsetzung der vorhergenden Seite

Headerdatei	Klassen	Bemerkungen
	Token_Format NonTerminal_Format Statemachine_Format Identtok_Format Keyowrdtok_Format Stringtok_Format Numbertok_Format Endtok_Format Epsilontok_Format Terminaltok_Format SymbolicIdTok_Format Production_Format Grammar_Format Compiler_Domain_Class_Map	
`store.h`	PDC_Object Object_Tag_Format PDC_Object_Format Object_Table_Format Object_Table Class_Map	
`typeinfo.h`	Type_info	
`grammar.h`	grammar	
`LRParser.h`	LRParser	
`scanner.h`	scanner	
`stateMachine.h`	stateMachine	
`error.h`	error	
`registry.h`	registry	
`prodSet.h`	prodSet	
`enhProduction.h`	enhProduction	
`attribute.h`	attribute concAttribute	
`classes.h`		Definition der Klassencodes
`definition.h`		Definition von NodeCreationFnct
`stmChar.h`	stmChar	
`stmIdentifier.h`	stmIdentifier	
`stmNumber.h`	stmNumber	
`stmTerminal.h`	stmTerminal	
`stmString.h`	stmString	
`stmKeyword.h`	stmKeyword	

Fortsetzung auf der nächsten Seite

Fortsetzung der vorhergenden Seite		
Headerdatei	Klassen	Bemerkungen

Literaturverzeichnis

[ASU92] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau Teil 1 und 2*. Addison-Wesley, 1992.

[BJ82] Michael Burke and Gerald A. Fisher Jr. A practical method for syntax error diagnosis and recovery. *ACM TOPLAS*, 6:67–77, 1982.

[Boo94] Grady Boock. *Object-Oriented Analysis and Design*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[Bud91] Timothy Budd. *An Introduction to Object-oriented Programming*. Addison-Wesley, 1991.

[Cho56] Norman Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:3:113–124, 1956.

[Cla94] Chris Clark. Oo compilation - what are the objects ? *Addenum to the Proceedings of OOPSLA*, pages 67–71, 1994.

[CN93] Peter Coad and Jill Nicola. *Object-Oriented Programming*. Yourdon Press computing Series, Prentice Hall Inc., 1993.

[Dai] J. A. Dain. A practical minimum distance method for syntax error handling. Departement of Computer Sience, University of Warwick.

[Flo63] R. W. Floyd. Syntax analysis and operator precedence. *Journal of the ACM*, 10:3:316–333, 1963.

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a compiler*. The Benjamin/Cummings Publishing Company, Inc., 1988.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GR83] Adele Goldberg and David Robson. *SMALLTALK-80, The Language and its Implementation.* Addison-Wesley, 1983.

[Hai86] Brent Hailpern. Multiparadigm languages and environments. *IEEE Software*, 3(1):6–9, 1986.

[Hol95] Jim Holmes. *Object-Oriented Compiler Construction*. Prentice Hall, 1995.

[HS94] Bruce Hahne and Hiroyuki Sato. Using yacc and lex with c++. *ACM SIGPLAN Notices*, 29(12):94–103, 1994.

[HU94] J.E. Hopcroft and J.D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1994.

[JF88] Ralph E. Johnson and Brian Foote. Designing reuseable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Joh94] Ralph Johnson. The rtl system. *Addenum to the Proceedings of OOPSLA*, pages 67–71, 1994.

[Joh97] Ralph E. Johnson. Components, frameworks, patterns. 1997.

[Kuh70] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 2nd edition, 1970.

[KW95] Basim M. Kadhim and William M. Waite. Maptool - mapping between concrete and abstract syntaxes. Technical Report CU-CS-765-95, University of Colorado at Boulder, Department of Computer Sience, February 1995.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., 1992.

[Loo95] Mary E.S. Loomis. *Object Databases, The Essentials*. Addison-Wesley, 1995.

[Mal94] Jawahar Malhorta. Viewing compilation as an object-oriented process. *Addendum to the Procceedings of OOPSLA*, pages 67–71, 1994.

[Mey88] Ware Meyers. Interview with wilma osborne. *IEEE Software*, 5(3):104–105, 1988.

[Mey94] Bertrand Meyer. *Reuseable Software, The Base object-oriented Component Library*. Prentice Hall, 1994.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[Par96] Terence Parr. *Language Translation using PCCTS & C++*. Automate Publishing Company, 1996.

[PDC90] Terence Parr, Henry Dietz, and Will Cohen. Purdue compiler-construction tool set. Technical Report TR-EE90-14, School Of Electrical Engineering,Purdue University,West Lafayette, IN, February 1990.

[PP92] Thomas Pittman and James Peters. *The Art of Compiler Design, Theory and Practice*. Prentice Hall, 1992.

[PT95] Benjamin C. Pierce and David N. Turner. *PICT Language Definition 3.6b*. Computer Laboratory University of Cambridge, Department of Computing Science University of Glasgow, June 1995.

[Rob94] Arch D. Robison. Iterators for abstract-syntax trees or recursion over trees considerd harmful. *Addenum of the Proceedings of OOPSLA*, pages 67–71, 1994.

[SSS83] Seppo Sippu and Eljas Soisalon-Soininen. A syntax-error-handling technique and its experimental analysis. *ACM TOPLAS*, 5(4):656–679, October 1983.

[Str92] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 1992.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[SU70] R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[WF74] Robert A. Wagner and Michael J. Fisher. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

[Zin94] Barbara Zino. Yacc++ and the language objects library. *Addendum to the Proceedings of OOPSLA*, pages 67–71, 1994.