Objekt-Orientierung im Compilerbau

Bernhard Kühl

Dissertation

Schriftliche Arbeit zur Erlangung des Doktorgrades der Naturwissenschaften
im Fachbereich Mathematik/Informatik der Universität Osnabrück

2001

Objekt-Orientierung im Compilerbau

Bernhard Kühl

Dissertation

Gutachter: Prof. Dr. Axel-Tobias Schreiner, Rochester Institute of Technology

Prof. Dr. Christoph Polze, Humboldt-Universität zu Berlin

Inhaltsverzeichnis

Abbildungs- und Tabellenverzeichnis

1 Einleitung

Aus Aho, Sethi und Ullman, “Compilerbau” ([Aho86]): Ein Compiler ist ein Programm, das ein in einer bestimmten Sprache — der Quellsprache — geschriebenes Programm liest und es in ein äquivalentes Programm einer anderen Sprache — der Zielsprache — übersetzt.

Seit der Entwicklung der ersten Compiler wurden für sehr viele Programmiersprachen Compiler entwickelt. Jeder Informatikstudierende ist zum Beispiel der Umgang mit einem C- oder vielleicht auch Java-Compiler gewohnt. Am Anfang galten Compiler als nur sehr schwer zu implementierende Programme. Seitdem wurden aber systematische Techniken und Werkzeuge zur automatischen Erzeugung von (Teilen von) Compilern entwickelt. Dank dieser Fortschritte stellt die Implementierung eines Compilers auch für eine höhere Programmiersprache kein großes Problem mehr dar.

Nur wenige Personen kommen allerdings in die Situation, für eine höhere Programmiersprache einen Compiler zu entwickeln. Aber auch andere “Dinge” sind wie Programme einer Sprache zu analysieren und zu verarbeiten:

Filter-Werkzeuge wie *grep*, *sed* oder *awk* agieren auf einem (zeilenweisen) Strom von Zeichen, analysieren die Zeichen und geben einen Strom von Zeichen aus. Applikationen lesen, untersuchen und schreiben Konfigurationsdateien. Pretty-Printer oder auch Präprozessoren erkennen Programme einer höheren Sprache und geben diese textuell abgeändert wieder aus.

Techniken des Compilerbaus finden daher in der Entwicklung von Software auch weiterhin häufig Verwendung. Allerdings ist die Entwicklung derartiger Software noch immer nicht ohne Probleme: Die Entwicklung von Übersetzer-Software ist von Hand sehr mühsam und fehleranfällig, und die entwickelten Übersetzer sind nur schwer zu erweitern. Es empfiehlt sich daher, Werkzeuge zur Generierung von Software zu verwenden. Diese sind allerdings für Anfänger nicht leicht zu erlernen und fest auf ihre Funktionalität begrenzt. Eine Erweiterung der Werkzeuge ist kaum möglich.

Xerox PARC ([Gol83]) hat, basierend auf Ideen der Sprache Simula, etwa um 1970 anhand der Programmiersprache Smalltalk die Technik der objekt-orientierten Programmierung erfunden. Objekt-orientierte Programmierung ist das Programmierkonzept der vergangenen Jahre. Viele der heute verwendeten Programmiersprachen sind entweder von Grund auf objekt-orientiert (zum Beispiel Java, C++ oder SmallTalk) oder wurden im Lauf der Zeit mit objekt-orientierten Erweiterungen versehen (zum Beispiel Basic oder Pascal). Selbst manche Skriptsprachen besitzen objekt-orientierte Eigenschaften (zum Beispiel JavaScript oder Python), und selbst nicht objekt-orientierte Sprachen wie C können objekt-orientiert verwendet werden ([Sch94]).

Es ist daher naheliegend und Inhalt der vorliegenden Arbeit, die Objekt-Orientierung auf das Problem der Verarbeitung von Sprachen anzuwenden, um zu zeigen, daß ...

... durch den Einsatz der objekt-orientierten Programmierung die Entwicklung von (Teilen von) Compilern vereinfacht werden kann.

... die so entwickelten Werkzeuge und deren Resultate an Mächtigkeit gegenüber den Werkzeugen und Resultaten der gängigen Techniken gewinnen.

... auch objekt-orientiert konzipierte Software des Compilerbaus der Vorteil der leichten Erweiterbarkeit zugute kommt.

... die Wiederverwendung von Klassen und Objekten die Entwicklung und Verwendung von Scannern oder Compilern erleichtert.

... die objekt-orientierte Programmierung ein Automatismus für *divide & conquer* ist und auch hier gewinnbringend genutzt werden kann. Selbst komplexe Algorithmen des Compilerbaus

zerfallen so in kleine, einfache Teilprobleme und sind dadurch in der Lehre für Studierende leicht zu verstehen bzw. sogar selbständig zu entdecken.

... der Einsatz verschiedener typischer Entwurfsmuster der Objekt-Orientierung die Flexibilität und Erweiterbarkeit zum Beispiel eines Scanners oder eines Compilers wesentlich erhöht.

2 Hintergrund

Dieses Kapitel schildert den Hintergrund der Arbeit. Es erläutert in einem Abschnitt straff die Phasen und klassischen Techniken des Compilerbaus und klärt dabei wichtige, in der Arbeit verwendete Begriffe. In einem weiteren Abschnitt wird kurz in die objekt-orientierte Programmierung eingeführt, die Vorteile der Objekt-Orientierung werden diskutiert, und die Konkurrenzprodukte zu den Ideen und Werkzeugen der Arbeit werden vorgestellt und bewertet. Abschließend folgt ein Überblick über den Inhalt aller Kapitel der Arbeit.

Die Abschnitte über den Compilerbau und über die Objekt-Orientierung sind keineswegs als vollständige Einführungen in die beiden Bereiche gedacht. Für ein ausführlicheres Studium der Themen wird weiterführende Literatur nahegelegt. Hinweise auf mögliche Literatur werden in den Abschnitten angegeben.

2.1 Compilerphasen und deren klassische Implementierung

Dieser Abschnitt führt kurz in den typischen Aufbau eines Compilers ein. Für eine vollständige, aber sehr komplexe und umfangreiche Einführung sei ([Aho86]) empfohlen.

Der Übersetzungsprozeß eines Quelltexts besteht im wesentlichen aus mehreren Phasen. Diese werden in Abbildung 2.1 dargestellt, wobei der zu analysierende bzw. zu übersetzende Quelltext die Eingabe der ersten Phase und die Ausgabe einer Phase die Eingabe der darauf folgenden Phase darstellt (blaue Pfeile).

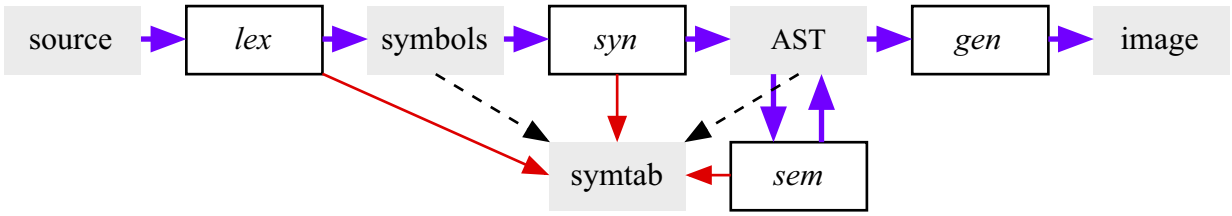

Abbildung 2.1: Die typischen Phasen eines Compilers.

Die lexikalische Analyse (in der Abbildung 2.1 mit *lex* bezeichnet) ist die erste Phase einer Übersetzung. Der Teil eines Compilers, der die lexikalische Analyse durchführt, wird als Scanner bezeichnet, da er die Eingabe nach vordefinierten Mustern durchsucht (engl. *scan*). Die Aufgabe des Scanners ist es dann, ein oder mehrere Zeichen des Quelltexts zu Symbolen zusammenzufassen und an die nächste Phase, die syntaktischen Analyse, weiterzugeben. Symbole sind die Grundelemente der syntaktischen Analyse.

Als Beispiel faßt ein typischer Scanner im Compilerbau Folgen von Leerzeichen, Tabulatoren und Zeilentrenner zu dem Symbol für Zwischenraum und Folgen von Ziffern zu dem Symbol für eine Zahl zusammen. Andere typische Symbole sind Zeichenfolgen als Namen für Variablen und Funktionen/ Methoden, Gleitkommazahlen, verschiedene Arten von Kommentaren oder die reservierten Wörter der zu analysierenden Sprache.[1]

Der Programmtext eines Scanners kann von Hand implementiert werden. Dies ist aber sehr mühsam und fehleranfällig, und das Resultat ist nur schwer zu erweitern und zu warten. Ein Scanner wird daher

1. Die reservierten Wörter einer Sprache werden aus Effizienzgründen normalerweise nicht alle als einzelne Symbole gescannt. Statt dessen wird ein Muster, welches alle reservierten Wörter und Variablen- und Methoden-/ Funktionsnamen beinhaltet, verwendet. Die Unterscheidung, welches Symbol nun genau erkannt wurde, wird dann später getroffen.

typischerweise von einem Werkzeug wie *lex* ([Les75]) oder *flex* ([Pax95]) generiert. Beide lesen als Eingabe Paare von Mustern und Programmtext. Die Muster beschreiben Symbole anhand von regulären Ausdrücken. *lex* und *flex* erzeugen als Resultat eine Funktion, die die per Muster beschriebenen Symbole erkennt und zu einem erkannten Symbol den assoziierten Programmtext zur Ausführung bringt.

In der Phase der syntaktischen Analyse (*syn*) wird geprüft, ob die von der lexikalischen Analyse gelieferte Folge von Symbolen syntaktisch korrekt ist, d.h., ob die Symbolfolge ein legales Programm der akzeptierten Sprache darstellt. Der Teil eines Compilers, der die syntaktische Analyse durchführt, wird als *Parser* bezeichnet, da er die Symbolfolge auf syntaktische Korrektheit untersucht (engl. *parse*). In der Regel arbeiten Parser und Scanner in der Weise zusammen, daß der Parser bei Bedarf das nächste Symbol anfordert, das der Scanner dann liefert.

Auch die syntaktische Analyse kann von Hand implementiert werden. Typischerweise kommt dabei die Technik des rekursiven Abstiegs zum Einsatz. Die gleichen Nachteile wie bei der Entwicklung eines Scanners von Hand gelten aber auch hier. Unter anderem ist die Implementierung von Hand mühselig und fehleranfällig. Daher wird auch diese Phase oder besser Teile der Phase üblicherweise mit Hilfe von Werkzeugen generiert. Als typische Werkzeuge seien dabei *yacc* ([Joh75]) oder *bison* ([Don91]) genannt. Beide verarbeiten als Eingabe eine BNF-Grammatik ([Knu64],[Nau63]), welche wiederum mit Programmtext versehen ist. Als Resultat wird eine Funktion erzeugt, welche mit einer lexikalischen Analyse als Partner die Symbolfolge gegen die Grammatik prüft und zu erkannten Teilen der Grammatik den assoziierten Programmtext zur Ausführung bringt.

Die Syntax einer Sprache wird zumeist in Form einer Grammatik beschrieben. Formal besteht eine Grammatik aus einer Menge von Eingabesymbolen, einer Menge von Grammatikbegriffen, daraus einem Startbegriff, und einer Menge von Regeln, d.h. bestimmten Paaren von Folgen von Grammatikbegriffen und Eingabesymbolen; alle Mengen und Folgen müssen endlich sein.

Typischerweise schreibt man nur die Regeln auf und verlangt bei kontextfreien Grammatiken, daß die linke Seite einer Regel immer ein Grammatikbegriff sein muß. Nach Konvention steht der Startbegriff auf der linken Seite der ersten Regel, und man faßt rechte Seiten zum gleichen Grammatikbegriff als Alternativen zusammen. Die Wiederholungen der Syntaxgraphen muß man durch rekursive Verweise modellieren. Derartige Grammatiken werden in Form von BNF-Grammatiken beschrieben. Für arithmetische Ausdrücke sieht das etwa so aus:

Hier ein Beispiel einer BNF-Grammatik für einen arithmetischen Ausdruck:

```
sum     : sum '+' product | sum '-' product | product ;
product : product '*' term | product '/' term | term ;
term    : NUMBER | '+' term | '-' term | '(' sum ')' ;
```

Ein Doppelpunkt trennt die linke und rechte Seite einer Regel, ein `|` trennt Alternativen, und ein Semikolon steht nach allen rechten Seiten zum gleichen Grammatikbegriff. Zitierte Eingabesymbole (*Literale* genannt) werden mit Anführungszeichen zitiert. Da `NUMBER` nicht links als Regelname vorkommt, muß `NUMBER` (implizit) eine Klasse von Eingabesymbolen (*Token* genannt) repräsentieren. Symbole werden auch *Terminal*-Symbole genannt.

EBNF (Extended oder Erweiterte Backus-Naur-Form) ist eine Erweiterung von BNF: Gegenüber BNF kommt ein Konstrukt für Wiederholungen hinzu, welche in BNF durch Rekursion ausgedrückt werden müssen. Leere Alternativen werden durch ein Konstrukt für Optionalität ersetzt.

Abweichend von der gängigen EBNF-Schreibweise markieren in dieser Arbeit auf der rechten Seite einer Regel eckige Klammern (`[ ... ]`) einen optionalen Teil (0-1) der Grammatik. Geschweifte Klammern (`{ ... }`) bezeichnen eine mindestens einmalige, aber beliebig häufige Wiederholung (1-n). Die Kombination der beiden Konstrukte markiert damit eine beliebig häufige Wiederholung (0-n). Runde Klammern dienen der Zusammenfassung.

Alle EBNF-Grammatiken sind als BNF-Grammatik darstellbar. EBNF-Grammatiken sind aber für die meisten Anwender wesentlich lesbarer. Kompakter sind sie auf jeden Fall. Das obige Beispiel verkürzt sich in EBNF zu:

```
sum     : product { ( '+' | '-' ) product } ;
product : term { ( '*' | '/' ) term } ;
term    : NUMBER | ( '+' | '-') term | '(' sum ')' ;
```

In der syntaktischen Analyse kann der Programmtext zu erkannten Teilen einer Grammatik interpretativ agieren, d.h., er führt bereits die Anweisungen des analysierten Programms aus. Typischerweise wird aber ein sogenannter abstrakter Syntax-Baum zur Repräsentation des erkannten Programms gebaut. In der semantischen Analyse (*sem*) wird daraufhin anhand des Baums das Programm untersucht: Stimmen zum Beispiel die Typen von Operanden eines Operators, wird auf eine nicht existente Variable zugegriffen usw. Resultat der semantischen Analyse kann ein neuer oder veränderter Baum sein, da zum Beispiel implizite Typen-Umwandlungen für verschiedene Operandentypen in den Baum einzubauen sind.

Im letzten Schritt des Compilers, welcher aus vielen kleinen Schritten inklusive verschiedener Arten von Optimierungen bestehen kann, wird ein ausführbares Programm anhand des Baums generiert (*gen*).

Die verschiedenen Phasen teilen sich eine Symboltabelle, in der Informationen über Eigenschaften (Attribute) der erkannten Symbole gespeichert werden. Typische Attribute sind: der Name einer Variablen oder Konstanten oder der Name einer Funktion/Methode — möglicherweise zusammen mit der Anzahl und dem Typ der Parameter. Diese Informationen werden ab dem Start der Übersetzung inkrementell von den Phasen vervollständigt. Dabei können etwa die Namen von Methoden in der lexikalischen Analyse, die Zahl der Parameter in der syntaktischen und die Typen in der semantischen Analyse ergänzt werden. Dieser Sachverhalt wird in Abbildung 2.1 durch die roten Pfeile dargestellt.

Bei der Analyse eines Quellprogramms können syntaktische, aber auch semantische Fehler auftreten. Aufgabe eines guten Compilers ist es, an dieser Stelle eine aussagekräftige Fehlermeldung für den Anwender zu liefern, damit dieser den Fehler schnell findet und korrigieren kann. Trotzdem sollte der Compiler in der Erkennung fortfahren, um so mögliche weitere Fehler dem Anwender berichten zu können. Durch beide Aspekte wird die Entwicklung von lauffähigen Programmen beschleunigt.

2.2 Objekt-Orientierung

Dieses Unterkapitel beinhaltet eine kurze Einführung (inklusive Beispiele) in die Konzepte der Objekt-Orientierung und führt in einem weiteren Abschnitt die Vorteile in der Verwendung der objekt-orientierten Programmierung auf. Genau diese Vorteile versucht die Arbeit, in der Anwendung der Objekt-Orientierung in bezug auf die Verarbeitung von Sprachen zu entdecken. Die Beispiele sind — wie der ganze Programmtext der Arbeit — in Java implementiert und führen so — wenn auch recht grob — in Teile der Programmiersprache Java ein. Thema der Arbeit ist aber nicht Java im Compilerbau, und alle in dieser Arbeit vorgestellten Ergebnisse sind von der Sprache unabhängig. Als Beweis wurde in einer Bachelorarbeit von Jan Kraneis ([Kra02]) der Programmtext aller entwickelten Werkzeuge und Beispiele nach C# ([Mic01], [Oba01], [Ell01]) portiert.

Das Unterkapitel stellt keinesfalls eine komplette Erläuterung aller Bestandteile und Feinheiten der objekt-orientierten Programmierung dar. Dazu sind klassisch [Gol83] oder modern und umfassend [Bud02] zu empfehlen.

2.2.1 Eine kurze Einführung in die Objekt-Orientierung

Objekt-Orientierung ist der Einsatz von Objekten und Klassen in der Analyse, Entwicklung und Programmierung.

Klassen, Objekte, Variablen und Methoden

Eine Klasse ist eine Zusammenfassung von Daten und darauf operierender Funktionen (die in der Objekt-Orientierung *Methoden* genannt werden). Instanzen einer Klasse werden *Objekte* genannt. Die Daten werden durch Variablen (welche *Attribute* oder *Instanzvariablen* genannt werden) repräsentiert, die für jedes instanziierte Objekt neu angelegt werden. Die Methoden sind im ausführbaren Programm nur einmal vorhanden. Die Aufrufe der Methoden werden aber immer an ein Objekt als Empfänger gesendet und operieren daher auf den Daten des Empfängerobjekts.

Die Instanzvariablen repräsentieren den Zustand eines Objekts. Deren Werte können bei jeder Instanz einer Klasse unterschiedlich sein und sich während der Lebensdauer des Objekts verändern. Die Methoden implementieren das Verhalten des Objekts. Sie sind — von gewollten Ausnahmen abgesehen, bei denen Variablen bewußt von außen zugänglich gemacht werden — die einzige Möglichkeit mit dem Empfängerobjekt zu kommunizieren und so Informationen über seinen Zustand zu gewinnen oder diesen zu verändern.

Die Zusammenfassung von Methoden und Variablen zu Klassen dient der Abstraktion, d.h., Klassen modellieren einen Aspekt der Anwendungswelt und dienen der Kapselung, d.h., Instanzvariablen (und durchaus auch die Methoden) werden vor der Außenwelt versteckt. Als Beispiel in Java eine Klasse `Point`, deren Objekte eine Abstraktion eines Punkts im zweidimensionalen Raum darstellen:

```
public class Point {
    private double x, y;

    public double getX() { return x; }
    public void setX(double new_x) { x = new_x; }

    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
}
```

`public` markiert Methoden und Variablen als öffentlich nutzbar. `private` beschränkt dagegen den Zugriff auf Instanzen der Klassen. Ein Objekt der Klasse `Point` kapselt in zwei Instanzvariablen die Koordinaten des Punkts und stellt über jeweils zwei öffentliche Methoden eine Schnittstelle zum Erfragen und Setzen der Koordinaten zur Verfügung. Die Werte der Instanzvariablen können dagegen nur vom Punkt selbst gelesen oder geändert werden.

In Java ist `this` ein Verweis auf das gerade die Methode ausführende Objekt. `this.name` ist die Instanzvariable namens `name` beim Objekt, auf das `this` verweist. Ist vor einem nicht lokalen Namen einer Instanzvariablen kein zugehöriges Objekt angegeben, wird immer `this.` vom Compiler vor dem Variablennamen hinzugefügt. Lokale Variablen verdecken Instanzvariablen des gleichen Namens. Daher wird im Programmtext der Methode `setY()` mit `this.y` explizit auf die Instanzvariable verwiesen.

Konstruktoren

Objekte einer Klasse fallen nicht vom Himmel, sondern werden konstruiert und dann initialisiert. Je nach Programmiersprache sind dies zwei getrennte Schritte (zum Beispiel in Objective-C ([Cox91])) oder sind wie in Java auch zu einem Schritt zusammengefaßt:

```
public class Point {
    ...
    public Point(double x, double y) {
        setX(x); this.setY(y);
    }
```

```
        public static void main(String args[]) {
            Point p = new Point(2, 3);
        }
        ...
    }
```

In Java wird immer direkt nach der Erzeugung eines Objekts ein Konstruktor zur Initialisierung des Objekts ausgeführt. Durch `new`, gefolgt von dem Namen der Klasse des zu erzeugenden Objekts, wird das Objekt erzeugt, und die Anzahl und die Typen der folgenden Parameter wählen, welcher Konstruktor zur Initialisierung des neuen Objekts verwendet wird. In dem Beispiel werden dem Konstruktor die beiden Koordinaten des neuen `Point`-Objekts als Parameter übergeben. In dem Programmtext des Konstruktors werden diese Initialwerte durch Aufruf von Methoden gesetzt.

Eine Methode hat immer einen Empfänger, d.h., für dieses Objekt wird der Code der Methode ausgeführt. In Java steht der Empfänger einer Methode analog zu Instanzvariablen immer vor dem Methodennamen — in dem Beispiel wieder mit `this` als Verweis auf das aktuelle Objekt. Wird in Java kein Empfängerobjekt angegeben, wird implizit `this` verwendet.

Klassenmethoden und Klassenvariablen

In objekt-orientierten Sprachen gibt es keine globalen Methoden oder globalen Variablen. Da es aber mitunter sinnvoll ist, Methoden oder Variablen zu verwenden, die nicht an eine Instanz einer Klasse gebunden sind, können beide auch zu einer Klasse gehören: Man spricht von (statischen) *Klassenmethoden* oder *Klassenvariablen*.

Jede Klassenvariable wird nur einmal pro Klasse angelegt und kann von allen Methoden — auch Instanzmethoden — der Klasse aufgerufen werden. Da sich alle Methoden die Variable teilen, sind Veränderungen, die eine Instanz vornimmt, auch für alle anderen Instanzen sichtbar. Klassenvariablen sind daher vergleichbar mit globalen Variablen, welche sich alle Instanzen einer Klasse teilen. Sind sie `public` deklariert, sind sie sogar vergleichbar mit programmweiten, globalen Variablen.

Neben Klassenvariablen gibt es auch Klassenmethoden, d.h., Methoden, die unabhängig von einer bestimmten Instanz der Klasse existieren. Sie haben daher von allein nur Zugriff auf Klassenvariablen, da kein ausführendes Objekt der Klasse mit der Methode assoziiert ist. Klassenmethoden und Klassenvariablen werden in Java mit Hilfe des `static`-Attributs deklariert und durch Voranstellen des Klassennamens aufgerufen.

Klassenmethoden werden da eingesetzt, wo Funktionalitäten zur Verfügung gestellt werden, die ohne einen Objektzustand auskommen. Ein Beispiel dafür ist die Klasse `Math` aus der Java-Klassenbibliothek, die eine Vielzahl an mathematischen Operationen zur Verfügung stellt. Eine weitere Anwendung für Klassenmethoden liegt in der Instanziierung von Objekten. Beispiele dafür finden sich in den Entwurfsmustern *Singleton* und *Factory* ([Gam95]).

Analog zu der Funktion `main()` in C startet die Ausführung eines Java-Programms bei der Klassenmethode `main()` einer angegebenen Klasse. Als Parameter wird vom System ein Array mit den Wörtern der Kommandozeile als String übergeben.

Als Beispiel eine Klassenmethode der Klasse `Point`, welche für ein Koordinatenpaar den Abstand zum Ursprung berechnet. In dem Programmtext der Methode wird die Klassenmethode `sqrt()` der Klasse `Math` zur Berechnung der Quadratwurzeln verwendet:

```
public class Point {
    ...
    public static double getDistanceFromOrigin(int x, int y) {
        return Math.sqrt(x*x + y*y);
    }
    ...
}
```

Alternativ könnte eine Instanzmethode ein `Point`-Objekt nach seinem Abstand zum Ursprung fragen. Dazu müßten die beiden primitiven Zahlenwerte der Koordinaten immer zuerst in eine `Point`-Instanz verwandelt werden. Das Vermeiden der Umwandlung eines primitiven Zahlenwerts in ein Objekt ist eine typische Anwendung für Klassenmethoden — siehe die Klassenmethoden der Klasse `Math`.

Vererbung, abstrakte Klassen

Ein weiteres wesentliches Merkmal objekt-orientierter Sprachen ist die Vererbung, also die Möglichkeit, Eigenschaften einer vorhandenen Klasse (die Oberklasse) auf neue Klassen (die Unterklassen) zu übertragen. Die Unterklasse erbt von ihrer Oberklasse deren Variablen und Methoden, welche somit für Objekte der Unterklasse bzw. bei Klassenmethoden und -variablen auch für die Unterklasse selbst anwendbar sind. Die Unterklasse kann eigene, neue Variablen und Methoden einführen. Sie erweitert damit die Oberklasse zu einem komplexeren Zustand — sie spezialisiert die Oberklasse.

Vererbungen können mehrstufig sein, d.h., eine abgeleitete Klasse kann Oberklasse für weitere Klassen sein. Auf diese Weise können vielstufige Vererbungshierarchien entstehen, die in natürlicher Weise die Struktur der zu modellierenden Anwendungswelt repräsentieren. Eine Klassenhierarchie für grafische Objekte zeigt Abbildung 2.2:

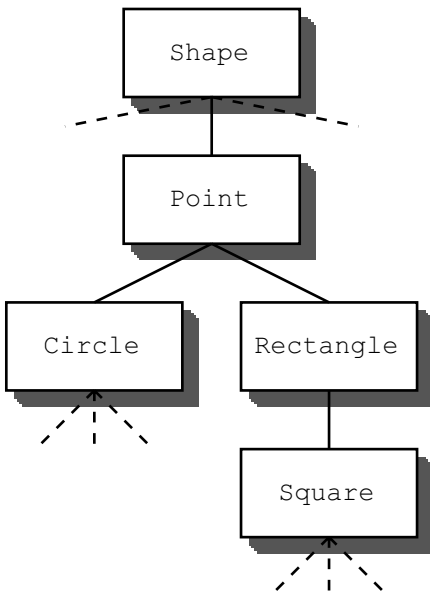

Abbildung 2.2: Eine Klassenhierarchie für grafische Objekte.

Man unterscheidet zwischen einfacher Vererbung, bei der eine Klasse von genau einer Oberklasse abstammt, und Mehrfachvererbung, bei der eine Klasse von mehr als einer anderen Klasse abgeleitet werden kann.

In Java gibt es lediglich Einfachvererbung, um den Mehrdeutigkeiten, die durch Mehrfachvererbung entstehen können — wie Methoden gleichen Namens in mehr als einer Oberklasse —, aus dem Weg zu gehen. Um eine neue Klasse von einer bestehenden abzuleiten, ist im Kopf der Klasse mit Hilfe des

Schlüsselworts `extends` ein Verweis auf die Basisklasse anzugeben. In dem Beispiel sollen alle grafischen Objekte direkt oder indirekt von der Klasse `Shape` abstammen:

```
public abstract class Shape {
    public abstract void translate(double dx, double dy);
    public abstract void translate(Point dp);
    public abstract void move(double x, double y);
    public abstract void scale(double factor);
}
```

Die Klasse `Shape` ist eine sogenannte *abstrakte Klasse*. Sie schreibt Methoden vor, ohne deren Körper zu implementieren. Von einer abstrakten Klasse kann es daher keine Objekte geben. In Java ist eine abstrakte Klasse mit `abstract` im Kopf der Klasse zu deklarieren. In Java stammen alle Klassen, welche nicht per `extends` ihre Oberklasse festlegen, von der Systemklasse `Object` ab, welche einige Methoden an Unterklassen vererbt, die somit für alle Objekte zur Verfügung stehen. Nicht abstrakte Unterklassen von `Shape` erben die Methoden der Oberklasse und müssen diese daher implementieren:

```
public class Point extends Shape {
    private double x, y;
    ...
    public void translate(double dx, double dy) {
        setX(this.x+dx); setY(getY()+dy);
    }
    public void translate(Point dp) {
        translate(dp.getX(), dp.getY());
    }
    public void move(double x, double y) { setX(x); setY(y); }
    public void scale(double factor) { }
    ...
}
```

Durch Hinzufügen neuer Methoden oder Instanzvariablen in einer Unterklasse kann die Funktionalität der Oberklasse erweitert bzw. spezialisiert werden:

```
public class Rectangle extends Point {
    ...
    protected double height;
    protected double width;

    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }

    public void scale(double factor) {
        height *= factor; width *= factor;
    }
    ...
}
```

Ein Rechteck ist eine linke, untere Ecke als Eckpunkt und eine Höhe und Breite. `Rectangle` stammt daher von `Point` ab und spezialisiert die Oberklasse um neue Methoden und Variablen für die Höhe und Breite des Rechtecks. In Java werden `protected` und `public` — im Gegensatz zu `private` — deklarierte Methoden und Variablen einer Klasse an Unterklassen vererbt. `protected` beschränkt den Zugriff aber auf die Klasse selbst oder auf Instanzen einer direkten oder indirekten Unterklasse. Die

Koordinaten `x` und `y` der Oberklasse `Point` sind daher in dem Programmtext eines `Rectangle` nicht direkt — da `private` deklariert —, sondern nur über die geerbten Methoden der Oberklasse ansprechbar.

Als weiteres, wichtiges Konzept in der Ableitung von Klassen können Unterklassen Methoden der Oberklasse überschreiben (*overriding*) und können so das Verhalten der Instanzen einer Klasse gegenüber der Oberklasse abändern. In dem obigen Beispiel wird die Methode `scale()` in `Rectangle` überschrieben. Ein Punkt bleibt beim Skalieren ein Punkt, ein Rechteck dehnt sich dagegen aus.

Abstrakte Klasse vs. Interface

Eine abstrakte Klasse — wie die obige Beispielklasse `Shape` — ist eine Sammlung von Methodenköpfen ohne Implementierung der Methodenkörper.

Ein Interface in Java ist analog eine Sammlung von Methodenköpfen, ohne deren Implementierung vorzugeben. Ein Interface kann nur konstante Variablen enthalten. Eine Klasse kann Interfaces annehmen — man spricht auch von adaptieren — und verspricht damit, die Methoden der adaptierten Interfaces zu implementieren.

Interface sind Teil der Sprache Java, da Java keine Mehrfachvererbung unterstützt. Durch das Adaptieren eines Interfaces können Klassen verschiedener Positionen einer Klassenhierarchie gleichartige Schnittstellen in Form von Methoden-Implementierungen versprechen.

Interfaces haben gegenüber abstrakten Klassen den Vorteil, daß ein Interface für eine Klasse keine Oberklasse vorschreibt. Jede neue Klasse kann unabhängig von ihrer Stelle in der Klassenhierarchie das gewünschte Interface adaptieren. Im Gegensatz dazu müssen abstrakte Klassen aber nicht nur abstrakte Methoden enthalten und können daher für Unterklassen bereits Variablen anlegen oder Methoden implementieren.

Als Beispiel in Java ein einfaches Interface `Dumpable`, welches nur eine Methode `dump()` vorschreibt. Instanzen von Klassen, die das Interface `Dumpable` annehmen, sollen als Resultat von `dump()` einen String mit einer textuellen Darstellung von sich selbst liefern:

```
public interface Dumpable {
    public String dump();
}
```

Als Beispiel adaptiert die Klasse `Point` das Interface. Dieses geschieht in Java durch `implements` im Kopf der Klasse, wobei eine Klasse durchaus mehr als ein Interface annehmen kann:

```
 public class Point extends Shape implements Dumpable {
    ...
    public String dump() {
        return "A "+getClass().getName()+" at ("+x+","+y+")";
    }
    ...
}
```

In Java sind Strings durch einfache Addition zu verketten, und `getClass().getName()` liefert den Namen der Klasse des die Methode ausführenden Objekts. `Rectangle` stammt von `Point` ab und erbt daher die Adaption von `Dumpable`. Das Resultat von `dump()` sollte aber für ein `Rectangle` gegenüber `Point` erweitert sein. Die Methode `dump()` wird daher überschrieben:

```
public class Rectangle extends Point {
    ...
    public String dump() {
        return super.dump()+" with "+getWidth()+" height "+getHeight();
    }
    ...
}
```

Das Beispiel zeigt eine weitere Technik der Objekt-Orientierung. Eine überschriebene Methode verdeckt den Code der gleichen Methode aus der Oberklasse. Damit Methoden den Code der Methode einer Oberklasse erweitern können, muß es eine Möglichkeit geben, auf die verdeckte Methoden-Implementierung zuzugreifen. Dies geschieht in Java durch Aufruf von `super.methodenName()`. Der Verweis `super` ist wie `this` mit dem Unterschied, daß die Bindung an Methoden und Instanzvariablen stattfindet, als wäre das Objekt ein echtes Objekt seiner Oberklasse. Zugriffsbeschränkungen durch `private`-Deklaration in der Oberklasse können damit nicht umgangen werden. Der spezielle Verweis `super` dient nur dazu, den Zugriff auf durch *overriding* versteckte Methoden und Instanzvariablen zu ermöglichen.

Overloading

Wie bereits gezeigt, können Klassen Methoden mit gleichem Namen, aber einer unterschiedlichen Parameterliste enthalten:

```
public class Point extends Shape {
    ...
    public void translate(double dx, double dy) {
        setX(this.x+dx); setY(getY()+dy);
    }
    public void translate(Point dp) {
        translate(dp.getX(), dp.getY());
    }
    ...
}
```

Die Klasse `Point` enthält zwei Methoden mit Namen `translate`, und bei Aufruf entscheidet allein die Parameterliste (Anzahl und Typen der Parameter) über die auszuführende Methode. Diese “Überladung” von Methodennamen und Konstruktoren wird *overloading* genannt.

Polymorphismus, dynamische Bindung

In den obigen Beispielklassen wurde die Methode `scale()` in der abstrakten Klasse `Shape` für alle Unterklassen festgelegt, in den Unterklassen `Circle` und `Point` von `Shape` implementiert und in der Unterklasse `Rectangle` von `Point` überschrieben. Im folgenden Beispiel wird jeweils eine Instanz der drei nicht abstrakten Klassen erzeugt, um dann jeweils `scale()` bei den Objekten aufzurufen:

```
public class Main {
    public static void main(String args[]) {
        Shape shape1 = new Point(2.0, 3.0);
        Shape shape2 = new Rectangle(-1.0, -8.0, 2.0, 3.0);
        Shape shape3 = new Square(1.0, 1.0, 4.0);

        shape1.scale(2);
        shape2.scale(2);
        shape3.scale(2);
```

```
            System.out.println(((Dumpable) shape1).dump());
            System.out.println(((Dumpable) shape2).dump());
            System.out.println(((Dumpable) shape3).dump());
        }
    }
```

Ein `Point` und ein `Circle` sind (spezialisierte) `Shape`-Objekte und können daher als `Shape`-Verweise deklariert werden. Gleiches gilt für ein `Rectangle` als Unterklasse von `Point`. Zunächst ist unklar, welche Implementierung der Methode `scale()` zum Beispiel für `shape2` ausgeführt wird. Die Variable `shape2` ist ein Verweis auf eine `Shape`-Instanz, in `Shape` ist `scale()` aber eine abstrakte Methode. `shape2` verweist allerdings in Wirklichkeit auf ein `Rectangle`. Zur Laufzeit wird in Java die Klasse des referierten Objekts bestimmt und die zu der Klasse gehörende Methode — also die aus `Rectangle` — ausgeführt. Dies wird die *dynamische Bindung* von Methoden genannt. Im Gegensatz dazu wird bei *statischer Bindung* die exakte Methode bereits zur Übersetzzeit ermittelt. Das obige Beispiel wäre daher für eine statische Bindung — welche in Java für `private` deklarierte Methoden und für Klassenmethoden verwendet wird — nicht geeignet. Der Compiler müßte dazu per Flußanalyse die exakte Klasse der Objekte bestimmen, um die richtige Methode zu binden. Dies ist nur für so einfache Beispiele wie oben möglich.

Für `shape1` und `shape3`, obwohl die Variablen beide auch Verweise auf eine `Shape`-Instanz sind, wird die Implementierung von `scale()` aus `Point` bzw. `Circle` ausgeführt. Dieses je nach Empfängerobjekt polymorphe Verhalten einer Methode — wie hier bei `scale()` — wird *Polymorphismus* genannt.

Die Klasse `Point` adaptiert und implementiert das Interface `Dumpable`. Die direkten oder indirekten Unterklassen `Rectangle` und `Square` erweitern die Methode sinnvoll. Die Methode `dump()` ist aber nicht in der Klasse `Shape` deklariert. Daher müssen die drei Verweise auf die `Shape`-Instanzen auf einen Verweis auf ein `Dumpaple` gewandelt werden. `System.out.println()` ist eine Methode des Java-Systems, die das Argument auf der Standardausgabe ausgibt. Sind alle Klassen korrekt übersetzt, hat das Beispiel wie erwartet folgende Ausgabe:

```
$ java Main
A Point at (2.0,3.0)
A Rectangle at (-1.0,-8.0) height 4.0 with 6.0
A Square at (1.0,1.0) height 8.0 with 8.0
```

Persistente Objekte

Unter der Serialisierung von Objekten bzw. unter persistenten Objekten versteht man die Möglichkeit, die Daten, d.h. den Zustand eines Objekts, in eine Folge von Bytes zu wandeln, so daß bei der Deserialisierung aus den Bytes wieder ein zum Ausgangsobjekt äquivalentes Objekt erzeugt werden kann ([Rig96]). Die serialisierten Bytes können zum Beispiel in einer Datei gespeichert oder über eine Netzverbindung gesendet werden. Damit kann ein Objekt “eingefroren” und zu einem späteren Zeitpunkt durchaus auf einer anderen Maschine wieder zum Leben erweckt werden.

Design-Patterns

Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides führen in ”Design Patterns — Elements of Reusable Object-Orientated Software” ([Gam95]) eine Vielzahl von Softwaretechniken für die objekt-orientierte Programmierung vor. Diese Standardtechniken finden in der Objekt-Orientierung immer wieder Verwendung. Als Beispiel sei hier kurz das *Factory*-Entwurfsmuster vorgestellt:

Eine Factory als ein Erzeugungs-Entwurfsmuster hat die Aufgabe, Objekte zu erschaffen, wobei sie die Art der erzeugten Objekte verkapselt und nur ein Verhalten in Form von Methoden verspricht. Das Verhalten wird in Java als Interface oder abstrakte Klasse formuliert. Auch die Factory-Schnittstelle wird durch ein Interface oder eine abstrakte Klasse definiert.

Hier als Beispiel ein Interface für Factories, welche grafische Objekte als Instanzen einer beliebigen Unterklasse der abstrakten `Shape`-Klasse liefern:

```
public interface ShapeFactory {
    public Shape createShape();
}
```

Verschiedene Klassen adaptieren das Factory-Interface und liefern unterschiedliche Objekte. Hier zwei Implementationen, die Punkte oder Kreise erzeugen:

```
public class PointShapeFactory implements ShapeFactory {
    public Shape createShape() {
        return new Point(0.0, 0.0);
    }
}
public class CircleShapeFactory implements ShapeFactory {
    public Shape createShape() {
        return new Circle(0.0, 0.0, 1.0);
    }
}
```

Eine Applikation verwendet zur Laufzeit eine konkrete Factory, wobei deren exakte Klasse für die Applikation unerheblich ist. Typischerweise kann der Anwender aber wählen, welche Factory verwendet wird und steuert damit, welche konkreten Objekte in der Applikation verwendet werden.

Exceptions

Ein Laufzeitfehler — berichtet und ausgelöst in Form einer Exception — oder eine vom Entwickler selbst erzeugte und geworfene Exception löst eine Ausnahme aus. Das Auslösen einer Exception bricht die Aufrufverschachtelung vom Punkt ihrer Auslösung zurück zum Ursprung ab. Eine Exception kann abgefangen werden, d.h., der Abbruch der Aufrufverschachtelung endet an dieser Stelle. Der Empfänger der Ausnahme hat in Java die Möglichkeit, sie entweder zu behandeln, zu ignorieren oder selbst neu auszulösen. Wird die Ausnahme von keinem Programmteil abgefangen, so führt sie zum Abbruch des ausführenden Threads und zur Ausgabe einer Fehlermeldung.

In Java werden Exceptions als Objekte der Klassen `Throwable` bzw. deren Unterklasse `Exception` repräsentiert und werden daher ganz normal mit `new` erzeugt. Ausgelöst wird eine Exception mit `throw`, und ein `try`/`catch`-Block fängt Exceptions nach Klassen als Muster ab.

Speicherverwaltung

In einem objekt-orientierten Programm werden sehr viele Objekte, welche zum Teil nur kurzzeitig verwendet werden, erzeugt. Je nach Sprache müssen nicht länger gebrauchte Objekte und damit der von den Objekten allokierte Speicher explizit oder implizit wieder freigegeben werden. In Java werden Objekte implizit entsorgt. In einem eigenen Thread entdeckt die *garbage collector* durch eine Referenzzählung Objekte bzw. ganze Inseln von Objekten, auf die keine Referenz mehr verweist, und gibt diese frei.

2.2.2 Vorteile und Nachteile der Objekt-Orientierung

Die objekt-orientierte Programmierung bietet aus meiner Sicht viele Vorteile, welche zum Teil bereits in der Einführung zur Objekt-Orientierung kurz skizziert worden sind:

Objekte und Klassen werden in der Entwicklung einer komplexen Software zur Analyse, zur Entwicklung und zur Programmierung des Systems verwendet. Einzelne Klassen kapseln bzw. modellieren kleine und überschaubare Einheiten der Systemwelt. Sie verbergen dabei nach außen die

exakte Repräsentation (*information hiding*) und bieten über eine Methodenschnittstelle ihre Funktionalität an. Diese Nähe zu natürlichen Dingen und die Kapselung der Interna machen die Objekte bzw. die Klassen in der Objekt-Orientierung für den Entwickler so attraktiv.

Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer* ([Aho83]). Durch die Modellierung in Klassen können Algorithmen in kleine Teilalgorithmen pro Klasse zerlegt werden, welche jeweils ein kleines Teilproblem lösen. Das Zusammenspiel aller Klassen und Methoden stellt dann den ganzen Algorithmus dar. Selbst komplexe Algorithmen zerfallen so in kleine, einfache Teilprobleme. Die Modellierung in Objekten und die Interaktionen zwischen Objekten sind im Kleinen wie im Großen eine Achse der Objekt-Orientierung.

Die Vererbung bringt zwei wesentliche Vorteile: Zum einen werden durch das Spezialisieren von Klassen durch Unterklassen sehr leicht nützliche neue Klassen erzeugt, welche wiederum ganz natürlich in das System von Objekten integriert werden können. Zum anderen können Unterklassen Methoden der Oberklassen überschreiben, wodurch zusammen mit der dynamischen Bindung eine Unterklasse das Verhalten der Oberklasse — zum Beispiel als Teil eines komplexeren Algorithmus — abändern kann (Template-Entwurfsmuster). Ein Objekt der neuen Klasse nimmt anstelle eines Objekts der Oberklasse ganz natürlich an dem Zusammenspiel des komplexen Algorithmus teil und ändert diesen so schnell und einfach ab.

Durch die Abstraktion und Kapselung wird die Wiederverwendung von Programmelementen gefördert. Ein einfaches Beispiel dafür sind *Collections*, also Objekte, die Sammlungen anderer Objekte aufnehmen und intern speichern (zum Beispiel Listen oder Hashtabellen). Collections sind intern oft kompliziert aufgebaut, besitzen aber eine einfache Schnittstelle in Form von Methoden und können so sehr einfach wiederverwendet werden. Immer wenn im Programm eine entsprechende Collection benötigt wird, muß lediglich ein Objekt der passenden Klasse erzeugt werden, und das Programm kann über die einfach zu bedienende Schnittstelle darauf zugreifen. Wiederverwendung erhöht die Effizienz und Fehlerfreiheit beim Programmieren. Das Entwicklungsrisiko für komplexe Projekte sinkt, und die Pflege bzw. Erweiterung existierender Software wird einfacher.

Strukturen sind durch Graphen von Objekten spezifischer Klassen (zum Beispiel als Baum von Objekten) leicht zu modellieren. Der Graph kann klassisch von außen durch Funktionen, die sich entlang der Struktur bewegen, oder über das Visitor-Entwurfsmuster traversiert und so analysiert bzw. bewertet werden. Klassen und damit Objekte besitzen aber Zustand und Methoden. Den Knoten eines Graphens — zum Beispiel der Wurzel eines Baums — kann daher eine Methode als Aufforderung zur Analyse geschickt werden, bzw. ein Knoten kann zur Analyse bei anderen Knoten Methoden — im Baum zum Beispiel bei Unterknoten — aufrufen. Durch den Einsatz von Vererbung ist diese Technik darüber hinaus schnell und einfach (auch nur lokal) zu variieren. Der Baum mit seinen Methoden repräsentiert alle Informationen bzw. beinhaltet alle nötigen Algorithmen.

Analog zur Wiederverwendung von Klassen macht die persistente Speicherung von Objekten die Objekte mit ihrem Zustand jederzeit und auf verschiedenen Maschinen wiederverwendbar. Die äquivalenten Objekte können so von verschiedenen Software-Systemen geteilt werden und müssen nicht erst erzeugt und in den passenden Zustand versetzt werden. Nicht alle beteiligten Systeme wollen und können wissen, wie das Objekt zu seinem aktuellen Zustand gekommen ist. Damit ist ein konformes Verhalten auf allen System garantiert.

Die von Gamma, et. al vorgestellten Entwurfsmuster der objekt-orientierten Programmierung sind mächtige Programmiertechniken der Objekt-Orientierung: Als Beispiel verschiebt der Einsatz des Factory-Entwurfsmusters die Erzeugung von Objekten während der Laufzeit eines Algorithmus auf die Factories. Durch vom Anwender zu implementierende Factories werden so die erzeugten Objekte in einem Algorithmus zur Laufzeit durch den Anwender frei wählbar. Dies macht die Algorithmen mächtiger und wesentlich flexibler.

Zwei wesentliche Nachteile werden der Objekt-Orientierung zugeschrieben: Komplexe objekt-orientierte Systeme bestehen aus sehr vielen Klassen und Objekten, wodurch der Anwender unter Umständen den Überblick verlieren kann. Dies ist aber für alle komplexe Systeme ein Problem.

Der große Nachteil der Objekt-Orientierung ist sicherlich der Performanceverlust gegenüber zum Beispiel Programmen einer prozeduralen Programmiersprache. Zur Laufzeit eines objekt-orientierten Programms werden sehr viele Objekte erzeugt und auch wieder vernichtet. Allein die Reservierung und Freigabe des Speicherplatzes der Objekte kosten Zeit. Auch weitere Aspekte eines objekt-orientierten Programms kosten Rechenzeit. Als Beispiel wird für einen Methodenaufruf per dynamische Bindung erst zur Laufzeit die zu verwendende Implementierung ermittelt.

2.3 Stand der Objekt-Orientierung im Compilerbau

Die Objekt-Orientierung eröffnet neue Möglichkeiten zur Realisierung von Compilern. Es existieren bereits einige Werkzeuge und Systeme des Compilerbaus, welche eine objekt-orientierte Sprache verwenden und teilweise auch eine objekt-orientierte Verwendung der Sprache nutzen. Diese unterscheiden sich aber teilweise wesentlich von meinen Ideen und Lösungen. Im Folgenden werden die Werkzeuge und Systeme vorgestellt und kurz aufgeführt, wo genau die einzelnen Schwachpunkte liegen und wieso diese Werkzeuge daher meines Erachtens alle nicht ideal in bezug auf den Einsatz der Objekt-Orientierung im Compilerbau sind.

2.3.1 A. Appel, Modern Compiler Implementation in Java

Das Buch “Modern Compiler Implementation in Java” von Andrew W. Appel ([App97]) ist eines von drei Büchern des Autors zu dem Thema “Modern Compiler Implementation”. Parallel beschreiben zwei weitere Bücher die analogen Inhalte für die Programmiersprachen C und ML.

Appel verwendet in der Java-Version seines Buchs als Scanner- und Parsergenerator die Tools *CUP* und *JLex*. Beide Werkzeuge werden in den nächsten beiden Abschnitten näher erläutert.

Der Autor geht Kapitel für Kapitel die verschiedenen Phasen eines Compilers durch. Er betrachtet nicht nur die lexikalische, syntaktische und semantische Analyse, sondern er geht auch auf die folgenden, teilweise recht kleinen Schritte bis hin zur Registerallokierung näher ein.

Appel baut für erkannte Programme und in den weiteren Schritten eines Compilers abstrakte Syntax-Bäume, die in Java natürlich aus Objekten bestehen. Dabei kann eine Phase des Compilers bei der Analyse des Baums durchaus für die nächste Phase einen neuen Baum über einer anderen Klassenbibliothek erzeugen. Bis auf Konstruktoren und Instanzvariablen zum Speichern von Werten besitzen die Objekte der Bäume aber keine weiteren, wesentlichen Eigenschaften. In einer Diskussion erläutert der Autor, daß seine Bäume ganz bewußt keine Methoden zur selbständigen Traverse des Baums besitzen. Dazu müßte bei einer Erweiterung der Klassenhierarchie um eine neue Funktionalität pro Klasse eine neue Methode implementiert werden. Er plädiert daher für eine von außen kommende Traverse der Bäume. Dabei nutzt er nicht einmal das Visitor-Entwurfsmuster, sondern untersucht die Knoten des Baums in einer `if`-Kette mit dem `instanceof`-Operator und verteilt bei Erfolg auf Klassen spezifische Methoden.

Derartige Klassen-Bibliotheken für die verschiedenen Phasen eines Compilers sind natürlich wiederverwendbar. Soll der Compiler Programme einer anderen Programmiersprache erkennen, müssen lediglich die semantische und syntaktische Analyse angepaßt werden, um für Programme der neuen Sprache Bäume über der existierenden Bibliothek zu bauen.

Allerdings implementiert Appel zum Beispiel die Regeln für die impliziten Typenwandlungen fest vor. Er sieht keine Technik vor, wie der Anwender der Klassen diese Regel nach seinen Bedürfnissen anpassen kann.

Seine Traverse der Bäume ist — wie er selbst zugibt — auf keinen Fall objekt-orientiert. Nur für Klassen, deren Funktionalität sich oft ändert, macht sein Ansatz Sinn. Seine Hierarchien sollen aber für verschiedene Compiler Wiederverwendung finden. Daher werden diese sich nicht ändern, und eine objekt-orientierte Traverse wäre vorzuziehen. Ich vermute viel mehr, daß er die gleichen Strukturen — mehr sind seine Bäume nicht — auch für die anderen Sprach-Versionen seines Buchs verwenden will.

CUP (Constructor of Useful Parsers)

CUP ([Cup]) ist analog zu *jay* eine an *yacc* angelehnte Neuimplementierung von LR(1) in Java. Der generierte Parser stammt von einer festen Klasse ab, d.h., man kann — zum Beispiel für eine andere Art von Fehlermeldungen — Methoden der Basisklasse überschreiben. Die Generierung kann durch sehr viele Anweisungen und Optionen beeinflußt werden, und jedes Symbol muß vor den Regeln definiert werden.

Die Generierung erzeugt eine Datei mit der Klasse des erzeugten Parsers und eine Datei mit einem Interface, welches für jedes Symbol einen konstanten, primitiven `int`-Wert festlegt. Damit kann eine *CUP*-Grammatik aber keine zitierten Zeichen oder gar zitierten Text als Symbol enthalten. Die Schnittstelle zum Scanner ist ein Interface mit einer Methode, welche, in einem `Symbol`-Objekt gekapselt, ein optionales Werte-Objekt und einen der `int`-Werte aus dem Interface liefert.

Die Quellen von *CUP* sind frei verfügbar und sind im Bereich der Homepage zu finden. Leider enthält *CUP* noch immer einige bekannte Fehler, und die letzte offizielle Version ist von Februar 1999; die letzte Beta-Version stammt von September 1999. Ein weiterer Support ist wohl nicht zu erwarten.

CUP kann Zeichenpositionen in der Eingabe verwalten, womit präzise Fehlermeldungen möglich sind. Leider verhält sich *CUP* in der Fehlerbehandlung in Feinheiten anders als aus *yacc* bekannt.

Insgesamt ist *CUP* lediglich ein in einer Klasse eingebetteter, klassischer Ansatz. Der Parser selbst nutzt die Objekt-Orientierung nicht. Er ist daher auch kaum um neue Konstrukte wie Wiederholungen eines Teils der Grammatik zu erweitern.

JLex (Java Lex)

JLex ([JLex]) ist eine Neuimplementierung von *lex* in Java. Die Syntax der Tabelle von Muster/Aktions-Paaren unterscheidet sich kaum von *lex*. Im Vorspann sind sehr viele Java-ismen zu beachten bzw. Anweisungen zu verwenden, welche die generierte Klasse beeinflussen. Insgesamt ist das System stark auf Zusammenarbeit mit einem Parser zugeschnitten.

Die Quellen von *JLex* sind analog zu *CUP* auch frei verfügbar. Leider enthält auch *JLex* einige bekannte Fehler, und die letzte offizielle Version ist von September 2000...

In Mustern repräsentiert `$` nicht das Zeilenende, was Muster für verschiedene Plattformen mühsam macht. Die Unterstützung von Unicode ist optional, führt aber zu wesentlich längeren Berechnungszeiten für den internen, endlichen Automaten. Unklar bleibt, wie zum Beispiel ein regulärer Ausdruck für Java-Bezeichner, welche zum Beispiel griechische oder kyrillische Zeichen enthalten können, kompakt anzugeben ist. Für eine echte Unicode-Unterstützung fehlen Makros für gängige Zeichen-Klassifizierungen.

Insgesamt ist *JLex* wie *CUP* lediglich ein in einer Klasse eingebetteter, klassischer Ansatz. Der erzeugte Scanner selbst nutzt die Objekt-Orientierung nicht.

2.3.2 jay (Java yacc)

jay ([Jay], [Küh99]) ist eine Umsetzung von *yacc* für Java und entstand durch Modifikation der *yacc*-Quellen von BSD-Lite. *jay* übernimmt daher die Algorithmen von *yacc*, liest also eine LR(1) BNF-Grammatik mit Aktionen, formuliert in Java, ein und generiert ein Java-Programm, das optional eine

verbesserte Ablaufverfolgung enthält. *jay* wurde von Axel-Tobias Schreiner im Rahmen der Vorlesung "Compilerbau mit Java" ([Sch98a]) entwickelt.

jay selbst ist in C implementiert und von der Homepage ([Jay]) frei verfügbar. *jay* kann zwar auf vielen Plattformen übersetzt werden, ist aber weniger portabel als eine neue Implementierung in Java.

jay verpackt den erzeugten Parser in eine vom Benutzer definierte Klasse. Die berechneten Tabellen sind in statischen inneren Klassen der Parser-Klasse gekapselt. Damit versucht *jay* die 64 KByte Grenze von Java zu umgehen.

Da *jay* die bewährte Implementierung der Algorithmen von *yacc* übernommen hat, ist zum Beispiel (und im Gegensatz zu *CUP*) die Fehlerbehandlung wirklich identisch, auch wenn die Plazierung der `error`-Token die Grammatiken mehrdeutig und schwer lesbar machen.

In C wird eine `union` als Typ des Werte-Stacks verwendet. In Java ist man ganz auf eine Klasse angewiesen. Da aber in der Regel wohl ein Baum gebaut wird, besteht der Werte-Stack aus Objekten, und das Fehlen einer `union` ist irrelevant.

Ein inneres Interface der erzeugten Parser-Klasse definiert die Schnittstelle für Scanner. Analog zu *yacc* sind alle Symbole durch ein `int`-Wert repräsentiert. In *jay* werden diese als öffentliche Klassenvariablen des Parsers für die Scanner hinterlegt. Damit können in Eingabe-Grammatiken für *jay* im Gegensatz zu *CUP* wenigstens zitierte Zeichen erkannt werden.

jay ist dank der Verwendung der originalen *yacc*-Quellen und dank der eleganteren Einbettung des erzeugten Parsers in eine Klasse dem Konkurrenten *CUP* vorzuziehen. Aber wie *CUP* stellt *jay* lediglich eine nicht objekt-orientierte Technik, verkleidet in einer Klasse, dar.

2.3.3 ANTLR (ANother Tool for Language Recognition)

ANTLR ([Antlr], [Par93]) ist ein in Java geschriebenes System von Terence Parr zum Erzeugen von Parsern und deren lexikalische Analyse für C++ und Java. *ANTLR* akzeptiert EBNF-artige Grammatiken zur Beschreibung der lexikalischen Analyse und zur Sprachbeschreibung. Optional konstruiert ein erzeugter Parser zur Laufzeit automatisch einen abstrakten Syntaxbaum für erkannte Programme der Sprache. Deren Traverse kann mit einer weiteren Grammatik beschrieben werden.

ANTLR erzeugt aus einer Beschreibung der lexikalischen Analyse in Form einer Grammatik eine Klasse, welche die spezifizierten Symbole erkennt. *ANTLR* erlaubt die Verwendung von Unicode, rät aber in der Dokumentation davon noch ab. Die generierte Scanner-Klasse stammt fest von einer Klasse des Systems ab und adaptiert ein Interface als Schnittstelle zwischen einem *ANTLR*-Parser und der lexikalischen Analyse. Jeder Scanner, der dieses Interface implementiert — also zum Beispiel auch ein handgeschriebener Scanner —, kann aber verwendet werden.

Als Parser erzeugt *ANTLR* eine Klasse, welche von einer festen Klasse abstammt und nach der Technik des rekursiven Abstiegs arbeitet. Dabei wird aus jeder Regel der Grammatik eine Methode im Parser. Lokal pro Regel bzw. Methode ist der lookahead, d.h. die Anzahl von Symbolen zur Entscheidungsfindung, einstellbar. Über weitere Anweisungen in der Grammatik kann der Kopf der Methoden (mögliche Exceptions, Anzahl und Typen der Parameter, ...) eingestellt werden. Aktionen werden in geschweiften Klammern nach Grammatikteilen angegeben, welche über eine weitere Syntax in der Grammatik Zugriff auf lokale Variablen und die Resultate von Unterregeln bzw. die Werte von Symbolen haben.

ANTLR erlaubt Grammatiken, welche über eine zusätzliche Syntax mit semantischen und syntaktischen Prädikaten versehen sind, d.h. mit Nebenbedingungen, die erfüllt sein müssen, damit ein Zweig der Grammatik erkannt wird. Ein semantisches Prädikat ist ein Java- (oder C++-) Ausdruck, der als Resultattyp `boolean` liefern muß. Nur wenn zur Laufzeit des Parsers der Ausdruck wahr ist, wird der mit diesem Prädikat versehene Teil der Grammatik bei der Parsierung berücksichtigt. Ein syntaktisches Prädikat beschreibt eine Symbolfolge, welche nach der möglichen Erkennung des markierten Teils der

Grammatik zu folgen hat. Alleine wegen der Prädikate, aber auch wegen des lokal einstellbaren lookahead, wird der Parser mit Backtracking arbeiten müssen. Die Prädikate sind eine mächtige Technik, machen die Grammatik als Ganzes aber teilweise nur schwer begreiflich. Darüber hinaus können Grammatiken von einer anderen Grammatik abstammen. Dies bedeutet nichts anderes, als daß die Summe der beiden Grammatikdateien die wirkliche Eingabe für *ANTLR* darstellt.

ANTLR-Parser bauen optional für erkannte Teile eines Programms einen abstrakten Syntaxbaum. Für sinnvolle Bäume sollte durch eine nochmals zusätzliche Syntax in der Grammatik die Konstruktion des Baums gesteuert werden. Dabei können allerdings nur Eingabesymbole als Knoten im Baum repräsentiert werden und kapseln den Typ (einen `int`-Wert) und den Text des Symbols. Die Knoten eines solchen Baums müssen Objekte von Klassen sein, die das `AST`-Interface implementieren. Das System verwendet eine Standardimplementierung, eigene Klassen können — wieder über eine Extrasyntax in der Grammatik — aber auch verwendet werden.

Durch eine weitere, mit Aktionen versehene Grammatik werden sogenannte `TreeWalker`-Klassen zur Traverse der Syntaxbäume von *ANTLR* generiert. Dabei enthält die rechte Seite einer Regel Entwurfsmuster, welche Knoten nach dem repräsentierten Eingabesymbol und dem Eingabesymbol der Unterknoten auswählen. Ein `TreeWalker` ist dann ein Parser, welcher nicht auf einer Symbolfolge, sondern auf dem Baum als Eingabe arbeitet.

Instanzen der Standardimplementierung des `AST`-Interfaces sind serialisierbar und damit zu einem späteren Zeitpunkt wiederverwendbar. Alternativ kann der Baum als XML-Datei serialisiert werden. Dabei wird aus dem Klassennamen des Knotens der Elementname, und aus dem Typ und dem Text des repräsentierten Symbols werden Attribute. Unterknoten werden zu Unterelementen.

Während der Parsierung löst ein Eingabefehler eine Exception aus. Pro Regelmethode generiert *ANTLR* automatisch einen Block, der derartige Exceptions abfängt, den Fehler berichtet und in der Eingabe fortschreitet, bis das aktuelle Eingabesymbol ein Element der follow-Menge ist. Über eine weitere Syntax in der Grammatik können aber eigene Blöcke für verschiedenste Teile der Grammatik hinterlegt werden.

Grammatiken werden von *ANTLR* auf ihre Verwendbarkeit (Mehrdeutigkeit, unendliche Rekursionen, ...) geprüft. Ein kurzer Test zeigt aber, daß *ANTLR* selbst einfache unendliche Rekursionen leider nicht immer korrekt findet. Zitierter Text in der Parser-Grammatik wird nicht wirklich unterstützt.

Die Vermischung der Grammatik mit Aktionen, mit Prädikaten, mit Anweisungen zum Einstellen der Methodenköpfe, mit `try`/`catch`-Blöcken für die Fehlerbehandlung, mit Steueranweisungen für die Baumgeneration und mit verschiedenen anderen Anweisungen machen die Grammatik bzw. die Anweisungen in der Grammatik sehr unleserlich und insgesamt unelegant. Darüber hinaus werden Symbole in der Grammatik lediglich daran erkannt, daß diese mit einem Großbuchstaben beginnen; analog haben Regelnamen mit einem kleinen Buchstaben anzufangen. All dies läßt einem die Eingabe für *ANTLR* recht eigenwillig vorkommen. Die Entwickler von *ANTLR* scheinen dies aber auch bemerkt zu haben, und *ANTLR* gibt über eine Option die undekorierte Grammatik als HTML-Datei aus, soweit das zum Beispiel bei semantischen Prädikaten überhaupt möglich ist.

Die erzeugten Scanner, Parser und Traversen nutzen selbst die Objekt-Orientierung nicht. Sie verwenden pro Methode große `switch`-Anweisungen, um den weiteren Fortschritt zu verteilen.

ANTLR selbst ist ein in sich abgeschlossenes System. Eine Erweiterung ist für den Anwender kaum möglich. Die Erkennung von allen möglichen Permutationen wie zum Beispiel den Attributlisten eines XML-Elements ist damit in *ANTLR* nur durch eine Grammatik möglich, welche alle diese Permutationen aufführt.

Die Traverse von erzeugten abstrakten Syntax-Bäumen ist keinesfalls objekt-orientiert. Zum Beispiel ist keine echte Unterstützung des Visitor-Entwurfsmusters vorhanden. Wünschenswert wäre eine

Technik, in der Klassen die Traverse ihrer Oberklassen erben könnten, oder eine Technik, in der nach Klassen als Muster selektiert werden kann.

Die Fehlerbehandlung durch das Abfangen von Exceptions läßt viele Wünsche offen. Die Standarderholung kann mit einem fehlenden Symbol nicht korrekt umgehen. Eine gute Technik muß dem Anwender die Möglichkeit bieten, Symbole zu verwerfen, fehlende Symbole zu substituieren und an einer anderen Stelle in der Methodenaktivierung fortzufahren. Für eine gut abgesicherte Grammatik sind daher alle kritischen Punkte der Sprache mit einem `try`/`catch`-Block zu versehen. Diese Punkte sind aber für komplexe Grammatiken nicht einfach zu finden, und der Programmtext für die Erholung macht die Grammatik noch unübersichtlicher.

Als Fazit ist *ANTLR* ein sehr mächtiges Werkzeug, auch wenn verschiedene Teile des Systems nicht objekt-orientiert bzw. elegant erscheinen. Die teilweise veraltete, sich auf Java 1.0 beziehende Dokumentation bzw. die Komplexität von *ANTLR* machen den Zugang zu dem System nicht einfach. Auch wenn das letzte offizielle Release von Oktober 2000 stammt, ist *ANTLR* in der Branche recht bekannt und wird sehr aktiv unterstützt durch die Mitglieder der *ANTLR*-Mailingliste, an der sich auch Terence Parr aktiv beteiligt.

2.3.4 Java Compiler Compiler™ (JavaCC)

JavaCC ([JavaCC], [Ens00]) ist ein vollständig in Java implementierter LL(k)-Parsergenerator für Java. *JavaCC* wurde ursprünglich von Sun und dann von Metamata (`http://www.metamata.com`) entwickelt. Metamata ging vor kurzem in die Firma Webgain (`http://www.webgain.com`) auf.

JavaCC liest eine Quelle und erzeugt daraus normalerweise eine Parser-Klasse, einen Scanner und eine Reihe von Hilfsklassen bzw. Interfaces. Die Quelle enthält in der Regel eine Gruppe von Optionen zur Steuerung der Übersetzung, Java-Programmtext (der mindestens die Parser-Klasse definieren muß), Regeln mit regulären Ausdrücken (die mindestens die uninteressanten Eingabezeichen definieren) und schließlich die Regeln der Grammatik in einer an eine Methode erinnernde Schreibweise.

Pro Regel bzw. Methode akzeptiert *JavaCC* EBNF-Konstrukte, welche mit einer recht eigenwilligen Syntax mit Aktionen in Java, Variablendefinitionen, lokalen Einstellungen, Bindung von lokalen Variablen an Teile der Grammatik und verschiedenen anderen Dingen vermischt sind. Aus jeder Regel wird in dem erzeugten Parser je eine Methode, welche sich nach der Technik des rekursiven Abstiegs gegenseitig aufrufen. Die Startregel der Grammatik spielt keine besondere Rolle, da jede Parser-Methode explizit aufgerufen werden kann.

Die Scanner-Definition beschreibt Symbole anhand einer an regulären Ausdrücken angelehnten Syntax. Leider ist zumindest der Punkt als Platzhalter für ein beliebiges Zeichen nicht erlaubt, und typische Zeichenklassen sind auch nicht voreingestellt. Erkannte Symbole werden in der Grammatik als `Token`-Objekte an den Parser geliefert und können, wenn sie an eine lokale Variable gebunden sind, in den Aktionen nach dem erkannten Text gefragt werden. Unicode wird unterstützt, aber ohne Zeichenklassen, zum Beispiel für alle Unicode-Letter, macht dies keinen Sinn. Überschneiden sich durch Ausdrücke definierte Symbole und direkt in der Grammatik zitierter Text, hat immer das Symbol Vorrang. Sind Symboldefinitionen mehrdeutig, sind zumindest mir die Regeln nicht klar...

Normalerweise prüft *JavaCC* Grammatiken auf LL(1). Mehrdeutigkeiten eliminiert man durch lokale oder globale, syntaktische oder semantische Maßnahmen. Leider entdeckt auch *JavaCC* analog zu *ANTLR* selbst einfache unendliche Rekursionen nicht, und auch Mehrdeutigkeiten, verursacht durch mehrdeutige Symboldefinitionen, werden nicht alle entdeckt.

Global oder auch lokal pro Methode kann der lookahead auf größer als eins eingestellt werden. Darüber hinaus unterstützt *JavaCC* einen sogenannten syntaktischen und semantischen lookahead, welche den syntaktischen und semantischen Prädikaten von *ANTLR* gleichkommen.

Tritt während der Parsierung ein Syntaxfehler auf, löst dies eine `ParseException` aus. Im Gegensatz zu *ANTLR* werden aber keine Standard-Exceptionhandler pro Regelmethode generiert. Somit durchbricht die erste Exception bereits die komplette Aufrufverschachtelung. Aber auch in *JavaCC* können `try`/`catch`-Blöcke zur Fehlerbehandlung in die Grammatik gestreut werden.

jjtree ist ein Präprozessor, der die Generierung von Parse-Bäumen weitgehend automatisieren soll und dessen Ausgabe als Eingabe für den eigentlichen Parsergenerator *javacc* verwendet wird. Die Quellen von *jjtree* sind mit Anmerkungen versehene Eingabedateien für *javacc*. *jjtree* wandelt die Anmerkungen in Aktionen in der Parser-Grammatik um, welche den Baum als Repräsentation eines erkannten Programms bauen.

jjtree legt Knoten per default für Nonterminals — und keine für Terminals — auf einem internen Stack ab und faßt sie typischerweise pro Nonterminal zusammen. Durch Anweisungen in der Eingabe von *jjtree* kann die Erzeugung der Knoten gesteuert werden und können die expliziten Knotenklassen und die Anzahl der Abkömmlinge, die der Baumknoten enthalten soll, spezifiziert werden. Knotenklassen müssen einem generierten Interface `Node` genügen. Per default werden Knoten einer ebenfalls erzeugten Standardimplementierung des Interfaces verwendet.

Optional wird von *jjtree* zusätzlich ein Interface, welches für jede Knoten-Klasse eine Methode enthält, zur Verwendung des Visitor-Entwurfsmusters erzeugt und in den erzeugten Knotenklassen unterstützt. Die Visitor-Schnittstelle ist sehr elegant, aber ohne Werkzeug kaum zuverlässig zu beherrschen. Man kann Visitor problemlos mehrmals implementieren und damit zum Beispiel für Programme einer Sprache Code für verschiedene Architekturen erzeugen.

Durch die Vermischung von EBNF, `try`/`catch`-Blöcken, verschiedenen lookahead-Einstellungen, Java-Programmtext in Form von Funktionsköpfen, -aufrufen, Variablendefinitionen und Zuweisungen entsteht eine sehr unübersichtliche Repräsentierung der Grammatik. Analog zu *ANTLR* kann daher die Kerngrammatik durch das Programm *jjdoc* als HTML oder reine Textdatei extrahiert werden, soweit diese außerhalb von Aktionen oder Tests für einen semantischen lookahead sichtbar ist.

jjtree ist im ersten Ansatz ein nützliches Werkzeug. Leider sind Syntax und Konventionen zur Manipulation der Knoten während der Konstruktion in der Regel nur durch Studium des generierten Codes zu erahnen, und die eigentliche Grammatik tritt noch einmal weiter in den Hintergrund. Insgesamt überwiegen für mich diese Probleme, denn die Konstruktion eines Baums an sich ist auch ohne Präprozessor eher trivial.

Der Parsergenerator erzeugt aus einer Quelle viele Interfaces und Klassen. Wenigstens einige davon sollten besser innere Klassen sein. Speziell bei der Fehlerbehandlung muß man unangenehm viele Interna kennen. Da die Fehlerbehandlung sehr ähnlich zu *ANTLR* ist, gilt hier die gleiche Kritik. Das Substituieren von fehlenden Zeichen ist nur mit Handarbeit möglich. Die für eine Erholung kritischen Stellen der Grammatik sind mühsam abzusichern und machen die Grammatik noch unleserlicher.

Die von *JavaCC* erzeugten Parser und Scanner nutzen selbst intern die Objekt-Orientierung nicht. Wiederum analog zu *ANTLR* wird ein Fortschreiten über mehr oder weniger große switch-Anweisungen bzw. if-Ketten verteilt. *JavaCC* kapselt lediglich klassische Ansätze in Klassen.

JavaCC ist wie *ANTLR* ein in sich abgeschlossenes System und kann vom Anwender kaum erweitert werden. Positiv ist, daß das Paket neben dem System auch eine Vielzahl von Beispielen enthält.

Auch wenn die *JavaCC*-Homepage etwas anderes aussagt, ist die letzte Version *JavaCC* 2.1 und wurde im August 2001 veröffentlicht. *JavaCC* war früher ein sehr beliebter Parsergenerator für Java. Obwohl *JavaCC* durch eine Mailing-Liste und eine Newsgroup betreut wird, scheinen die Alternativen *JavaCC* Anwender abgenommen zu haben.

2.3.5 SableCC

SableCC (benannt nach einer kanadischen Insel namens Sable, ([SableCC], [Gag98a], [Gag98b])) akzeptiert als Eingabe eine Grammatik zusammen mit einer Beschreibung der lexikalischen Schicht. *SableCC* generiert als Ausgabe vier Java-Pakete mit Java-Klassen bzw. -Interfaces:

Das Paket `parser` beinhaltet unter anderem den aus einer (sehr) eingeschränkten EBNF-artigen Grammatik erzeugten LALR(1)-Parser. Der Parser arbeitet auch klassisch auf den aus der Grammatik berechneten Tabellen, welche serialisiert in dem Katalog des Pakets hinterlegt sind.

Das `lexer`-Paket sammelt unter anderem den aus der ebenfalls EBNF-artigen Beschreibung erzeugten Scanner. Der Scanner arbeitet als deterministischer, endlicher Automat auf einer Tabelle, welche wiederum serialisiert als Datei in dem Katalog des Pakets zu finden ist.

SableCC-Grammatiken sind nicht mit Aktionen zu erkannten Teilen einer Grammatik vermischt. Ein *SableCC*-Parser erzeugt immer automatisch einen abstrakten Syntax-Baum als Repräsentation eines erkannten Programms. Das Paket `node` sammelt Klassen und Interfaces, deren Instanzen die Knoten der Bäume darstellen bzw. eine Traverse der Bäume mit Hilfe des Visitor-Entwurfsmusters beschreiben.

Zusätzlich enthält das vierte Paket `analysis` fertige Adapter-Klassen als Visitor-Implementierung, welche die erzeugten Bäume “depth first” oder “reversed depth first” traversieren. Durch Ableiten der Klassen und Ersetzen von Methoden wird der Anwender so schnell zu einem Ergebnis kommen.

Die Namen der Pakete können über eine Anweisung in der Grammatik am Anfang um weitere Komponenten erweitert werden. Darüber hinaus sind in der Grammatik jede Alternative und mehrfach auftretende Elemente einer Sequenz mit jeweils unterschiedlicher Syntax mit Namen zu versehen. Diese Namen werden die Klassennamen der zugehörigen Knotenklassen bzw. zu Namen von Instanzvariablen der Knoten, welche die Elemente enthalten.

Die erzeugten Scanner unterstützen Unicode, bieten aber wieder keine vordefinierten Zeichenklassen, womit die Verwendung von Unicode eher beschränkt bleiben muß. Darüber hinaus sind Scanner mit einem `PushbackReader` als Argument zu konstruieren, da ein Scanner je nach Symbol Zeichen in die Eingabe zurückstecken muß. Die Puffergröße eines `PushbackReader` ist aber statisch und wird bei Konstruktion angegeben. Damit kann der Anwender nur hoffen, daß sein Puffer groß genug ist.

Die Analyse, d.h. Traverse, eines erzeugten Parse-Baums besteht aus der Implementierung des erzeugten Visitor-Interfaces oder aus dem Ableiten von einer der beiden Adapter-Klassen. Der Baum selbst ist leider statisch, d.h., er kann nicht umgebaut werden, oder Informationen im Baum können nicht verändert werden. Mehrere Visitor können zwar in beliebiger Reihenfolge den Baum zu verschiedenen Zwecken traversieren, doch deren Berechnungen müssen außerhalb des Baums gespeichert werden und können somit nicht elegant die Arbeit füreinander vorbereiten.

Wer mit der Traverse von Bäumen als Resultat der Parsierung auskommt, wird mit *SableCC* vielleicht glücklich. Es bleiben aber einige, zum Teil gravierende Nachteile:

Die Schnittstelle zwischen dem Parser und dem Scanner ist nicht offen. Handgeschriebene Scanner bzw. von anderen Generatoren erzeugte Scanner können daher nicht verwendet werden.

SableCC sollte ein einfaches und dadurch schnell zu lernendes System sein. Leider sind nur recht eigenwillige Grammatiken erlaubt, und EBNF wird nicht wirklich unterstützt. Eine Grammatik für reguläre Ausdrücke mit korrektem Vorrang ist schon nicht einfach zu schreiben.

SableCC prüft Grammatiken zwar auf Mehrdeutigkeiten, kann aber unendliche Rekursionen in der Grammatik nicht erkennen. Ein entscheidender Nachteil ist, daß der erste Syntaxfehler die Parsierung sofort mit einer Exception abbricht, da die erzeugten Parser generell nicht fähig sind, eine Fehlerbehandlung durchzuführen. Darüber hinaus unterstützt *SableCC* keine zitierten Literale in der Grammatik.

Sehr interessant ist der Ansatz, die Aktionen aus der Grammatik fernzuhalten, da automatisch ein Parse-Baum gebaut wird und es daher erst keine Anwender-Aktionen gibt. Dies läßt auch komplexe Grammatiken in einer lesbaren Form. Leider sind Grammatiken mit Alternativen- und mit Elementbenennungen und optional mit einer Paket-Anweisung vermischt und werden selbst dadurch sehr schnell unübersichtlich. Eine Grammatik ohne Aktionen sollte für Portierungen von *SableCC* auf andere Ausgabesprachen ohne Änderungen wiederverwendbar sein.

SableCC-Parser bauen immer einen Parse-Baum. Ein reiner syntaktischer Check eines Programms und damit eine direkte Interpretation eines Programms oder gar eine interaktive Verarbeitung von Eingaben sind damit nicht möglich. Außerdem sind die erstellten Bäume auch nicht serialisierbar.

Wie die anderen Werkzeuge kapselt auch *SableCC* in seinen Parsern und Scannern lediglich klassische Techniken in Klassen und nutzt die Objekt-Orientierung nicht. Das System selbst bzw. die Parser und Scanner sind in sich abgeschlossen und vom Anwender nicht zu erweitern.

Die aktuelle Version ist 2.16.2 und stammt von Juni 2001. *SableCC* ist in der Branche bekannt und wird aktiv von einer Mailingliste begleitet.

2.3.6 Steven Metsker, Building Parsers with Java

Steven John Metsker beschreibt in seinem Buch “Building Parser with Java” ([Met01a], [Met01b]) meiner Meinung nach den am meisten objekt-orientierten Ansatz der hier aufgeführten Systeme bzw. Werkzeuge. Ein Parser ist für ihn ein Objekt bzw. besser ein Baum von Objekten, welcher Elemente einer Sprache erkennt und dabei Ziel-Objekte für die erkannten Teile baut. Dazu führt er verschiedene Klassenhierarchien ein:

Die Parser-Hierarchie besteht aus den vier Hauptklassen `Sequence`, `Alternation`, `Repetition` und `Terminal`. Parser werden als Baum aus Instanzen dieser Klassen bzw. deren Unterklassen zusammengesetzt, welche alle von der abstrakten Klasse `Parser` als Wurzel der Hierarchie abstammen. Eine `Repetition` kapselt dabei eine `Parser`-Instanz und wiederholt deren Erkennung beliebig oft. `Alternation` bzw. `Sequence` sind Container vieler `Parser`-Objekte und parsieren diese alternativ bzw. nacheinander. Ein `Terminal` erkennt als ein Blatt des Baums ein Eingabe-Symbol. Zur Erkennung verschiedener Arten von Symbolen (Zahlen, Wörter, einzelne Zeichen, ...) existieren jeweils eigene Unterklassen von `Terminal`. Eine weitere Unterklasse `Empty` von `Parser` akzeptiert die leere Eingabe.

Ein `Assembly`-Objekt kapselt einen kompletten Strom von Eingabe-Symbolen, einen Stack von Objekten und optional einen Verweis auf ein Ziel-Objekt. Werden Symbole erkannt, wird vom System pro erkanntes Symbol ein `Token`-Objekt als Repräsentant des Symbols auf dem Stack hinterlegt. `Assembler`-Klassen sind vom Anwender als Aktionen zu erkannten Teilen einer Grammatik zu implementieren. Pro Knoten eines Parser-Baums kann eine `Assembler`-Instanz hinterlegt werden. Wird nun während der Parsierung eines Programms ein Knoten erkannt, ruft dieser beim zugehörigen `Assembler` eine Methode mit einem Verweis auf das aktuelle `Assembly` als Parameter auf. Der `Assembler` kann dann als Aktion zum Beispiel auf dem `Assembly`-Stack arbeiten oder das Ziel-Objekt erfragen und verwenden.

Ein Parser wird in Form von Java-Programmtext aus den Instanzen der Klassen instanziiert. Die Knoten des so erzeugten Parser-Baums rufen nach der Technik des rekursiven Abstiegs Methoden bei den Unterknoten auf, bzw. eine Terminal-Instanz erkennt letztendlich ein Symbol.

Die Knoten-Klassen der Parser-Bäume sind wiederverwendbar. Für unterschiedlichste Grammatiken können Repräsentationen der Grammatik in Form eines Objekt-Baums der Klassen konstruiert werden. Leider sind die Instanzen der Klassen nicht serialisierbar. Damit sind die erzeugten Parser selbst nicht wiederverwendbar und müssen immer wieder neu instanziiert werden.

Es existiert kein Parsergenerator, der eine BNF- oder gar EBNF-Grammatik in den Baum der Objekte wandelt, obwohl Metsker die dazu nötige Technik in einem Kapitel über die Erkennung von regulären Ausdrücken (unbewußt?) skizziert. Damit ist die Grammatik eines Parsers nur in Form von Java-Programmtext vorhanden. Dies ist für komplexe Grammatiken sehr unübersichtlich.

Für eine elegante Repräsentation von EBNF-Grammatiken fehlen der Klassenhierarchie Klassen zum optionalen Erkennen eines Grammatikteils, bzw. es fehlt eine nicht optionale Wiederholung. Dies ist momentan als `Alternation` mit einer leeren Alternative bzw. nur durch Kopieren des Grammatikteils zu erreichen. Da ein Parser aber aus Instanzen von Unterklassen der Klasse `Parser` besteht, könnte das System recht einfach um die beiden fehlenden Klassen (zum Beispiel `Optional` und `Some`) ergänzt werden.

Ein großer Nachteil ist, daß die Parsierung nicht deterministisch arbeitet. `Assembly` und `Assembler` müssen klonbar sein, da zum Beispiel jede Alternative einer `Alternation` parallel mit kopierten Ausgangs-`Assembly` und -`Assembler` verfolgt wird. Analog kann eine `Repetition` nicht testen, wann sie mit dem Erkennen zu enden hat und die Erkennung in nachfolgenden Knoten fortzusetzen ist. Auch hier werden alle möglichen Wege parallel verfolgt. Dies führt dazu, daß das System nur für kleinere Programme geeignet ist. Die Parsierung größerer Eingaben benötigt Speicherplatz, wie er auf keinem mir zur Verfügung stehenden Rechner vorhanden ist.

Es besteht keine Möglichkeit, die durch einen Baum von Parser-Instanzen repräsentierte Grammatik zu prüfen. Metsker rät Link-Rekursionen — wenn man sie kennt — durch Wiederholungen zu ersetzen. Mehrdeutigkeiten versucht er dadurch entdecken zu lassen, daß er den Baum viele, zufällige Eingabe-Folgen generieren läßt und diese dann zu erkennen versucht. Wird eine dieser erzeugten Symbol-Folgen auf mehr als einem Weg von dem Parser-Baum akzeptiert, liegt eine Mehrdeutigkeit vor. Dieser Ansatz findet vorhandene Mehrdeutigkeiten nur zufällig.

Metsker hat keine Technik zur Fehlerbehandlung parat. Lediglich ein möglicher Ansatz zur Ausgabe der im Fehlerfall akzeptierten Symbole wird skizziert. Dieser entdeckt aber nicht alle möglichen Symbole. Für Produktionszwecke sollte ein Parser dagegen bei einer Fehleingabe nicht nur die korrekte Menge an erlaubten Symbolen ausgeben, sondern sich auch erholen und mögliche weitere Fehler finden.

Vor dem Start der Parsierung liest das erste `Assembly`-Objekt den kompletten Strom von Symbolen ein, was für ein komplexes Programm sehr viel Speicherplatz benötigen kann. Eine interaktive Erkennung ist damit sogar ausgeschlossen, und dadurch kann während der Parsierung die Art der erkannten Symbole nicht gesteuert werden. Als Beispiel sind bereits deklarierte Variablen nicht als gesonderte Symbole zu erkennen.

Da ein `Assembly` immer einen Stack, optional ein Ziel-Objekt und den kompletten Eingabestrom kapselt, ist das vielfache Klonen der `Assembly` für echte Programme sehr zeit und speicherplatzaufwendig. Das Kopieren der `Assembler`-Instanzen birgt die gleichen Probleme, da diese während der probeweisen Parsierung den Stack bearbeiten bzw. an dem Ziel-Objekt weiterbauen.

Interessant ist die lexikalische Schicht seines Systems. Ein `Assembly` beauftragt einen `Tokenizer` mit dem Zerlegen der Eingabe in Symbole (`Token`-Instanzen). Ein `Tokenizer` kapselt dazu pro ASCII-Zeichen als Index in einer Tabelle eine Instanz einer Unterklasse von `State`. Um ein Symbol zu finden, liest `Tokenizer` nun ein Zeichen, sucht mit dem Zeichen als Index in der Tabelle nach dem zugehörigen `State`-Objekt und beauftragt dieses, das Symbol zu finden. Dazu liest das `State`-Objekt zumeist weitere Zeichen von der Eingabe. Hat das `State`-Objekt ein Symbol erkannt, liefert es als Repräsentant des Symbols ein `Token`-Objekt.

Dieser Ansatz ist meines Erachtens recht interessant, da objekt-orientiert. Einzelne Objekte bzw. deren Klassen kapseln die Aufgabe, ein bestimmtes Symbol zu erkennen. Erst durch die Tabelle wird aus den vielen, einzelnen Symbol-Erkennern ein komplexes System.

Das System beinhaltet für die meisten Symbole im Compilerbau Erkenner-Klassen, welche von Projekt zu Projekt wiederverwendet werden. Leider sind `State`-Instanzen bzw. `Tokenizer` nicht serialisierbar und damit nicht wiederverwendbar.

Das System kann durch Unterklassen von `State` vom Anwender erweitert werden. Leider muß dazu fast immer auch eine zugehörige Unterklasse von `Terminal` im Parser implementiert werden.

Bedauerlicherweise hat Metsker auch hier den Ansatz nicht komplett durchdacht:

Das Eintragen der `State`-Objekte in die Tabelle des `Tokenizer`-Objekts geschieht nicht automatisch. Der Anwender muß diese über eine Methode selbst in die Tabelle an allen gewünschten Positionen eintragen. Besser wäre es, wenn der `Tokenizer` die beteiligten `State`-Objekte nach möglichen, ersten Zeichen fragen und dann entsprechend in die Tabelle eintragen würde.

Nicht immer bestimmt das erste Zeichen das nächste Symbol. So kann ein < das einzelne Zeichen-Symbol <, das Symbol für <= oder auch das Symbol für <> beginnen. In der Tabelle kann für < aber nur ein `State`-Objekt eingetragen sein. Metsker hat keine allgemeine und automatische Technik, welche einen Stellvertreter der konkurrierenden Erkenner für die möglichen Symbole erzeugt und in die Tabelle einträgt. Diese Stellvertreter sind von Hand selbst zu implementieren, und nur für einige Fälle hat Metsker dies vorbereitet.

Die lexikalische Analyse verwendet einen `java.io.PushbackReader` zum Kapseln der Eingabezeichen. Damit kann ein Symbol-Erkenner Zeichen probehalber lesen und wieder zurück in die Eingabe stecken. Leider muß bei Konstruktion des `PushbackReader`-Objekts die maximale Puffergröße angegeben werden. Damit ist die maximale Länge für Symbole, welche probehalber Zeichen in der Eingabe lesen, beschränkt.

Der Einsatz von `PushbackReader` hat noch einen weiteren Nachteil. Da die Symbol-Erkenner keinen Zugriff auf den Puffer im `PushbackReader` haben, kopieren diese die erkannten Zeichen intern noch einmal, um sie in einem `Token` zu hinterlegen. Damit wird jedes Zeichen der Eingabe mindestens einmal kopiert.

Alles in allem hat Metsker einige analoge Ideen zu meiner Arbeit entdeckt und entwickelt: Parser bestehen aus einem Baum von Instanzen von für verschiedene Parser wiederverwendbaren Klassen. Die Erkennung von Symbolen wird an viele Erkenner-Objekte delegiert, welche jeweils genau ein Symbol erkennen können. Auch diese Klassen sind wiederverwendbar.

2.3.7 Fazit

Alle vorgestellten Werkzeuge sind entweder selbst nicht objekt-orientiert und kapseln lediglich klassische Techniken in Kapseln, oder die Umsetzung der Ideen ist nicht geglückt.

Zeitlich parallel zu meiner Arbeit hat zum Beispiel *SableCC* die Trennung von Aktionen und Grammatik entwickelt. Leider ist die Grammatik immer noch mit Anweisungen für den Bau des Parse-Baums gemischt. Eine reine Grammatik wäre diesem Ansatz vorzuziehen, da diese dann lesbar bleibt und für Portierungen des Werkzeugs auf andere Plattformen oder für andere Werkzeuge wiederzuverwenden ist. Außerdem sollte der Benutzer eigene Aktionen während der Parsierung zu erkannten Teilen der Grammatik anschließen können.

Metsker hat auch zeitlich parallel zu meiner Arbeit zwei interessante Ideen untersucht: So werden Parser aus Objekten existenter Klassen konstruiert, und Scanner bestehen aus einer Sammlung von Erkenner-Objekten, welche jeweils ein Symbol scannen können. Leider ist sein Ansatz des Parsers für praktische Zwecke nicht zu verwenden. Außerdem ist keine Fehlerbehandlung und Serialisierung des Parsers möglich. Entscheidet das erste Zeichen eines Symbols nicht eindeutig, welches Erkenner-Objekt das Symbol zu scannen hat, muß der Anwender diese Verteilung von Hand machen. Nur für einige spezielle Fälle sind Klassen bereits implementiert worden.

Generell kommt in allen Ansätzen neben der Wiederverwendung von Klassen die Wiederverwendung von Objekten oder besser die einer Sammlung von Objekten zu kurz. Die Serialisierung von Objekten sollte dies sehr elegant bringen.

2.4 Thesen

Inhalt der Arbeit ist es, die Anwendung der objekt-orientierten Programmierung auf den Compilerbau zu untersuchen. Die Arbeit soll zeigen, daß unter anderem folgende aufgeführte Vorteile der Objekt-Orientierung den Compilerbau zu neuen Techniken und Lösungen vorantreiben:

> Die Wiederverwendung von Klassen, aber auch die von Objekten vereinfachen die Entwicklung und die Verwendung von Software aus dem Bereich des Compilerbaus.
>
> Der Einsatz verschiedener typischer Entwurfsmuster der Objekt-Orientierung bringen auch den Compilerbau voran. Sie erhöhen zum Beispiel die Flexibilität und Erweiterbarkeit eines Scanners oder eines Compilers.
>
> Objekt-orientiert programmierte Software ist leicht erweiterbar, und wird Software aus dem Bereich des Compilerbaus objekt-orientiert konzipiert, so gilt dies auch für dieses Einsatzgebiet. Dank der Vererbung erben Klassen die Instanzvariablen und Methoden ihrer Oberklasse und können zum Beispiel durch das Überschreiben von Methoden das Verhalten wie gewünscht anpassen. Instanzen der neuen Klassen sind wie Instanzen der Oberklasse in das System zu integrieren, erweitern dieses aber um das neue Verhalten bzw. um die neue Funktionalität.
>
> Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer*, was auch im Compilerbau gewinnbringend genutzt werden kann. Selbst komplexe Algorithmen des Compilerbaus zerfallen so in kleine, einfache Teilprobleme und sind dadurch in der Lehre für Studierende leicht zu verstehen bzw. sogar selbständig zu entdecken.

2.5 Überblick

Kapitel 3 (“Lexikalische Analyse”) untersucht den Einsatz der Objekt-Orientierung in der lexikalischen Analyse. Durch Vererbung werden Erkenner-Klassen verfeinert und erkennen so gewünschte Symbole. Dank der Wiederverwendung von Klassen können Klassen für typische Symbole in verschiedenen Scanner-Projekten genutzt und durch Serialisierung bzw. Deserialisierung Scanner in verschiedenen Projekten oder in verschiedenen Durchläufen wiederverwendet werden. Dies führt zu mächtigeren Werkzeugen in der Implementierung von Scannern.

Kapitel 4 (“Ein Parser aus Objekten”) analysiert, wie eine EBNF-Grammatik durch einen Baum von Objekten repräsentiert werden kann. Es wird gezeigt, daß (durchaus komplexe) Algorithmen wie das Berechnen von lookahead- und follow-Mengen, wie das Prüfen der repräsentierten Grammatik auf LL(1)-Konformität und wie das Parsieren von Programmen über der Grammatik durch die Zerlegung der Algorithmen in je Knoten-Klasse spezifische, aber kleine Teilprobleme leicht zu implementieren sind. Die Knoten-Klassen sind für jede Grammatik wiederverwendbar. Der eine Grammatik repräsentierende Baum ist serialisierbar und somit auch wiederverwendbar.

Kapitel 5 (“Eine objekt-orientierte Scanner-Schnittstelle”) beschreibt einen objekt-orientierten Ansatz für eine Scanner-Schnittstelle der in Kapitel 4 vorgestellten Parser. Objekte identifizieren vom Scanner erkannte Symbole beim Parser, wobei die genaue Klasse der Identifizierungs-Objekte dem Scanner-Entwickler verborgen bleibt. Die Symboltabelle eines Compilers war schon immer eine Art Objekt: Über eine Methoden-Schnittstelle kann auf die gespeicherte Information zugegriffen werden. Inhalt des Kapitels ist darüber hinaus aber auch die Entwicklung einer Symboltabelle, welche durch den Einsatz von *Closure*-Objekten effizient zu benutzen ist.

Kapitel 6 (“Parser-Aktionen”) führt die Mächtigkeit in der Vererbung von Klassen im Compilerbau vor. Die in Kapitel 3 beschriebenen Klassen, aus deren Instanzen Bäume als Repräsentation einer Grammatik gebaut werden, sind durch einfache Unterklassen um vom Benutzer anzugebende Aktionen zu erkannten Teilen einer Grammatik während der Parsierung erweiterbar. In dem Kapitel wird die Implementierung von drei verschiedenen Arten von Aktions-Ideen bzw. -Mustern erläutert, und weitere mögliche Ansätze für Aktions-Muster werden diskutiert.

Kapitel 7 (“Die opi-Schnittstelle”) versucht analog zu Sun's *Java APIs for XML Processing* (kurz JAXP, [JAXP]), eine Verkapselung von Parsern, deren Eingabe und eine Erzeugung von Parsern durch das Factory-Entwurfsmuster zu erreichen. Das Kapitel beschreibt die Interfaces und Klassen und führt diese an einem Beispiel näher vor.

Kapitel 8 (“Fehlerbehandlung durch Objekte und Exceptions”) beschreibt zwei Ansätze einer möglichen Fehlerbehandlung in Parsern nach der Technik des rekursiven Abstiegs. Eine Kette von Exception-Objekten, welche spezielle Methoden je Knoten implementieren, modellieren die Aufrufverschachtelung während des rekursiven Abstiegs. Bei einem Syntaxfehler kann entlang der Kette nach Punkten für eine mögliche Fehlerbehandlung gefragt werden, welche durch das Auslösen der Exception dann auch erreicht werden. Weiterhin kann die Kette auch zur Berechnung der an einem Punkt in der Eingabe möglichen Symbole verwendet werden.

Kapitel 9 (“Grammatik-Notationen, Erweiterung durch Klassen”) führt weitere mögliche Notationen für Grammatiken vor und erläutert dabei, daß die ursprüngliche Notation der durch einen Parser-Baum repräsentierten Grammatik für den Baum unerheblich ist. Der Baum selbst enthält alle Informationen und Algorithmen. Darüber hinaus zeigt das Kapitel auf, daß die Erweiterung der Parser-Funktionalität um neue Konstrukte einfach ist. Neben der von EBNF bekannten Verknüpfung von Alternativen durch | wird die Idee dreier weiterer, mächtiger und zum Beispiel zur Beschreibung von XML-Dokumenten auch notwendiger Konstrukte und deren aufgrund der Vererbung nicht schwierigen Implementierung erläutert. Dies führt zu einer Erweiterung von EBNF zu Extended EBNF oder kurz XEBNF.

Die Erzeugung der Bäume von Objekten als Repräsentation einer Grammatik sollte von einem Parsergenerator geschehen, da dies “von Hand” sicherlich nicht produktiv ist. Kapitel 10 (“oops als Parsergenerator”) untersucht, wie aus den bereits bekannten Techniken Parsergeneratoren für unterschiedliche Arten von Eingabegrammatiken erzeugt und genutzt werden können. Als Beispiel werden Parsergeneratoren für EBNF und XEBNF aufgezeigt.

Das letzte Kapitel 11 (“Resümee und Ausblick”) faßt die Arbeit zusammen, bewertet den allgemeinen Ansatz sowie die konkreten Realisierungen und gibt Anregungen für weiterführende Arbeiten.

3 Lexikalische Analyse

Die lexikalische Analyse hat die Aufgabe, aus einem Strom von Eingabezeichen eine Folge von Symbolen zu bilden. Ein Symbol ist eine Klassifizierung von Zeichenfolgen. Typische Symbole im Compilerbau sind verschiedene Arten von Zahlen, unterschiedliche Kommentare, Zwischenraum, Bezeichner, Wörter, aber auch einzelne Zeichen.

Dieses Kapitel untersucht den Einsatz der Objekt-Orientierung in der lexikalischen Analyse, stellt zwei implementierte Tools vor und diskutiert die Vor- und Nachteile der neuen Technik gegenüber dem herkömmlichen Ansatz à la *lex*.

3.1 Motivation, Idee

Zwei Grammatiken beschreiben einen Compiler einer Sprache: Eine Grammatik legt fest, wie die lexikalische Analyse Zeichen zu Symbolen zusammenfaßt und eine zweite Grammatik beschreibt, welche Symbolfolgen legale Programme der Sprache darstellen. Es hat sich gezeigt, daß reguläre Grammatiken zur Beschreibung der ersten Grammatik ausreichen und daher durch endliche Automaten erkannt werden können, welche wiederum durch reguläre Ausdrücke beschrieben werden können.

Der herkömmliche praktische Ansatz zur Implementierung der lexikalischen Analyse ist der Einsatz eines Codegenerators wie *lex*, welcher aus Paaren von Mustern und Aktionen als Eingabe zum Beispiel eine C- oder Java-Funktion erzeugt. Die Muster beschreiben als reguläre Ausdrücke Symbole, und die assoziierten Aktionen kommen zur Ausführung, wenn eines der Muster erkannt wird. Bei Mehrdeutigkeiten gewinnt das längere Symbol bzw. bei gleich langen Symbolen das erste in der Tabelle der Paare.

In der nächsten Version von Java (momentaner Entwicklungsstand ist Java 1.4 Beta 3) wird es ein Paket namens `java.util.regex` zum Umgang mit regulären Ausdrücken in Java geben. Ein regulärer Ausdruck wird dabei durch eine Instanz einer `Pattern`-Klasse repräsentiert, welche dann mit einem `Matcher`-Objekt gegen eine Zeichenfolge verglichen werden kann. Ein Scanner könnte dann viele `Pattern` parallel mit `Matcher`-Objekten ab dem aktuellen Index im Eingabestrom untersuchen. Dies würde aber lediglich den klassischen Ansatz à la *lex* pro Symbol in ein Objekt verlegen und an der Technik an sich nichts ändern.

Die Beschreibung der Muster durch reguläre Ausdrücke ist kryptisch und für Anfänger nur schwer zu erlernen. Um komplizierte Muster bzw. Muster von einer dritten Person zu verstehen, bedarf es einiger Erfahrung. Als Beispiel hier ein regulärer Ausdruck für einen C-Kommentar:

```
"/*"([^*]|"*"+[^/*])*"*"+"/"
```

Ein komplizierter regulärer Ausdruck ist fehleranfällig und muß daher ausgiebig getestet werden. So ist zum Beispiel der folgende reguläre Ausdruck für einen *javadoc*-Kommentar nicht korrekt:

```
"/**"([^*]|"*"+[^/*])*"*"+"/"
```

lex erzeugt als Ausgabe eine Funktion, welche die durch die Paare beschriebene lexikalische Analyse darstellt. Diese Funktion muß in einem nächsten Schritt übersetzt werden. Um den Scanner zu erweitern bzw. um Fehler zu beseitigen, muß der ganze Entwicklungszyklus (Editieren der Paare, Erzeugen der Funktion und Übersetzung) erneut ausgeführt werden.

Weiterhin sind einige realistische Symbole im Compilerbau — wie geschachtelte Kommentare — mit dieser Technik nur schwer zu erkennen.

Diese Beobachtungen führen zu einem neuen objekt-orientierten Ansatz:

Ein Scanner wird als Wettbewerb von Objekten in einem Raum modelliert. Jedes der Objekte ist in der Lage, genau ein Symbol zu erkennen. Der Raum, gefüllt mit diesen Erkenner-Objekten, sendet Zeichen für Zeichen aus der Eingabe zu den Objekten im Raum (Push-Prinzip). Ein Erkenner-Objekt verläßt den Raum, wenn es mit dem aktuellen oder mit zukünftigen Zeichen nichts anfangen kann. Die Erkenner-Objekte teilen dem Raum (möglicherweise mehrmals) mit, wenn sie mit dem aktuellen Zeichen ein Symbol erkannt haben. Die Erkennungs-Runde endet, wenn alle Erkenner keine weiteren Zeichen akzeptieren, also alle den Raum verlassen haben.

Gewinner der Runde und damit Repräsentant des gescannten Symbols ist das Objekt, welches als letztes Erkennung signalisiert hat, wenn alle den Raum verlassen haben. Sollten in der zugehörigen Zeichen-Runde mehrere Objekte die Erkennung eines Symbols angezeigt haben, kann die Mehrdeutigkeit analog zu *lex* entschieden werden: Das Objekt, welches zuerst den Raum betreten hat, ist dann der Gewinner.

Als Beispiel ein Raum mit zwei Erkenner-Objekten `A` und `B`: `A` sucht die Zeichenfolge `abca`, und `B` sucht das Zeichen `a`. Nach Eingabe von `a` signalisiert `B`, daß es ein Symbol erkannt hat und daß es keine weiteren Zeichen akzeptiert, d.h., es verläßt den Raum. Da `A` weitere Zeichen zu erkennen hat, verbleibt es im Raum und akzeptiert bei der weiteren Eingabe von `bc` die beiden Zeichen. Wenn nun als nächstes ein `a` kommt, signalisiert `A`, daß es ein Symbol erkannt hat und verläßt auch den Raum. Da kein weiteres Objekt im Raum verbleibt, endet die Erkennungsrunde, und `A` ist der Gewinner der Runde, da es die längste Zeichenfolge als Symbol erkannt hat. Kommt aber nach `bc` kein `a`, verläßt `A` auch den Raum, da es keine weitere Zeichen akzeptiert. `A` hat in diesem Fall aber kein Symbol erkannt, und `B` ist der Gewinner.

Der neue Ansatz soll für den Benutzer einfach zu handhaben sein. Das System beinhaltet daher neben den Klassen zur Modellierung des Raums und der Eingabe eine fertige Bibliothek von Klassen, deren Instanzen die gängigen Symbole im Compilerbau wie Wörter, verschiedene Arten von Zahlen, Kommentare, Zwischenraum, Bezeichner, aber auch einzelne Zeichen erkennen können. Der Anwender stellt den Raum aus Objekten dieser Klassen zusammen und muß sich keine Gedanken über die Erkennung der Symbole oder die Verwaltung der Eingabezeichen machen. Dies ist in den Klassen fertig gekapselt.

Da die Erkenner-Objekte Instanzen von Klassen sind, können durch das Ableiten von bestehenden Erkenner-Klassen spezialisierte Symbole erkannt werden. Ein Anwender des Systems wird also kaum eine komplette neue Klasse zu schreiben haben, falls die Bibliothek keine passende Klasse enthält.

Objekte haben darüber hinaus einen Zustand und können daher mächtigere Erkenner als reguläre Ausdrücke darstellen. Dies macht das Erkennen von Symbolen wie geschachtelte Kommentare leicht.

Der Raum als Container von Objekten ist im laufenden Betrieb umkonfigurierbar.

3.2 lolo

lolo (**l**anguage-**o**riented **l**exer **o**bjects, ([Lolo], [Küh01b])) stellt eine Implementierung der oben beschriebenen Idee in Java dar.

lex-artige Tools erzeugen zuerst einen nicht-deterministischen, endlichen Automaten: Jedes Eingabezeichen bewirkt einen Wechsel von einem Zustand in einen oder mehrere neue Zustände. Mengen von neuen Zuständen werden dann zu einem größeren, äquivalenten, deterministischen Automaten kombiniert. Der neue Automat wird abschließend noch reduziert.

Die Erkenner-Klassen dagegen beschränken die Implementierung der Erkennung von Symbolen auf kleine, einfache und überschaubare Probleme. Erst der Raum kombiniert Instanzen der Klassen zu einem großen, nicht-deterministischen System, welches aufgrund der parallelen Nutzung aller Erkenner-Objekte einen Performance-Nachteil zu haben scheint.

Wie sich herausstellte, entscheidet glücklicherweise bei Verwendung der typischen Symbole des Compilerbaus aber zumeist bereits das erste Zeichen, welcher Erkenner den Wettbewerb gewinnt: `0` bis `9` ist der Anfang einer Zahl, ein Buchstabe startet ein reserviertes Wort oder einen Bezeichner (welche später anhand der Symboltabelle differenziert werden), Zwischenraumzeichen beginnt Zwischenraum usw.

Aus diesem Grund arbeitet *lolo* auf einer nach Eingabezeichen indizierten Tabelle von Erkenner-Instanzen, welche das Index-Zeichen als erstes Zeichen des erkannten Symbols akzeptieren. Dadurch wird der (potentielle) Gewinner einer Runde schnell durch einen Zugriff in die Tabelle bestimmt. Ein spezielles Multiplex-Objekt wird stellvertretend für mehrere mit dem gleichen Zeichen startende Erkenner in die Tabelle eingetragen und spielt so die Rolle eines kleinen Unterraums.

Als Beispiel zeigt Abbildung 3.1 eine Tabelle eines *lolo*-Scanners mit Instanzen der Klasse `Flt` zur Erkennung von Gleitkommazahlen, mit `Int`-Instanzen für ganze Zahlen und einem `Identifier`:

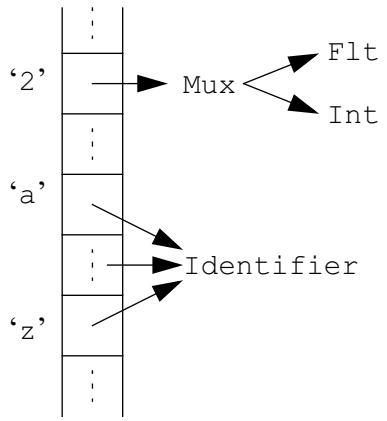

Abbildung 3.1: Eine Tabelle eines lolo-Scanners.

Steht der Gewinner einer Runde fest, könnte dieser alle darüber hinaus akzeptierten Zeichen des Symbols selbständig aus der Eingabe lesen (Pull-Prinzip). Ich habe mich in *lolo* gegen den Einsatz der Pull-Technik entschieden, was ich später in diesem Kapitel erläutern werde.

Die lexikalische Analyse als Black-Box erkennt Symbole und löst zu erkannten Symbolen Aktionen aus. So kommt es zu den Muster/Aktions-Paaren in *lex*. Bei *lolo* geht die Eingabe in den Objektraum, der dann eine Aktion auslöst, für die ein Objekt als Träger der Aktion bereitgestellt wird. Optional kennt daher jede Erkenner-Instanz ein Aktions-Objekt, dessen Klasse vom Anwender zu implementieren ist und welche lediglich einem Interface genügen muß. Dieses Aktions-Objekt ist im Betrieb jederzeit austauschbar.

3.2.1 Implementierung

Das Paket `lolo` stellt Klassen als Framework des Wettbewerbs von Erkenner-Instanzen zur Verfügung und umfaßt die Klassen `lolo.Input`, `lolo.Scanner`, `lolo.Scan` und `lolo.Mux`:

Input

Die Klasse `Input` stellt die Eingabeseite für den Wettbewerb von Erkenner-Objekten dar.

```
package lolo;
...
public class Input {
    protected final Reader reader;
    protected int next, filled;
```

```
        protected char[] buffer;
        public Input(Reader reader, int bufferSize) { ... }
```

Ein `Input` puffert in `buffer` Zeichen aus dem Eingabestrom `reader`. Damit werden bereits alle Zeichen als Unicode-Zeichen gelesen, und der Anwender hat sich um ein mögliches Encoding zu kümmern. Der Puffer besitzt initial für `bufferSize` Zeichen Platz, kann sich aber im laufenden Betrieb vergrößern. `next` und `filled` markieren das aktuelle, unverarbeitete Zeichen bzw. die Position im Puffer, ab der neue Zeichen von `reader` gelesen werden müssen.

```
        public int next() throws IOException { ... }
```

Ein `Scanner` liest durch Aufruf von `next()` die einzelnen Zeichen von einem `Input`. Analog zu den `read()`-Methoden aus dem `java.io`-Paket wird als Resultat der `char`-Wert des Zeichens als `int` und ein negativer Wert am Dateiende geliefert.

```
        protected int startOfSymbol = -1;
        protected void setStartSymbol() { ... }
        protected void unsetSymbol() { ... }
```

Ein `Input` soll nicht alle Zeichen der Eingabe in dem Puffer sammeln, sondern bereits verarbeitete Zeichen aus dem Puffer entfernen dürfen, wenn der Platz im Puffer verbraucht ist. Allein durch die Aufrufe von `next()` kann ein `Input` nicht entscheiden, welche Zeichen verworfen werden können. Der `Scanner`, der die Zeichen von dem `Input` abholt, steuert dies über Methoden:

Die Methode `setStartSymbol()` markiert in der Instanzvariablen `startOfSymbol` als Index das nächste Zeichen als das erste Zeichen eines Symbols. Alle Zeichen eines Symbols müssen im Puffer bleiben, da die Aktion zu einem Symbol mit diesen Daten arbeiten will. `unsetSymbol()` löscht die Markierung wieder, was vom `Scanner` aber momentan nicht verwendet wird.

```
        protected int mark = -1;
        public void mark() { ... }
        public void unmark() { ... }
        public void pushBackToMarked() { ... }
    } // end of Input
```

`mark()` markiert in `mark` die Position des nächsten Zeichens im Puffer, so daß nach Aufruf von `pushBackToMarked()` in der Eingabe wieder zu der markierten Position zurückgesprungen wird. Dies ist notwendig, da es vorkommt, daß ein Symbol-Erkenner mehr Zeichen liest, als das erkannte Symbol lang ist. Zum Beispiel liest ein Gleitkomma-Erkenner von der Zeichenfolge `23exz...` die Zeichen `23ex`, da nach dem `e` noch eine Zahl kommen könnte. Dies ist nicht der Fall, und nur `23` wird als Symbol erkannt, und in der Eingabe muß zum `e` zurückgesprungen werden.

Alle Zeichen ab der so markierten Position dürfen natürlich auch nicht aus dem Puffer gelöscht werden. Normalerweise sollte diese Markierung aber immer nach der Markierung für ein neues Symbol kommen.

Scanner

Die Klasse `Scanner` modelliert den Wettbewerb der Erkenner-Instanzen und stellt damit einen Scanner dar. Sie sammelt viele Erkenner-Objekte und kann nach dem nächsten Symbol gefragt werden.

```
    package lolo;
    ...
    public class Scanner implements Serializable {
        public Scanner() {}
        public Scanner(Scan scan) throws IllegalArgumentException { ... }
        public Scanner(Scan [] scans) throws IllegalArgumentException { ... }
```

Bei Erzeugung des `Scanner` kann dieser optional bereits mit Erkenner-Instanzen, die Objekte von Unterklassen der abstrakten Klasse `Scan` sein müssen, gefüllt werden.

```
protected Scan[] scans = new Scan[1];  // array to maintain order
protected int next, max = scans.length;

public boolean add(Scan scan) throws IllegalArgumentException {
    if (scan == null) throw new IllegalArgumentException("scan is null");
    if (contains(scan)) return false;
    if (max == next) {
        Scan [] help = new Scan[max*=2];
        System.arraycopy(scans, 0, help, 0, max/2);
        scans = help;
    }
    scans[next++] = scan;
    packed = false;
    return true;
}
public boolean add(Scan [] scans) throws IllegalArgumentException { ... }
public boolean remove(Scan scan) { ... }
public boolean contains(Scan scan) { ... }
```

Die zwei `add()`-Methoden `remove()` und `contains()` dienen der Verwaltung der in einem `Scanner` versammelten Erkenner. `contains()` erfragt, ob das Argument Teil des `Scanner` ist. Dabei werden Objekte mit `equals()` auf Gleichheit überprüft. Mit `add()` können weitere Erkenner zum Scanner hinzugefügt werden, falls diese nicht bereits Teil des `Scanner` sind. `remove()` entfernt das Argument aus dem `Scanner`. Das Resultat der `add()`- und `remove()`-Methoden reflektiert, ob sich der `Scanner` geändert hat.

Bei der Verwaltung der in dem Scanner gekapselten Erkenner ist die Reihenfolge des Einfügens der Objekte zu bewahren, da diese bei Mehrdeutigkeiten entscheidet. Daher können die `Scan` nicht in einem `Set` gesammelt werden. Alternativ hätten die Objekte aber auch in einer `List` an Stelle eines Arrays gespeichert werden können.

```
protected boolean packed;
public transient Scan[] table;
public void pack() { ... }
public boolean packed() { return packed; }
```

Ein `Scanner` arbeitet zur Effizienzgewinnung auf der bereits beschriebenen Tabelle von Erkennern, welche bei jeder Änderung der Erkenner-Menge neu berechnet werden muß. Dies geschieht durch Aufruf der Methode `pack()`, welche die Tabelle `table` füllt. Dabei wird jedem Erkenner einmal jedes Zeichen als erstes Zeichen des Symbols mitgeteilt, und je nach Antwort wird das Objekt in die Tabelle an der Position eingetragen. Gehen mehr als ein Erkenner an die gleiche Stelle in der Tabelle, werden diese von einem `Mux`-Objekt verwaltet, und das `Mux` wird in die Tabelle eingetragen. `packed` reflektiert, ob die Tabelle aktuell ist oder neu berechnet werden muß. Der Wert von `packed` kann mit `packed()` erfragt werden.

```
protected boolean unicode;
public void setUnicode(boolean unicode) { ... }
public boolean getUnicode() { ... }
```

Nicht alle `Scanner` werden den kompletten Unicode Zeichensatz verwenden wollen. Ist der Wert von `unicode` wahr, wird die Tabelle `table` für alle Unicode-Zeichen als Index gefüllt. Ist der Wert `false`, wird lediglich für den ASCII-Zeichensatz die Tabelle gefüllt. Dies spart Platz, spart Berechnungszeit

während der Tabellenerzeugung, und das Serialisieren bzw. Deserialisieren eines `Scanner` ist schneller.

```
    public Scan scan(Input input) throws IllegalCharacterException,
        IOException { ... }

    public static class IllegalCharacterException extends Exception {
        protected char ch;
        public IllegalCharacterException(char ch, String message) {
            super(message); this.ch = ch;
        }
        public char getChar() { return ch; }
    }
```

Durch Aufruf von `scan()` wird der `Scanner` zum Erkennen des nächsten Symbols aufgefordert. Das Argument `input` gibt dabei die Zeichenquelle an und kann von Aufruf zu Aufruf ausgetauscht werden. Als Resultat liefert `scan()` den Erkenner des erkannten Symbols oder `null` am Dateiende. Ist die Tabelle der `Scanner`-Instanz bei Aufruf von `scan()` noch nicht aktuell, geschieht dies implizit durch Aufruf von `pack()`.

Paßt das nächste Zeichen zu keinem der zu erkennenden Symbole, wirft die Methode eine `IllegalCharacterException`, welche das illegale Zeichen enthält. Kommt es zu einem Eingabe/ Ausgabe-Fehler, wird eine `IOException` ausgelöst.

`scan()` ermittelt anhand des ersten Zeichens des Symbols und anhand der Tabelle das eine `Scan`-Objekt bzw. die eine `Mux`-Instanz. Diese könnte nun alle weiteren Zeichen selbständig lesen (Pull-Prinzip). Steht in dem `Mux` der Gewinner fest, könnte dieser auch folgende Zeichen selbständig verarbeiten.

Der Pull-Ansatz würde aber eine weitere Methode analog zu `nextChar()` nach sich ziehen, in der die `Scan`-Instanz als Zeichenquelle das `Input` als Argument erhalten würde.

Weiterhin müßten dann die `Scan` selbst `mark()` und `setStartSymbol()` beim `Input` aufrufen, was somit in die Verantwortung des Entwicklers von Erkenner-Klassen fallen würde. Diese Aufrufe sind derzeit fehlerfrei in `Scanner` gekapselt.

```
    private void readObject(ObjectInputStream stream) throws IOException,
        ClassNotFoundException { ... }
    private void writeObject(ObjectOutputStream stream) throws
        IOException { ... }
} // end of Scanner
```

Bei der Serialisierung eines `Scanner`-Objekts würde das standardmäßige Serialisieren der Tabelle zu einer großen Menge an Daten führen. So sind bei einem Scanner über dem ganzen Unicode-Zeichensatz oft sehr viele der Tabelleneinträge `null`-Verweise. Wie Messungen gezeigt haben, dauert darüber hinaus das Deserialisieren recht lange. Damit wäre die Startphase eines Compilers, welcher *lolo* für die lexikalische Analyse verwendet, inakzeptabel groß.

Daher wird bei der Serialisierung in `readObject()` die Information der Tabelle in Paaren von jeweils einem String und einer `Scan`-Instanz verpackt, welche serialisiert und dementsprechend in `writeObject()` deserialisiert werden. Pro Paar ist die `Scan`-Instanz an all den Positionen eingetragen, die als Zeichen im String vorkommen, d.h., der String enthält all die Zeichen, die als erstes Zeichen des Symbols auftreten können. `null`-Verweise werden dabei nicht kodiert.

Scan, Scan.State, Scan.Action

Die Klasse `Scan` ist die abstrakte Basisklasse aller Symbol-Erkenner.

```
package lolo;
...
public abstract class Scan implements Serializable {
    /** Action code for a recognized symbols is represented by Action
     * objects.
     */
    public interface Action {
        /** Called to perform action code for a scanned symbol. */
        public void action(Scan sender, char [] buffer, int off, int len);
    }

    /** The action object; can be null. */
    protected transient Action action;

    /** Returns the action object. */
    public Action getAction() { return action; }

    /** Sets the action object.  */
    public Scan setAction(Action action) { this.action = action; return this; }

    /** If action is not null, then the action method is called for action. */
    public void action(char [] buffer, int off, int len) {
        if (action != null) action.action(this, buffer, off, len);
    }
```

Das innere Interface `Scan.Action` definiert eine Schnittstelle für Aktion-Code, der bei Erkennen eines Symbols ausgeführt werden soll. Möchte der Anwender von *lolo* Aktionen zu erkannten Symbolen, hat er Klassen für die Aktionen zu schreiben und kann Instanzen der Klassen durch `setAction()` bei der Erkenner-Instanz hinterlegen. `setAction()` hat als Resultat den Empfänger der Methode zu liefern. Dadurch kann die `Action` — zum Beispiel durch ein Objekt einer anonymen Klasse — direkt beim Erzeugen der `Scan`-Instanz bzw. beim Hinzufügen zum `Scanner` gesetzt werden und eignet sich damit zur Kaskadierung.

`getAction()` liefert eine hinterlegte `Scan.Action`-Instanz, und `action()` wird vom `Scanner` beim Gewinner einer Runde aufgerufen. Als Parameter wird ein Ausschnitt in einem `char`-Array mit den Zeichen des erkannten Symbols übergeben. Das Kopieren der erkannten Zeichen in einen eigenen Puffer oder alternativ in eine `String`-Instanz würde jedes Zeichen einmal kopieren und wäre ineffizient. Daher ist das Array der Puffer des verwendeten `Input` und sollte daher nur lesend benutzt werden. Ist eine `Scan.Action` hinterlegt, wird aus `action()` der `Scanner`-Instanz bei dem `Scan.Action`-Objekt `action()` mit dem analogen Ausschnitt im Zeichen-Puffer aufgerufen.

```
    /** Marks if the symbol will be ignored. */
    protected boolean ignore;

    /** Sets the ignore state. */
    public Scan setIgnore(boolean ignore) {
        this.ignore = ignore; return this;
    }

    /** Returns the ignore state. */
    public boolean getIgnore() { return ignore; }
```

Zwischenraum und Kommentare werden im Compilerbau typischerweise bereits in der lexikalischen Analyse verworfen und nicht als Symbole an die Syntax-Analyse geliefert. Um dies zu unterstützen, kann ein Symbol mit `setIgnore()` eingestellt werden. Ist das Argument wahr, wird das Symbol zwar erkannt und eine zugehörige Aktion aufgerufen, aber es beendet nicht den Aufruf von `scan()` des

`Scanner`-Objekts — es wird ignoriert. Analog zu `setAction()` liefert `setIgnore()` als Resultat das Empfänger-Objekt selbst und eignet sich daher zur Kaskadierung.

```
/** Resets this Scan instance. */
public abstract void reset();

/** Called by a Scanner for every new character. Returns a State object
 * to signal whether more characters are wanted and whether a symbol
 * is found.
 */
public abstract State nextChar(char ch);

/** A simple class to mark the result for nextChar(). */
public static class State implements Serializable {
    /** Sets the object to signal whether more characters are wanted
     * and whether a symbol was found. */
    public State set(boolean more, boolean found) {
        this.more = more; this.found = found;
        return this;
    }

    /** The Scan instance wants more characters? */
    public boolean more;

    /** The Scan instance just found a symbol? */
    public boolean found;
}
```

`Scan` als abstrakte Basisklasse definiert für Unterklassen vier abstrakte Methoden. `reset()` wird vor jeder neuen Erkennungsrunde beim Erkenner zur Initialisierung aufgerufen. Anschließend werden dem Objekt durch Aufrufe von `nextChar()` Zeichen für Zeichen vorgeworfen. Das Objekt antwortet mit einem `Scan.State`-Objekt als Antwort zu `nextChar()`. Ein `Scan.State` kapselt als Information, ob weitere Zeichen erwünscht sind und ob mit dem aktuellen Zeichen gerade das Symbol erkannt worden ist.

```
/** Subclasses have to implement this method to indicate whether some
 * other object is equal to this one.
 */
public abstract boolean equals(Object o);

/** Subclasses have to implement this method to return a hash code
 * value for the object.
 */
public abstract int hashCode();
```

Die Methoden `equals()` und `hashCode()` sind bereits in der Oberklasse `java.lang.Object` definiert. Sie sind hier als abstrakt markiert, damit Unterklassen gezwungen sind, diese zu implementieren. `equals()` wird in `add()`, `remove()` und `contains()` eines `Scanner`s verwendet. Da `equals()` und `hashCode()` in Abhängigkeit zueinander stehen, sind beide Methoden zu implementieren.

```
/** Serializes the <tt>Scan</tt> object and it's action object
 * (if that is serializable). */
private void writeObject(ObjectOutputStream stream) throws IOException {
    stream.defaultWriteObject();
```

```
            boolean b = action != null && action instanceof Serializable;
            stream.writeBoolean(b);
            if (b) stream.writeObject(action);
        }

        /** Deserializes the Scan-Object and optional reads the serialized
         * action object. */
        private void readObject(ObjectInputStream stream) throws IOException,
            ClassNotFoundException {
            stream.defaultReadObject();
            if (stream.readBoolean()) action = (Action) stream.readObject();
        }
    } // end of Scan
```

Ist das `Scan.Action`-Objekt einer `Scan`-Instanz gesetzt und serialisierbar, wird es bei der Serialisierung des `Scan`-Objekts mitserialisiert. Analog wird beim Deserialisieren der `Scan`-Instanz ein serialisiertes `Scan.Action`-Objekt wieder deserialisiert. Damit ist ein Scanner inklusive der Aktionen speicherbar und kann immer wieder — auch auf verschiedenen Plattformen — zum Leben erweckt werden.

Mux

Ein `Mux`-Objekt ist ein Container für `Scan`-Objekte und wird stellvertretend für diese gekapselten `Scan`-Instanzen in die Tabelle eingetragen. Ein Benutzer erzeugt normalerweise niemals selbst `Mux`-Instanzen. Daher ist Mux keine öffentliche Klasse:

```
package lolo;

/** A Mux mulitplexes two or more Scan instances. */
class Mux extends Scan {
    /** Creates a Mux object. */
    public Mux(Scan s1, Scan s2) throws IllegalArgumentException {  ... }
```

Ein `Mux` wird von einem `Scanner` immer mit zwei `Scan` erzeugt, da ein `Mux` mindestens zwei `Scan` kapselt. Weitere `Scan` werden durch Aufruf von `add()` hinzugefügt:

```
    /** Collects all managed Scan instances. */
    protected Scan [] scans = new Scan[ 2];
    /** Indexes into scans. */
    protected int max = scans.length, next;
    /** Array displaying active Scan objects.  */
    protected boolean done [];

    /** Adds a Scan object. */
    public void add(Scan scan) throws IllegalArgumentException {  ... }
```

Die `Scan` werden in einem Array gesammelt, da die Reihenfolge wieder entscheidend ist. `done` wird in `nextChar()` verwendet und reflektiert die noch aktiven, gekapselten Erkenner einer Runde.

```
    /** Used to mark allready reseted objects. */
    protected transient boolean reset;

    /** Resets all managed Scan instance. */
    public void reset() {
        if (reset) return;
```

```
        for (int i = 0; i < next; i++) {
            scans[ i] .reset(); done[ i] = false;
        }
        reset = true;
    }
```

`Mux` delegiert `reset()` nacheinander an alle Insassen und markiert diese wieder als aktiv.

```
    /** At the end of one round winner holds the winning Scan object. */
    protected transient Scan winner;
    /** A State object which is used for nextChar(). */
    protected final State stateObject = new State();

    public State nextChar(char ch) {
        boolean more = false, found = false;
        reset = false;
        for (int i = 0; i < next; i++) {
            if (done[ i] ) continue;
            Scan scan = scans[ i] ;
            State state = scan.nextChar(ch);
            if (!found && state.found) {
                found = true;
                winner = scan;
            }
            if (!state.more) done[ i] = true;
            else more = true;
        }
        return stateObject.set(more, found);
    }
```

In `nextChar()` werden anhand von `done` alle noch aktiven Erkenner befragt. Möchte einer der aktiven Erkenner weitere Zeichen und hat analog eines ein Symbol gefunden, so liefert ein `Mux` dieses nach außen.

Als Resultat zu `nextChar()` wird immer wieder dasselbe Objekt `stateObject` verwendet. Wie Messungen gezeigt haben, spart dies viel Zeit, da die Erzeugung immer neuer Resultat-Objekte vermieden wird.

```
    /** Calls action() for the winning recognizer. */
    public void action(char [] buffer, int off, int len) {
        winner.action(buffer, off, len);
    }

    /** Returns the winning Scan object. */
    public Scan winner() { return winner; }
```

Ein `Mux` muß natürlich einen Aufruf des Aktion-Codes an den Gewinner der aktuellen Runde weiterleiten. Der Gewinner wurde in `nextChar()` immer aktuell in der Instanzvariablen `winner` hinterlegt und kann durch `winner()` auch von außen erfragt werden.

```
    /** Returns a hash code value for the object. */
    public int hashCode() {
        int hashCode = 0;
        for (int i = 0; i < next; i++) hashCode += scans[ i] .hashCode();
        return hashCode;
    }
```

```
        /** Indicate whether some other object is &quot;equal to&quot; this one. */
        public boolean equals(Object obj) {
            if (obj == null || !(obj instanceof Mux)) return false;
            if (this == obj) return true;

            Mux mux = (Mux) obj;
            if (next != mux.next) return false;
            for (int i = 0; i < next; i++)
                if (scans[ i] != mux.scans[ i] )  // check identity,
                                                  // Scan order is always the same...
                    return false;
            return true;
        }
    }
```

Auch `Mux` hat als Unterklasse von `Scan` die Methoden `equals()` und `hashCode()` zu implementieren. Als simpler Hashwert wird die Summe der Hashwerte der gekapselten `Scan`-Instanzen berechnet. Da aufgrund des Algorithmus in `pack()` der Klasse `Scanner` die Reihenfolge der gekapselten `Scan`-Instanzen gleicher `Mux` gleich ist und da dieselben `Scan`-Instanzen gekapselt werden, kann in `equals()` der Vergleich in einer `for`-Schleife auf Identität erfolgen.

3.2.2 Typische Symbol-Erkenner

Der Anwender benutzt *lolo* in folgenden Schritten: Zuerst wird ein `Scanner` erzeugt und mit `Scan`-Instanzen als Symbol-Erkenner gefüllt. Die Erkenner-Objekte werden optional mit Aktions-Objekten verbunden. Nach dem expliziten oder impliziten Berechnen der Tabelle durch `pack()` wird Symbol für Symbol durch Aufruf von `scan()` mit einem `Input` als Zeichenquelle erkannt, was wiederum Aufrufe von `action()` bei Aktions-Objekten beinhaltet.

lolo soll für den Benutzer einfach zu handhaben sein. Daher beinhaltet das System ein Paket `lolo.scans`, welches fertige Erkenner-Klassen für die typischen Symbole im Compilerbau sammelt. Natürlich kann der Benutzer jederzeit eigene Klassen implementieren und nutzen.

Im Folgenden wird die Implementierung einiger Klassen aus `lolo.scans` aufgezeigt:

Scan

Die Klasse `lolo.scans.Scan` erweitert die abstrakte Basisklasse `lolo.Scan` aller Symbol-Erkenner. Alle anderen Klassen aus dem Paket `lolo.scans` stammen von dieser Klasse ab. Die Klasse bereitet Methoden und Instanzvariablen für ihre Unterklassen vor.

```
package lolo.scans;

/** A package base class for Scan objects. */
abstract class Scan extends lolo.Scan {
    /** A State object which can be used in subclasses as the result
     * for nextChar().
     */
    protected final State stateObject = new State();

    /** May be used in subclasses to mark already reseted objects. */
    protected transient boolean reset;
```

In der Instanzvariablen `reset` kann sich eine Instanz merken, ob sie bereits für eine neue Erkennungsrunde initialisiert ist. Falls ja, kann durch einen Test von `reset` am Anfang von `reset()` der weitere Programmtext der Methode übergangen werden. Das eine `Scan.State`-Objekt

`stateObject` benutzen Objekte der Unterklassen immer wieder als Resultat von `nextChar()`. Dies erspart das ständige Erzeugen sehr vieler `Scan.State`-Objekte.

```
        /** Indicates whether some other object is equal to this one. */
        public boolean equals(Object obj) {
            if (obj == null || obj.getClass() != this.getClass()) return false;
            return true;
        }

        /** Returns the hash code of the Class-object as hash code value
         * for the object.
         */
        public int hashCode() {  return this.getClass().hashCode(); }
    } // end of Scan
```

Bei vielen Klassen der Bibliothek sind Objekte schon äquivalent, wenn sie Instanzen der gleichen Klasse sind, als Beispiel alle Arten von Kommentaren oder Zwischenraum. Für diese Erkenner kann kein pro Objekt spezifisches Verhalten gesetzt werden. Daher implementiert `Scan equals()` und `hashCode()` dementsprechend für Unterklassen vor. Klassen, welche ein anderes Verhalten der Methoden brauchen, haben diese zu ersetzen.

Set

`Set`-Objekte erkennen ein Zeichen, welches innerhalb oder außerhalb einer Menge von Zeichen liegt.

```
    package lolo.scans;

    /** A class to scan for a character from a set of characters. */
    public class Set extends Scan {
        /** The set of characters. */
        protected final String set;

        /** Marks if the symbol characters have to be inside or outside set. */
        protected final boolean inside;

        /** Just calls this(set, true).*/
        public Set(String set) throws IllegalArgumentException {  this(set, true); }

        /** Constructs a recognizer which scans characters from a set
         * of characters. If inside is true the characters have to be inside set,
         * else they have not to be in set.
         */
        public Set(String set, boolean inside) throws IllegalArgumentException {
            if (set == null) throw new IllegalArgumentException("set == null");
            this.set = set; this.inside = inside;
        }
```

Bei Konstruktion wird die Zeichenmenge als `String` angegeben. `inside` bestimmt, ob das erkannte Zeichen innerhalb oder außerhalb der Menge zu suchen ist.

```
        public void reset() {}

        public State nextChar(char ch) {
            return stateObject.set(
                false,                      // more characters ?
                inside ?                    // found symbol ?
                    set.indexOf(ch) >= 0 :
                    set.indexOf(ch) < 0
            );
        }
```

Ein `Set`-Objekt hat in der Resetphase nichts zu tun, da es keinen Zustand während der Erkennung von Symbolen braucht. Bereits das erste Zeichen entscheidet den Ausgang der Erkennung.

Die Klasse `Set` stammt von `Scan` aus dem aktuellen Paket ab. Sie erbt damit die Instanzvariable `stateObject`, welche immer wieder als Resultat von `nextChar()` genutzt wird. Da `Set` nur ein Zeichen als Symbol erkennt, fordert das Resultat von `nextChar()` keine weiteren Zeichen, und je nach Wert von `inside` wird das aktuelle Zeichen innerhalb oder außerhalb der Menge `set` gesucht.

```
        /** Indicates whether some other object is equal to this one. */
        public boolean equals(Object o) {
            if (o == null || !(o instanceof Set)) return false;
            Set s = (Set) o;
            return this == s || (set.equals(s.set) && inside == s.inside);
        }

        /** Returns a hash code value. */
        public int hashCode() {
            return set.hashCode();
        }
    }
```

Wie alle Erkenner hat auch `Set` die Methoden `equals()` und `hashCode()` zu implementieren. In `equals()` werden `set` und `inside` verglichen, und als Hashwert wird der Hashwert von `set` verwendet.

Char

`Char` zeigt die Mächtigkeit der Objekt-Orientierung. Durch Vererbung wird aus der Klasse `Set` sehr einfach eine Klasse zum Erkennen einzelner Zeichen:

```
    package lolo.scans;

    /** A simple class to scan a single character. */
    public class Char extends Set {
        /** Creates an object which accepts any character. */
        public Char() { super("", false); }

        /** Creates an object which scans for the character ch. */
        public Char(char ch) { super(ch+"", true); }
    }
```

Ein `Char` kann irgendein oder ein spezifisches Zeichen erkennen. Bis auf die Konstruktoren erbt die Klasse alles von `Set`.

SetMN

Die Klasse `SetMN` ist eine weitere Erweiterung zu `Set`. Ein `SetMN` erkennt zwischen `m` und `n` Zeichen, welche innerhalb oder außerhalb einer Menge von Zeichen liegen:

```
package lolo.scans;

/** A class to scan for many character from a set of characters. */
public class SetMN extends Set {
    /** The limits. */
    protected final int m, n;

    /** Constructs a recognizer which scans minimal m and maximum n characters.
     * If inside is true the characters have to be inside set,
     * else they have not to be in set.
     */
    public SetMN(String set, boolean inside, int m, int n) throws
        IllegalArgumentException {
        super(set, inside);
        if (m < 1) throw new IllegalArgumentException("m < 1");
        if (n < m) throw new IllegalArgumentException("n <= m");
        this.m = m; this.n = n;
    }
```

Da `SetMN` von `Set` abstammt, hat `SetMN` nur `m` und `n` als neue Information zu verwalten.

```
    /** Counts the current number of characters. */
    protected transient int jog;

    public void reset() {  jog = 0; }

    public State nextChar(char ch) {
        jog ++;
        boolean b = inside ? set.indexOf(ch) >= 0 : set.indexOf(ch) < 0;
        return stateObject.set(
            b && jog < n,  // more characters ?
            b && jog >= m  // found symbol ?
        );
    }
```

Ein `SetMN` zählt in der Variablen `jog` die Anzahl der bereits erkannten Zeichen. In `reset()` ist daher `jog` wieder auf Null zu setzen. In `nextChar()` ist lediglich das aktuelle Zeichen gegen die Menge und `jog` gegen die Grenzen zu testen.

```
    public boolean equals(Object o) {  ... }
    public int hashCode() {  ... }
} // end of SetMN
```

Die Implementierung von `equals()` prüft alle vier Informationen (`set`, `inside`, `m` und `n`), und in `hashCode()` wird aus allen vier Informationen ein Hashwert berechnet.

Die Methoden `equals()` und `hashCode()` sind oft recht einfach bzw. in `lolo.scans.Scan` bereits vorimplementiert. Ich werde diese daher bei folgenden Klassen zumeist nicht weiter erwähnen.

Int

Auch bei `SetMN` führt eine einfache Spezialisierung durch eine Unterklasse zu einem sinnvollen Erkenner:

```
package lolo.scans;

/** A class to scan for integer numbers. */
public class Int extends SetMN {
    /** Constructs a recognizer which scans one ore more digits as symbol. */
    public Int() {  this(Integer.MAX_VALUE); }

    /** Constructs a recognizer which scans one or up to maxDigits digits
     * as symbol.
     */
    public Int(int maxDigits) {  super("0123456789", true, 1, maxDigits); }
}
```

`Int` erkennt Integer-Zahlen. Die Anzahl der maximal erkannten Zeichen pro Zahl kann über einen der beiden Konstruktoren gesetzt werden.

Word

`Word`-Objekte erkennen vorgegebene Zeichenfolgen:

```
package lolo.scans;

/** A class to scan for a string (a word) as symbol. */
public class Word extends Scan {
    /** The word. */
    protected final char[] word;

    /** Constructs a Word to find the string word. */
    public Word(String word) throws IllegalArgumentException {
        if (word == null) throw new IllegalArgumentException("null word");
        if (word.length() == 0) throw
            new IllegalArgumentException("word.length() == 0");
        this.word = word.toCharArray();
    }
```

Für einen schnellen Zugriff auf die einzelnen Zeichen des Worts werden diese in einem Array abgelegt.

```
    /** Marks next character inside word to find. */
    protected transient int index;

    public void reset() {  index = 0; }

    public State nextChar(char ch) {
        boolean okay = ch == word[ index++];
        return stateObject.set(
            okay && index < word.length,// more
            okay && index == word.length// found
        );
    }
} // end of Word
```

`index` verweist während einer Runde auf das nächste zu erkennende Zeichen und wird daher in `reset()` mit Null initialisiert. Paßt das nächste Zeichen zu der gesuchten Folge und sind alle Zeichen erkannt, signalisiert das Resultat-Objekt, daß das Symbol erkannt worden ist. Paßt das Zeichen und sind noch nicht alle Zeichen erkannt, werden weitere angefordert.

SimpleWhitespace, JavaWhitespace

Ein weiteres Beispiel für den sinnvollen Einsatz von Vererbung sind Zwischenraum-Erkenner. Die Klasse `SimpleWhitespace` akzeptiert ein oder mehrere Zeichen mit einem ASCII-Code kleiner oder gleich dem Leerzeichen:

```
package lolo.scans;

/** Instances of this class scan for one or more character from 0x00 - 0x20.
 * As default these symbols will be ignored.
 */
public class SimpleWhitespace extends Scan {
    /** Sets ignore to true. */
    { ignore = true; }
```

Typischerweise wird in einem Compiler Zwischenraum oder Kommentar ignoriert. *lolo* zielt als Haupteinsatzgebiet auf den Compilerbau. Daher werden Instanzen von `SimpleWhitespace` per default ignoriert. Dies geschieht durch das Setzen der Instanzvariablen `ignore` in einem Initialisierungsblock.

```
    public void reset() {}

    /** Returns whether ch is a whitespace character. */
    protected boolean isWhitespace(char ch) { return ' ' >= ch; }

    public State nextChar(char ch) {
        boolean b = isWhitespace(ch);
        return stateObject.set(b, b);
    }
} // end of SimpleWhitespace
```

In `nextChar()` wird durch Aufruf der Methode `isWhitespace()` getestet, ob das aktuelle Zeichen ein Zwischenraum-Zeichen ist.

Die Methode `isWhitespace()` bietet Unterklassen durch Ersetzen der Methode einen einfachen Weg, die Klassifizierung von Zwischenraum-Zeichen zu ändern. Neben der Spezialisierung von Klassen durch Unterklassen liegt in dieser Technik der Abänderung von Algorithmen durch das Überschreiben von entsprechenden Methoden die Macht der Objekt-Orientierung.

Die Klasse `JavaWhitespace` nutzt dieses aus. Sie stammt von `SimpleWhitespace` ab und hat nur die Methode `isWhitespace()` zu überschreiben. Die gesamte Funktionalität wird von der Oberklasse geerbt:

```
package lolo.scans;

/** Instances of this class scan for one or more Java whitespace
 * characters. The test is done by the java.lang.Character.isWhitespace()
 * method. As default these symbols will be ignored.
 */
public class JavaWhitespace extends SimpleWhitespace {
    protected boolean isWhitespace(char ch) {
        return Character.isWhitespace(ch);
    }
}
```

Flt

Die Klasse `Flt` stellt als Gleitkommazahl-Erkenner eine komplexere Klasse dar und ist intern durch einen endlichen Automaten realisiert.

```
package lolo.scans;

/** A class to scan for floating point numbers. */
public class Flt extends Scan {
    /** The legal characters. */
    protected static final String chars = "+-.0123456789eE";

    /** The state table. */
    protected static final int newState[][] = {
        // current state, input => newstate
        //+  -  .  0  1  2  3  4  5  6  7  8  9  e  E
        { 8, 8, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 8, 8 },  // state 0
        ...
        { 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 }   // state error
    };
```

Die möglichen Zustandswechsel sind in Form eines zweidimensionalen Arrays kodiert. Weiterhin enthält die Instanzvariable `chars` alle möglichen Zeichen eines Symbols.

```
    /** The current state. */
    protected transient int state;

    public void reset() { state = 0; }
```

Der aktuelle Zustand wird in `state` mitgeführt. In `reset()` wird daher `state` auf den Startwert Null zurückgesetzt.

```
    public State nextChar(char ch) {
        // is ch a legal input character?
        int input;
        if ((input = chars.indexOf((int) ch)) < 0)
            return stateObject.set(false, false);

        boolean found = false, more = false;
        state = newState[state][input];
        switch(state) {
            case 8:  break;    // error state, parse ends
            ...
            case 7:  found = more = true; break;
            default: new RuntimeException("illegal state "+state);
        }

        return stateObject.set(more, found);
    }
} // enf of Flt
```

In `nextChar()` wird zuerst geprüft, ob das aktuelle Zeichen eines der möglichen Zeichen des Symbols ist. Falls ja, wird der neue Zustand anhand der Tabelle ermittelt und das entsprechende `Scan.State`-Objekt zurückgeliefert.

Analog zu `Flt` sind einige andere Klassen implementiert. So sind die meisten Klassen zum Finden von Kommentaren in Form von endlichen Automaten realisiert.

Ein Zustandswechsel kann aus mehr als dem Ermitteln eines neuen Zustands bestehen, und das Ermitteln des neuen Zustands kann mehr als den Zugriff in die Tabelle beinhalten: Eine Instanz von `CComment`, welche (optional geschachtelte) C-Kommentare erkennt, zählt bei jedem neuen Kommentar-Start eine Variable hoch. Erst wenn für einen Erkenner, der geschachtelte Kommentare erkennt, beim Schließen eines Kommentars und dem Dekrementieren der Variablen diese wieder auf Null zurückgeht, ist der Kommentar erkannt.

3.2.3 Ein Beispiel

Das folgende Programm verwirklicht einen Scanner für einige Symbole einer simplen Programmiersprache:

```
...
public class Example {
    public static void main(String args[]) throws Exception {
        Scan.Action action = new Scan.Action () {
            public void action(Scan sender, char [] buffer, int off, int len) {
                System.out.println("\tfound -"+new String(buffer, off, len)+
                                   "-\tsender is "+sender);
            }
        };
```

Alle Erkenner werden sich das eine Aktions-Objekt `action` teilen. Dieses gibt das erkannte Symbol und den Symbol-Erkenner auf der Standardausgabe aus.

Im nächsten Schritt wird der Scanner erzeugt und mit einigen Erkennern gefüllt.

```
        Scanner scanner = new Scanner(
            new Scan[] {
                new Word("<=").setAction(action),
                new Word(">=").setAction(action),
                new Word("==").setAction(action)
        });
        scanner.add(new Int().setAction(action));
        scanner.add(new Flt().setAction(action));
        scanner.add(new SimpleIdentifier().setAction(action));
        scanner.add(new SimpleWhitespace());
        scanner.add(new CComment(true).setAction(action).setIgnore(false));
        scanner.add(new Set("+-=/*<>;()").setAction(action));
```

Der Scanner wird bereits bei Konstruktion oder nachträglich per `add()` mit `Scan`-Instanzen gefüllt. Vor dem Start des Scanners wird einmalig explizit (oder vom Scanner bei Aufruf der Erkennung einmalig implizit) `pack()` zur Berechnung der Tabelle aufgerufen.

Per default wird `CComment` ignoriert. Um die Kommentare zu sehen, wird im Beispiel der Erkenner anders eingestellt. Natürlich sollen geschachtelte C-Kommentare erlaubt sein.

```
        scanner.pack();
```

Als Eingabe wird ein `Input` erzeugt, der von der Standardeingabe liest, und in einer `while`-Schleife wird anschließend der Scanner abgefahren. Zu Beginn hat der Puffer des Input für 2 Zeichen Platz. Da einige Symbole sicherlich länger sein werden, muß und wird sich der Puffer dynamisch vergrößern.

```
            Input input = new Input(new InputStreamReader(System.in), 2);
            Scan winner;
            while ((winner = scanner.scan(input)) != null)
                ;
        }
    } // end of Example
```

Das Beispiel in Ausführung:

```
$ export CLASSPATH=.:$HOME/Promotion/jars/lolo.jar
$ java Example
while (a*3.4 <= 23 ) /* a /* nested */ comment */
        found -while-   sender is lolo.scans.SimpleIdentifier
        found -(-       sender is lolo.scans.Set[ +-=/*<>;(),true]
        found -a-       sender is lolo.scans.SimpleIdentifier
        found -*-       sender is lolo.scans.Set[ +-=/*<>;(),true]
        found -3.4-     sender is lolo.scans.Flt
        found -<=-      sender is lolo.scans.Word[ <=]
        found -23-      sender is lolo.scans.Int[ maxDigits 2147483647]
        found -)-       sender is lolo.scans.Set[ +-=/*<>;(),true]
        found -/* a /* nested */ comment */-
                        sender is lolo.scans.CComment[ nested]
```

Zwischenraum wird hier nicht ausgegeben, da der Zwischenraum-Erkenner ignoriert wird. Die Ausgabe zu dem erkannten C-Kommentar wurde von mir auf zwei Zeilen verteilt.

```
print a = a + 1;
         found -print-   sender is lolo.scans.SimpleIdentifier
         found -a-       sender is lolo.scans.SimpleIdentifier
         found -=-       sender is lolo.scans.Set[ +-=/*<>;(),true]
         found -a-       sender is lolo.scans.SimpleIdentifier
         found -+-       sender is lolo.scans.Set[ +-=/*<>;(),true]
         found -1-       sender is lolo.scans.Int[ maxDigits 2147483647]
         found -;-       sender is lolo.scans.Set[ +-=/*<>;(),true]
^D
$
```

Das Beispiel zeigt, daß ein Scanner — wie hier für eine kleine Programmiersprache — mit der objektorientierten Technik von *lolo* sehr leicht, fehlerfrei und schnell zu verwirklichen ist.

3.2.4 Serialisierung

Die Klassen `Scanner` und `Scan` adaptieren beide das Interface `java.io.Serializable`. Weiterhin serialisiert `Scan` eine zugehörige `Scan.Action`-Instanz, wenn diese auch `Serializable` ist. Bei allen Klassen ist sehr darauf geachtet, daß nur die nötigen Informationen möglichst kompakt serialisiert werden.

Ein Scanner kann als Kollektion von Objekten serialisiert und zu einem späteren Zeitpunkt deserialisiert und wiederverwendet werden. Dabei muß die Tabelle nicht neu berechnet werden, und der `Scanner` ist sofort einsatzbereit.

3.2.5 Zeiten

lolo wurde für den Einsatz im Compilerbau konzipiert. Die Bibliothek umfaßt daher Klassen zum Erkennen der typischen Symbole im Compilerbau. Dem Anwender steht es aber frei, andere Klassen zu implementieren oder durch die Aktionen *lolo* als Filter zu benutzen.

Aus Messungen weiß man, daß ein Compiler teilweise über 50 Prozent der Gesamtzeit in der lexikalischen Analyse verbringt [Bor79]. Aus diesem Grund sollten die Vorteile von *lolo* nicht zu inakzeptablen Geschwindigkeitseinbußen führen.

Das Paket `lolo.test` beinhaltet unter anderem Klassen zum Ausmessen von Scannern, die alle Symbole eines Java-Programms erkennen: alle zwei Zeichen Operatoren (>=, <=, ==, && und ||), Integer- und Gleitkommazahlen, Bezeichner, alle Arten von Kommentaren (C++, C und *javadoc*), Zeichenkonstanten, in Anführungszeichen eingeschlossene Zeichen, einzelne Zeichen (aus der Menge `+-=/@*/!<>?:;|[]{}().,%&`) und Zwischenraum.

Der Scanner `lolo.test.TypicalCompilerScannerByLolo` ist mit *lolo* implementiert. Ist der Wert der Property `PrintTypicalCompilerSymbols.print` darüber hinaus `true`, werden die erkannten Symbole textuell auf der Standardausgabe ausgegeben.

Der Scanner `lolo.test.TypicalCompilerScannerByJLex` verhält sich analog und ist mit *JLex* ([JLex], [Küh00c]), einem Java *lex*-Port als konventionelle Technik, implementiert.

Beide Scanner haben über 2 MByte an Java-Quelltext in Symbole in fünf Durchgängen auf verschiedenen Rechnern und Plattformen zerlegt. Die folgende Tabelle zeigt die gemessenen Durchschnittszeiten in Sekunden:

	suleika Pentium III, Linux, 2 x 450Mhz, 256MB	tom PowerPC, MacOSX, 450Mhz, 512MB	luna2 Pentium III Katmai, Linux, 500Mhz, 128MB
lolo	2.649 s	3.575 s	2.909 s
JLex	1.283 s	2.162 s	1.385 s
lolo (mit Symbolausgabe)	32.893 s	147.582 s	34.655 s
JLex (mit Symbolausgabe)	27.620 s	148.945 s	27.853 s

Tabelle 3.1: Scanner-Durchschnittszeiten.

Geht es um die reine Erkennung der Symbole, ist *lolo* um den Faktor 1.65 bis 2.10 langsamer als *JLex*. Aber im Compilerbau kommen zum Erkennen der Symbole immer auch Aktionen hinzu, welche unabhängig für alle Scanner gleich lange dauern. Gibt man die Symbole als Text aus, ist *lolo* um den Faktor 0.99 bis 1.24 langsamer.

3.2.6 Fazit lolo

Es scheint, daß der objekt-orientierte Ansatz von *lolo* zur Scanner-Konstruktion viele Vorteile und nur wenige Nachteile gegenüber der herkömmlichen Technik à la *lex* bietet:

Enthält die Bibliothek `lolo.scans` nicht die benötigte Klasse, dauert das Implementieren einer neuen Klasse sicherlich länger als das Hinzufügen eines regulären Ausdrucks in *lex*. Aber für den Einsatz im Compilerbau sollte die Bibliothek komplett sein, und durch Bildung von Unterklassen sollten neue Erkenner schnell zu programmieren sein.

Der Anwender von *lolo* muß sich einen Überblick über die Klassen der Bibliothek und des Framework verschaffen. Dagegen dauert das Erlernen von *lex* meines Erachtens etwas länger, da die Syntax von regulären Ausdrücken und die Syntax und Funktionen von *lex* zu erlernen sind.

Ein *lolo*-Scanner braucht im Gegensatz zu einem *lex*-Scanner zur Laufzeit die Klassen-Bibliothek als Laufzeitunterstützung. Dies bedeutet aber lediglich die Erweiterung des Klassenpfads um eine weitere Komponente.

Der einzige echte Nachteil stellt die Geschwindigkeitseinbuße dar. Ein *lex*-Scanner ist ein deterministischer, endlicher Automat, beschrieben durch eine Zustandstabelle. Ein *lolo*-Scanner mit `Mux`-Instanzen dagegen betrachtet einige Symbol-Erkenner parallel, was für Symbole im Compiler glücklicherweise aber kaum der Fall ist. Weiterhin muß nach jeder Runde den beteiligten Erkennern `reset()` geschickt werden. Beides führt dazu, daß der objekt-orientierte Ansatz etwas langsamer ist.

Alternativ könnten `Mux` analog zu `Scanner` wiederum eine Tabelle ihrer gekapselten `Scan`-Instanzen unterhalten. Die Tabellen würden die `Scan` je nach Verschachtelungstiefe nicht nach dem ersten Zeichen, sondern nach weiteren Zeichen indizieren. Momentan ist `Mux` eine recht einfache Klasse. Durch die Erweiterung würde sich das aber ändern, da ein `Mux` dann auch die `Input`-Instanz kennen und vielleicht auch deren Zustände bezüglich des Puffers setzen müßte. `Scanner` sollte dann wohl von `Mux` abstammen.

Der objekt-orientierte Ansatz bietet dagegen eine Vielzahl von Vorteilen und Möglichkeiten:

lolo kann sehr einfach an durch Parsergeneratoren (wie zum Beispiel *oops* ([Oops]) oder *jay* ([Jay], [Küh99]) erzeugte Parser angeschlossen werden.

Der Ansatz ist definitiv intuitiver im Gebrauch von objekt-orientierten Sprachen wie Java, Objective C oder C++.

Die Erkenner-Bibliothek beinhaltet einfache, aber mächtige Symbol-Erkenner. Für *lex* kann analog nur eine Sammlung von regulären Ausdrücken angegeben werden, welche kryptisch sind und nicht immer fehlerfrei kopiert werden müssen. Die Klassen dagegen sind ausgiebig getestet und somit sofort einsetzbar.

Die Klassen der Erkenner können von Scanner zu Scanner wiederverwendet werden. Reguläre Ausdrücke dagegen sind von Hand zu kopieren.

Firmen können ihre eigene Bibliothek von Erkennern zur Verfügung stellen. Zur Anwendung oder Erweiterung der Klassen braucht der Benutzer nicht die Quellen. Reguläre Ausdrücke dagegen sind immer als Quelle anzugeben.

Objekte kapseln Zustand und können daher mächtiger als einfache endliche Automaten sein. Das Erkennen von geschachtelten Kommentaren stellt damit kein Problem dar.

Eine Klasse, welche reguläre Ausdrücke als Erkenner-Objekte kapselt, würde das Verwenden bereits existenter regulärer Ausdrücke vereinfachen.

In *lolo* können die Aktions-Objekte im laufenden Betrieb des Scanners ausgetauscht werden. In einem *lex*-Aktionsblock müßte dazu eine ähnliche Technik nachgebildet werden, welche einen Verweis auf ein Objekt als Aktion verwendet.

Die Klassen `Scanner` und `Input` sind lose gekoppelt. Ein `Scanner` kann mit verschiedenen `Input` aufgerufen werden, und der `Scanner` kann direkt vor dem Aufruf umkonfiguriert werden. Allerdings würde dies eine Neuberechnung der Tabelle bedeuten. Daher ist der Einsatz verschiedener `Scanner` mit dem gleichen `Input` ein billigerer Weg, um Zustände zu modellieren. Dies entspricht dem Hinzufügen oder Ändern der regulären Ausdrücke eines *lex*-Scanners bzw. dem Einsatz von Zuständen. Zumindest der erste Teil zieht aber das Beenden, das erneute Erzeugen und Übersetzen des Scanners nach sich.

`Scanner`- und `Scan`-Instanzen sind serialisierbar. Damit kann ein Scanner wiederverwendet werden. Dies für verschiedene Projekte und sogar auf verschiedenen Plattformen. Keine erneute Übersetzung wie bei *lex* ist nötig.

`Input` arbeitet auf einem `Reader` und damit ganz natürlich auf Unicode-Ebene. Die Tabelle eines `Scanner`s kann für Index-Zeichen aus dem ASCII- oder Unicode-Zeichensatz gefüllt werden. Soll zum Beispiel ein *JLex*-Scanner auf Unicode-Ebene arbeiten, dauert das Berechnen der Tabellen wesentlich

länger. Außerdem ist zum Beispiel das Spezifizieren eines regulären Ausdrucks für Java-Bezeichner so gut wie unmöglich.

Unterklassen wie `Char` oder `JavaWhitespace` spezialisieren oder ändern das Verhalten der Oberklasse. Bei Mehrdeutigkeiten gewinnt der erste der längsten Erkenner. Dies könnte als Beispiel eine Unterklasse von `Mux` ändern, was in *lex* nicht möglich ist.

3.3 oolex

oolex (**o**bject-**o**rientated **lex**er, ([Oolex], [Küh00b])) ist der Vorgänger von *lolo*. Anhand von *oolex* wurde die Idee der konkurrierenden Erkenner zum ersten Mal umgesetzt und untersucht.

3.3.1 Vergleich zu lolo

Das Framework von `oolex` ist analog zu *lolo*. Alle Erkenner-Klassen stammen von einer abstrakten Basisklasse `Scan` ab, von der hier nur die entscheidenden Unterschiede zu der *lolo*-Version gezeigt werden:

```
package oolex;
...
public abstract class Scan implements Serializable, Cloneable {
    public abstract boolean next (int ch);

    protected transient int length;
    public int getLength() { return length; }

    protected transient boolean accepted;
    public boolean getAccepted() { return accepted; }
```

Die Methode `next()` dient zur Verarbeitung eines Zeichens. `next()` liefert als Resultat, ob das Objekt weitere Zeichen akzeptiert. Durch die Methoden `getAccepted()` und `getLength()` wird erfragt, ob das Objekt mit dem letzten durch `next()` empfangenen Zeichen gerade ein Symbol erkannt hat und falls ja, wie viele Zeichen das Symbol lang ist. Dazu müssen in `next()` die Instanzvariablen `length` und `accepted` unterhalten werden.

```
    public Object clone () { ... }
```

Unter anderem nach jeder Erkennungsrunde wird durch Aufruf von `clone()` eine Kopie des Empfängers für die nächste Erkennungsrunde erzeugt. Das neue Objekt muß fertig initialisiert für die Erkennung des Symbols sein.

```
    public String toJLexRe () throws RuntimeException { ... }
} // end of Scan
```

Einige *oolex*-Erkenner können sich selbst durch `toJLexRe()` als regulären Ausdruck darstellen. Dies kann nicht für alle Erkenner gelten. Ein Erkenner für geschachtelte Kommentare wird bei Aufruf der Methode eine Exception auslösen.

`Scan`-Instanzen kennen wiederum `Action`-Objekte und sind optional als zu ignorieren markierbar.

3.3.2 Kombination von Scan-Instanzen

oolex bietet neben der Bibliothek typischer Symbol-Erkenner weiterhin Klassen zum Kombinieren von `Scan`-Instanzen:

`Scan.Alt` `public Alt (Scan[] entry) { ... }`	Sammelt analog zu `lolo.Mux` viele Erkenner, welche parallel betrieben werden.
`Scan.BOL, Scan.EOL`	Zeilenanfang und -ende.
`Scan.Loop` `public Loop (Scan node, int m, int n) { ... }`	Kapselt eine `Scan`-Instanz, welche `m`- bis `n`-mal erkannt wird.
`Scan.Opt` `public Opt (Scan node) { ... }`	Erkennt optional einen gekapselten Erkenner.
`Scan.Seq` `public Seq (Scan first, Scan second) { ... }`	Sequenz von zwei `Scan`-Instanzen; durch Kaskadierung kann daraus eine beliebig lange Folge werden.

Tabelle 3.2: Kombinationsklassen von oolex.

Durch die Verwendung dieser Klassen ist die Implementierung vieler Symbole ein Kinderspiel. Ein C++-Kommentar von `//` bis Zeilenende besteht damit aus einer Sequenz von `//`, gefolgt von einem beliebigen Zeichen (außer dem Zeilenende) beliebig oft und abschließend dem Zeilenende.

Nachteil dieser Technik ist, daß zum Beispiel innerhalb von `Loop` oder `Seq` mehrere Instanzen der gekapselten Erkenner erzeugt und parallel betrieben werden. Zum Erzeugen der neuen Aktivierungen wird `clone()` verwendet.

Da `clone()` ein neues und zum Erkenner vorinitialisiertes Objekt liefert, besteht die Resetphase nach einer Runde aus dem Erzeugen neuer Objekte per `clone()` gegen alle beteiligten Objekte.

Es zeigt sich, daß reguläre Ausdrücke mit Hilfe der Kombinationsklassen und einigen primitiven Erkennern wie `Word` oder `Char` als Baum von `Scan`-Instanzen darstellbar sind. *oolex* beinhaltet daher einen rudimentären Parser, welcher reguläre Ausdrücke zu `Scan`-Instanzen wandeln kann. Somit kann ein Symbol-Erkenner unmittelbar als regulärer Ausdruck geschrieben werden.

3.3.3 Fazit oolex

Zeitmessungen zeigen, daß *oolex* in der reinen Symbol-Erkennung um den Faktor 28 bis 50 langsamer als *JLex* ist. Dies hat vier Gründe:

Bei *oolex* sind an jeder Runde alle Erkenner beteiligt, was zumindest für das erste oder die ersten Zeichen viel Zeit beansprucht. In *lolo* wurde dies durch die im voraus berechnete Tabelle vermieden.

Das Klonen eines neuen Objekts per `clone()` dauert in Java relativ lange. Da `clone()` auch zum Zurücksetzen der Erkenner nach jeder Runde benutzt wird und da im Gegensatz zu *lolo* alle Erkenner an jeder Runde teilnehmen, wird `clone()` sehr oft aufgerufen, was Zeit und wegen der vielen Objekte

auch Speicherplatz (und damit Zeit im garbage collector von Java) kostet. Effektiver wäre das Trennen des Klonens und der Resetphase in zwei Methoden.

In *lolo* kapselt ein Resultat-Objekt den Zustand des Erkenners nach dem Verarbeiten eines Zeichens. In *oolex* muß der Scanner immer wieder `getLength()` und `getAccepted()` aufrufen, um an diese Information zu gelangen.

oolex war ein “proof of concepts” und wurde nie auf Geschwindigkeit optimiert. In *lolo* dagegen wurde an vielen Stellen (auch mit Hilfe von Profiling) auf Geschwindigkeit geachtet. So erzeugen die Erkenner der Bibliothek nicht pro Aufruf von `nextChar()` ein neues Resultat-Objekt, sondern benutzen immer wieder dasselbe Objekt.

Die Technik des Kombinierens von Symbol-Erkennern bedingt, daß alle Erkenner klonbar sein müssen. Diese Technik ist zwar sehr mächtig, aber langsam.

oolex zeigte aber, daß der objekt-orientierte Ansatz aus der Sicht des Anwenders einfach zu benutzen und sehr mächtig ist, was zur Entwicklung von *lolo* führte. *lolo* reduziert *oolex* auf das Wesentliche.

3.4 Fazit

Der objekt-orientierte Ansatz bietet viele Vorteile gegenüber auf regulären Ausdrücken basierten Scannern. Der kleine Geschwindigkeitsnachteil wird durch die höhere Flexibilität (mächtigere Erkenner, Austausch von Erkennern und Aktionen im Betrieb, echte Unicode-Unterstützung, Serialisierung, ...) und durch einen kürzeren Entwicklungszyklus mehr als ausgeglichen.

Der Ansatz von *lolo* ist schnell und einfach. *oolex* dagegen verwendet komplexere Erkenner, welche daher wesentlich mächtigere Kombinationen ermöglichen. Dies hat aber eine wesentliche Geschwindigkeitsminderung als Konsequenz.

Die Thesen der Arbeit, siehe Kapitel 2.4 (“Thesen”), werden durch *lolo* (und auch durch *oolex*) bestätigt:

lolo beinhaltet neben den Klassen, welche das Framework eines *lolo*-Scanners ausmachen, eine Bibliothek fertiger Erkenner für den Einsatz im Compilerbau, welche von Scanner zu Scanner wiederverwendbar sind.

lolo ist leicht erweiterbar. Neue Erkenner-Klassen müssen lediglich von der abstrakten Basisklasse `Scan` abstammen. Dank der Vererbung können neue Klassen aber auch bestehende Erkenner-Klassen ableiten und so diese spezialisieren oder erweitern. Instanzen der neuen Klassen nehmen wie alle `Scan`-Objekte an dem Wettbewerb eines *lolo*-Scanners teil.

Ein *lolo*-Scanner als Gesamtheit arbeitet nach dem *divide & conquer*-Prinzip. Ein `Scanner` delegiert an eine `Input`-Instanz die Verwaltung und Beschaffung der Eingabezeichen und delegiert das Erkennen der einzelnen Symbole an viele Objekte von `Scan`-Unterklassen.

Alles zusammen ist der objekt-orientierte Ansatz eine Klasse für sich, wenn es um lexikalische Analyse geht.

4 Ein Parser aus Objekten

Die Phase der Syntax-Analyse eines Compilers untersucht, ob die von der lexikalischen Analyse gelieferte Symbolfolge syntaktisch korrekt ist. Zumeist beschreibt eine Grammatik die vom Compiler akzeptierte Sprache, und die Syntax-Analyse prüft, ob die Symbolfolge einen legalen Satz über der Grammatik bildet.

4.1 Motivation, Idee

Der Programmtext der Syntax-Analyse wird typischerweise von Tools generiert, welche in zwei Arten zu unterteilen sind:

Parsergeneratoren wie *yacc*, *jay* oder *cup* arbeiten auf einer mit Programmcode versehenen Tabelle von Grammatikregeln. Resultat ist eine tabellengetriebene Funktion, die während der Parsierung zu erkannten Teilen der Grammatik die assoziierten Programmstücke zur Ausführung bringt. *JavaCC* oder *ANTLR* erzeugen Parser, die nach der Technik des rekursiven Abstiegs agieren. Beide Generatoren erzeugen zwar Klassen, nützen aber die Vorteile der objekt-orientierten Programmierung nicht.

Inhalt dieses Kapitels ist es zu untersuchen, ob ein Parser aus Objekten gebaut werden kann. Wie sich zeigt, bringt dieser objekt-orientierte Ansatz Vorteile gegenüber den tabellengetriebenen Parsern, bringt eine Entschlackung gegenüber den zum Beispiel von *JavaCC* erzeugten Parsern, zeigt die Effizienz der Zerlegung von Algorithmen in kleine und einfache Teilalgorithmen und zeigt die Macht in der Wiederverwendung von Objekten für Compiler. Die Inhalte dieses Kapitels wurden in [Küh00a] und als Thema von zwei Vorträgen ([Küh01a], [Sch99b]) veröffentlicht.

4.2 Von EBNF zu Klassen, von Grammatiken zu Objekten

Es ist bekannt, daß BNF den Bau eines Syntaxgraphen steuert. Da man EBNF in BNF ausdrücken kann, gibt es auch hierfür einen Syntaxgraphen. Ziel dieses Kapitels ist es zu untersuchen, ob die Knoten eines Syntaxgraphen durch Instanzen spezifischer Klassen dargestellt werden können. Wir betrachten dazu eine EBNF-Grammatik für EBNF-Grammatiken, wobei die Klammern `{}` eine Wiederholung der Form `1-n` und `[]` einen optionalen Teil beschreiben:

```
parser  : {  rule } ;                  // a parser is one ore more rules.
rule    : ID ":" alt ";" ;             // rule is a name and a right hand side
alt     : seq [{  "|" seq }] ;         // alternatives
seq     : {  ID | LIT | TOKEN | some | opt | "(" alt ")" } ; // sequence
some    : "{ " alt "} " ;              // one or more
opt     : "[ " alt "] " ;              // zero or one
```

Ein Blick auf die Grammatik offenbart die Klassen, welche Abschnitte eines Syntaxgraphen repräsentieren können:

Ein Objekt der Klasse `Parser` stellt einen Parser bzw. die Grammatik als Sammlung vieler `Rule`-Instanzen dar, welche wiederum Grammatikregeln repräsentieren. Eine der `Rule` ist im `Parser` als Startregel ausgezeichnet. Eine `Rule` kennt ihren Namen und ein Objekt als rechte Seite.

`Alt`-Instanzen sammeln Alternativen, und `Seq`-Objekte repräsentieren Sequenzen.

`Some`- und `Opt`-Objekte kapseln einen Unterbaum als Wiederholung bzw. als optionalen Teil der Grammatik.

Als Token der Grammatik, also nicht literale, zitierte Symbole, können Bezeichner als Namen oder als Verweis auf Regeln, Literale oder Tokennamen auftreten. Token werden als Objekte der Klasse `Token`, Literale als `Lit` und Verweise auf Regeln als `Id` repräsentiert.

Eine Grammatik kann durch einen Baum von Objekten dieser Klassen abgebildet werden. Der Baum sollte optimiert werden: Ein-elementige `Seq` oder `Alt` werden durch ihr eines Element und optionale Wiederholungen bzw. Wiederholungen eines optionalen Teils durch ein `Many` ersetzt. `Many` repräsentiert eine `0`- bis `n`- malige Wiederholung des Unterbaums.

Als Beispiel eine Grammatik für arithmetische Ausdrücke:

```
sum     : product [{  ( "+" | "-" ) product }] ;
product : term [{  ( "*" | "/" ) term }] ;
term    : NUMBER | ( "+" | "-") term | "(" sum ")" ;
```

Diese Grammatik wird durch folgenden optimierten Baum repräsentiert, wobei die Einrücktiefe die Verschachtelung im Baum wiedergibt:

```
Parser
   Rule sum
      Seq
         Id product
         Many
            Seq
               Alt
                  Lit "+"
                  Lit "-"
               Id product
   Rule product
      Seq
         Id term
         Many
            Seq
               Alt
                  Lit "*"
                  Lit "/"
               Id term
   Rule term
      Alt
         Token NUMBER
         Seq
            Alt
               Lit "+"
               Lit "-"
            Id term
         Seq
            Lit "("
            Id sum
            Lit ")"
```

Das Beispiel zeigt, daß EBNF-Grammatiken durch Objekte der skizzierten Klassen zu repräsentieren sind. Nützlich wird diese Repräsentation aber erst, wenn sie auch genutzt wird. Wirth führt vor, daß auf den Syntaxgraphen von außen gearbeitet werden kann ([Wir86]). Bestehen die Knoten des Graphen aber nun aus Objekten und arbeiten diese Objekte mit ihren Methoden als Teil des Graphen im Graphen selbst, kommt es plötzlich — wie in diesem Kapitel gezeigt werden wird — zu *divide & conquer* der verschiedensten Algorithmen, und der Zustand der Objekte kann vielfältig genutzt werden:

Eine Methode `parse()` veranlaßt den Baum, eine Eingabe gegen den Baum zu prüfen, was der Parsierung eines Programms über der Grammatik gleichkommt. Die Objekte rufen dazu gegenseitig `parse()` auf, womit der Parser ein in Objekten gekapselter Parser für rekursiven Abstieg ist.

Während der Parsierung müssen die Knoten-Objekte lokal Entscheidungen über das weitere Fortschreiten der Parsierungen treffen. Dazu wird bei einem LL(1)-Parser das aktuelle Eingabesymbol gegen die jeweilige lookahead-Menge verglichen. Die Instanzen der Knotenklassen besitzen daher eine Methode zur Berechnung der Mengen und speichern diesen als lokalen Zustand in einer Instanzvariablen.

Damit eine Parsierung Erfolg haben kann, sollte die durch den Baum repräsentierte Grammatik auf LL(1)-Konformität geprüft werden. Eine weitere Methode aller Baumknoten fordert daher den Baum vor einer Parsierung auf, sich selbst zu prüfen. Dazu werden neben der Parsierung auch hier die lokal in den Knoten berechneten und gespeicherten lookahead-Mengen verwendet.

Zur Prüfung der Grammatik werden unter anderem die lokalen lookahead- und follow-Mengen verglichen. Die Knoten sind daher über eine weitere Methode auch zur Berechnung der follow-Menge in der Lage.

4.3 Prüfen einer Grammatik

4.3.1 Basisklasse Node, lookahead und follow

Alle Klassen, deren Instanzen eine Grammatik repräsentieren, sind im Paket `oops.parser` gesammelt und stammen von der abstrakten Basisklasse `Node` ab. `oops` steht dabei für **o**bject-**o**rientated **p**arser **s**ystem ([Oops]).

Im Folgenden wird kurz der Aufbau von `Node` skizziert. Alle Argumente bzw. die Implementierung der Methoden werden später genauer betrachtet.

```
package oops.parser;
...
public abstract class Node implements Serializable {
    public void add(Node node) throws ParserBuildException {  ... }
    public Node node() {  return this; }
```

Während der Konstruktion des Baums kann einem Knoten — wie zum Beispiel `Seq` oder `Alt` — durch Aufruf von `add()` ein weiterer Unterbaum hinzugefügt werden. `add()` ist keine abstrakte Methode, da ansonsten alle Unterklassen `add()` zu implementieren hätten. Für Klassen wie `Token`, `Id` oder `Lit` macht dies aber keinen Sinn. Daher wirft die Implementierung von `add()` in `Node` lediglich eine `ParserBuildException`. Klassen wie `Alt` und `Seq` ersetzen die Methode.

Eine `ParserBuildException` reflektiert, daß beim Bau einer Grammatik ein Fehler aufgetreten ist. Als Beispiel werden `ParserBuildException` ausgelöst, wenn eine `Rule` mit dem gleichen Namen mehr als einmal zu einem `Parser` hinzugefügt wird.

Durch Aufruf von `node()` wird ein Knoten veranlaßt, einen optimierten Baum als Resultat zu liefern. Knoten mit Unterknoten rufen `node()` nach unten auf. `Seq` und `Alt` liefern bei nur einem Unterknoten diesen als Resultat. Damit werden ein-elementige `Seq` und `Alt` im Baum eliminiert. Ist ein `Opt` in ein `Some` geschachtelt (oder umgekehrt), werden die beiden Knoten durch ein `Many` ersetzt, wobei der Unterbaum des inneren Knotens der Unterknoten des neuen `Many` wird. Durch Aufruf von `node()` gegen die Wurzel wird so der gesamte Baum optimiert.

```
    protected Set lookahead;
    protected transient Set follow;
```

```
    public final Set getLookahead() {
        if (lookahead == null) throw new RuntimeException("lookahead missing");
        return lookahead;
    }

    protected abstract Set setLookahead(Parser parser) throws
        CheckLL1Exception;
    protected abstract Set setFollow(Parser parser, Set succ) throws
        CheckLL1Exception;
```

Wirth ([Wir86]) führte die *first*- und *follow*-Mengen einer Grammatik ein und zeigt, daß anhand dieser Menge eine Grammatik auf LL(1) geprüft werden kann. Außerdem kann während der Parsierung anhand der lookahead-Menge die Entscheidung über den Fortgang der Parsierung getroffen werden.

Daher wird jedes Objekt eines Grammatikbaums durch Aufruf von `setLookahead()` bzw. `setFollow()` zur Berechnung der Mengen aufgerufen. Die Methoden haben als `Set`-Objekt in den Instanzvariablen `lookahead` und `follow` die berechnete Menge zu hinterlegen. Auf die Klasse `Set` wird später noch näher eingegangen. Beide Methoden liefern die berechnete Menge als Resultat.

Eine `CheckLL1Exception` reflektiert, daß eine Grammatik nicht LL(1)-konform ist. Bereits bei der Berechnung der lookahead- oder follow-Mengen kann in bestimmten Situationen eine Verletzung festgestellt werden.

```
    protected void checkLL1(Parser parser) throws CheckLL1Exception { ... }
    protected boolean checkDeadLoop() throws CheckLL1Exception { ... );
```

Jede Knotenklasse muß zwei Methoden zur Prüfung der Grammatik implementieren. `checkLL1()` bestimmt anhand der berechneten lookahead- und follow-Mengen, ob die Grammatik LL(1)-konform ist. `checkDeadLoop()` ist Teil eines Markierungs-Algorithmus über den Baum, welcher unendliche Rekursionen in der Grammatik findet.

```
    public void parse(Parser parser, Activation caller, Object action) throws
        ParseException, IOException, Activation { ... }
} // end of Node
```

`parse()` fordert einen Knoten zur Parsierung auf. Durch Aufruf von `parse()` gegen einen `Parser` wird eine Eingabe über der Grammatik erkannt. Die Objekte reflektieren damit nicht nur die Grammatik, sie stellen außerdem einen Grammatikprüfer und einen Parser für die Grammatik dar.

4.3.2 Berechnung des lookahead

Die Berechnung des lookahead ist lokal auf die Klassen verteilt und geschieht pro Knoten durch Aufruf von `setLookahead()`. Alle Knoten mit Unterknoten rufen `setLookahead()` für die Unterknoten auf und propagieren so die Berechnung der lookahead-Mengen in den Baum.

Die Berechnung beginnt beim `Parser` als Wurzel des Baums. Ein `Parser` fordert alle Regeln zur Berechnung des lookahead auf und übernimmt den lookahead von der Startregel.

Eine `Rule` übernimmt den lookahead von der rechten Seite und eine `Id` von der zugehörigen `Rule`.

Ein `Some` übernimmt den lookahead des Unterknotens als eigenen lookahead.

`Alt` berechnet den lookahead durch Vereinigung der lookahead-Mengen der Alternativen.

`Opt` und `Many` übernehmen den lookahead von ihrem Unterknoten und fügen die leere Eingabe hinzu.

`Seq` nimmt den lookahead des ersten Sequenz-Elements und addiert den lookahead der folgenden Elemente, so lange die leere Eingabe akzeptiert wird. Nur wenn alle Elemente die leere Eingabe akzeptieren, ist die leere Eingabe auch Teil des lookahead der `Seq`-Instanz.

`Lit` und `Token` als Symbole kennen ihren lookahead, welches nur aus dem Symbol selbst besteht. Knoten dieser Klassen bringen damit die ersten lookahead-Mengen in die Berechnung ein.

EBNF-Grammatiken entsprechen Nassi-Shneiderman-Diagrammen ([Nas73], [DIN66261]), da EBNF verschachtelte Strukturen beschreibt, welche wiederum durch Nassi-Shneiderman-Diagramme dargestellt werden können. Die Berechnung der lookahead-Mengen für die interessanten Klassen zeigt Abbildung 4.1 daher in einer Art Nassi-Shneiderman-Diagramm:

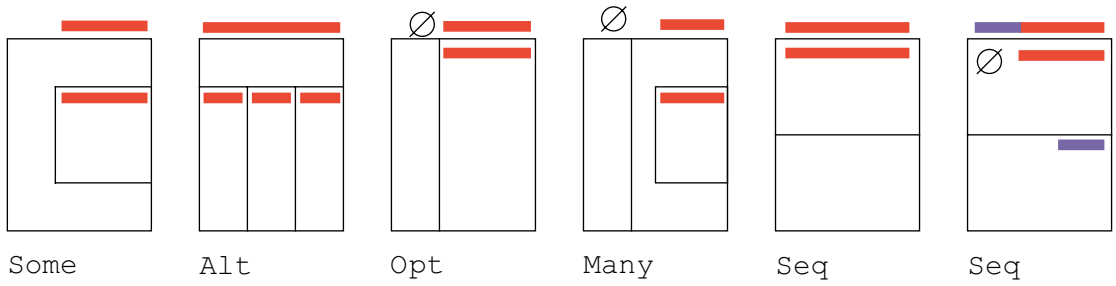

Abbildung 4.1: Die Berechnung von lookahead pro Knoten.

Intern in den Objekten werden die lookahead- und follow-Mengen durch Instanzen der Klasse `Set` modelliert:

```
package oops.parser;
...
public class Set implements Serializable {
    ...
    public Set (int token) {  ... }
    public Set (Set s) {  ... }

    public void addEmpty () {  ... }
    public void removeEmpty () {  ... }
    public boolean matchesEmpty () {  ... }
    public boolean add (Set s) {  ... }
    public boolean remove (Set s) {  ... }

    public boolean matches (Set tokenSet) {  ... }
    public boolean accepts (Set s) {  ... }
    ...
} // end of Set
```

Symbole werden in einem `Set` als nicht negative `int`-Werte repräsentiert. Drei Methoden setzen, löschen und erfragen die Information, ob die leere Eingabe Teil der Menge ist. `add()` fügt die Symbole des Arguments dem Empfänger hinzu, und `remove()` löscht die Symbole analog im Empfänger. Zwei Methoden testen ein Symbol bzw. eine Menge von Symbolen gegen ein `Set`. `matches()` erwartet als Argument ein `Set` aus nur einem Symbol und testet, ob die Empfängermenge das Symbol enthält. `accepts()` dagegen akzeptiert beliebige `Set` als Argument und testet, ob mindestens ein Symbol der Argumentmenge auch Teil der Empfängermenge ist.

Die Verteilung der lookahead-Berechnung auf die Klassen des Baums zerlegt das Problem in kleine und einfache Teilprobleme. Als Beispiel die Berechnung der lookahead-Menge in der Klasse `Many`:

```
package oops.parser;
...
public class Many extends Node {
    protected Node node;
    public Many(Node node) {  this.node = node; }

       public Set setLookahead(Parser parser) throws CheckLL1Exception {
        if (lookahead == null) {
            lookahead = new Set(node.setLookahead(parser));
            lookahead.addEmpty();
        }
        return lookahead;
    }
    ...
} // end of Many
```

Wie bereits im Text bzw. in der Grafik erklärt, übernimmt ein `Many` einmalig den lookahead seines Unterbaums und fügt die leere Eingabe zur lookahead-Menge hinzu.

Als Beispiel für eine etwas komplexere `setLookahead()`-Methode hier die der Klasse `Seq`:

```
package oops.parser;
...
public class Seq extends Node {
    protected Vector nodes = new Vector();
    ...
```

Ein `Seq` sammelt in einem `Vector` die Elemente der Sequenz. Die Anfänge von *oops* stammen aus der Zeit von Java 1.1. Mit Java 1.2 wurden die Collection-Klassen eingeführt, und `Vector` könnte durch eine Implementierung von `List` ersetzt werden.

```
    public Set setLookahead(Parser parser) throws CheckLL1Exception {
        if (lookahead == null) {
            if (nodes.size() == 0)
                throw new CheckLL1Exception(this+
                                            ": no empty sequences allowed");

            lookahead = new Set(((Node) nodes.elementAt(0)).
                                setLookahead(parser));
            if (! lookahead.matchesEmpty())
                return lookahead;
```

Zu Beginn wird der lookahead von dem ersten Element der Sequenz übernommen. Akzeptiert diese Menge aber die leere Eingabe, werden sukzessive die lookahead-Mengen der nächsten Elemente hinzugefügt. Dies endet beim letzten Element oder beim ersten Element, dessen lookahead nicht die leere Eingabe akzeptiert. Im letzten Fall wird die leere Eingabe aus dem lookahead des `Seq` gelöscht:

```
            for (int n = 1; n < nodes.size(); ++ n) {
                Set set = ((Node) nodes.elementAt(n)).setLookahead(parser);
                lookahead.add(set);
                if (! set.matchesEmpty()) {  lookahead.removeEmpty(); break; }
            }
        }
        return lookahead;
    }
    ...
} // end of Seq
```

4.3.3 Berechnung von follow

Auch die Berechnung der follow-Mengen ist lokal auf die Klassen verteilt. Während der lookahead aber durch Übernahme der lookahead-Mengen eines oder mehrerer Unterknoten berechnet wird, werden die follow-Mengen durch (wiederholtes) Übergeben von Mengen an die Unterknoten bestimmt.

Die Methode `setFollow()` wird pro Knoten zur Bestimmung von follow aufgerufen. Als zweites Argument wird dem Knoten in `succ` die follow-Menge mitgeteilt:

```
public Set setFollow(Parser parser, Set succ)
```

Das `parser`-Argument als Verweis auf das eine `Parser`-Objekt wird zum Beispiel von `Id` zum Auffinden der referierten `Rule` benutzt.

Wie Abbildung 4.2 zeigt, sind auch die Algorithmen zur Bestimmung der follow-Mengen pro Klasse grafisch darstellbar:

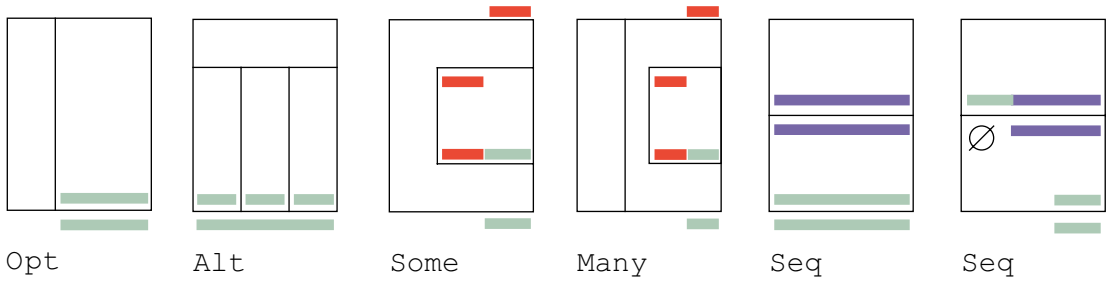

Abbildung 4.2: Die Berechnung von follow pro Knoten.

`Alt` und `Opt` speichern die übergebene follow-Menge und reichen diese als follow-Menge an den oder die Unterknoten weiter:

```
public class Opt extends Many {
    ...
    public Set setFollow(Parser parser, Set succ) throws
        CheckLL1Exception {
        ...
        follow = succ;                  // store follow
        node.setFollow(parser, succ); // send follow to subnode
        return lookahead;
    }
    ...
}
```

`Some` und `Many` verhalten sich analog. Da sie Wiederholungs-Knoten sind, fügen sie alle Symbole aus ihrem lookahead bzw. analog aus dem lookahead des Unterknotens der follow-Menge des Unterknotens hinzu:

```
public class Many extends Node {
    protected Node node;
    ...
    public Set setFollow(Parser parser, Set succ) throws CheckLL1Exception {
        ...
        follow = succ;                  // store follow

        Set s = new Set(lookahead);
        s.add(follow);
```

```
            s.removeEmpty();
            node.setFollow(parser, s); // send combined follow to subnode

            return lookahead;
        }
         ...
    }
```

`Seq` reicht seine follow-Menge zuerst an das letzte Element weiter. Ist die leere Eingabe in dem lookahead des Elements, wird die Vereinigung von follow und lookahead an das nächstletzte Element überreicht. Diese Kette wird bis zum Anfang der Sequenz ausgeführt:

```
    public class Seq extends Node {
        protected Vector nodes = new Vector();
        ...
        public Set setFollow(Parser parser, Set succ) throws CheckLL1Exception {
            setLookahead(parser);
            follow = succ;                  // store follow

            for (int n = nodes.size(); -- n >= 0; ) {
                Set prev = ((Node)nodes.elementAt(n)).setFollow(parser, succ);
                if (prev.matchesEmpty()) {
                    prev = new Set(prev); prev.removeEmpty(); prev.add(succ);
                }
                succ = prev;
            }

            return lookahead;
        }
        ...
    }
```

`setFollow()` liefert als Resultat den lookahead des Empfängers, was hier ganz elegant genutzt wird.

`Id` reicht follow an die zugehörige `Rule`, und diese reicht die Menge an das Objekt der rechten Seite weiter. Eine Rule markiert in `followChanged`, ob sich im Zuge von `setFollow()` die follow-Menge verändert hat:

```
    public class Rule extends Node {
        ...
        protected transient boolean followChanged;

        public Set setFollow(Parser parser, Set succ) {
            if (follow == null) {
                followChanged = true; follow = new Set(succ);
            } else if (follow.add(succ))
                followChanged = true;

            return lookahead;
        }

        public void setFollow(Set s) throws CheckLL1Exception {  follow = s; }
    }
```

`Rule` bildet eine Ausnahme gegenüber den anderen Bausteinen. Auf eine `Rule` kann durch mehrere `Id`-Instanzen von verschiedenen Stellen einer Grammatik aus verwiesen werden. Damit ist follow eine Vereinigung der durch die `Id`-Knoten übergebenen Mengen.

Der Parser startet die Berechnung der follow-Mengen, indem er für die Startregel zuerst die einelementige Menge mit dem EOF-Symbol für die Start-Regel hinterlegt.

```
public class Parser extends Node {
    protected Vector rules = new Vector();

    public Set setFollow (Parser parser, Set succ) throws CheckLL1Exception {
        start.setFollow(Set.getEOFSet());
```

Anschließend wird jede `Rule` durch `setFollow(Parser)` zur Weitergabe seiner follow-Menge an die rechte Seite aufgefordert.

```
        boolean followChanged;
        do {
            for (int n = 0; n < rules.size(); ++ n)
                ((Rule) rules.elementAt(n)).setFollow(this);
```

Alle Regeln markieren, ob sich die follow-Menge der Regel gerade geändert hat, was durch Aufruf von `followChanged()` zu erfragen ist. Dies geschieht, wenn eine `Id` auf der rechten Seite einer Regel als Verweis auf eine andere Regel im Zuge der Abarbeitung der rechten Seite die follow-Menge der verwiesenen Menge abändert.

```
            followChanged = false;
            for (int n = 0; n < rules.size(); ++ n)
                followChanged |= ((Rule) rules.elementAt(n)).followChanged();
```

Hat sich follow auch nur einer Regel geändert, so wird die Schleife erneut durchlaufen. Da die Grammatik nur endlich viele Regeln und auch nur endlich viele Symbole enthält, muß der Algorithmus letztendlich terminieren, wenn alle follow-Mengen berechnet worden sind.

```
        } while (followChanged);

        for (int n = 0; n < rules.size(); ++ n) {
            Rule rule = (Rule)rules.elementAt(n);
            if (rule.follow == null)
                throw new CheckLL1Exception(rule.nt+": not connected");
        }

        return null;
    }
}
```

Bereits an dieser Stelle können nicht verbundene Regeln gefunden werden, da sie an der follow-Berechnung nicht teilgenommen haben. Keine `Id` verweist auf die `Rule`, und somit ist die Instanzvariable `follow` ein `null`-Verweis.

4.3.4 Prüfen einer Grammatik

Durch Aufruf von `check()` gegen einen `Parser` kann die Prüfung der repräsentierten Grammatik veranlaßt werden:

```
public class Parser extends Node {
    private boolean check;
    protected Vector rules = new Vector();
```

```
...
public void check() throws CheckLL1Exception {
    setSets();
    setLookahead(this); setFollow(this, null);
    checkLL1(this);
    checkDeadLoop();
    check = true;
}
```

In `check()` wird zuerst die Methode `setSets()` aufgerufen, welche für alle Symbole der Grammatik — also Objekte der Klassen `Lit` und `Token` — die ein-elementigen lookahead-Mengen erzeugt. Genauer werden in `setSets()` sogar erst die `Token`-Instanzen kreiert. Während des Aufbaus des Baums für eine Grammatik werden für Namen auf der rechten Seite einer Regel immer `Id`-Instanzen in den Baum eingebaut. Erst nachdem der Baum komplett aufgebaut ist, kann berechnet werden, welche der Verweise zu Regeln korrespondieren und welche Namen `Token` als Symbole repräsentieren.

Als nächste Schritte werden die lookahead- und follow-Mengen berechnet. Bereits hierbei wird die Grammatik teilweise geprüft. So wird in `setLookahead()` eines `Alt` erkannt, wenn die lookahead-Mengen der Alternativen nicht paarweise disjunkt sind. In `setFollow()` kann, wie bereits erklärt, zum Beispiel geprüft werden, ob alle Regeln der Grammatik auch verbunden sind.

Zwei weitere Methoden, welche von allen Klassen zu implementieren sind, prüfen nun weitergehend die Grammatik. Am Ende wird in `check` markiert, daß die Grammatik geprüft worden ist. Diese Information kann zum Beispiel vor dem Parsieren abgefragt werden.

In `checkLL1(Parser)` prüft jeder Knoten des Baums, ob er LL(1)-konform ist. Ein `Parser` ruft dazu die Methode bei allen Regeln auf:

```
public void checkLL1(Parser parser) throws CheckLL1Exception {
    for (int n = 0; n < rules.size(); ++ n)
        ((Rule) rules.elementAt(n)).checkLL1(this);
}
```

Selbst wenn eine Grammatik allen LL(1)-Bedingungen genügt, kann eine unendliche Rekursion die Parsierung eines Programms über der Grammatik in eine Endlosschleife verwandeln. Ein Beispiel:

```
x :  "x" y ;
y :  "y" x ;
```

Keine endliche Symbolfolge wird von dieser Grammatik akzeptiert, da am Ende der Regel `x` die Regel `y` erkannt werden muß. Diese verweist aber am Ende ihrer rechten Seite wieder auf die Regel `x` usw...

Daher soll die Grammatikprüfung unendliche Rekursionen finden, was durch einen Markierungs-Algorithmus über den Baum geschieht. Jeder Knoten des Baums hat dazu die Methode `checkDeadLoop()` zu implementieren. Das Resultat der Methode reflektiert als Boolschen Wert, ob eine unendliche Rekursion vorliegt.

```
protected boolean checkDeadLoop() throws CheckLL1Exception {
    for (int n = 0; n < rules.size(); ++ n) {       // test every rule
        for (int i = 0; i < rules.size(); ++ i)     // unmark every rule
            ((Rule) rules.elementAt(i)).marked = false;
```

Der `Parser` testet über eine Schleife jede `Rule` als Repräsentant einer Grammatikregel. Als erstes wird dazu in `marked` jede Regel als nicht aktiv markiert, und daraufhin wird die Regel durch `checkDeadLoop()` getestet:

```
                Rule r = (Rule) rules.elementAt(n);
                if (r.checkDeadLoop())                    // test current rule
                    throw new CheckLL1Exception("aborting for unlimited "+
                                                "recursion Rule "+r+".");
            }
            return false;
        }
        ...
    } // end of Parser
```

Im Folgenden wird auf die beiden Methoden zum Prüfen einer Grammatik genauer eingegangen.

Die einzelnen checkLL1()-Methoden

Unter anderem zeigt Wirth die notwendigen Bedingungen für eine LL(1)-konforme Grammatik. Ausgehend von diesem Ansatz testen Instanzen der Klassen eines Grammatikbaums in `checkLL1()` anhand der beiden Mengen, ob der jeweilige Knoten im Baum den Kriterien entspricht.

Als Beispiel eine Grammatik mit nur einer Regel, welche binäre Zahlen beschreibt:

```
bits : { "0" | "1" } ( "0" | "1" ) ;
```

Die rechte Seite der Regel `bits` wird durch folgenden Baum repräsentiert. In den nächsten beiden Spalten werden die follow- und lookahead-Mengen der einzelnen Knoten dargestellt:

```
                  lookahead   follow
Seq               0 1         [ empty]
  Some            0 1         0 1
    Alt           0 1         0 1
      Lit 0       0           0 1
      Lit 1       1           0 1
  Alt             0 1         [ empty]
    Lit 0         0           [ empty]
    Lit 1         1           [ empty]
```

Die Grammatik ist keine LL(1)-Grammatik, da mit einem Zeichen Vorschau nicht entschieden werden kann, wann die erste Wiederholung (`Some`) endet.

Anhand der aufgezeigten Mengen wird ein Kriterium sofort klar: Für Wiederholungen müssen follow und lookahead disjunkt sein, d.h., kein Symbol der einen Menge darf in der anderen vorkommen.

Abbildung 4.3 zeigt grafisch, wie pro Klasse die Grammatik geprüft wird:

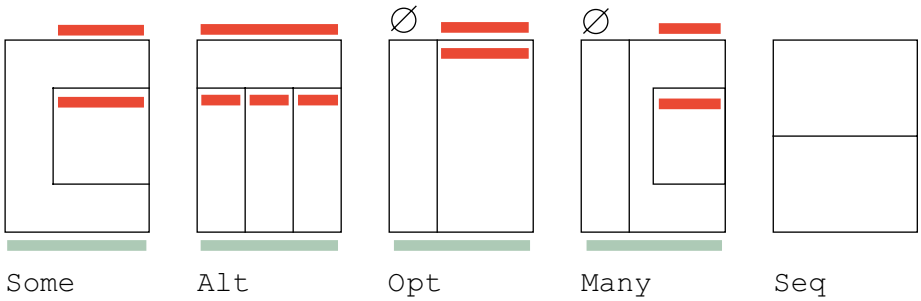

Abbildung 4.3: Die Prüfung der Grammatik pro Knoten.

Alle Klassen prüfen lokal auf LL(1) und fordern Unterknoten zum Test auf.

`Some`, `Opt` und `Many` prüfen, ob die lookahead- und follow-Menge disjunkt sind. Allerdings löst ein Konflikt keine `CheckLL1Exception` aus, sondern es wird lediglich eine Warnung ausgegeben. Diese lokale Mehrdeutigkeit entspricht einem shift/reduce-Konflikt in *yacc*. Typischerweise wird im

Compilerbau an dieser Stelle nicht reduziert, damit zum Beispiel ein `else`-Zweig immer zum innersten `if` gehört. Analog verhalten sich die Klassen: Vor dem Konflikt wird gewarnt, und der Knoten versucht selbst das Symbol zu parsieren.

```
public class Some extends Many {
    ...
        public void checkLL1(Parser parser) throws CheckLL1Exception {
            if (follow == null) throw new CheckLL1Exception(this+
                                ": follow not set");
            if (lookahead.accepts(follow))
                System.err.println(getClass().getName()+", warning:\n"+
                                this+": ambiguous"+
                                "\n  lookahead "+lookahead.toString(parser)+
                                "\n  follow "+follow.toString(parser));
            node.checkLL1(parser);
    }
    ...
}
```

Ein `Alt` testet, daß die lookahead-Mengen der Alternativen paarweise disjunkt sind. Enthält der lookahead des `Alt` die leere Eingabe, müssen darüber hinaus follow und lookahead disjunkt sein.

`Seq` ruft lediglich bei allen Elementen, `Rule` bei der rechten Seite, `Id` bei der referierten `Rule` und `Parser` bei allen `Rule` die Prüfmethode auf.

`Lit` und `Token` haben nichts zu testen.

Die einzelnen checkDeadLoop()-Methoden

Die Methode `checkDeadLoop()` dient zum Auffinden unendlicher Rekursion in einer Grammatik, d.h., keine unendliche Symbolfolge darf von der Grammatik akzeptiert werden.

Der `Parser` ruft für alle Regeln `checkDeadLoop()` auf. Eine `Rule` markiert zu Beginn der Methode in `marked`, ob sie aktiv ist. Aktiviert der Algorithmus (über eine `Id`) erneut eine bereits als aktiv markierte Regel, liegt eine unendliche Rekursion vor.

```
public class Rule extends Node {
    protected transient boolean marked;
    ...
    protected boolean checkDeadLoop() throws CheckLL1Exception {
        if (marked)
            return true;
        marked = true;
```

Wird die Methode aufgerufen und ist die Regel aktiv, wird `true` als Signal für eine unendliche Rekursion zurückgeliefert. Wird die Methode zum ersten Mal aktiviert, wird `mark` auf `true` gesetzt, und die rechte Seite wird zum Test aufgefordert:

```
        boolean result = rhs.checkDeadLoop();
        marked = false;
        return result;
    }
    ...
}
```

Nach dem Test der rechten Seite ist die Regel nicht länger aktiv, und das Ergebnis des Tests der rechten Seite wird zurückgeliefert.

Eine `Id` ruft in `checkDeadLoop()` bei der referierten `Rule` und ein `Some` beim Unterknoten wieder `checkDeadLoop()` auf. Beide liefern das Resultat des Aufrufs zurück:

```
public class Id extends Node {
    ...
    protected boolean checkDeadLoop() throws CheckLL1Exception {
        return rule.checkDeadLoop();
    }
    ...
}
```

`Many` und `Opt` sind optionale Knoten und können daher immer eine Rekursion abbrechen. `Many`, `Opt`, `Token` und `Lit` liefern als Resultat von `checkDeadLoop()` immer `false`.

Interessant sind lediglich `Seq` und `Alt`. Beide testen am Anfang der Methode, ob der lookahead die leere Eingabe beinhaltet. Falls ja, kann der Knoten eine Rekursion abbrechen, und die Knoten liefern daher in diesem Fall `false`.

Ist ein `Alt` nicht optional, werden die Alternativen durch Aufruf von `checkDeadLoop()` getestet. Liefert der Test bei einer der Alternativen `false`, so bricht der Test der Alternativen ab, und das Resultat des `Alt`-Knotens ist `false`, sonst `true`:

```
public class Alt extends Node {
    protected Vector nodes = new Vector();
    ...
    protected boolean checkDeadLoop() throws CheckLL1Exception {
        if (lookahead.matchesEmpty())
            return false;

        for (int n = 0; n < nodes.size(); ++ n)
            if (! ((Node) nodes.elementAt(n)).checkDeadLoop())
                return false;
        return true;
    }
    ...
}
```

Ein `Seq` testet mit `checkDeadLoop()` der Reihe nach alle Elemente der Sequenz. Liefert einer der Tests `true`, bricht der Test der Elemente ab, und das Resultat des `Seq` ist `true`, ansonsten `false`.

4.4 Parsierung

Eine weitere Methode `parse()` erweitert den Baum um die Möglichkeit der Parsierung von Programmen über der repräsentierten Grammatik. In der abstrakten Basisklasse `Node` der Baumklassen ist die Methode für alle Unterklassen vordefiniert:

```
public void parse(Parser parser, Activation caller, Object action) throws
        ParseException, IOException, Activation
```

`parse()` erhält als erstes Argument einen Verweis auf das eine `Parser`-Objekt des Baums. Dieser Verweis wird in dem rekursiven Aufruf von `parse()` weitergereicht, da zum Beispiel alle Klassen vom Parser das aktuelle Symbol erfragen.

Das zweite Argument `caller` dient zur Fehlerbehandlung bei einem Syntaxfehler während der Parsierung. Daher wird in diesem Abschnitt die Implementierung der einzelnen `parse()`-Methoden nicht vollständig gezeigt. Das Kapitel 8 (“Fehlerbehandlung durch Objekte und Exceptions”) erklärt die

Idee und den Mechanismus hinter der Fehlerbehandlung und zeigt komplette Methoden-Implementierungen.

Die hier vorgestellten `parse()`-Methoden erkennen ausschließlich, ob die untersuchte Eingabe einem legalen Programm über der Grammatik entspricht oder nicht. Noch werden keine Aktionen zu erkannten Teilen der Grammatik ausgeführt. Um Aktionen aber durch einfache Ableitung der Baum-Klassen zu ermöglichen, ist das dritte Argument `action` ein Objekt, welches für eine spätere Erweiterung der Klassen um Aktionen genutzt werden kann. Es wird entlang der Aktivierungen von `parse()` durchgereicht, und dem Empfänger von `parse()` bleibt die Interpretation (zum Beispiel ein gelegentlicher Austausch) überlassen.

Lokal implementieren alle Klassen `parse()` und treffen aufgrund des aktuellen Symbols und aufgrund des eigenen lookahead ihre Entscheidungen.

Die follow-Mengen werden während der Parsierung nicht mehr benötigt. Daher ist die Instanzvariable `follow` in `Node` als `transient` markiert, und damit werden die follow-Mengen nicht mitserialisiert. Dies führt zu einer kleineren Datenmenge für einen serialisierten Parser.

Der eine `Parser` stellt für alle Instanzen des Baums einige hilfreiche Methoden zur Verfügung:

```
public class Parser extends Node {
    ...
    protected final void error(Node n, Activation activation, Object action)
        throws ParseException { ... }
    protected boolean atEnd() { ... }
    protected Set tokenSet()  { ... }
    protected boolean advance() throws IOException { ... }
}
```

Während der Parsierung kann eine Instanz den `Parser` durch `tokenSet()` nach dem aktuellen Symbol fragen und mit `advance()` ein Symbol in der Eingabe vorrücken. `atEnd()` reflektiert, ob die Eingabe bereits am Ende ist. Bei einem Syntaxfehler geben die beteiligten Instanzen durch `error()` eine Fehlermeldung aus.

Eine `Rule` fordert in `parse()` lediglich ihre rechte Seite zur Parsierung auf:

```
public class Rule extends Node {
    protected Node rhs;
    ...
    public void parse (Parser parser, Activation caller, Object action) throws
        ParseException, IOException, Activation {
        rhs.parse(parser, caller, action);
    }
    ...
}
```

Ein `Alt` steht bei Aufruf von `parse()` vor der Wahl, in welche der Alternativen die Parsierung absteigen soll:

```
public class Alt extends Node {
    protected Vector nodes = new Vector();
    ...         // error recovery, chapter 7
    public void parse(Parser parser, Activation caller, Object action)
        throws ParseException, IOException, Activation {
        ...     // error recovery, chapter 7
        if (lookahead.matches(parser.tokenSet())) {
```

Paßt das aktuelle Symbol zum lookahead des `Alt`, werden die lookahead-Mengen der Alternativen untersucht, und es wird in die eine — der Baum ist geprüft — passende Alternative abgestiegen:

```
            ... // error recovery, chapter 7
            for (int i = 0; i < nodes.size(); ++i) {
              Node node = (Node) nodes.elementAt(i);
              if (node.lookahead.matches(parser.tokenSet())) {
                node.parse(parser, ..., action);
                return;
              }
            }
            throw new ParseException("This can not happen...");
          }
```

Eine `ParseException` reflektiert, daß während der Parsierung kein Syntaxfehler, sondern ein Fehler im Algorithmus der Parsierung aufgetreten ist. Das Auslösen der Exception ist an dieser Stelle eine rein defensive Programmierung, da dieser Punkt nie erreicht werden sollte. Während der Entwicklung des Systems diente das Auslösen der `ParseException` zur Fehlersuche.

Paßt das aktuelle Symbol nicht zum lookahead des `Alt`, so ist dies ein Syntaxfehler. Über einen Aufruf von `error()` beim `Parser` wird dieses dem Anwender berichtet:

```
            parser.error(this, ..., action);
            ... // error recovery, chapter 7
        }
    }
```

Einige Stellen von `parse()` sind mit Punkten ausgeblendet, da dort der Programmtext für die Fehlerbehandlung stattfindet. Analog sind in den Methoden der folgenden Klassen Teile ausgeblendet.

Die Methode für `Seq` ist etwas komplexer:

```
public class Seq extends Node {
    protected Vector nodes = new Vector();
    ...
    public void parse(Parser parser, Activation caller, Object action)
        throws ParseException, IOException, Activation {
        ...
```

In einer äußeren `for`-Schleife werden der Reihe nach die Elemente der Sequenz abgearbeitet. Paßt das aktuelle Symbol zum lookahead des gerade betrachteten Elements, steigt die Parsierung in dieses Element ab.

```
        for (i = 0; i < nodes.size(); ) {
            Node node = (Node) nodes.elementAt(i);
            if (node.lookahead.matches(parser.tokenSet())) {
                i++;
                node.parse(parser, ..., action);
                continue;
            }
```

Elemente der Sequenz können optional sein, d.h., sie akzeptieren die leere Eingabe als lookahead. Ein Beispiel:

```
start: "a" [ "b" ] "c";
```

Die Eingabe `a c` ist legal, und die Sequenz muß das mittlere Element auslassen. Paßt daher ein Symbol nicht zu einem Element, wird in einer inneren `for`-Schleife untersucht, ob die Parsierung gerade optionale Elemente zu überspringen hat:

```
                int j;
                for (j = i; j < nodes.size(); j++) {
                    Node n = (Node) nodes.elementAt(j);
                    if (n.lookahead.matches(parser.tokenSet())) {
                        i = j + 1;
                        n.parse(parser, ..., action);
                        continue;
                    }
                    if (!n.lookahead.matchesEmpty())
                        break;
```

Ist eines der Elemente nicht mehr optional, bricht die innere Schleife ab, und ein Syntaxfehler liegt vor.

```
                }
                ...
                parser.error(this, ..., action);
                ...
            }
            return;
        }
    }
```

Paßt das aktuelle Symbol zum lookahead eines `Some`, `Many` oder `Opt`, so steigt die Parsierung in den Unterknoten ab. `Some` und `Many` wiederholen den Abstieg beliebig oft, `Some` aber mindestens einmal.

```
public class Opt extends Many {
    ...
    public void parse(Parser parser, Activation caller, Object action)
        throws ParseException, IOException, Activation {
        ...
        if (lookahead.matches(parser.tokenSet())) {
            ...
            node.parse(parser, ..., action);
            return;
        }
        ...
        parser.error(this, ..., action);
        ...
        }
    }
}
```

`Some` und `Many` brauchen ein Kriterium zum Abbruch der Wiederholung. Der Abbruchtest nutzt die Technik der Fehlerbehandlung lokal aus und wird daher erst im Kapitel 8 (“Fehlerbehandlung durch Objekte und Exceptions”) erklärt.

Letztendlich obliegt es `Lit` und `Token` als Repräsentation von Symbolen, im Baum das aktuelle Eingabe-Symbol zu akzeptieren:

```
public class Lit extends Node {
    ...
    protected void shift(Object action, Parser parser) {}

    public void parse(Parser parser, Activation caller, Object action)
        throws ParseException, IOException, Activation {
        ...
        shift(action, parser);
        parser.advance();
        return;
        ...
    }
}
```

`Lit` oder `Token` werden aktiviert, wenn das aktuelle Symbol zum lookahead des Objekts paßt. `Lit` und `Token` haben aber genau das `Set` des repräsentierten Symbols im lookahead. Das Objekt kann also das aktuelle Symbol akzeptieren und den `Parser` durch `advance()` zum Vorrücken in der Eingabe auffordern.

Erkennt ein `Lit` oder ein `Token` ein Symbol, wird als Repräsentation des Ereignisses intern die Methode `shift()` aufgerufen. Der Name der Methode ist in Anlehnung an *yacc* gewählt: Die Methode wird zu dem Zeitpunkt geschickt, zu dem bei *yacc* shift stattfindet. Der Körper der `shift()`-Methode ist momentan in `Lit` und `Token` leer implementiert. Unterklassen bietet diese Methode aber eine Möglichkeit, um diesem Ereignis durch Überschreiben der Methode mit Programmtext Rechnung zu tragen (Template-Entwurfsmuster).

Die Parsierung startet mit Aufruf von `parse(Scanner,TableFactory)` gegen den `Parser`. Ein `Scanner` ist eine Instanz einer Klasse, die das `oops.scanner.Scanner`-Interface implementiert. Auf dieses Interface und auf das Interface `TableFactory` geht das folgende Kapitel 5 (“Eine objekt-orientierte Scanner-Schnittstelle”) ein.

```
package oops.parser;
...
public class Parser extends Node {
    protected transient Scanner scanner;
    ...
    public boolean parse(Scanner scanner, TableFactory tf) throws
        ParseException, IOException {
        if (!check)
            throw new ParseException("You try to run an unchecked parser...");

        this.scanner = scanner;
        ...
        parse(self);
        if (!scanner.atEnd())
            throw new ParseException(scanner+": trash at end");
        ...
        return numberOfErrors == 0;
    }
```

Resultat der Methode ist, ob die Parsierung fehlerfrei abgelaufen ist. Dazu zählt der Parser in `numberOfErrors` die Anzahl der erkannten Syntaxfehler mit.

Die eigentliche Parsierung startet in `parse(Activation)`. Diese Methode kann (und wird) von Unterklassen ersetzt werden. Hier wird die Startregel zur Parsierung aufgefordert.

```
    protected void parse(Activation caller) throws
        ParseException, Activation, IOException {
        start.parse(this, caller, null);
    }
}
```

4.5 Beispiel arithmetische Ausdrücke

Als Beispiel wird hier für eine Grammatik für arithmetische Ausdrücke ein Baum von Objekten erzeugt und geprüft. Die Regeln sind in der Grammatikdatei `arith.ebnf` aufgeschrieben:

```
$ cat arith.ebnf
sum     : product [{  ( "+" | "-" ) product }] ;
product : term [{  ( "*" | "/" ) term }] ;
term    : NUMBER | ( "+" | "-") term | "(" sum ")" ;
```

oops beinhaltet eine Klasse `oops.EBNF`, die eine Grammatik aus einer Datei oder von der Standardeingabe liest, daraufhin den Baum der Objekte als Repräsentation der Grammatik baut, diesen zur Prüfung der Grammatik auffordert und abschließend den von der Prüfung akzeptierten Baum auf der Standardausgabe serialisiert:

```
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=$JARS/oops.jar
$ java -Doops.oops.actionType=none oops.EBNF arith.ebnf > arith.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using ebnf parser generator generated by oops on Wed Aug  8 14:25:16 MEST 2001
$ ls -l arith.ser
-rw-r--r--   1 bernd    staff        2885 Aug  9 14:22 arith.ser
$
```

`oops.EBNF` kann verschiedene Arten von Parsern erzeugen. Durch die Angabe der Property `oops.oops.actionType` wird ein Parser ausschließlich aus den in diesem Kapitel vorgestellten, zu keinen Benutzer-Aktionen fähigen Klassen erzeugt. Darauf und auf die Implementierung des Parsergenerators wird in Kapitel 10 (“oops als Parsergenerator”) genauer eingegangen.

Das Werkzeug `oops.tools.Dump` dient zur textuellen Ausgabe eines serialisierten Parsers. Als Bestätigung der korrekten Arbeitsweise des Generators hier die gekürzte Ausgabe des Werkzeugs bezüglich des Beispiels:

```
$ java oops.tools.Dump arith.ser
oops.parser.Parser
   oops.parser.Rule sum : ( product [{  ( ( "+" | "-" ) product ) }] )
      oops.parser.Seq
         oops.parser.Id product
         oops.parser.Many
            oops.parser.Seq
               oops.parser.Alt
                  oops.parser.Lit "+"
                  oops.parser.Lit "-"
               oops.parser.Id product
   oops.parser.Rule product : ( term [{  ( ( "*" | "/" ) term ) }] )
      ...
   oops.parser.Rule term : (NUMBER | ( ( "+" | "-" ) term ) | ( "(" sum ")" ))
      ...
```

4.6 Zeiten

Sowohl *oops* als auch *ANTLR* und *JavaCC* erzeugen Parser, die nach der Technik des rekursiven Abstiegs arbeiten. Zum Abschätzen der Performance von *oops*-generierten Parsern wurde ein *oops*-Parser für arithmetische Ausdrücke mit analogen *ANTLR*- und *JavaCC*-Parsern verglichen. Alle drei haben eine 904 KByte große Datei mit arithmetischen Ausdrücken auf drei verschiedenen Rechnern mehrfach parsiert. Tabelle 4.1 zeigt die gemessenen Durchschnittszeiten:

	suleika Pentium III, Linux, 2 x 450Mhz, 256MB	tom PowerPC, MacOSX, 450Mhz, 512MB	luna2 Pentium III Katmai, Linux, 500Mhz, 128MB
oops-Parser zusammen mit *lolo*-Scanner	7.441 s	6.797 s	7.520 s
ANTLR-Parser mit generiertem Scanner	18.325 s	9.492 s	20.690 s
JavaCC-Parser mit generiertem Scanner	1.560 s	7.454 s	1.660 s

Tabelle 4.1: Parser-Durchschnittszeiten

Der *ANTLR*-Parser ist auf allen Maschinen teilweise mit deutlichem Abstand der langsamste Parser. Die Ergebnisse von suleika und luna2 decken sich: *oops* ist ca. um den Faktor 4.6 langsamer als *JavaCC*, aber auch ca. 12-mal schneller als *ANTLR*. Lediglich die Messungen von tom verhalten sich merkwürdig. Dort ist *oops* sogar schneller als *JavaCC*, und der Performance-Nachteil von *ANTLR* ist wesentlich geringer. Die Java-Implementierung von Apple unter MacOSX muß bestimmte Operationen besonders gut (*ANTLR*) bzw. schlecht (*JavaCC*) umsetzen.

Ein möglicher Grund, warum *oops* Performance-Nachteile gegenüber *JavaCC* besitzt, wird in Abschnitt 8.4 (“Zeiten”) näher erläutert. Auch könnte durch eine geschicktere Repräsentierung der Symbolmengen Zeit gespart werden. Die momentanen Mengen in Form von `Set`-Instanzen sind nicht auf Performance optimiert, werden aber während der Parsierung sehr aktiv verwendet.

SableCC- und Metsker-Parser wurden nicht näher untersucht, da dieser Test noch einmal sehr deutlich die Schwächen der beiden Systeme aufzeigt: *SableCC*-Parser sind zu einer reinen Erkennung eines Programms gar nicht in der Lage, da immer automatisch ein Parse-Baum gebaut wird. Für die gleiche Eingabe brauchte der *SableCC*-Parser einige Minuten und 113 MByte an Speicherplatz. Noch schlimmer ist das Resultat bei einem Metsker-Parser. Dieser versucht, Alternativen in der Grammatik parallel zu erkennen und kopiert dazu `Assembly`-Objekte als Repräsentanten der Eingabe. Da diese aber immer die komplette Eingabe enthalten, kostet das Klonen der Objekte viel Zeit und Speicherplatz. Mir stand leider nur ein Rechner mit einem Hauptspeicher von 1 GByte zur Verfügung, welcher für das Beispiel auch komplett gebraucht wird. Doch der Parser braucht noch mehr Speicher, und das System beginnt zu swappen. Da die Maschine der Server für alle *diskless*-Maschinen in unserem Netz ist, habe ich den Test daher abgebrochen.

4.7 Fazit

Wie das Kapitel gezeigt hat und die Thesen versprochen haben, sind die Vorteile der Objekt-Orientierung auch in der Entwicklung von Parsern wiederzufinden:

Das Kapitel zeigt, daß eine beliebige EBNF-Grammatik als Baum von Objekten repräsentiert werden kann. Die Knoten-Klassen sind unabhängig von der durch die Instanzen der Klassen repräsentierten Grammatik. Sie sind als Klasse allgemein für verschiedenste Grammatiken, ja sogar für verschiedenste

Grammatik-Notationen — siehe Kapitel 9 (“Grammatik-Notationen, Erweiterung durch Klassen”) — wiederverwendbar.

Der Baum als Repräsentation einer Grammatik ist serialisierbar. Damit kann ein bereits geprüfter Parser immer wiederverwendet werden. Der Baum muß nicht neu erzeugt und nicht neu geprüft werden. Aufgrund der Verwendung von Java kann der Parser, ohne neu erzeugt zu werden, direkt auf unterschiedlichsten Plattformen zum Einsatz kommen.

Aufgrund des Einsatzes der Objekt-Orientierung können Klassen voneinander erben. So stammt als Beispiel die Klasse `Some` von `Many` ab, übernimmt Instanzvariablen und Methodenimplementierungen und erweitert die Oberklasse und damit mögliche Parser schnell um die neue Funktionalität.

Die Modellierung eines so komplexen Systems wie ein Parser in Form von Objekten ist ein Beispiel für *divide & conquer*. Die Objekte der Parser-Klassen teilen die Verantwortung untereinander auf und entscheiden lokal über den Aufruf von Methoden bei anderen Objekten. Die Algorithmen sind in Form von Methoden Teil des Baums und nicht versteckt in einem Programm, welches den Baum erzeugt hat.

Da die Knoten im Baum lokal nur einen kleinen Bereich eines komplexen Algorithmus verwirklichen, sind die Knotenklassen leicht zu implementieren, und im Parser-Baum sind die Mengenberechnungen, die Prüfung der repräsentierten Grammatik und das Parsieren einer Eingabe über der Grammatik Beispiele dafür.

Das Beispiel des Findens von unendlichen Rekursionen in einer Grammatik zeigt, daß der Ansatz, daß ein Baum durch Methoden auf sich selbst arbeitet, ein mächtiges Konzept ist. Andere Parsergeneratoren wie *yacc* oder *JavaCC* finden diese Problemfälle nicht.

Diese Technik drängt sich daher geradezu für den Einsatz in der Lehre auf. Selbst ein so komplexes Problem wie das Berechnen der follow-Mengen ist lokal pro Knoten-Klasse jeweils von den Lernenden leicht zu verstehen, und der gesamte Algorithmus kann Schritt für Schritt von Klasse zu Klasse entdeckt werden.

5 Eine objekt-orientierte Scanner-Schnittstelle

Ein Parser prüft, ob eine Folge von Symbolen (Token als Kategorien oder Literale) zu der zugrundeliegenden Grammatik paßt. Die Aufgabe eines Scanners (in Zusammenarbeit mit einem Screener) ist es, die Zeichen in der Eingabe in eine Folge von Symbolen zu zerlegen. Darüber hinaus hinterlegt der Scanner optional zu erkannten Symbolen Werte-Objekte. Dieses Kapitel beschreibt die Interfaces und Klassen zur Implementierung eines Scanners inklusive Screener als Partner eines *oops*-Parsers.

5.1 Design der Schnittstelle

Ein Scanner faßt Zeichen zu Wörtern zusammen, und ein Screener klassifiziert diese zu Symbolen. Anschließend wird dem Parser das aktuell erkannte Symbol mitgeteilt — gewöhnlich als Resultat einer Methode/Funktion.

Parsergeneratoren wie *yacc* oder *jay* verwenden für die erzeugten Parser zur Repräsentation der Symbole `int`-Werte, welche vom Scanner als Repräsentation des aktuell erkannten Symbols geliefert werden. Literale bestehen aus einem Zeichen, so daß der ASCII-Wert des Literals direkt als `int`-Wert zur Repräsentation dient. Token sind durch konstante `int`-Variablen mit Werten oberhalb von `256` und damit außerhalb des ASCII-Bereichs repräsentiert. Der Name der Variablen in *yacc* oder *jay* entspricht dabei dem Namen des Token in der Grammatik.

Dieser Ansatz birgt für *oops* Probleme:

Die Sammlung der `int`-Werte muß für den Scanner in einer Klasse oder besser einem Interface geschehen. Dieses Interface ist bei Erzeugung des Parsers mitzugenerieren. *oops* könnte das Interface immer gleich benennen und keinen Paketnamen setzen, was aber die Anwendung unflexibel macht. Sollen der Name und das Paket vom Anwender vorzugeben sein, müßten diese Namen zum Beispiel in zwei System-Properties gesetzt werden.

Weiterhin bestehen Literale in *oops*-Grammatiken aus mehr als einem Zeichen. Damit kann ein Literal nicht für sich selbst als Repräsentation stehen, und auch hier müßte ein berechneter `int`-Wert anhand einer Variablen hinterlegt sein. Da ein Tokenname und der Text eines Literals gleich sein können, kann der Variablenname für ein Literal nicht einfach der Text des Literals sein, zumal dies nicht immer ein legaler Variablenname ist. Für Literale wäre damit neben der Berechnung eines `int`-Werts eine Regel zur Namensfindung nötig.

Pro Baustein eines Parsers (`Some`, `Seq`, ...) dienen `Set`-Objekte zur Berechnung und Repräsentation des lookahead und der follow-Menge. Für Symbole im Parser (`Token` bzw. `Lit`) sind die `Set`-Objekte einelementig und eindeutig. Es liegt daher nahe, diese Objekte direkt anstelle von `int`-Werten zur Identifizierung eines erkannten Symbols zu verwenden.

5.2 Die Scanner-Schnittstelle

Alle Interfaces und Klassen, welche zum Scanner-Komplex eines *oops*-Parsers gehören, sind im Paket `oops.scanner` gesammelt.

In der ersten Version des Scanner-Interfaces als Schnittstelle zwischen dem Parser und der Scanner/Screener-Kombination war eine Methode `tokenSet()` definiert:

```
public Set tokenSet ();
```

Ein Parser fragte über `tokenSet()` den Scanner nach dem `Set`-Objekt des aktuellen Symbols. Damit mußte der Entwickler die Klasse `Set` kennen und hatte ein Wissen über die interne Repräsentation der Objekte im Parser. Er mußte weiterhin den Parser über zwei Methoden (mit einem Tokennamen oder dem Text eines Literals als Argument) nach den `Set`-Instanzen zu einem Symbol fragen und besaß daher auch noch einen Verweis auf das `Parser`-Objekt des Parsers.

Im aktuellen Interface `oops.scanner.Scanner` sind beide Schwachpunkte der ersten Schnittstellen-Version bereinigt:

```
package oops.scanner;
...
public interface Scanner {
  public void scan(SymTab symtab, Hashtable tokens);
  public boolean advance() throws java.io.IOException;
  public boolean atEnd();
  public Object symbol();
}
```

Der Parser ruft die Methoden des Interfaces bei einem `Scanner` auf. Eine `Scanner`-Implementierung stellt die Kombination aus einem Scanner, der Zeichen zu Wörtern zusammenfaßt, und aus einem Screener, welcher Symbole zu den Wörtern erzeugt, dar.

Auf welchen Zeichen der `Scanner` arbeitet, ist nicht Teil des Interfaces. Dies ist bei Konstruktion einer `Scanner`-Instanz zu regeln. Als Beispiel erzeugt das Kommandozeilen-Tool `oops.RunParser` als Konvention aus dem Namen der `Scanner`-Klasse einen Scanner durch Verwendung des Konstruktors mit einer `oops.opi.InputSource`-Instanz als Argument. Auf die Klasse `InputSource` geht das Kapitel 7 (“Die opi-Schnittstelle”) ein.

Der Parser ruft einmal `scan()` beim Scanner zur Initialisierung des Scanners auf.

Bei Aufruf von `advance()` hat der Scanner ein Symbol in der Eingabe vorzurücken, wobei eine Ausnahme in Form einer `IOException` auftreten kann. Die Methode muß `false` am Dateiende, sonst `true`, liefern.

`atEnd()` zeigt als `boolean` an, ob das Dateiende erreicht worden ist.

Das Resultat von `symbol()` ist ein Objekt, welches das momentane Symbol repräsentiert. Der Entwickler eines Scanners will und soll die genaue Klasse des Objekts nicht kennen; er liefert es einfach. Der Scanner erhält die Resultat-Objekte für `symbol()` über die Argumente `symtab` und `tokens` zu `scan()`, deren Schlüssel die Literale- und Tokennamen als `String`-Objekte sind.

`symtab` ist ein Objekt einer Klasse, welche das Interface `oops.scanner.SymTab` implementiert. Um zu erklären, wie die Identifizierungs-Objekte für `symbol()` in `symtab` und `tokens` hinterlegt worden sind, wird hier zunächst die Idee hinter dem Interface `SymTab` erläutert.

5.3 oops.scanner.SymTab, oops.scanner.SymTab.Handle

Screener agieren zumeist auf einer Symboltabelle. Als Beispiel verwendet ein Screener einer Programmiersprache eine Symboltabelle zur Verwaltung der Schlüsselwörter und zur Verwaltung von Werte-Objekten zu Variablennamen.

Eine Instanz einer Implementierung von `SymTab` dient als Symboltabelle. In ihr kann unter einem String als Schlüssel nach einem Eintrag gesucht werden:

```
package oops.scanner;
...
public interface SymTab extends Serializable {
  public Handle get(String key);
```

```
    public java.util.Set keySet();
    ...
  }
```

Die Methode `keySet()` liefert alle eingetragenen Schlüssel in Form eines Containers, und `get()` liefert als Ergebnis einer Anfrage ein Objekt des inneren Interface `SymTab.Handle`:

```
public interface SymTab extends Serializable {
  ...
  public interface Handle {
    public boolean isEntered();
    public void enter(Object symbol, Object value);
    public void remove();
    public Object symbol();
    public Object value();
  }
}
```

Ein `Handle` kann über die Methode `enter(Object,Object)` zwei Objekte zu einem Schlüssel in die Tabelle eintragen. `value` stellt zu dem Schlüssel ein mögliches Werte- und `symbol` ein mögliches Identifizierungs-Objekt dar. Beide dürfen `null` sein.

Von den mit `enter()` eingetragenen Objekten liefert `symbol()` das Identifizierungs- und `value()` das Werte-Objekt.

`isEntered()` erfragt als `boolean`, ob zu dem Schlüssel bereits mit `enter()` Objekte hinterlegt worden sind bzw. ob mit `remove()` der Eintrag zum Schlüssel noch nicht gelöscht worden ist.

Ein Scanner im Compilerbau faßt Zeichen zu Wörtern zusammen, und die Aufgabe eines Screeners ist es, zu einem Wort das zugehörige Symbol- und Werte-Objekt zu finden bzw. zu erzeugen. Bei der Implementierung der Scanner-Schnittstelle dient eine `SymTab`-Instanz als Symboltabelle als Werkzeug des Screener-Programmcodes. Die Methode `get(String)` liefert zu einem Wort als String-Schlüssel ein `Handle`-Objekt (`handle`). Der Screener untersucht, ob bereits Informationen vorhanden (`handle.isEntered()`) sind oder ob diese in der Tabelle zu hinterlegen sind (`handle.enter()`). Ist bereits ein Symbol- oder auch ein Werte-Objekt zu einem Symbol vorhanden, werden diese erfragt (`handle.symbol()` und `handle.value()`) und im weiteren Verlauf benutzt.

Dabei ist wichtig, daß die Symboltabelle für einen Schlüssel immer das identische `Handle`-Objekt liefert. Verschiedene Phasen eines Compilers (lexikalische, syntaktische und semantische Analyse oder die Code-Generierung) werten das Resultat von `get()` als `Handle` aus und ändern nicht den Verweis selbst, sondern die in dem Handle hinterlegten Informationen.

Alle diese Operationen sollen nach Möglichkeit mit nur einem Lookup in der Tabelle auskommen. Die Erzeugung des `Handle`-Objekts zu einem Schlüssel stellt bereits eine Suche in der Tabelle dar, und daher sollten `Handle`-Instanzen als *Closure* ([Ben99]) arbeiten. Closure-Objekte besitzen Kenntnis über ihre “Umgebung” und wissen, wie sie auf bzw. mit ihrer Umgebung arbeiten. Als Beispiel sollte ein `Handle` wissen, wo es in der Tabelle ohne eine erneute Suche die Informationen findet, ablegt bzw. den Eintrag zu einem Schlüssel löscht. Damit stellt nur `get()` eine Suche in der Tabelle dar.

Ein Handle setzt sich aus dem Identifizierungs-Objekt und einem weiteren Objekt zusammen. Das weitere Objekt muß alles enthalten, was der Anwender vom `Handle` an Informationen bekommen möchte. Für eine reine Parsierung wäre das zweite Objekt nicht nötig. Anstelle einer Komposition von `Handle` aus zwei Objekten wäre ein Vererbungsansatz eine mögliche Alternative:

Für eine Parsierung würde es reichen, wenn eine `SymTab` ein `Handle` liefert, welches lediglich nach dem Identifizierungs-Objekt gefragt werden kann:

```
public interface SymTab extends Serializable {
    public Handle get(String key);
    public interface Handle {
        public void enter(Object symbol);
        public Object symbol();
        ...
    }
}
```

Ein `Scanner` würde mit einer Instanz dieses `SymTab`-Interfaces initialisiert:

```
public interface Scanner {
    public void scan(SymTab symtab, Hashtable tokens);
    ...
}
```

Für sinnvolle Benutzer-Aktionen zu erkannten Teilen einer Grammatik hat der Scanner dann optional zu Symbolen auch ein Werte-Objekt zu liefern. Man könnte dies im Gegensatz zur jetzigen Lösung auch durch Vererbung modellieren:

```
public interface ValueSymTab extends SymTab {
    // geerbt: public Handle get(String key);
    // illegal: public get(String key);
    public interface ValueHandle extends SymTab.Handle {
        public void enter(Object symbol, Object value);
        public Object value();
    }
}

public interface ValueScanner extends Scanner {
    // geerbt: public void scan(SymTab symtab, Hashtable tokens);
    // public void scan(ValueSymTab symtab, Hashtable tokens);
    public Object value();
}
```

In `ValueSymTab` müßte `get(String)` ein `Handle` liefern, da Overloading nur für die Parameter, aber nicht für den Resultattyp erlaubt ist. Damit müßte der Anwender das Resultat von `get()` für ein `ValueSymTab` selbständig immer nach `ValueHandle` wandeln.

Adaptiert eine Klasse das Interface `ValueScanner`, muß es auch `scan(SymTab,...)` als Bestandteil von `Scanner` implementieren. Daher macht eine zweite Methode `scan(ValueSymTab,...)` keinen Sinn, obwohl diese bezüglich des Typs des ersten Parameters besser wäre. In `scan(SymTab,...)` muß dann aber wieder selbständig das erste Argument nach `ValueSymTab` gewandelt werden.

Beide Umwandlungen sind Absprachen, die der Anwender kennen muß und die nicht in Form von Klassennamen im Programmtext fixiert sind. Außerdem möchte der Anwender möglicherweise bereits während der Parsierung triviale Dinge, wie das Abzählen der Symbole in einer Eingabe über die `value`-Einträge der `Handle`, machen. Das Feld kann bereits hier sehr praktisch sein. Daher habe ich mich für die Komposition der beiden Objekte in einem `Handle` entschieden.

5.4 oops.scanner.DefaultSymTab

Die Klasse `oops.scanner.DefaultSymTab` stellt eine Implementierung von `SymTab` zur Verfügung. Sie verwendet intern eine `java.util.Hashtable` als Tabelle und hinterlegt ein `Object`-Array als Wert zum Schlüssel. In `get()` wird eine Instanz einer anonymen `Handle`-Klasse mit einem Verweis auf dieses Array erzeugt und zurückgeliefert. Das `Handle` greift als Closure auf die Elemente des Arrays zu, was eine erneute Suche in der Hashtabelle für alle Operationen des `Handle` vermeidet.

Das Array besteht aus drei Elementen: `value`, `symbol` und einem `Boolean`-Objekt. Die `Boolean`-Instanz markiert, ob der Eintrag zu diesem Schlüssel als eingetragen in die Symboltabelle gilt.

Die Klasse besitzt ein einfaches Hauptprogramm. `g` key erfragt per `get()` ein Handle zum `key`, `s` erfragt und druckt das `symbol`-Objekt zum `Handle` vom letzten `get()`-Aufruf, `v` druckt das `value`-Objekt, `p` ruft für `Handle`-Instanz `toString()` auf, `e symbol value` trägt Texte ein, `r` ruft `remove()` und `?` ruft `isEntered()` beim `Handle` auf:

```
$ export CLASSPATH=$HOME/Promotion/jars/oops.jar
$ java oops.scanner.DefaultSymTab
g var1
Handle[ var1,false,null,null]
s
symbol null
v
value null
?
isEntered false
e symbol value
p
Handle[ var1,true,symbol,value]
g var2
Handle[ var2,false,null,null]
g var1
Handle[ var1,true,symbol,value]
?
isEntered true
r
p
Handle[ var1,false,symbol,value]
?
isEntered false
```

5.5 symtab und tokens

Ein Screener schaut für alle Wörter der Eingabe in der Symboltabelle nach Informationen und unterscheidet so reservierte Wörter von anderen. Daher ist bei Aufruf von `scan()` die Tabelle `symtab` mit den reservierten Wörtern vorgeladen worden. Für jedes Literal der Grammatik (die reservierten Wörter sind eine Teilmenge der Literale) ist unter dem Text des Literals als Schlüssel `null` als Werte- und ein für `symbol()` passendes Identifizierungs-Objekt hinterlegt.

In `tokens` ist für alle Tokennamen als Schlüssel jeweils ein zugehöriges Objekt für `symbol()` zur Repräsentation des Token abgelegt worden.

Bei diesem Ansatz besitzt ein Scanner keinen Verweis in den Parser hinein und weiß auch nichts über die interne Repräsentation der Identifizierungs-Objekte. Dies sind natürlich weiterhin `Set`-Objekte.

Die Objekte für Literale und Token sind in zwei verschiedenen Tabellen gespeichert, da der Name eines Token und der Text eines Literals gleich sein können. Wären die Informationen in einer Tabelle hinterlegt, käme es in diesem Fall zu einem Konflikt.

Die Token sind per default nicht in der `SymTab`-Tabelle, da zum Beispiel für ein Token mit dem Namen `identifier` als Symbol für Variablen der Name `identifier` durchaus als legaler Variablenname auftreten kann. In `symtab` wäre in diesem Fall ein Werte-Objekt und das zugehörige Identifizierungs-Objekt zum Variablennamen als Schlüssel zu hinterlegen.

Die Literale sind in der `SymTab`-Tabelle, da Schlüsselwörter normalerweise nicht als Variablennamen auftreten. Per `remove()` gegen ein `Handle` können diese aber auch jederzeit aus der Tabelle gelöscht werden.

Die Verwendung der Symboltabelle und die der Token-Tabelle sind ein kritisch wichtiger Teil eines Compilers. Daher sollte nicht nur die Suche in der Tabelle durch den Einsatz von Closure-Instanzen optimiert werden. Scanner sollten aus Effizienzgründen die Objekte zur Repräsentierung von Symbolen einmalig von den beiden Tabellen beschaffen und danach nur noch über direkte Verweise verwenden.

5.6 Ein Beispiel

Ein Beispiel erläutert die Interfaces `Scanner` und `SymTab` noch einmal. Die Klasse `ArithScanner` dient als Scanner für einen Parser für eine (mehrdeutige) Grammatik, die arithmetische Ausdrücke mit Variablen beschreibt:

```
expr    : {[ [ sum ] ";" ]} ;
sum     : product [{ ( "+" | "-" ) product }] ;
product : term [{ ( "*" | "/" ) term }] ;
term    : NUMBER | ID [ "=" sum ] | ( "+" | "-") term | "(" sum ")" ;
```

`ArithScanner` ist mit *lolo* implementiert und adaptiert das `Scanner`-Interface:

```
public class ArithScanner implements oops.scanner.Scanner {
    protected Object NUMBER, ID;
    protected SymTab symtab;

    public void scan(SymTab symtab, Hashtable tokens) {
        this.symtab = symtab;
        NUMBER = tokens.get("NUMBER");
        ID = tokens.get("ID");
    }
```

Die Objekte, welche die Token `ID` und `NUMBER` identifizieren, werden aus Effizienzgründen nur einmal in `tokens` gesucht und an Instanzvariablen gebunden. In `symtab` wird ein Verweis auf die Symboltabelle hinterlegt.

```
    protected Object symbol;

    protected final Scanner scanner = new Scanner(new Scan[] {
        new Flt().setAction(new Action() {
            public void action(Scan sender, char[] buf, int off, int len) {
                symbol = NUMBER;
            }
        }),
        new SimpleIdentifier().setAction(new Action() {
            public void action(Scan sender, char[] buf, int off, int len) {
                SymTab.Handle handle = symtab.get(new String(buf, off, len));
                if (handle.isEntered()) // use old information
                    symbol = handle.symbol();
                else {                  // enter new identifier
                    handle.enter(symbol = ID, null);
                    System.err.println("new var "+new String(buf, off, len));
                }
            }
        }),
        new SimpleWhitespace(),
```

```
            new Set("()+-/*=;").setAction(new Action() {
                public void action(Scan sender, char[] buf, int off, int len) {
                    symbol = symtab.get(""+buf[off]).symbol();
                }
            }),
        });
```

Der *lolo*-Scanner erzeugt den Raum mit vier Erkenner-Objekten: eines zum Erkennen von Zahlen, eines für Wörter, eines zum Ignorieren von Zwischenraum und eines zum Finden einzelner Zeichen. Der Erkenner für Zahlen und der für einzelne Zeichen hinterlegen als Aktion in der Instanzvariablen `symbol` lediglich das richtige Identifizierungs-Objekt. In der Aktion zu erkannten Wörtern wird ein `Handle` für das Wort bei der `SymTab` angefordert. Ist bereits Information zu dem Schlüssel in der Symboltabelle hinterlegt, wird diese verwendet. Ansonsten handelt es sich bei dem Wort um einen neuen Variablennamen, und dieser wird in die Tabelle mit dem richtigen Identifizierungs-Objekt eingetragen.

```
        protected final lolo.Input input;

        public ArithScanner(InputSource is) {
            input = new lolo.Input(is.getReader(), 1);
        }
```

Ein `ArithScanner` wird mit einer `InputSource` als Datenquelle erzeugt. Eine `InputSource` repräsentiert eine Datenquelle und wird im Kapitel 7 (“Die opi-Schnittstelle”) erläutert. Hier wird die `InputSource` nach einem `java.io.Reader` für die Daten gefragt, mit dem ein `lolo.Input` erzeugt wird.

Damit steht das Gerüst, und die Methoden des `Scanner`-Interfaces stützen sich auf den *lolo*-Scanner und dessen Aktionen ab:

```
        protected boolean atEnd;

        public boolean advance() throws IOException {
            if (atEnd) return false;
            try {
                return scanner.scan(input) == null ? !(atEnd = true) : true;
            } catch (lolo.Scanner.IllegalCharacterException e) {
                throw new RuntimeException(e.toString());
            }
        }
        public boolean atEnd() { return atEnd; }
        public Object symbol() { return symbol; }
    }
```

5.7 oops.scanner.TableFactory, oops.scanner.DefaultTableFactory

Die genauen Klassen der beiden Tabellen zu `scan()` sollten vom Anwender frei wählbar sein. Daher erzeugt eine `TableFactory` die beiden Tabellen:

```
package oops.scanner;

public interface TableFactory {
  public SymTab symbolTable();
  public java.util.Hashtable tokenTable();
}
```

Ein Parser aus Klassen des `oops.parser`-Pakets wird durch Aufruf der Methode

```
public Object parse(Scanner scanner, TableFactory tf) throws
    ParseException, IOException
```

bei dem einen `oops.parser.Parser`-Objekt des Parsers ausgeführt. Der Anwender gibt als zweites Argument eine Instanz einer beliebigen `TableFactory` an. Ist das Argument `null`, wird eine Instanz der Klasse `DefaultTableFactory` verwendet, welche als Symboltabelle eine Instanz der Klasse `DefaultSymTab` und als Token-Tabelle eine `java.util.Hashtable` erzeugt. Alternativ hätte die Methode die beiden Tabellen auch als Parameter akzeptieren können:

```
public Object parse(Scanner scanner, SymTab s, Hashtable t) throws ...
```

Die `TableFactory` macht aber zum Beispiel die Auswahl der Tabellen für die Kommandozeile einfach. Die Klasse `oops.RunParser` führt einen serialisierten Parser von der Kommandozeile aus:

```
$ java oops.RunParser
usage:
  java oops.RunParser [ -t TableFactory] parser.ser scannerClass [ <] source
```

Durch die Option `-t` kann der Anwender optional den Namen der `TableFactory`-Klasse auswählen:

```
$ JARS=$HOME/p/jars
$ export CLASSPATH=$JARS/oops.jar:$JARS/lolo.jar:.
$ flags="-t oops.scanner.DefaultTableFactory"
$ java oops.RunParser $flags arith.ser ArithScanner
TableFactory is oops.scanner.DefaultTableFactory
a = 2 + 3;
new var a
b = a * 2;
new var b
b * a;
^D
$
```

5.8 Fazit

Die Verwendung von einfachen `int`-Werten ist bei Literalen mit mehr als einem Zeichen umständlich. Auch für Literale müßten `int`-Werte berechnet und in einer Variablen hinterlegt werden. Der Name der Variablen kann nicht der Text des Literals sein. Aus diesem Grund sind in anderen Compilergeneratoren wie zum Beispiel *SableCC* oder praktisch auch *ANTLR* keine Literale als Teil der Grammatik erlaubt. Alle Symbole müssen dort Token sein.

Die `int`-Variablen müßten als Klassenvariablen einer Klasse oder eines Interfaces bekannt sein. Der Name der Klasse bzw. des Interfaces und auch der Paketname sollten aber von außen beeinflußbar sein.

Da im *oops*-Parser intern Token und Literale durch `Set`-Objekte bereits eindeutig repräsentiert werden, bietet sich die Verwendung dieser Objekte zur Identifizierung von Symbolen zwischen der lexikalischen Analyse und dem Parser ganz natürlich an, und die Berechnung von `int`-Werten wie in *yacc* erübrigt sich damit.

Nachteil dieses Ansatzes ist, daß selbst für einfache Literale das Identifizierungs-Objekt in der Symboltabelle gesucht werden muß. Für häufig auftretende Literale kann aber zur Effizienzsteigerung ein Verweis auf das Identifizierungs-Objekt einmalig erfragt und in einem konstanten Verweis hinterlegt werden.

Der Entwickler einer `Scanner`-Implementierung sollte kein unnötiges Wissen über den Parser bzw. über die Repräsentation von Objekten im Parser besitzen bzw. zu verwenden haben (*information*

hiding). Eine einfache `Object`-Schnittstelle für `symbol()` und bereits gefüllte Tabellen lösen dieses Problem.

Beim Design der Symboltabelle vom Typ `SymTab` wurde auf nur eine Suche in der Tabelle pro Schlüssel für mehrere Operationen geachtet. `Handle`-Instanzen sollten deshalb als Closure arbeiten und dies dadurch gewährleisten.

Die Verwendung einer eigenen Hashtabelle `tokens` zur Abbildung von Tokennamen zu `Set`-Instanzen hält die Symboltabelle `symtab` anfänglich klein und von den Tokennamen als Schlüssel sauber.

Durch den Einsatz eines Factory-Musters für die beiden Tabellen können eigene Implementierungen von `SymTab` bzw. Unterklassen von `Hashtable` als Tabellen verwendet werden. Der Einsatz dieses Entwurfsmusters macht die Schnittstelle, wie in der These vorhergesagt, flexibel und für den Nutzer frei wählbar.

6 Parser-Aktionen

Ein Baum von Objekten der in Kapitel 4 (“Ein Parser aus Objekten”) vorgestellten Klassen repräsentiert Grammatiken und ist in der Lage, die repräsentierten Grammatiken zu prüfen und Programme über der Grammatik zu erkennen. Allerdings ist die Parsierung ohne vom Benutzer zu spezifizierende Aktionen zu erkannten Teilen der Grammatik von eher akademischem Wert.

Dieses Kapitel zeigt, wie die Klassen des Parser-Baums dank der Zerlegung in Klassen und dank der Vererbung leicht um unterschiedlichste Arten von Aktionen zu erweitern sind. Drei Arten von Aktionen sind implementiert, und weitere mögliche Arten von Aktions-Mustern werden diskutiert.

6.1 Die Idee

Ein oops-Parser kann als *black box* angesehen werden. Er enthält Algorithmen zur Prüfung der repräsentierten Grammatik und zur syntaktischen Erkennung von Programmen über der Grammatik. Soll der Parser nun nicht nur parsieren, sondern auch Aktionen zu erkannten Teilen der Grammatik ausführen, kann der Parser als eine Art Kontrollstruktur bei der Erkennung von Programmen angesehen werden:

Der Parser ruft während der Parsierung Nachrichten bei gewissen Knoten des Parser-Baums auf, um das Erkennen von Symbolen oder die komplette Erkennung von Regeln mitzuteilen. Diese Methoden sind in den bereits eingeführten Klassen leer vorimplementiert. Als Beispiel wird die Methode `shift()` bei einem `Lit` aufgerufen, wenn das `Lit` das repräsentierte Literal erkennt.

Unterklassen können diese Methoden nun mit Programmtext für Aktionen zu erkannten Teilen der Grammatik überschreiben (Template-Entwurfsmuster). Werden Instanzen der neuen Klassen anstelle von Instanzen der Oberklasse in den Baum integriert, werden dadurch mögliche *oops*-Parser um Aktionen erweitert. Da die Unterklassen die Instanzvariablen und Methoden der Oberklassen erben, kann ein Baum mit Instanzen der neuen Klassen weiterhin die repräsentierte Grammatik prüfen.

Verschiedenste Arten von Aktionsmustern sind denkbar: Eine erste und einfache Aktionsidee ist die Ausgabe einer Ablaufverfolgung während der Parsierung. Weitergehende Aktionsmuster könnten pro aktiver Regel ein Aktions-Objekt erzeugen, dessen Klasse vom Anwender implementiert wird, und teilen diesem über Methoden den Fortschritt in der Parsierung der zugehörigen Regel mit. In den Methoden sind die Aktionen vom Anwender dann frei zu programmieren. Als weiteres Beispiel könnten in Anlehnung an die Oberflächen-Programmierung in Java pro Regel Objekte nach dem Listener-Entwurfsmuster eingetragen werden. Die Listener werden durch Events über das Fortschreiten der Parsierung informiert und implementieren in ihrer Eventbehandlung die gewünschten Aktionen.

Diese und andere Ideen für Aktionsmuster werden im Folgenden im Detail vorgestellt.

6.2 Ablaufverfolgung als Aktion

6.2.1 Idee

Um die Idee bzw. die Technik der Erweiterung der Klassen eines Parser-Baums zu testen, wurden in einem ersten Test Unterklassen implementiert, welche als Aktionen eine Ablaufverfolgung während der Parsierung ausgeben.

Dazu sind im Paket `oops.parser.trace` die Klassen `Parser`, `Rule`, `Lit` und `Token` als Unterklassen der gleichnamigen Klassen aus dem Paket `oops.parser` gesammelt.

Für eine Grammatik wird ein Baum aus Instanzen der neuen Klassen und aus Instanzen der nicht abgeleiteten Klassen gebaut. Zum Beispiel sieht der Parser-Baum für die bereits bekannte Grammatik

```
sum     : product [{  ( "+" | "-" ) product }] ;
product : term [{  ( "*" | "/" ) term }] ;
term    : NUMBER | ( "+" | "-") term | "(" sum ")" ;
```

für arithmetische Ausdrücke mit Instanzen der neuen und alten, nicht erweiterten Klassen damit so aus:

```
oops.parser.trace.Parser
   oops.parser.trace.Rule sum
      oops.parser.Seq
         oops.parser.Id product
         oops.parser.Many
            oops.parser.Seq
               oops.parser.Alt
                  oops.parser.trace.Lit "+"
                  oops.parser.trace.Lit "-"
               oops.parser.Id product
   oops.parser.trace.Rule product
      oops.parser.Seq
         oops.parser.Id term
         oops.parser.Many
            oops.parser.Seq
               oops.parser.Alt
                  oops.parser.trace.Lit "*"
                  oops.parser.trace.Lit "/"
               oops.parser.Id term
   oops.parser.trace.Rule term
      oops.parser.Alt
         oops.parser.Token NUMBER
         oops.parser.Seq
            oops.parser.Alt
               oops.parser.trace.Lit "+"
               oops.parser.trace.Lit "-"
            oops.parser.Id term
         oops.parser.Seq
            oops.parser.trace.Lit "("
            oops.parser.Id sum
            oops.parser.trace.Lit ")"
```

Alle `Parser`, `Rule`, `Lit` und `Token` des Baums sind Instanzen aus dem neuen Trace-Paket. Wird dieser Baum — hier serialisiert in der Datei `arith.ser` — zur Ausführung gebracht, gibt er eine Ablaufverfolgung der Parsierung aus:

```
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=.:$JARS/oops.jar:$JARS/lolo.jar
$ java oops.RunParser arith.ser ArithScanner
2+
   Rule sum start, rule level 3
      Rule product start, rule level 6
         Rule term start, rule level 9
         Token accept a NUMBER
         Rule term end, rule level 9
      Rule product end, rule level 6
   Lit accept +
3
      Rule product start, rule level 6
         Rule term start, rule level 9
         Token accept a NUMBER
```

```
^D
          Rule term end, rule level 9
        Rule product end, rule level 6
    Rule sum end, rule level 3
$
```

Der Scanner ignoriert allen Zwischenraum. Pro Aktivierung einer Unterregel wird der Text der Ablaufverfolgung tiefer eingerückt, und im Text wird die Zahl der benutzten Leerzeichen zum Einrücken als Ebene der Einrücktiefe ausgegeben. Eine `Rule` zeigt den Start und das Ende ihrer Aktivierung an und signalisiert damit, daß die repräsentierte Regel erkannt werden soll bzw. erkannt wurde. Instanzen der Klassen `Lit` und `Token` geben das akzeptierte, repräsentierte Symbol aus.

6.2.2 Implementierung

Die Erweiterung des Parsers um die Ablaufverfolgung ist dank der Zerlegung des Parsers in Klassen und durch die vorbereiteten, zu überschreibenden Template-Methoden recht einfach. Lediglich die vier bereits erwähnten Klassen sind zu erweitern. Da die neuen Klassen die Instanzvariablen und Methoden der jeweiligen Oberklasse erben, beschränkt sich der Programmtext der neuen Klassen auf das Implementieren der Aktionen.

Während der Parsierung in `parse()` ruft ein `Lit` für jedes akzeptierte Literal eine Methode `shift()` mit zwei Argumenten auf (siehe Kapitel 4.4 (“Parsierung”)). In `oops.parser.Lit` war der Körper der Template-Methoden noch leer implementiert. Die Ablaufverfolgung wird nun in der Unterklasse `oops.parser.trace.Lit` ausgegeben:

```
package oops.parser.trace;

public class Lit extends oops.parser.Lit {
    public Lit(String body) {  super (body); }

    protected void shift(Object action, oops.parser.Parser parser) {
        Rule.printIndent((Integer) action);
        System.err.println("Lit accepts "+body);
    }
}
```

Die `parse()`-Methoden der einzelnen Objekte im Baum rufen sich während der Parsierung rekursiv auf:

```
public void parse(Parser parser, Activation caller, Object action) throws ...
```

Dabei wird der Verweis des Arguments `action` immer an rekursive Aktivierungen weitergegeben, und `parse()` aus `Lit` gibt den Wert auch als erstes Argument zu `shift()` weiter.

Der `action`-Verweis ist die zweite, vorbereitete Erweiterungsmöglichkeit zum Einbau von Aktionen, da die verschiedenen Objekte des Baums über `action` ein Objekt in die rekursive Parsierung mitgeben können. Hier wird `action` als `Integer`-Objekte der aktuellen Einrücktiefe im Baum verteilt. Eine Klassenmethode `printIndent()` in `Rule` akzeptiert ein `Integer` als Argument und gibt eine dem Wert des Arguments entsprechende Anzahl Leerzeichen aus.

`oops.parser.trace.Token` stammt von `oops.parser.Token` ab und ersetzt analog zu `Lit` die `shift()`-Methode durch den Programmtext der Ablaufverfolgung für ein erkanntes Token-Symbol.

Die Unterklasse `Rule` stellt, wie bereits erwähnt, allen Klassen des Pakets eine Methode `printIndent()` zum Ausgeben der Einrücktiefe zur Verfügung:

```
public class Rule extends oops.parser.Rule {
    public Rule(Id nt, Node rhs) { super(nt, rhs); }
    protected static void printIndent(Integer level) { ... }
```

Zwei weitere Methoden geben die Ablaufverfolgung für die Aktivierung bzw. für die komplette Parsierung einer Regel aus. Pro neuer Aktivierung wird die Einrücktiefe vergrößert:

```
    protected static final int INCR = 3;

    protected Integer startRule(Integer oldLevel) {
        Integer newlevel = new Integer(oldLevel.intValue()+INCR);
        printIndent(newlevel);
        System.err.println("Rule "+getNt()+" start, rule level "+newlevel);
        return newlevel;
    }

    protected void endRule(Integer newlevel) {
        printIndent(newlevel);
        System.err.println("Rule "+getNt()+" end, rule level "+newlevel);
    }
```

Die neue `Rule`-Klasse überschreibt nun einfach die `parse()`-Methode der Oberklasse und ruft die beiden Methoden zur Ausgabe der Ablaufverfolgung auf. Die Parsierung wird nicht neu implementiert, sondern an die Methode der Oberklasse delegiert:

```
    public void parse(oops.parser.Parser parser, Activation caller,
        Object oldLevel) throws ParseException, IOException, Activation {
        Integer newlevel = startRule((Integer) oldLevel);
        super.parse(parser, caller, newlevel);
        endRule(newlevel);
    }
}
```

Als letztes muß lediglich die `Parser`-Klasse noch erweitert werden. Wie bereits erklärt, kann erst für einen kompletten Baum entschieden werden, welche Namen auf den rechten Seiten der Regeln einer Grammatik Symbole und welche Verweise auf Regeln darstellen. Daher werden zu Beginn für Namen immer `Id`-Instanzen in den Baum eingefügt, und während der Grammatikprüfung werden für Symbole `Token`-Objekte erzeugt. Dieses geschieht im Parser durch die Methode `createToken()`. Die neue `Parser`-Klasse überschreibt die Methode und erzeugt Instanzen der neuen `Token`-Klasse des Pakets `oops.parser.trace`:

```
public class Parser extends oops.parser.Parser {
    ...
    protected oops.parser.Token createToken(String id, Set set) {
        return new Token(id, set);
    }
    ...
```

Als letztes wird `parse(Activation)` überschrieben und mit einem `Integer` als `action`-Objekt gestartet.

```
    protected void parse(Activation caller) throws Activation,
        ParseException, IOException {
        start.parse(this, caller, new Integer(0));
    }
}
```

6.2.3 Fazit

Die Klassen des `oops.parser`-Pakets sind lediglich zum Parsieren eines Programms über der Grammatik in der Lage. Allerdings stellen die Klassen vorbereitete Möglichkeiten zur Erweiterung zur Verfügung:

An kritischen Punkten werden von den Algorithmen Template-Methoden aktiviert. So wird `shift()` in `Lit` und `Token` bei jedem erkannten Symbol und `createToken()` in `Parser` zum Erzeugen der `Token`-Instanzen aufgerufen. Unterklassen können diese Methode überschreiben und so in die Algorithmen eingreifen bzw. die Algorithmen erweitern.

In dem rekursiven Aufruf von `parse()` wird ein `Object`-Verweis `action` weitergegeben. Dieses Objekt kann in beliebiger Weise für Aktionen genutzt werden. Hier wird damit die aktuelle Einrücktiefe in die rekursive Aktivierung durchgereicht. Wie der nächste Abschnitt aber zeigt, kann dieser Objektverweis auch für viel mächtigere Aktionsideen genutzt werden.

Die Unterklassen erben die Methoden ihrer Oberklassen. Damit erben sie die durch die Methode implementierten Algorithmen. Die vier Klassen des neuen Pakets sind also sofort zur Grammatikprüfung in der Lage und integrieren sich ganz harmonisch in die Parser-Bäume.

6.3 Aktionen durch Goal-Instanzen

6.3.1 Idee, Goal-Interface

Pro Aktivierung einer Regel existiert ein zugehöriges `Goal`-Objekt. Während die Parsierung voranschreitet, werden dem `Goal` der Regel verschiedene Nachrichten bezüglich erkannter Teile der Grammatik gesendet:

```
package oops.parser.goal;

public interface Goal {
    public void shift(Goal sender, Object value);
    public void shift(Token sender, Object value);
    public void shift(Lit sender, Object value);
    public Object reduce ();
    public void error();
}
```

Die Methode `shift(Lit,Object)` wird beim `Goal` aufgerufen, wenn zitierter Text (ein Literal) auf der rechten Seite der Regel erkannt wird. Der Absender der Methode `sender` ist das Objekt im Parser-Baum, welches das Symbol gerade erkannt hat, und kann mit `getBody()` nach dem erkannten Text gefragt werden. `value` ist das zugehörige Werte-Objekt des Symbols vom Scanner.

`shift(Token,Object)` wird analog für erkannte Token-Symbole aufgerufen. `sender` ist als zugehöriges `Token`-Objekt des Baums der Absender der Methode und kann mit `getName()` nach dem Namen des Token befragt werden. `value` ist wieder das vom Scanner gelieferte Werte-Objekt.

Ist eine Regel komplett erkannt, wird beim zur Regel gehörenden `Goal` die Methode `reduce()` aufgerufen. `reduce()` hat als Wert ein Objekt zu liefern, welches das Resultat der erkannten Regel darstellt.

Die Methode `shift(Goal,Object)` wird aufgerufen, wenn auf der rechten Seite einer Regel ein Verweis auf eine andere Regel abgearbeitet ist, d.h., wenn die Unterregel komplett erkannt worden ist. In diesem Fall wurde dem `Goal` der Unterregel `reduce()` geschickt. Das Ergebnis von `reduce()` für die Unterregel ist das zweite Argument zu `shift()` und die `Goal`-Instanz der Unterregel selbst das erste Argument.

Tritt während der Parsierung der rechten Seite der zu einem `Goal` gehörigen Regel ein Syntaxfehler auf, wird `error()` beim `Goal` aufgerufen.

Abbildung 6.1 stellt grafisch die Methodenaufrufe exemplarisch dar:

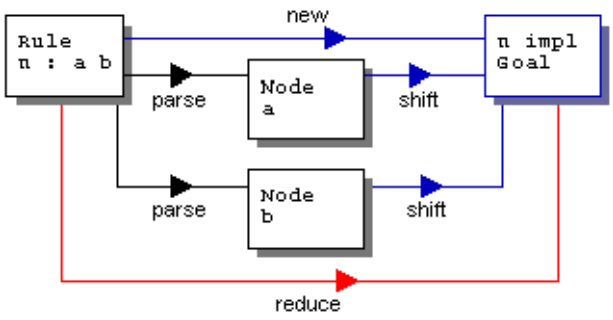

Abbildung 6.1: Methodenaufrufe bei einer Goal-Instanz.

In *yacc* sammelt ein Werte-Stack die im Scanner gescannten und vom Anwender für Regeln berechneten Werte. Am Ende einer Regel kann über eine `$i`-Syntax auf Elemente des Stacks zugegriffen werden, wird eine der Anzahl der Elemente der erkannten rechten Seite der Regel entsprechende Anzahl Elemente vom Stack genommen und wird ein neues, möglicherweise vom Anwender berechnetes Element wieder auf den Stack plaziert. Ein `Goal` kann ganz analog ein lokaler Stack für die repräsentierte Regel sein, da eine Implementierung von `Goal` die Information der `shift()`-Methoden speichern und bei `reduce()` verarbeiten kann. Analog kann ein `Goal` die bei `shift()` übergebenen Informationen aber auch sofort bewerten.

Aktionen zu erkannten Teilen einer Grammatik machen erst dann ernsthaft Sinn, wenn der Scanner zu Symbolen assoziierte Werte-Objekte erzeugt. Daher ist das `Scanner`-Interface um eine Methode erweitert:

```
package oops.scanner;

public interface ValueScanner extends Scanner {
    public Object value();
}
```

Während der Parsierung wird ein `ValueScanner` durch `symbol()` (als Bestandteil des Interface `Scanner`) nach dem aktuellen Symbol und parallel dazu durch `value()` nach einem zugehörigen Werte-Objekt gefragt.

6.3.2 Adapter-Klassen

Die Klassen zu `Goal`-Aktionen sind im Paket `oops.parser.goal` gesammelt. Das Paket beinhaltet drei Standardimplementierungen des `Goal`-Interface:

Ein `GoalAdapter` merkt sich das erste nicht `null` Werte-Objekt der `shift()`-Methoden in der Instanzvariablen `result`. In der Boolschen Instanzvariablen `error` markiert ein `GoalAdapter`, ob `error()` aufgerufen worden ist. Ist `error` wahr, liefert `reduce()` als Resultat `null`, sonst den Wert von `result`, welcher natürlich auch `null` sein kann.

`GoalDebugger` stammt von `GoalAdapter` ab und verhält sich daher analog. Zusätzlich gibt ein `GoalDebugger` für jede aufgerufene Methode einen Text als Ablaufverfolgung aus.

Die Klasse `NopGoal` adaptiert auch das `Goal`-Interface und implementiert alle Methoden mit einem leeren Körper. Lediglich `reduce()` liefert `null` als Resultat.

6.3.3 GoalMakerFactory, GoalMaker

Je aktive Regel existiert eine `Goal`-Instanz, bei der bei Fortschreiten der Parsierung auf der rechten Seite der Regel die Methoden des Interfaces aufgerufen werden.

Die Erzeugung der `Goal`-Instanzen geschieht über ein zweischichtiges Factory-Muster. Eine `GoalMakerFactory` wird bei Start des Parsers pro Regel nach einer `GoalMaker`-Instanz gefragt:

```
public interface GoalMakerFactory {
    public GoalMaker goalMaker (String ruleName);
}
```

Die `Rule` zu einer Regel fragt dann bei jeder neuen Aktivierung ihren einen `GoalMaker` nach einer `Goal`-Instanz als Empfänger der `Goal`-Nachrichten:

```
public interface GoalMaker {
    public Goal goal ();
}
```

Normalerweise möchte ein Anwender von *oops* keine eigene Implementierung von `GoalMakerFactory` schreiben. Daher enthält das Paket fertige Implementierungen des Interfaces:

Eine `DefaultGoalMakerFactory` erzeugt einen `GoalMaker`, der als Resultat von `goal()` ein Objekt einer Klasse liefert, deren Klassenname dem Regelnamen entspricht. Konstruiert wird das Objekt dabei über den parameterlosen Konstruktor. Existiert keine Klasse mit diesem Namen auf dem Klassenpfad oder implementiert die gefundene Klasse nicht das `Goal`-Interface, wird ein `GoalAdapter` verwendet.

Factory-Klassen sollten im Betrieb — zum Beispiel wie hier in Java über System Properties — konfigurierbar sein. Damit verschiedene Factory-Klassen koexistieren können, sollten die Namen der Properties den Namen der Factory enthalten.

Für eine `DefaultGoalMakerFactory` können folgende Properties gesetzt werden, um das Verhalten der erzeugten `GoalMaker`-Instanzen zu steuern:

`oops.parser.goal.DefaultGoalMakerFactory.goalPackage`
: Setzt einen Paketnamen, der vor den Namen der Regel gesetzt wird, wenn eine `Goal`-Klasse für eine Regel gesucht wird.

`oops.parser.goal.DefaultGoalMakerFactory.verbose`
: Falls der Wert `true` ist, geben die erzeugten `GoalMaker` eine kurze Nachricht aus, wenn keine `Goal`-Klasse zu einer Regel auf dem Klassenpfad gefunden wird und daher ein `GoalAdapter` verwendet wird. Der default-Wert ist `true`.

`oops.parser.goal.DefaultGoalMakerFactory.verbose2`
: Falls der Wert `true` ist, geben die erzeugten `GoalMaker` eine sehr verbale Ablaufverfolgung ihrer Arbeit aus. Der default-Wert ist `false`.

Eine `DebuggerGoalMakerFactory` agiert wie eine `DefaultGoalMakerFactory`, außer daß `GoalDebugger` anstelle von `GoalAdapter` als default verwendet werden. Analog zu `DefaultGoalMakerFactory` werden wieder drei Properties unterstützt:

`oops.parser.goal.DebuggerGoalMakerFactory.goalPackage`

`oops.parser.goal.DebuggerGoalMakerFactory.verbose`

`oops.parser.goal.DebuggerGoalMakerFactory.verbose2`

Als weitere Implementierung von `GoalMakerFactory` bildet eine `HashtableGoalMakerFactory` Regelnamen auf Klassennamen für die `Goal`-Instanzen ab:

```
public class HashtableGoalMakerFactory implements GoalMakerFactory {
    protected HashMap map = new HashMap();
    ...
    public Object add(String ruleName, String className) { ... }
    public GoalMaker goalMaker (String ruleName) { ... }
}
```

Die erzeugten `GoalMaker` suchen in der internen Hashtabelle mit dem Regelnamen als Schlüssel nach dem Namen der zu verwendenden `Goal`-Klasse und liefern ein Objekt dieser Klasse als `Goal`.

Auch eine `HashtableGoalMakerFactory` wird über Properties gesteuert:

`oops.parser.goal.HashtableGoalMakerFactory.verbose`
: Ist der Wert `true`, wird eine verbale Ablaufverfolgung der Factory und der erzeugten `GoalMaker` ausgegeben. Der default-Wert ist `false`.

`oops.parser.goal.HashtableGoalMakerFactory.debug`
: Ist zu einem Regelnamen als Schlüssel kein `Goal`-Klassenname eingetragen, wird je nach Wert der Property ein `GoalAdapter` (`false`) oder ein `GoalDebugger` (`true`) verwendet.

`HashtableGoalMakerFactory` könnte noch dahingehend erweitert werden, daß zu einem Regelnamen als Schlüssel auch `Goal`-Instanzen oder `Class`-Objekte hinterlegt werden. Im ersten Fall würde der zugehörige `GoalMaker` die `Goal`-Instanz immer wieder klonen und im zweiten Fall ein Objekt über den parameterlosen Konstruktor erzeugen.

Eine `NopGoalMakerFactory` liefert `GoalMaker`, die immer neue `NopGoal`-Instanzen erzeugen.

Mit diesen Implementierungen braucht ein *oops*-Benutzer keine eigene `GoalMakerFactory` und `GoalMaker` schreiben.

6.3.4 Implementierung der Goal-Aktionsschnittstelle

Um den Parser um Aktionen in Form von vom Anwender zu implementierende `Goal`-Klassen zu erweitern, müssen wiederum nur die Klassen `Parser`, `Rule`, `Lit` und `Token` erweitert werden.

Als `action`-Objekt wird als Argument zu `parse()` und zu `shift()` in `Lit` und `Token` das `Goal`-Objekt der momentan aktiven Regel übergeben. Die neue `Parser`-Klasse kann mit `value()` nach dem Werte-Objekt zum aktuellen Symbol gefragt werden. In der überschriebenen `shift()`-Methode der Klasse `oops.parser.goal.Lit` kann damit die entsprechende `shift(Lit,Object)`-Methode beim `Goal` aufgerufen werden:

```
public class Lit extends oops.parser.Lit {
    public Lit(String body) { super (body); }

    protected void shift(Object goal, oops.parser.Parser parser) {
        ((Goal) goal).shift(this, ((Parser)parser).value());
    }
}
```

Analog wird `oops.parser.goal.Token` als Unterklasse von `oops.parser.Token` implementiert und ruft beim `Goal` die Methode `shift(Token,Object)` mit sich als Absender und dem Werte-Objekt vom Scanner auf.

Die neue `Rule`-Klasse kennt ihre `GoalMaker`-Factory zum Erzeugen von `Goal`-Instanzen:

```
public class Rule extends oops.parser.Rule {
    public Rule(Id nt, Node rhs) { super(nt, rhs); }

    protected transient GoalMaker gm;

    public void setGoalMaker(GoalMakerFactory gmf) {
        if (gmf != null) gm = gmf.goalMaker(getNt());
    }
```

In `parse()` erfragt eine `Rule` für die neue Aktivierung beim `GoalMaker` ein `Goal` und übergibt dieses Objekt in die `parse()`-Aufrufe. Die konkrete Parsierung wird wie gehabt von der Oberklasse übernommen.

```
    public void parse(oops.parser.Parser parser, Activation caller,
        Object goal) throws ParseException, IOException, Activation {
        if (gm == null)
            throw new ParseException("No GoalMaker for Rule "+getNt());
        Goal oldGoal = (Goal) goal, subGoal = gm.goal();
        super.parse(parser, caller, subGoal);
        oldGoal.shift(subGoal, subGoal.reduce());
    }
}
```

Der `Parser` erzeugt analog zu der Ablaufverfolgung als `Token`-Instanzen Objekte der `Token`-Klasse aus dem `oops.parser.goal`-Paket. Weiterhin kann der `Parser` mit `value()` nach dem Werte-Objekt zum aktuellen Symbol gefragt werden:

```
public class Parser extends oops.parser.Parser {
    public Parser (oops.parser.Rule start)  { super(start); }
    ...
    protected oops.parser.Token createToken(String id, Set set) {
        return new Token(id, set);
    }

    protected Object value() { return ((ValueScanner)scanner).value(); }
```

Die Parsierung lieferte bislang lediglich als `boolean`, ob die Parsierung fehlerfrei, d.h. ohne einen Syntaxfehler, abgelaufen ist. Dies ist sicherlich eine erwünschte Information. Andererseits liefern die `Goal`-Instanzen als Ergebnis von Regeln Objekte, und das Resultat der `Goal`-Instanz der Startregel stellt so etwas wie das Ergebnis der Parsierung dar. Daher liefert die Parsierung als Resultat ein Objekt der inneren `Result`-Klasse, welche beide Informationen kapselt:

```
    public static class Result {
        Result(boolean error, Object result) {
            this.error = error; this.result = result;
        }
        public boolean error; public Object result;
    }
```

Die Klasse `Parser` des neuen Pakets implementiert eine neue `parse()`-Methode. Diese erwartet einen `ValueScanner` als Scanner und neben der `TableFactory` eine `GoalMakerFactory`. Wird `null` als `GoalMakerFactory` angegeben, wird eine Instanz von `DefaultGoalMakerFactory` benutzt. Alle Regeln werden zum Setzen ihrer `GoalMaker`-Instanz aufgefordert, und die Parsierung startet:

```
        public Result parse(ValueScanner scanner, GoalMakerFactory gmf,
            TableFactory tf) throws ParseException, IOException {
            if (gmf == null)
                gmf = new DefaultGoalMakerFactory();
            for (int i = 0; i < rules.size(); i++)
                ((Rule) rules.elementAt(i)).setGoalMaker(gmf);
            return new Result (
                !super.parse(scanner, tf), // returns boolean parse result
                result                     // set by local Goal; see parse(Activation)
            );
        }
```

`result` ist ein Verweis auf das Ergebnis von `reduce()` gegen die Startregel. Dieser Verweis wird in der überschriebenen `parse(Activation)` in einem lokalen `Goal` gesetzt:

```
        protected transient Object result;

        protected void parse(Activation caller) throws
            ParseException, IOException, Activation {
            start.parse(this, caller, new GoalAdapter() {
                {  Parser.this.result = null; }
                public void shift(Goal goal, Object result) {
                    Parser.this.result = result;
                }
            });
        }
    }
```

6.3.5 Bewertung arithmetischer Ausdrücke

Wie immer sollen arithmetische Ausdrücke als Beispiel nun nicht nur erkannt, sondern von `Goal`-Instanzen während der Erkennung auch ausgewertet werden. Folgende Grammatik liegt dem Beispiel zugrunde:

```
exprs       : [{ [  sum ]  ";" }];

sum         : product [{  sum.add | sum.sub }];
sum.add     : "+" product;
sum.sub     : "-" product;

product     : term [{  product.mul | product.div }];
product.mul : "*" term;
product.div : "/" term;

term        : NUMBER | term.id | "+" term | term.minus | "(" sum ")";
term.id     : ID [  "=" sum ];
term.minus  : "-" term;
```

Um einen großen Test in Form einer `if`-Kette in den `Goal` zu den Regeln zu vermeiden, sind die Grammatikregeln hier teilweise auf Unterregeln aufgeteilt worden. Pro Unterregel gibt es ein `Goal`, welches nun aber genau weiß, ob es zum Beispiel addieren (`sum.add`) oder multiplizieren (`product.mul`) muß.

Exemplarisch werden hier die `Goal`-Klassen zum Addieren und Subtrahieren, die `Goal`-Klasse für den Umgang mit Variablen, die `Goal`-Klassen für unäre Vorzeichen und die `Goal`-Klasse zur Regel `exprs` vorgestellt.

Die Klasse `term` sammelt als Oberklasse der Klassen `term.id` und `term.minus` diese Klassen als statische innere Klassen und befindet sich, wie alle `Goal`-Klassen des Beispiels, im Paket `arith.goals`. Die drei inneren Klassen wissen, welche Aufgabe sie in der Bewertung eines Ausdrucks übernehmen. Daher können alle Literale wie unäre + oder – bzw. das = einer Zuweisung ignoriert werden. Aus diesem Grund implementiert `term` die Methode `shift(Lit,Object)` mit leerem Körper für die Unterklassen vor:

```
package arith.goals;

public class term extends oops.parser.goal.GoalAdapter {
    public void shift (oops.parser.goal.Lit sender, Object value) { }

    public static class id extends term {  ... }
    public static class minus extends term {
        public Object reduce () {
            return error ? null : new Double(-((Number)result).doubleValue());
        }
    } // end of minus
} // end of term
```

In diesem Beispiel sollen die `Goal`-Instanzen den Ausdruck während der Erkennung als `Double` bewerten. Der Scanner liefert daher als Werte-Objekt zum Symbol `NUMBER` die erkannte Zahl als `Double`. Als Ergebnis von `reduce()` eines `term.minus` wird der Wert der erkannten `term`-Regel negiert und der neue Wert wieder als `Double`-Resultat der Regel geliefert. Trat ein Fehler während der Parsierung auf, was die Methode `error()` des `GoalAdapter` in der Instanzvariablen `error` markiert, liefert `reduce()` lediglich `null`.

`term` stammt von `GoalAdapter` ab, erbt damit die Methoden der Oberklassen und ersetzt die eine `shift()`-Methode. Damit ist ein `term` als `Goal` aber bereits in der Lage, das Token `NUMBER`, das unäre Plus und die in runden Klammern geschachtelten Ausdrücke erfolgreich zu bearbeiten. Das erste Werte-Objekt, also hier die Zahl des Symbols `NUMBER`, der Wert des Ausdrucks hinter dem + oder der Wert des geklammerten Ausdrucks, merkt sich `term` als `GoalAdapter` in der Instanzvariablen `result` und liefert den Wert von `result` bei Aufruf von `reduce()` zurück.

Das Beispiel zeigt, daß dank der Vererbung und der Bereitstellung der `GoalAdapter`-Klasse die Entwicklung von `Goal`-Klassen einfach ist. Lediglich die Bewertung bzw. die Zuweisung an Variablen ist etwas komplexer:

Pro Variable wird ein Objekt der inneren Klassen `Var` erzeugt und in der Symboltabelle hinterlegt. Eine `Var` kennt den Namen der von ihr vertretenen Variablen und den Wert der Variablen als `Number`-Instanz:

```
public class term extends oops.parser.goal.GoalAdapter {
    ...
    public static class id extends term {
        public static class Var extends Number {
            protected final String name;
            protected Number value;

            public Var(String name) {  this.name = name; }
```

Vor der ersten Zuweisung an eine Variable und damit an die zugehörige `Var`-Instanz ist der Wert der Variablen undefiniert. Daher prüft `checkValue()`, ob die Variable ohne Wert ist, und `setValue()` setzt einen Wert:

```
            protected void checkValue() {
                if (value == null)
                    throw new RuntimeException("variable "+name+
                                               " has no value");
            }

            public void setValue(Number value) {
                if (value instanceof Var) {  // copy value and not
                    Var var = (Var) value;  // a ref to the value
                    var.checkValue();
                    this.value = var.value;
                } else
                    this.value = value;
            }
```

Ein `Var` ist selbst eine `Number` und implementiert daher die entsprechenden Methoden:

```
            public int intValue() {
                checkValue(); return value.intValue();
            }
            public long longValue() { ...}
            public float floatValue() { ...}
            public double doubleValue() { ...}
        }  // end of term.id.Var
        ...
    } // end of term.id
    ...
} // end of term
```

Der Scanner ist ein Beispiel für eine lexikalische Analyse, die aktiv Objekte in der Symboltabelle einträgt und diese Einträge später wiederverwendet. Als Scanner wird eine *lolo*-Implementierung mit einem `SimpleIdentifier`-Erkenner für Variablennamen verwendet. Wird ein Variablenname erkannt, sucht der Scanner mit dem Namen der Variablen in der Symboltabelle nach einem Eintrag:

```
package arith;
...
public class ArithScanner implements oops.scanner.ValueScanner {
    protected Object symbol, value, NUMBER, ID;
    protected final  Scanner scanner = new Scanner(new Scan[] {
        new SimpleIdentifier().setAction(new Action() {
            public void action(Scan sender, char[] buf, int off, int len) {
                String id = new String(buf, off, len);
                SymTab.Handle handle = symtab.get(id);
```

Sind zu dem Variablennamen bereits Informationen eingetragen worden, werden Verweise auf das Identifizierungs- und das Werte-Objekt (hier eine `Var`-Instanz) für `symbol()` und `value()` hinterlegt. Anderenfalls wird das `Var`-Objekt für den Variablennamen erzeugt und zusammen mit dem Identifizierungs-Objekt in die Symboltabelle eingetragen:

```
                if (handle.isEntered()) {
                    symbol = handle.symbol();
                    value = handle.value();
                } else
                    handle.enter(symbol = ID, value = new Var(id));
            }
        }),
```

```
            new Flt().setAction(...),
            new SimpleWhitespace(),
            new Set("()+-/*;=").setAction(...)
        });
        ...
        public Object value() { return value; }
    }
```

Drei weitere Erkenner-Objekte des *lolo*-Scanners scannen Zahlen, Zwischenraum und die einzelnen Operatoren. Der Programmtext des Scanners zeigt noch einmal deutlich, wie einfach die Schnittstelle `oops.scanner.Scanner` inklusive der Symboltabelle zu handhaben ist.

Damit ist die `Goal`-Klasse `term.id` einfach. Eine Instanz der Klasse merkt sich das `Var`-Objekt des Variablennamens als Ergebnis für `reduce()` und führt die optionale Zuweisung aus. Alle `shift()`- und die `reduce()`-Methode achten dabei auf mögliche Eingabe-Fehler

```
public class term extends oops.parser.goal.GoalAdapter {
    ...
    public static class id extends term {
        public static class Var extends Number { ... }
        protected Var var = null;

        public void shift(oops.parser.goal.Token sender, Object value) {
            if (error |= (value == null)) return;
            var = (Var) value;
        }

        public void shift(oops.parser.goal.Goal goal, Object value) {
            if (error |= (value == null)) return;
            var.setValue((Number) value);
        }

        public Object reduce() { return error ? null : var; }
    } // end of term.id
    ...
} // end of term
```

Die Regel der Grammatik wurde in Teilregeln zerlegt, damit die `Goal`-Instanzen wissen, welche Aufgabe sie zu übernehmen haben. `sum.add` hat zum Beispiel zwei Operatoren zu addieren. Allerdings wird der linke Operator in der `sum`-Regel erkannt. Mit einem kleinen Trick läßt sich das Problem leicht umgehen:

Die `Goal`-Instanzen zu `sum.add` und `sum.sub` stammen beide von der Klasse `eval` ab. `eval` stammt von `GoalAdapter` ab und überimplementiert eine der `shift()`-Methoden, um die Operatoren als Literale zu ignorieren. Als `GoalAdapter` merken sich damit `sum.add` und `sum.sub` den rechten Operanden in der Instanzvariablen `result`:

```
package arith.goals;

public class sum extends oops.parser.goal.GoalAdapter {
    ...
    public abstract static class eval extends oops.parser.goal.GoalAdapter {
        public abstract Object eval (Object a, Object b);
        public void shift (oops.parser.goal.Lit sender, Object value) {}
    }
```

```
        public static class add extends eval {
            public Object eval (Object a, Object b) {
                return new Double(((Number)a).doubleValue() +
                                  ((Number)b).doubleValue());
            }
        }

        public static class sub extends eval {
            public Object eval (Object a, Object b) {
                return new Double(((Number)a).doubleValue() -
                                  ((Number)b).doubleValue());
            }
        }
    }
```

`eval` schreibt für nicht abstrakte Unterklassen die Methode `eval()` mit zwei Operatoren vor. `sum.add` und `sum.sub` implementierten beide die Methode und liefern das Ergebnis der Addition bzw. der Subtraktion zurück.

Ein `sum` merkt sich nun den ersten Operanden und ruft lediglich `sum.add` bzw. `sum.sub` zur Bewertung ihrer Operation auf:

```
public class sum extends oops.parser.goal.GoalAdapter {
    public void shift (oops.parser.goal.Goal sender, Object value) {
        if (error |= (value == null)) return;
        if (result == null) result = value;
        else result = ((eval)sender).eval(result, value);
    }
    ...
}
```

In `exprs` wird das Ergebnis der Bewertung des gerade parsierten, arithmetischen Ausdrucks auf der Standardausgabe ausgegeben:

```
package arith.goals;

public class exprs extends oops.parser.goal.GoalAdapter {
    public void shift (oops.parser.goal.Goal sender, Object value) {
        if (value != null)
            System.out.println("\t"+((Number) value).doubleValue());
    }

    public Object reduce () { return null; }
}
```

Um die Bewertung zu starten, muß einmalig der Parser anhand der Grammatik generiert werden. Dies geschieht, wie bereits gezeigt, durch `oops.EBNF`:

```
$ # print current directory
$ pwd
/examples/arith
$ # set class path
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=..:$JARS/oops.jar:$JARS/lolo.jar
```

```
$ # generate parser from grammar
$ java -Doops.oops.actionType=goal oops.EBNF arith.ebnf > arith.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using ebnf parser generator generated by oops on Fri Aug 17 10:19:53 CEST 2001
oops.parser.Many, warning:
[{ ( sum.add | sum.sub ) }]: ambiguous, will shift
  lookahead {[empty], "+", "-"}
  follow {"+", "*", ")", ";", "/", "-"}

oops.parser.Many, warning:
[{ ( product.mul | product.div ) }]: ambiguous, will shift
  lookahead {[empty], "*", "/"}
  follow {"+", "*", ")", ";", "/", "-"}
$
```

Die Property `oops.oops.actionType` steuert die Parser-Generierung. Ist der Wert der Property `goal`, wird der Parser-Baum aus Instanzen der Klassen des `oops.parser.goal`- und aus den nicht überschriebenen Klassen des `oops.parser`-Pakets erzeugt.

Der erzeugte Parser wird auf der Standardausgabe serialisiert und in diesem Beispiel in der Datei `arith.ser` hinterlegt. Von dort kann der Parser — wie hier vorgeführt wird — immer wieder zum Leben erweckt und ausgeführt werden.

Die Grammatik ist natürlich mehrdeutig. Als Beispiel kann bei dem Ausdruck

```
2 + b = 9 * 3
```

die `3` mit `9` multipliziert und dann `27` an `b` zugewiesen werden, oder alternativ wird `9` an `b` zugewiesen und dann das Resultat der Zuweisung mit `3` multipliziert. Der Parsergenerator warnt vor der Mehrdeutigkeit der Grammatik und erzeugt einen Parser, der in der Konfliktsituation in der Erkennung des momentanen Grammatikteils fortfährt und diesen nicht beendet. Es wird also `27` an `b` zugewiesen.

Weiterhin sind einmalig die Goal-Klassen und der Scanner zu übersetzen:

```
$ # compile classes
$ javac ArithScanner.java
$ javac goals/*.java
```

Die Klasse `oops.RunGoalParser` dient analog zu `oops.RunParser` zum Deserialisieren und Ausführen eines Parsers mit Klassen des `oops.parser.goal`-Pakets:

```
$ java oops.RunGoalParser
java oops.RunGoalParser [-g GoalMakerFactory] [-t TableFactory] parser.ser
                        scannerClassName [<] source [> tree.ser]
$ export PROPS=\
> -Doops.parser.goal.DefaultGoalMakerFactory.goalPackage=arith.goals
$ # run the parser
$ java $PROPS oops.RunGoalParser arith.ser arith.ArithScanner
2 + b = 9 * 3;
        29.0
b;
        27.0
^D
$
```

Per default wird eine `oops.parser.goal.DefaultGoalMakerFactory` als `GoalMakerFactory` verwendet. Damit die erzeugten `GoalMaker` zur Laufzeit die oben erklärten `Goal`-Klassen finden, wird über eine Property der Paketname der `Goal`-Klassen gesetzt.

Verläuft die Parsierung fehlerfrei und liefert das `Goal` der Startregel als Ergebnis von `reduce()` ein Objekt als Gesamtresultat der Erkennung, wird dieses Objekt, falls es serialisierbar ist, auf der Standardausgabe serialisiert. In dem Beispiel liefert `reduce()` in `exprs` aber `null`, und daher kommt es hier zu keiner Ausgabe von `oops.RunGoalParser`.

Neben der `oops.parser.goal.DefaultGoalMakerFactory` kann aber auch eine passend initialisierte `HashtableGoalMakerFactory` verwendet werden:

```
package arith;

public class HashtableGoalMakerFactory extends
    oops.parser.goal.HashtableGoalMakerFactory {
    {
        add("exprs", "arith.goals.exprs");
        add("sum", "arith.goals.sum");
        add("sum.add", "arith.goals.sum.add");
        add("sum.sub", "arith.goals.sum.sub");
        add("product", "arith.goals.product");
        add("product.mul", "arith.goals.product.mul");
        add("product.div", "arith.goals.product.div");
        add("term", "arith.goals.term");
        add("term.id", "arith.goals.term.id");
        add("term.plus", "arith.goals.term.plus");
        add("term.minus", "arith.goals.term.minus");
    }
}
```

Durch die Option `-g` kann bei Aufruf von `oops.RunGoalParser` die `GoalMakerFactory` gesetzt werden:

```
$ # compile the factory
$ javac HashtableGoalMakerFactory.java
$ # run the parser
$ java oops.RunGoalParser -g arith.HashtableGoalMakerFactory arith.ser \
> arith.ArithScanner
GoalMakerFactory is arith.HashtableGoalMakerFactory
2+3*4;
        14.0
^D
$
```

6.3.6 Fazit

Analog zu der Ablaufverfolgung zeigen die `Goal`-Aktionen, daß das Template-Entwurfsmuster, also durch Vererbung von Klassen bestehende Algorithmen um Funktionalität zu erweitern, ein gute und einfach anzuwendende Technik ist. Die Ablaufverfolgung und die `Goal`-Aktionen sind sehr unterschiedliche Aktionsmuster, welche aber beide dem Parser, der hier als eine Art Kontrollstruktur zur Aktivierung der Aktionsmuster angesehen werden kann, aufgeprägt werden können.

Der Einsatz des Factory-Musters ist eine mächtige Technik zur Erzeugung von Instanzen (Klasse des Objekts, Anzahl der Objekte) der gewünschten Klassen, und die von den Factories unterstützten Properties bieten eine Flexibilisierung zur Laufzeit. Der Parser zu einer Grammatik wird mit *oops* einmalig erzeugt. Zur Ausführung des Parsers können dann verschiedene `GoalMakerFactory`- bzw.

`GoalMaker`-Instanzen unterschiedlichen Programmtext als Aktionen ausführen. In *yacc* müßte dazu der Parser mit neuen Aktionen neu übersetzt werden.

Die Trennung von Aktionen und Grammatik hält die Grammatik sauber und damit lesbar. Damit ist sie auch für Portierungen von *oops* auf andere Sprachen wie C# nutzbar. Die Grammatik bleibt sprachunabhängig.

Durch verschiedene Arten von `GoalMakerFactory`-Instanzen kann es zu ganz unterschiedlichem Verhalten kommen: Pro Parser kann immer dasselbe `Goal`-Objekt (analog zu einem SAX-Handler ([SAX]) im XML-Bereich), pro Regel immer dasselbe `Goal` oder wie in den default-Implementierungen der Factory pro Regel-Aktivierung jeweils ein eigenes Goal erzeugt werden. Je nach Lösung entspricht die `Goal`-Instanz damit einem mehr oder weniger lokalen Stack für die Werte-Objekte.

6.4 Aktionen durch Reducer-Instanzen

6.4.1 Idee, Reducer- und ReducerLit-Interface

Das `Goal`-Interface stellt vier Methoden als Schnittstelle zu erkannten Teilen einer Grammatik zur Verfügung. Als weitere Idee eines Aktions-Musters reduziert das `Reducer`-Interface des `oops.parser.reducer`-Pakets die Schnittstelle auf genau eine Aktions-Methode:

```
package oops.parser.reducer;

public interface Reducer {
    public Object reduce(Object[] values);
    public void error();
}
```

Pro Regel existiert ein `Reducer`, welcher am Ende der Erkennung der Regel durch das Resultat von `reduce()` nach einem Werte-Objekt als Ergebnis der Regel gefragt wird. Als Parameter zu `reduce()` erhält der `Reducer` ein Array von Werte-Objekten zu erkannten Teilen der rechten Seite der zugehörigen Regel übergeben. Pro erkanntes Token-Symbol (wie `NUMBER` für einen arithmetischen Ausdruck) wird das vom Scanner gelieferte Werte-Objekt in dem Array hinterlegt. Für erkannte Teilregeln ist das Resultat von `reduce()` gegen die `Reducer`-Instanz der Teilregel im Array eingetragen worden. Die Reihenfolge der Objekte im Array entspricht natürlich der der Erkennung.

Analog zum `Goal`-Interface wird `error()` beim `Reducer` aufgerufen, wenn bei der Parsierung der rechten Seite der zum `Reducer` gehörigen Regel ein Syntaxfehler aufgetreten ist.

Sollen auch zu erkannten Literalen als Symbole die zugehörigen Werte-Objekte vom Scanner im Array vorhanden sein, muß das Aktions-Objekt das `ReducerLit`-Interface adaptieren:

```
public interface ReducerLit extends Reducer {}
```

6.4.2 Adapter-Klassen

Auch für diese simple Aktions-Schnittstelle gibt es fertige Implementierungen der beiden Interfaces:

Ein `ReducerAdapter` implementiert das `Reducer`-Interface und liefert als Resultat zu `reduce()` das übergebene Array. Tritt während der Parsierung der zum `ReduceAdapter` zugehörigen Regel ein Eingabefehler auf, wird dies in der Boolschen Instanzvariablen `error` markiert, und `reduce()` liefert `null`:

```
public class ReducerAdapter implements Reducer {
    protected boolean error;
    public void error() {  error = true; }
```

```
        public Object reduce(Object[] values) { return error ? null : values; }
    }
```

Ein `ReducerDebugger` verhält sich analog zu einem `ReducerAdapter`. Allerdings wird eine Ablaufverfolgung, d.h. zum Beispiel der Inhalt der Argument-Arrays, auf der Standardfehlerausgabe ausgegeben.

Die Klasse `ReducerLitAdapter` stammt von `ReducerAdapter` und `ReducerLitDebugger` von `ReducerDebugger` ab. Beide Klassen adaptieren das leere `ReducerLit`-Interface. Da keine Methoden überschrieben werden, verhalten sich Instanzen der Klasse analog zu ihrer Oberklasse. Als `ReducerLit` werden aber auch Werte-Objekte zu Literalen gesammelt.

6.4.3 ReducerMakerFactory, ReducerMaker

Der Einsatz des doppelten Factory-Musters zum Erzeugen der Aktions-Objekte zur Laufzeit des Parsers hatte sich bereits bei der `Goal`-Aktionsschnittstelle als äußerst mächtig — da sehr flexibel — bewährt. Aus diesem Grund gibt es auch für die `Reducer`-Schnittstelle analoge Fabriken.

Pro Regel erzeugt eine `ReducerMakerFactory` einen `ReducerMaker`:

```
public interface ReducerMakerFactory {
    public ReducerMaker reducerMaker(String name);
}
```

Die `Rule` der Regel fragt bei Aktivierung der Parsierung ihren `ReducerMaker` nach einem `Reducer` (bzw. `ReducerLit`) als Aktions-Partner:

```
public interface ReducerMaker {
    public Reducer reducer();
}
```

Analog zu der `Goal`-Schnittstelle kann der Anwender Factories mit verschiedenen Strategien (Anzahl der `Reducer` bzw. `ReducerLit` pro Parser bzw. pro Regel, Finden der Klassen der Aktions-Objekte, ...) zur Laufzeit ins Spiel bringen.

Um aber dem Anwender das Implementieren einer Factory nicht aufzuzwingen, existieren analog zu `Goal` wieder fertige und sinnvoll nutzbare Implementierungen:

Eine Instanz der Klasse `DefaultReducerMakerFactory` erzeugt `ReducerMaker`, welche `Reducer` bzw. `ReducerLit` als Objekt einer Klasse erzeugen, deren Klassenname dem Namen der Regel entspricht. Ist keine Klasse mit dem entsprechenden Namen auf dem Klassenpfad, wird ein `ReducerAdapter` verwendet.

Um den Einsatz der `DefaultReducerMakerFactory` flexibel zu halten, ist das Verhalten einer Instanz über Properties zur Laufzeit steuerbar:

`oops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.verbose`

`oops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.verbose2`
: Analog zu `Goal` kann über diese beiden Properties eine mehr oder weniger verbale Ablaufverfolgung aktiviert werden.

`oops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.goalPackage`
: Ist die Property gesetzt, wird deren Wert als Paketname der gesuchten Klassen verwendet.

Die Klassen `DefaultLitReducerMakerFactory`, `DebuggerReducerMakerFactory` und `DebuggerLitReducerMakerFactory` stammen von `DefaultReducerMakerFactory` ab und

verhalten sich analog zur Oberklasse. Finden die erzeugten `ReducerMaker` keine Klasse mit dem Namen der Regel, werden allerdings `ReducerLitAdapter`, `ReducerDebugger` bzw. `ReducerLitDebugger` verwendet. Die gleichen drei Properties werden jeweils unterstützt. Lediglich der Klassenname ist im Namen der Property auszutauschen.

6.4.4 Implementierung der Reducer-Aktionsschnittstelle

Analog zur `Goal`-Aktionsschnittstelle war die Erweiterung der Parser-Klassen um `Reducer`-Aktionen wiederum recht einfach. Lediglich die Klassen `Token`, `Lit`, `Rule` und `Parser` mußten erweitert werden.

Ich gehe hier nicht näher auf die Implementierung der vier Klassen ein, da dies keine neuen Ideen aufzeigt.

6.4.5 Repräsentation arithmetischer Ausdrücke

Als Beispiel von Aktionen mit der Reducer-Schnittstelle sollen wiederum arithmetische Ausdrücke erkannt werden. Anstelle der direkten Bewertung analog zum `Goal`-Beispiel wird diesmal aber ein Baum von Objekten als Repräsentierung des erkannten Ausdrucks gebaut.

Alle Objekte, die einen arithmetischen Ausdruck als Baum reflektieren, sind Instanzen von Klassen, welche direkt oder indirekt von der abstrakten Klasse `Node` abstammen. `Node` selbst ist eine `Number` und bereitet einige Methoden für die Unterklassen vor:

```
package arith;

public abstract class Node extends Number implements java.io.Serializable {
    public Node() {}

    public byte byteValue ()   { return (byte)longValue(); }
    public short shortValue () { return (short)longValue(); }
    public int intValue ()     { return (int)longValue(); }
    public float floatValue () { return (float)doubleValue(); }
    ...
}
```

Eine `Node` und damit Instanzen aller Unterklassen sind serialisierbar. `Node` implementiert einige Methoden von `Number` und bildet diese auf `longValue()` bzw. `doubleValue()` ab. Unterklassen müssen damit nur noch diese beiden fehlenden Methoden implementieren.

Pro mathematischer Operation bzw. für Variablen existiert eine innere Klasse zur Repräsentation im Baum. Hier als Beispiel ausführlich die Klasse `Node.Add` und ihre Oberklasse `Binary` für Additionen bzw. zur Modellierung von binären Operatoren:

```
public abstract class Node extends Number implements java.io.Serializable {
    ...
    protected abstract static class Binary extends Node {
        protected Number left, right;

        protected Binary (Number left, Number right) {
            this.left = left; this.right = right;
        }
    }

    public static class Add extends Binary {
        public Add (Number left, Number right) { super(left, right); }
```

```
            public long longValue () {
                return left.longValue() + right.longValue();
            }
            public double doubleValue () {
                return left.doubleValue() + right.doubleValue();
            }
        }
        public static class Sub extends Binary { ... }
        public static class Mul extends Binary { ... }
        public static class Div extends Binary { ... }

        protected abstract static class Unary extends Node { ... }
        public static class Minus extends Unary { ... }
        public static class Var extends Unary { ... }
    }
```

Ziel der Aktion ist es, einen Baum des erkannten Ausdrucks aus Instanzen der `Node`-Klassen zu bauen. Setzt man als Grammatik die in Unterregeln zerlegte Grammatik aus dem `Goal`-Beispiel voraus, so existiert pro Operation eine Unterregel. Damit können `Reducer` anstelle von `ReducerLit` verwendet werden, da die Literale als Information über die Art des Operators nicht benötigt werden.

Der Scanner erzeugt für erkannte Integer- ein `Long`- und für Gleitkomma-Zahlen ein `Double`-Objekt als Werte-Objekt.

Die Klasse `term` stammt von `ReducerAdapter` ab und liefert als Resultat zu `reduce()` das erste Element des Arrays. Damit funktioniert eine Instanz der Klasse bereits als Aktion für erkannte Zahlen, für das unäre Plus und für geklammerte Ausdrücke, da für ein `term` als `Reducer` zu erkannten Literalen kein Verweis auf ein Werte-Objekt im Array hinterlegt worden ist:

```
package arith.reducer;

public class term extends oops.parser.reducer.ReducerAdapter {
    public Object reduce(Object[] values) { // for NUMBER, + term, ( sum )
        return error || values == null ||
               values.length == 0 ? null : values[0];
    }
```

In `reduce()` und auch im Folgenden Programmtext wird defensiv programmiert. So wird hier untersucht, ob ein Syntaxfehler auftrat, ob das Argument zu `reduce()` als Wert `null` ist bzw. ob das Array überhaupt ein Element enthält. Der letzte Fall kann eigentlich nie eintreten.

Für das unäre Minus und für den Umgang mit Variablen werden eigene `Reducer`-Klassen implementiert. Ein `term.id` liefert als Ergebnis das `Node.Var`-Objekt und weist optional den erkannten Wert zu:

```
        public static class id extends term {
            public Object reduce (Object[] values) {
                if (error || values.length == 0 || values[0] == null ||
                    (values.length > 1 && values[1] == null)) return null;
                arith.Node.Var var = (arith.Node.Var) values[0];
                if (values.length > 1)
                    var.setValue((Number) values[1]);
                return var;
            }
        }
```

Ein `term.minus` erzeugt einen `Node.Minus`-Knoten mit dem bereits erzeugten Baum als Unterknoten und liefert diesen als Resultat zurück:

```
        public static class minus extends oops.parser.reducer.ReducerAdapter {
            public Object reduce (Object[] values) {
                return error || values.length == 0 ?
                    null :
                    new arith.Node.Minus((Number) values[0]);
            }
        }
    }
```

Analog bauen die `Reducer` der binären Operatoren die entsprechenden Knoten der `Node`-Klassen. Hier als Beispiel die `Reducer`-Klasse `sum.add` für die Addition:

```
    package arith.reducer;

    public class sum extends oops.parser.reducer.ReducerAdapter {
        public Object reduce(Object[] values) {
            if (error || values == null || values.length == 0) return null;
            Number result = (Number) values[0];
            for (int i = 1; i < values.length; i++)
                result = ((eval) values[i]).eval(result);
            return result;
        }

        public static abstract class eval extends
            oops.parser.reducer.ReducerAdapter {
            protected Number right;
            public abstract Number eval (Number left);
            public Object reduce(Object[] values) {
                if (error || values == null || values.length == 0 ||
                    values[0] == null) return null;
                right = (Number) values[0];
                return this;
            }
        }
        public static class add extends eval {
            public Number eval (Number left) {
                return new arith.Node.Add(left, right);
            }
        }
        public static class sub extends eval { ... }
    }
```

Abschließend gibt der `Reducer` zu `exprs` diesmal nicht den Wert der erkannten Ausdrücke aus, sondern liefert als Ergebnis zu `reduce()` das Array mit den gebauten Bäumen der erkannten Ausdrücke zurück:

```
    package arith.reducer;

    public class exprs extends oops.parser.reducer.ReducerAdapter {
        public Object reduce (Object [] values) {
            return error || values == null || values.length == 0 ?
                   null : values;
        }
    }
```

Damit sind alle Klassen implementiert und werden einmalig übersetzt. Analog wird wiederum mit `oops.EBNF` einmalig ein `Reducer`-fähiger Parser aus der Grammatik generiert und in `arith.ser` serialisiert:

```
$ # set the classpath
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=.:$JARS/oops.jar:$JARS/lolo.jar
$ # compile all classes
$ javac ArithScanner.java Node.java
$ javac reducer/exprs.java reducer/sum.java reducer/product.java \
> reducer/term.java
$ # generate parser from grammar
$ java -Doops.oops.actionType=reducer oops.EBNF arith.ebnf > arith.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using ebnf parser generator generated by oops on Fri Aug 17 10:19:53 CEST 2001
oops.parser.Many, warning:
[{ ( sum.add | sum.sub ) }]: ambiguous, will shift
  lookahead {[empty], "+", "-"}
  follow {"+", "*", ")", ";", "/", "-"}

oops.parser.Many, warning:
[{ ( product.mul | product.div ) }]: ambiguous, will shift
  lookahead {[empty], "*", "/"}
  follow {"+", "*", ")", ";", "/", "-"}
$
```

Das Deserialisieren des Parsers geschieht analog durch die Klasse `oops.RunReducerParser`:

```
$ java oops.RunReducerParser
usage:
java oops.RunReducerParser [-r ReducerMakerFactory] [-t TableFactory]
                           parser.ser scannerClassName [<] source [> tree.ser]
```

Durch Angabe der Option `-r` kann der Klassenname der `ReducerMakerFactory` angegeben werden. Als Default wird eine `DefaultReducerMakerFactory` verwendet. Verläuft die Parsierung fehlerfrei und liefert der `Reducer` bzw. der `ReducerLit` der Startregel als Resultat zu `reduce()` ein Objekt, welches serialisierbar ist, wird dieses Objekt auf der Standardausgabe serialisiert. Hier enthält `expr.ser` daher ein serialisiertes Array mit Bäumen aus `Node`-Klassen als Elemente:

```
$ # run the parser with reducer action
$ PROPS=-Doops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.\
> goalPackage=arith.reducer
$ java $PROPS oops.RunReducerParser arith.ser arith.ArithScanner > exprs.ser
12.34 + 56.7 * 89;
9.999 + 0.001 - 10.;
^D
$ ls -l exprs.ser
-rw-r--r--   1 bernd    staff         440 Aug 22 12:34 exprs.ser
$
```

Das Serialisieren der Objekte des Parsers macht die erzeugten Parser persistent und wiederverwendbar. Aus dem gleichen Grund ist in diesem Beispiel ein Baum für erkannte Ausdrücke gebaut worden. Dieser Baum kann serialisiert, später wiederverwendet und zum Beispiel in verschiedenen Typen bewertet werden.

Als Beispiel deserialisiert die Klasse `Go` ein Array von `Number`-Objekten und bewertet die Elemente in verschiedenen Typen:

```
package arith;

public class Go {
    public static void main (String args[]) {
        try {
            java.io.ObjectInputStream in = new java.io.ObjectInputStream(
                args.length == 0 ?
                System.in : new java.io.FileInputStream(args[0])
            );
            Object[] expressions = (Object[])in.readObject();
            System.out.println("long\tdouble");
            for (int i = 0; i < expressions.length; ++ i) {
                Number n = (Number)expressions[i];
                System.out.println(n.longValue()+"\t"+n.doubleValue());
            }
        } catch (Exception e) { System.err.println(e); }
    }
}
```

Als Beispiel deserialisiert `Go` im Folgenden die in `expr.ser` gespeicherten zwei Bäume:

```
$ # compile Go
$ javac Go.java
$ # run Go
$ java arith.Go exprs.ser
long    double
4996    5058.64
-1      0.0
$
```

6.4.6 Fazit

Die Reducer-Aktionsschnittstelle zeigt drei Aspekte:

Zum einen ist die Erweiterung der Parser um andere Aktionsmuster genauso einfach wie die Erweiterung um `Goal`-Aktionen. *oops* unterstützt fest eingebaute Ablaufverfolgungs-, `Goal`- und `Reducer`-Aktionen. Ein Anwender von *oops* kann aber durch das Implementieren eigener Unterklassen sehr leicht seine eigene Aktionsidee verwirklichen.

`Goal` ist ein wesentlich granularerer Ansatz als Reduce. Beide Ansätze liefern dem Programmierer von Aktionen zu Regeln aber alle nötigen Informationen. Bei `Goal` wird der Anwender sofort über erkannte Symbole bzw. über erkannte Unterregeln informiert. Bei `Reducer` erfährt er dies erst am Ende der kompletten Regel. Das Ignorieren von zum Beispiel erkannten Literalen geschieht in `Goal` durch eine leere `shift(Lit,Object)`-Methode und bei `Reducer` durch das Adaptieren des `ReducerLit`-Interfaces.

Meines Erachtens sind `Goal`- und `Reducer`-Aktionen sehr mächtige Techniken. Weitere Aktionsideen werden von mir daher im Folgenden nur diskutiert und können mit einer geeigneten Factory aber auch durch `Goal` bzw. `Reducer` imitiert werden.

Gamma et al. führen weitere Muster zur Benachrichtigung bzw. Beobachtung von Ereignissen vor, welche auch als Aktions-Muster verwendet werden könnten. Im Folgenden werden einige auf ihre mögliche Verwendung als Aktions-Muster untersucht.

6.5 Aktionen per Listener-Muster

Beim *Listener*-Muster melden sich mehrere Listener-Objekte bei dem einen zu observierenden Objekt an und werden per Methodenaufruf über Ereignisse informiert. Die Methode bzw. die Methoden sind dabei durch ein Interface fest vorgeschrieben.

Erkennt ein `Parser` ein Symbol oder eine Regel, können diese Ereignisse auch über das Listener-Muster in Benutzer-Aktionen gewandelt werden. Dies hat gegenüber der `Goal`-Schnittstelle den Vorteil, daß mehr als ein Listener-Objekt sich für das gleiche Ereignis interessieren kann.

Es bietet sich analog zu `Goal` an, die drei verschiedenen shift-Ereignisse in dem Listener-Interface als einzelne Methoden zu definieren:

```
public interface RuleListener {
    public void startRule(String ruleName);
    public void shift(String ruleName, Token token, Object valueFromScanner);
    public void shift(String ruleName, Lit lit, Object valueFromScanner);
    public void shift(String ruleName, String parsedSubRuleName);
    public void endRule(String ruleName);
}
```

Analog zu den AWT-Klassen könnten `RuleListener` bei einer `Rule` eingetragen werden:

```
    void addRuleListener(String ruleName, RuleListener l)
    void deleteRuleListener(String ruleName, RuleListener l)
```

Listener hören zu und werden nicht — wie zum Beispiel per `reduce()` bei einem `Goal` — nach Ergebnissen befragt. Sind mehrere Listener eingetragen worden, wäre auch nicht klar, welcher von diesen Vorrang hätte. Die verschiedenen Listener müssen daher über einen globalen Bereich Daten austauschen. Die Listener könnten sich einen Verweis teilen oder eine Klasse dazu verwenden.

Weiterhin ist eine Regel aufgrund von Rekursion durchaus mehr als einmal aktiv. Um die verschiedenen `shift()`-Nachrichten der aktuell aktiven Regel zuzuordnen, wird ein Listener einen lokalen Stack vorhalten müssen. Dies ist ein entscheidender Nachteil gegenüber dem `Goal`- oder `Reducer`-Muster mit einer `Goal`- bzw. einer `Reducer`-Instanz pro Regelaktivierung.

In dem obigen Ansatz werden Listener pro Regelname der Grammatik bei den `Rule`-Instanzen eingetragen. Als Alternative könnte eine Listener-Instanz aber auch global für alle Regeln oder auch nur für das Erkennen von einzelnen Symbolen eingetragen werden. Im zweiten Fall beständen die Interfaces dementsprechend aus nur einer Methode:

```
public interface TokenListener {
    public void shift(Token token, Object valueFromScanner);
}

public interface LitListener {
    public void shift(Lit lit, Object valueFromScanner);
}
```

6.5.1 Fazit

Viele Listener pro Regel stellen einen Vorteil gegenüber der `Goal`- oder der `Reducer`-Schnittstelle dar. Aber ein Multiplexer-`Goal` bzw. -`Reducer` behebt einfach und schnell den Nachteil.

Der aktuelle Wert zu den bereits erkannten Teilen der Eingabe muß über eine globale Datenfläche verwaltet werden.

Da eine Regel gleichzeitig mehr als einmal aktiv sein kann, ist ein lokaler Stack zur Unterscheidung der Aktivierungen nötig.

Die mögliche Verwendung des Listener-Musters für Aktionen zu erkannten Teilen der Grammatik stellt eine interessante Idee dar. Auf jeden Fall zeigt sie die Stärke des objekt-orientierten Ansatzes. Die Erweiterung des Parsers um Listener als Benutzer-Aktionen ist durch die Erweiterung der Klasse `Rule` bzw. auch `Parser`, `Lit` und `Token` schnell und einfach zu implementieren.

6.6 Aktionen per Delegate

Das *Delegate*-Muster ist ein weiteres Beobachtungsmuster und käme daher auch als Aktions-Muster für einen *oops*-Parser in Frage. Das Delegate-Muster ist neben dem *target/action*-Muster das Aktions-Muster für die Programmierung von NeXTSTEP bzw. dem Nachfolger OPENSTEP. Beide Betriebssysteme wurden von der Firma NeXT entwickelt und gingen mittlerweile in das neue Betriebssystem Mac OS X ([App01]) von Apple auf. Auch in der Programmierung von Mac OS X sind das Delegate- und das target/action-Muster weiterhin die Standard-Beobachtungsmuster.

Bei dem Delegate-Muster kennt ein zu beobachtendes Objekt optional genau ein Delegate-Objekt. Der Delegate kann vorgegebene Methoden implementieren, muß es aber nicht. Kommt es zu einem Ereignis in dem zu beobachtenden Objekt und kennt das Objekt ein Delegate, so delegiert er dieses Ereignis oder besser eine Aktion zu dem Ereignis per Methodenaufruf an den Delegate. Dies allerdings nur, wenn der Delegate die zum Ereignis zugehörige Methode implementiert hat.

Dieses Muster bringt auch kaum etwas Neues gegenüber dem `Goal`- oder `Reducer`-Muster:

Bei dem `Goal`-Muster sind wegen des `Goal`-Interfaces die Methoden zwar alle zu implementieren, aber für eine konkrete Implementierung von `Goal` können uninteressante Methoden mit leeren Körpern implementiert werden. Gleiches gilt für `Reducer`, wo darüber hinaus Werte-Objekte zu Literalen erst bei einem `ReducerLit` mitgeführt werden.

Weiterhin gibt es mit den Standard-Factories genau eine `Goal`- oder `Reducer`-Instanz je aktive Regel. Beim Delegate-Muster kann dies noch entworfen werden. Das optionale Delegate-Objekt kann es pro `Parser`, pro Regel, über eine Factory pro Regelaktivierung oder auch pro `Lit` oder `Token` geben.

Da Delegate-Methoden eigentlich keinen Wert liefern, müßten sich mehrere Delegate-Objekte wiederum Informationen über eine globale Fläche teilen. Gibt es nicht pro Regel-Aktivierung einen Delegate, ist wieder ein lokaler Stack nötig.

6.7 Aktionen per target/action

Beim target/action-Muster kennt ein zu observierendes Objekt optional ein target-Objekt. Tritt das zu beobachtende Ereignis ein, wird beim target-Objekt eine vereinbarte, aber vom Namen her frei wählbare Methode aufgerufen.

Dieses Muster wäre für einen Parser eine sehr schmale Brücke, da über nur eine Methode alle shift-Nachrichten und die zugehörigen Werte-Objekte fließen müßten. Daher sollte es besser pro Regel und pro `Lit` und `Token` jeweils target-Objekte geben.

Wie bei den anderen Mustern bleibt die Frage nach der Anzahl von target-Instanzen pro Regel bzw. pro Regelaktivierung und damit, ob target-Objekte einen lokalen Stack brauchen oder nicht.

Alle target-Objekte müßten wieder über eine globale Fläche Informationen austauschen.

6.8 Aktionen per Observer

Das Observer-Muster ist sehr analog zum target/action-Muster: Ein zu observierendes Objekt kennt maximal ein Observer-Objekt als Beobachter. Tritt das zu beobachtende Ereignis beim observierten Objekt ein, wird beim Observer eine fest vereinbarte Methode aufgerufen.

Der einzige Unterschied zum target/action-Muster ist also, daß der Name der Methode nicht frei wählbar ist.

6.9 AST und Visitor

Parser anderer Compiler-Systeme, wie zum Beispiel *ANTLR*, *JavaCC* oder *SableCC*, erzeugen (teilweise optional) als Resultat einen abstrakten Syntax-Baum (*abstract syntax tree* oder kurz AST) als Repräsentation eines parsierten Programms. Ein Anwender traversiert anschließend den AST mit einem Visitor-Objekt nach dem Visitor-Entwurfsmuster ([Gam95]).

Das Visitor-Entwurfsmuster bietet den Vorteil, daß verschiedene Visitor von außen auf einen AST aufgesetzt werden können. Allerdings ist das Visitor-Entwurfsmuster — im Gegensatz zu einem Baum mit Knoten, deren Methoden den Baum von innen bewerten, — nicht wirklich objekt-orientiert:

Als Beispiel könnten Programme der Grammatik

```
expr:  term {[ "+" term ]} ;
term:  NUMBER;
```

durch ein AST aus Instanzen der Klassen `Expr` und `Term` als Knoten bzw. aus `String`- und `Number`-Objekten für Literale und Werte zu `NUMBER`-Symbolen als Blätter des Baums repräsentiert werden. Das Visitor-Interface würde für die Traverse pro Knoten-Klasse eine Methode definieren:

```
public interface Visitor {
    public void visit(Expr expr);
    public void visit(Term term);
}
```

Die Knoten des AST akzeptieren einen `Visitor` als Argument einer Methode — wie hier `apply()` — und rufen in der Methode beim `Visitor`-Argument die entsprechende `visit()`-Methode des Interfaces auf. Eine abstrakte Basisklasse aller Knoten-Klassen stellt sicher, daß alle Knoten-Klassen die Methode `apply()` implementieren:

```
public abstract class Node {  // abstract class for node classes
    ...
    public abstract void apply(Visitor visitor);
}
public class Expr extends Node {
    ...
    public void apply(Visitor visitor) {
        visitor.visit(this, o);
    }
}
```

`Visitor`-Implementierungen verarbeiten in den `visit()`-Methoden die Informationen des Argument-Knotens und rufen gegebenfalls `apply(this)` gegen Unterknoten zur Traverse der Unterknoten auf.

Das Problem beginnt, wenn die Grammatik erweitert wird und damit neue Knoten-Klassen ins Spiel kommen. Bestehende Knoten-Klassen und `Visitor`-Implementierungen sollten in einem objekt-orientierten Ansatz wiederverwendet und nicht abgeändert werden. Eine Unterklasse einer `Visitor`-

Implementierung sollte die neuen Knoten behandeln und alles andere von der bestehenden Implementierung erben.

An dieser Stelle gibt es drei Möglichkeiten: Das Visitor-Interface zur Traverse eines AST der neuen, ausgebauten Grammatik erweitert das alte Interface um die neuen `visit()`- oder es beinhaltet alle `visit()`-Methoden. Im zweiten Fall kann der Name des neuen Interfaces darüber hinaus zum Namen des alten Interfaces gleich oder verschieden sein.

Leider kommt es in allen drei Fällen zu Problemen: Im ersten und dritten Fall müssen die neuen Knoten-Klassen die `apply()`-Methode mit einem Parameter des alten Visitor-Interfaces implementieren. Hier ist zumindest ein Cast des Arguments auf das zugehörige Interface notwendig. Im zweiten Fall sind die alten Visitor-Klassen unvollständig, da sie die neuen `visit()`-Methoden nicht implementieren.

Die erste der drei Möglichkeiten wäre gangbar. Dazu müßten aber in der Grammatik Entwicklungsschritte notiert werden, damit eine entsprechende Klassengenerierung möglich ist. Alle mir bekannten Werkzeuge bieten hier keine objekt-orientierte Lösung an.

Da *oops*-Grammatiken keine zusätzliche Auszeichnungen enthalten sollen, ist in *oops* ganz bewußt keine automatische AST-Unterstützung integriert. Eine eigene, unveröffentlichte Studie hat gezeigt, daß *oops* aber sehr leicht um einen Generator für das Visitor-Interface und für die Knoten-Klassen des AST zu erweitern ist. Eine `GoalMakerFactory` bzw. von deren `GoalMaker`-Instanzen erzeugte `Goal`-Objekte bauen dann zur Laufzeit des Parsers den AST als Repräsentation des parsierten Programms. Eine strikte objekt-orientierte Lösung ist aber auch hier nicht möglich.

6.10 Fazit

Der Einsatz der Objekt-Orientierung, d.h. der Einsatz von Klassen oder besser von Instanzen der Klassen zur Repräsentierung einer Grammatik und damit gleichzeitig als Parser von Programmen über der Grammatik, läßt Erweiterungen der Klassen und damit eine Erweiterung der Parsierung offen.

Zur Erweiterung der Parser um verschiedenste Arten von Aktionen bedarf es lediglich der Erweiterung von maximal vier Klassen eines Parser-Baums. Die Erweiterung war durch den Einsatz des Template-Entwurfsmuster in Form von zu überschreibenden Methoden und durch ein frei nutzbares Argument der `parse()`-Methode vorbereitet und damit leicht durchführbar.

Die `Goal`- oder die `Reducer`-Aktionsschnittstelle bindet auf eine für die Objekt-Orientierung natürliche Art und Weise Aktionen an Teile des Parsers. Beide Techniken trennen die Grammatik vom Programmtext der Aktionen, da die Grammatik nicht — wie in *yacc* — mit Programmtext für Aktionen zu versehen ist. Die Grammatik bleibt rein und lesbar. Aus der Grammatik können Parser mit verschiedenen Arten von Aktions-Mustern erzeugt werden. Das ist Wiederverwendung von Grammatiken! Die Grammatik ist darüber hinaus von der Implementierungssprache von *oops* unabhängig.

oops-Parser, welche `Goal`- bzw. `Reducer`-Aktionen unterstützen, lassen pro Regel die zugehörige `Goal`- bzw. `Reducer`-Instanz durch eine Factory erzeugen. Dadurch bleibt der Parser vom Aktions-Programmtext unabhängig. Durch die Verwendung verschiedener Factories kann der gleiche Parser verschiedenartige Aktionen zu erkannten Teilen einer Grammatik ausführen.

Dem Anwender von *oops* steht es frei, eigene Aktionsideen durch Unterklassen der Klassen eines Parser-Baums in die Parsierung einzubringen. `Goal`- und `Reducer`-Aktionen sind aber beide ausreichend mächtige Techniken, um alle anderen gängigen Beobachtungsmuster abbilden zu können.

7 Die opi-Schnittstelle

7.1 Die Idee

Dem Entwickler eines Programms, das XML-Dokumente analysieren bzw. transformieren soll, ist es egal, wer der Hersteller des von ihm verwendeten XML-Parsers bzw. XSLT-Prozessors ist. Mehr noch, er möchte ein Programm, welches von der genauen Version des Parsers bzw. des Prozessors unabhängig ist. An diesem Punkt setzt Sun's *Java API for XML Processing* (kurz JAXP, [JAXP]) an. JAXP beinhaltet Factory-Klassen und Interfaces, welche für einen standardisierten Zugriff auf XML-Parser und XSLT-Prozessoren sorgen. Die Auswahl des XML-Parsers bzw. XSLT-Prozessors erfolgt lediglich über den Wert einer System Property, was den XML-Parser bzw. den XSLT-Prozessor sehr leicht austauschbar und außerhalb vom Programmtext wählbar hält.

Dieser Ansatz war Ausgangspunkt für eine allgemeine API für Parser bzw. Compiler: Die *opi*-API (**o**bject-orientated **p**arser **i**nterface). Ziel der API ist wiederum, verschiedene Implementierungen von Parsern bzw. Compilern unter eine einheitliche Schnittstelle zu packen und auch über eine Property außerhalb vom Programmtext wählbar zu halten.

7.2 Die opi-Schnittstelle

Ein Parser arbeitet über eine Eingabe und prüft, ob die Eingabe einem legalen Programm über der repräsentierten Grammatik entspricht. Das Resultat einer Parsierung ist lediglich ein Boolscher Wert, welcher das Ergebnis der Prüfung reflektiert. Ein Parser kann daher allgemein und unabhängig von seiner exakten Implementierung durch folgendes Interface beschrieben werden.

```
public interface Parser {
    public boolean parse(InputSource input) throws OpiException;
}
```

Für eine bezüglich der Implementierung neutrale Beschreibung eines Parsers muß auch die Eingabe unabhängig beschrieben werden, was hier durch eine `oops.opi.InputSource` geschieht. Eine Vielzahl von Konstruktoren läßt eine `InputSource` die Daten einer Datei, einer Netzverbindung oder eines beliebigen Daten-Stroms (`InputStream` als Byte-Strom oder `Reader` als Strom von Unicode-Zeichen) repräsentieren:

```
package oops.opi;
...
public class InputSource {
    ...
    public InputSource(String fileName, String encoding) throws
        FileNotFoundException, UnsupportedEncodingException { ... }
    public InputSource(String fileName) throws FileNotFoundException { ... }
    public InputSource(InputStream input, String encoding) throws
        UnsupportedEncodingException { ... }
    public InputSource(InputStream input) { ... }
    public InputSource(URL input, String encoding) throws
        IOException, UnsupportedEncodingException { ... }
    public InputSource(URL input) throws IOException { ... }
    public InputSource(Reader reader) throws
        IOException, UnsupportedEncodingException { ... }
```

Die Methode `getReader()` einer `InputSource` liefert als Resultat einen `java.io.Reader`, der als Resultat seiner `read()`-Methoden die Daten liefert. Sollten die Daten nicht bereits als Unicode-

Zeichen vorliegen, kann bei Konstruktion oder später durch `setEncoding()` das Encoding der Bytes zu Unicode-Zeichen ausgewählt werden. Liegen die Daten bereits als Unicode-Zeichen vor, liefert `getInputStream()` als Resultat `null`, ansonsten aber einen `InputStream`, der die Daten als Bytes zu den `read()`-Methoden liefert:

```
        public InputStream getInputStream() {  ... }
        public Reader getReader()  {  ... }
        public String getEncoding() {  ... }
        public void setEncoding(String encoding) throws
            UnsupportedEncodingException {  ... }
    }
```

Während der Parsierung können Ausnahmen auftreten — zum Beispiel findet der ausgewählte Parser notwendige Ressourcen nicht. Auch diese sind neutral zu beschreiben. Da das `Parser`-Interface alle möglichen Ausnahmen aller möglichen Implementierungen des Interfaces nicht vorhersehen kann, wird eine Ausnahme in Form einer `OpiException` berichtet. Eine `OpiException` kapselt neben einem Text optional auch eine beliebige andere Exception (oder exakt ein `Throwable`). Über diese Brücke können von `Parser`-Implementierungen beliebige Exception, gekapselt in eine `OpiException`, nach außen berichtet werden:

```
public class OpiException extends Exception {
    public Throwable nest;
    public OpiException() {}
    public OpiException(String message) {  ... }
    public OpiException(String message, Throwable nest) {  ... }
    public String getMessage() {  ... }
    public Throwable getNestedThrowable() {  ... }
    ...
}
```

Unterschiedliche Parser unterstützen das vorgestellte `Parser`-Interface und sind somit im Programmtext vollkommen gleich zu benutzen. Für einen von der Implementierung unabhängigen Programmtext muß dies aber auch für die Erzeugung der `Parser`-Objekte gelten. Analog zu JAXP werden daher `Parser`-Instanzen über ein Factory-Muster kreiert:

```
public abstract class ParserFactory {
    public abstract Parser parser() throws OpiException;
    public static ParserFactory newInstance() throws ClassNotFoundException,
        InstantiationException, ... {  ...}
}
```

Unterklassen von `ParserFactory` implementieren die abstrakte Methode `parser()`, erzeugen in der Methode ein `Parser`-Objekt für eine konkrete Implementierung und liefern den `Parser` als Resultat zurück. Dadurch wird die Erzeugung der `Parser`-Instanz auf die Factory verschoben. Damit die exakte Klasse der Factory nicht Teil des Programms ist, kann über die Klassenmethode `newInstance()` eine Instanz der gewünschten Factory angefordert werden. `newInstance()` wertet dazu die Property `oops.opi.ParserFactory` als den Klassennamen der gewünschten Factory aus, erzeugt über den parameterlosen Konstruktor ein Objekt der Klassen und liefert dieses Objekt als Resultat.

Parser prüfen lediglich die Eingabe gegen die repräsentierte Grammatik. Compiler im weitesten Sinne erweitern die Funktionalität von Parsern dahingehend, daß sie während der Parsierung zu erkannten Teilen eines Programms zum Beispiel einen Parse-Baum bauen oder auch das Programm sofort bewerten. Das Interface `Compiler` beschreibt diesen Umstand wieder unabhängig:

```
public interface Compiler {
    public Result compile(InputSource input) throws OpiException;
```

```
    public static class Result {
        public boolean parseOk; public Object result;
        public Result(parseOk, result) {
            this.parseOk = parseOk; this.result = result;
        }
    }
}
```

Das Resultat eines Compilerlaufs ist ein `Result`-Objekt, welches unter anderem als Information kapselt, ob die Parsierung erfolgreich war. Darüber hinaus verweist die Instanzvariable `result` auf die vom Compiler erzeugte Information für das erkannte Programm.

Eine `CompilerFactory` dient wiederum zur von der Implementierung unabhängigen Erzeugung von `Compiler`-Instanzen, wobei innerhalb von `newInstance()` der Wert der Property `oops.opi.CompilerFactory` als Klassenname der zu erzeugenden `Compiler` verwendet wird:

```
public abstract class CompilerFactory {
    public abstract Compiler compiler() throws OpiException;
    public static CompilerFactory newInstance() throws ClassNotFoundException,
        InstantiationException, ... { ... }
}
```

7.3 Beispiel arithmetische Ausdrücke

Die Interfaces und Klassen der *opi*-API sollen die Verwendung verschiedener Parser und Compiler transparent halten. Als Beispiel eine Klasse `Main`, welche über eine `ParserFactory` einen `Parser` instanziiert, eine Eingabequelle als `InputSource` erzeugt, die Eingabe vom `Parser` analysieren läßt und abschließend den Erfolg oder Mißerfolg der Parsierung anzeigt:

```
public class Main {
    public static void main(String args[]) throws Exception {
        oops.opi.ParserFactory factory = oops.opi.ParserFactory.newInstance();
        oops.opi.Parser parser = factory.parser();
        oops.opi.InputSource source = new oops.opi.InputSource(System.in);

        boolean ok = parser.parse(source);
        System.out.println("input was "+(ok ? "ok" : "not ok"));
    }
}
```

Im Gegensatz zu JAXP, wo immer XML-Parser bzw. XSLT-Prozessoren erzeugt und benutzt werden, ist über das `Parser`-Interface nicht festgelegt, welche Art von Parsern betrieben werden. Der Anwender hat daher bei der Wahl der `ParserFactory` darauf zu achten, daß diese auch einen zum gewünschten Verwendungszweck passenden Parser — hier also einen für arithmetische Ausdrücke — liefert.

Als Beispiel führt im Folgenden `Main`, gesteuert über die Property `oops.opi.ParserFactory`, zwei verschiedene Parser aus. In dem Beispiel parsieren beide Parser einen arithmetischen Ausdruck, wobei die `Parser` intern einen *oops*- bzw. einen *jay*-Parser kapseln:

```
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=.:$JARS/oops.jar:$JARS/lolo.jar
$ export PROPERTIES=-Doops.opi.ParserFactory=OopsParserParserFactory
$ java -classpath $CLASSPATH $PROPERTIES Main
2+3;
input was ok
```

```
$ export PROPERTIES=-Doops.opi.ParserFactory=JayParserParserFactory
$ java -classpath $CLASSPATH $PROPERTIES Main
7-4;
input was ok
$
```

7.4 Fazit

Wie das Kapitel gezeigt hat, kann der auf XML-Parser bzw. XSLT-Prozessoren beschränkte Ansatz von JAXP, auch auf allgemeine Parser und Compiler erweitert werden. Parser und Compiler sind mit *opi* neutral bezüglich ihrer Implementierung zu verwenden und sind über den Wert einer Property leicht austauschbar.

Im Gegensatz zu JAXP unterliegen modellierte Parser hinsichtlich ihrer Anwendung keinerlei Beschränkung. Der Anwender muß daher bei der Auswahl der `ParserFactory` sicherstellen, daß diese auch Parser für den gewünschten Zweck erzeugt.

8 Fehlerbehandlung durch Objekte und Exceptions

Ein Syntaxfehler tritt auf, wenn ein Parser während der Parsierung eines Programms auf ein unpassendes Symbol trifft. In diesem Fall sollte die Erkennung nicht einfach mit einer Fehlermeldung abbrechen. Gewünscht ist die Möglichkeit einer Fehlerbehandlung, das heißt, die Erkennung setzt an einer geeigneten Stelle in der Eingabe und an einer geeigneten Stelle in der Grammatik wieder auf, und dem Programmierer werden mögliche weitere Fehler berichtet.

Da die Knoten eines von *oops* generierten Parser-Baums ihre lookahead- und follow-Mengen kennen, könnte man diese für eine automatische und mächtige Fehlerbehandlung verwenden.

Unter automatischer Fehlerbehandlung sind zwei Dinge zu verstehen: Es soll keine zusätzliche Syntax wie das `error`-Token in *yacc* in der Eingabegrammatik von *oops* für die Fehlerbehandlung geben. Diese Extrasyntax muß vom Anwender erlernt werden, und sie macht die eigentliche Grammatik weniger gut sichtbar. Weiterhin soll die Fehlerbehandlung ohne ein Eingreifen des Entwicklers stattfinden – eben automatisch. Beides hält die Anwendung von *oops* für den Entwickler einfach und ist in der Benutzung mächtig.

Wie gut kann eine automatische Fehlerbehandlung sein? Die Steuerung der Erholung in *yacc* durch `error`-Token ist sicher eine mächtige Technik, muß aber erlernt und für jede Grammatik neu entworfen werden. Die Plazierung der `error`-Token erfordert Erfahrung und macht oft die Grammatik mehrdeutig. Eine automatische Erholung vereinfacht die Anwendung, aber im Gegensatz zu einem von Hand auf Fehlerbehandlung optimierten Parser wird die Qualität der Erholung voraussichtlich schlechter sein. Dies ist meiner Meinung nach kein echter Nachteil: Das Programm ist ohnehin falsch, der Parser erholt sich sinnvoll und automatisch, und mögliche weitere Fehler werden dem Entwickler berichtet.

Als weitere Idee könnte die Fehlerbehandlung dahingehend erweitert werden, daß der Entwickler optional in den Fehlerbehandlungs-Mechanismus eingreifen und diesen steuern kann.

8.1 Fehlerbehandlung

Der Benutzer einer Sprache ist bei einem Syntaxfehler in erster Linie an einer aussagekräftigen Fehlermeldung interessiert, welche zum Beispiel das falsche Symbol und die Menge der aktuell akzeptablen Eingabesymbole enthalten könnte. Weiterhin möchte er für seine Programme nach Möglichkeit nicht nur den ersten Syntaxfehler mitgeteilt bekommen.

Der Programmierer eines Compilers einer Sprache ist an einer möglichst einfachen Umsetzung einer Fehlerbehandlung interessiert, welche aber trotzdem die Wünsche kommender Anwender abdeckt. Eventuell möchte er an einigen Stellen des Compilers in die Fehlerbehandlung eingreifen können.

Der Entwickler eines Parsergenerators muß eine sinnvolle Möglichkeit der Fehlerbehandlung bei einem Syntaxfehler als Bestandteil generierter Parser einplanen. Er muß sich überlegen, wie weit seine Anwender, also die Programmierer eines Compilers, in die Fehlerbehandlung eingreifen können sollen und welche Informationen (zum Beispiel das falsche Symbol und die Menge der aktuell akzeptablen Eingabesymbole) er bei einem Syntaxfehler zur Verfügung stellt.

Im Folgenden sollen für eine Parsierung nach der Technik des rekursiven Abstiegs mögliche Situationen beim Auftreten eines Syntaxfehlers und deren alternativen Behandlungen anhand eines *oops*-Parsers für primitive arithmetische Ausdrücke erläutert werden. Dem Parser liegt folgende Grammatik zugrunde:

```
expr    : product {[ "+" product ]} ";" ;
product : NUMBER {[ "*" NUMBER ]} ;
```

Abbildung 8.1 zeigt den Parser, nachdem bereits `2+3` aus der Eingabe akzeptiert worden ist. Schwarze Knoten wurden bereits erfolgreich erkannt, Blau symbolisiert aktive und Grün markiert (möglicherweise) noch zu aktivierende Knoten:

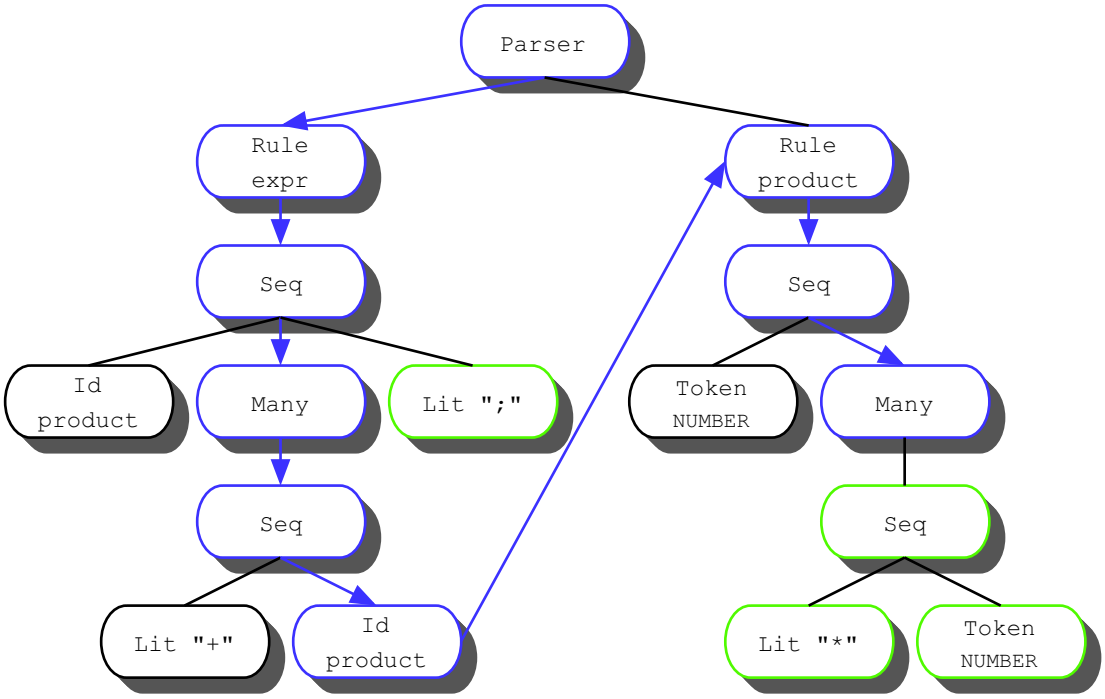

Abbildung 8.1: Erkannte, aktive und deaktive Parser-Knoten.

Der momentan aktive Knoten ist der `Many`-Knoten der `product`-Regel. Sollte das nächste Eingabesymbol zu seinem lookahead passen, steigt die Parsierung in dessen `Seq`-Unterknoten ab.

Grundsätzlich sind bei einem Syntaxfehler je aktive Methode des rekursiven Abstiegs, d.h. hier pro aktiven Knoten, drei mögliche Vorgehensweisen zur Fehlerbehandlung denkbar: Die Methode beginnt mit der Erkennung der kompletten, repräsentierten Phrase von vorne (*retry*), sie endet und delegiert die Verantwortung weiter nach außen in der Aufrufverschachtelung (*abort*), oder sie versucht, an der aktuellen Stelle mit der Parsierung fortzufahren (*resume*), was das Verwerfen des illegalen Symbols beinhalten muß, da sonst sofort der gleiche Syntaxfehler erneut auftritt.

Im Folgenden werden alle drei Vorgehensweisen anhand einer weiteren Eingabe zu dem obigen Beispiel aus Abbildung 8.1 näher beleuchtet. Ziel der Untersuchung ist es herauszufinden, welche Informationen nötig sind, um sich für eine weitere Vorgehensweise entscheiden zu können.

Abbildung 8.1 zeigt die Situation, nachdem bereits die Eingabe `2+3` parsiert worden ist. Folgt als nächstes in der Eingabe `*`, steigt die Erkennung in den `Many`-Knoten ab und aktiviert das `Seq`-Objekt (siehe Abbildung 8.2). Dessen erstes Element akzeptiert den Stern, und danach versucht das `Token`-Objekt, eine Zahl zu erkennen.

Folgt in dieser Situation als Rest der Eingabe `x9;` (also insgesamt `2+3*x9;`), erkennt die `Token`-Instanz das `x` als illegales Symbol, und ein Syntaxfehler liegt vor. Da das x nirgendwo sonst sinnvoll erkannt und die folgende Zahl vom `Token` akzeptiert werden kann, ist in diesem Beispiel die beste Fehlererholungsstrategie, das `x` zu verwerfen und lokal die Erkennung der `Token`-Instanz zu wiederholen (*retry*).

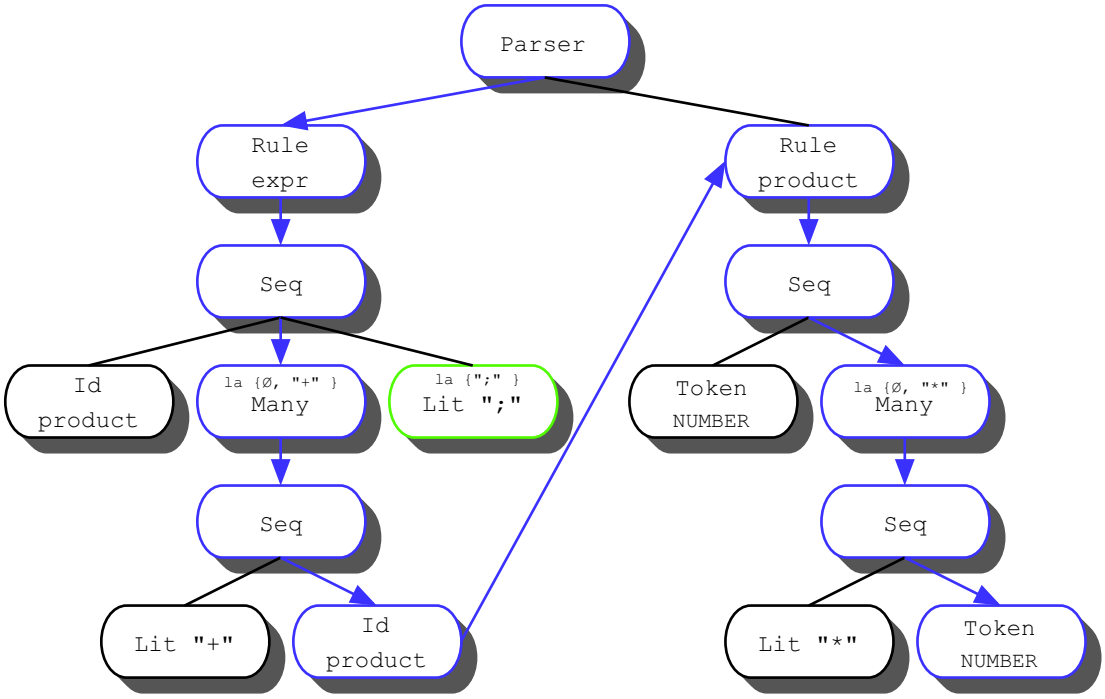

Abbildung 8.2: Die Situation nach `2+3*`

Folgt aber als Rest der Eingabe ein Semikolon (also insgesamt `2+3*;`), sollte eine ideale Fehlererholung hier die aktiven Parsierungsmethoden bis zu der Methode der Sequenz der `expr`-Regel abbrechen (*abort*) und dort mit dem nächsten Element mit der Erkennung des Semikolons fortfahren (*resume*). Die beiden Fehlersituationen sind sich sehr ähnlich, da in beiden Fällen nach dem Stern ein illegales Symbol folgt. Der Unterschied ist, daß das Semikolon im Gegensatz zum `x` von einem weiter außen aktiven Knoten akzeptiert wird. Die Entscheidung für eine solche Erholung kann aber nicht lokal in dem `Token`-Objekt getroffen werden. Zur Entscheidungsfindung wurden hier die Position der Sequenz der `expr`-Regel und die lookahead-Menge des nächsten Elements der Sequenz verwendet.

Folgt als weiteres Beispiel `*9;` (also insgesamt `2+3**9;`), ist dies ein weiteres Beispiel für die Wiederholung einer Erkennung. Die `Token`-Instanz kann den zweiten Stern nicht akzeptieren. Weiter außen paßt aber der Stern zum lookahead des `Many`-Knotens, und die Parsierung kann mit dessen Erkennung fortfahren. Aber auch diese Situation ist nicht lokal in dem `Token`-Knoten zu klären.

Allgemein wird eine ausschließlich lokale Entscheidung über die Art der Fehlerbehandlung grundsätzlich schlechter sein, als wenn eine globalere Sicht berücksichtigt wird. Zur Entscheidungsfindung sind zumindest die Zustände (Position einer Sequenz, lookahead-Mengen von Knoten bzw. deren Elemente, ...) aller aktiven oder noch zu aktivierenden Knoten nötig. Eine rein lokale Fehlererholung wird kaum sinnvolle Ergebnisse liefern.

8.2 Ein erster Ansatz – error() und recover()

In diesem Abschnitt wird ein erster Ansatz zur automatischen Fehlerbehandlung innerhalb eines von *oops* erzeugten Parsers geschildert. Dieser Ansatz wurde von mir allerdings zugunsten einer meiner Ansicht nach besseren und allgemeineren Technik verworfen. Ich schildere den Ansatz dennoch kurz, um meine Arbeit zu dokumentieren und um zu erklären, wieso ich den zweiten Ansatz bevorzuge. Der Ansatz stammt aus einer frühen Version von *oops*. Daher sind die Methodenköpfe einiger Methoden verschieden zu denen der aktuellen Version.

8.2.1 Die Idee

In einem Parser-Baum treibt die Methode `parse()` die Parsierung rekursiv voran. Trifft ein Knoten des Parser-Baums (z.B. `Lit` oder `Token`) auf ein unpassendes Eingabesymbol, so löst er eine `ParseException` aus. Zur Fehlerbehandlung wird der Programmtext innerhalb der verschiedenen `parse()`-Methoden in einen `try/catch`-Block eingebettet, der bei `catch` auf eine `ParseException` fängt.

Das `Goal`-Interface beinhaltet eine Methode `error()`:

```
int error (Node sender, Scanner scanner, Parser parser,
           ParseException pe) throws IOException ;
```

Innerhalb der `catch`-Blöcke der Knoten wird beim assoziierten `Goal`-Objekt `error()` aufgerufen, wobei der Knoten sich selbst als `sender` mitschickt. Ein Aufruf von `error()` signalisiert dem `Goal`, daß bei der Parsierung ein Fehler aufgetreten ist. Eine `Goal`-Instanz weiß durch die bereits erfolgten `shift()`-Meldungen, welcher Teil der Erkennung schon stattgefunden hat bzw. analog, wo der Fehler aufgetreten ist.

Der Rückgabewert von `error()` ist nun entscheidend für die Art der Erholung in dem Knoten des Parser-Baums. Hier als Beispiel die Methode `parse()` der Klasse `Seq`. Je nach Rückgabewert von `error()` kann es zu vier Arten von Fehlerbehandlungen kommen. `RESUME` setzt die Erkennung an der gleichen Stelle fort, `RETRY` wiederholt die Erkennung der ganzen Sequenz, `RESUME_NEXT` fährt mit der Erkennung hinter dem aktuellen Element der Sequenz fort, und `ABORT` beendet die Erkennung der Sequenz.

```
static public final int RESUME = 0, RETRY = 1, ABORT = 2, RESUME_NEXT = 3;

protected Vector nodes = new Vector();  // sequence of subtrees.
protected int cur;                      // current position in sequence

public void parse (Scanner scanner, Goal goal, Parser parser)
                  throws IOException {
    int n;
    recover:
        for (n = 0; n < nodes.size(); ++ n)
            try {
                ((Node) nodes.elementAt(n)).parse(scanner, goal, parser);
            } catch (ParseException pe) {
                cur = n;                // needed in recover()
                switch(goal.error(this, scanner, parser, pe)) {
                    case RESUME:      n--; continue recover;
                    case RESUME_NEXT: continue recover;
                    case RETRY:       n = -1; continue recover;
                    case ABORT:       return;
                    default:          throw pe;
                }
            }
}
```

Das `Goal` kann anhand des Rückgabewerts von `error()` die Art der Fehlerbehandlung beim Sender bestimmen. Die verschiedenen Klassen der Objekte des Baums definieren analog zu `Seq` mögliche Rückgabewerte für `error()` und assoziieren mit den Werten unterschiedliche Fehlerbehandlungen.

Das `Goal`-Objekt erhält über die Argumente von `error()` bzw. über `shift()`-Aufrufe alle nötigen Informationen, um eine Entscheidung fällen zu können. Es kann zum Beispiel bei einem Fehler während der Parsierung einer Sequenz über `scanner` in der Eingabe weiterlesen und durch den

anschließenden Rückgabewert `RESUME` Symbole in der Eingabe verwerfen. Durch `RESUME_NEXT` kann das nicht erkannte Symbol (bis auf ein fehlendes `shift()` gegen die `Goal`-Instanz) synthetisiert werden.

Die Fehlerbehandlung innerhalb von `error()` ist durch den Entwickler ohne eine Änderung der Grammatik frei programmierbar. Ziel war aber eine automatische Fehlerbehandlung. Die Entscheidung über die Art der Fehlerbehandlung sollte – wenn überhaupt – nur eine Option für den Entwickler sein.

Die Klasse `Node` schreibt als Oberklasse die Methode `recover()` für die Knoten des Parser-Baums vor:

```
public int recover (Scanner scanner, Parser parser, ParseException pe)
```

Damit ist eine automatische Fehlerbehandlung einfach. `GoalAdapter` und `GoalDebugger` implementieren den Körper von `error()` durch die Zeile

```
return sender.recover(scanner, parser, pe);
```

und lassen so `recover()` der Knoten des Parser-Baums die Art der Fehlerbehandlung bestimmen. Alle Klassen des Parser-Baums bestimmen in `recover()` eine möglichst sinnvolle Strategie zur Fehlerbehandlung. Stammen die `Goal`-Klassen zum Beispiel von `GoalAdapter` ab, braucht der Entwickler im Normalfall keine Zeile Programmtext für die Fehlerbehandlung implementieren, und die automatische Fehlerbehandlung steht.

8.2.2 Stopsymbole

In vielen Sprachen sind einige Literale wichtige Bestandteile der Sprache, welche sich sehr gut als Punkte zur Fehlerbehandlung eignen und keinesfalls verworfen werden sollten. Beispiel: Die schließende geschweifte Klammer als Abschluß eines Anweisungsblocks oder ein Semikolon zum Trennen oder Terminieren von Anweisungen.

Solche Literale nannte Wirth ([Wir86]) Stopsymbole bzw. Ammann ([Amm78]) “global key symbols”. In der Fachliteratur wird die Suche nach einem Stopsymbol zur Fehlerbehandlung auch “panic mode” genannt.

Wegen der Existenz der Stopsymbole habe ich die Eingabe für *oops* probeweise doch um eine Extrasyntax erweitert: Literale, welche anstelle von doppelten mit einfachen Anführungszeichen in der Grammatik spezifiziert werden, werden damit als Stopsymbole oder besser als Stop-Literale markiert.

Als Beispiel auszugsweise die Grammatik einer Sprache, in der Anweisungen durch ein Semikolon als Stop-Literal zu trennen sind:

```
...
stmts :  stmt {[  ';' stmt ]} ;
stmt  :  "if" ... | "while" ... | ... ;
...
```

Tritt während der Parsierung eines Programms über der Grammatik ein Fehler auf, soll spätestens beim nächsten Semikolon eine Erholung stattfinden. Das Semikolon wird daher in einfachen Anführungszeichen als Stopsymbol markiert und wird daher nie verworfen.

8.2.3 Die recover()-Methoden der Klassen

Da nun die Idee der Stopsymbole bekannt ist, gehe ich in diesem Abschnitt exemplarisch auf die `recover()`-Methode der `Seq`-Klassen ein.

Die Methode `recover()` prüft das aktuelle Eingabesymbol gegen den lookahead des aktuellen Elements der Sequenz, gegen den lookahead des nächsten Elements in der Sequenz, gegen den eigenen

lookahead und dann gegen die eigene follow-Menge und liefert dementsprechend als Resultat `RESUME`, `RESUME_NEXT`, `RETRY` oder `ABORT`. Trifft keiner der Fälle zu, wird bei einem Stopsymbol in der Eingabe die aktuelle `ParseException` erneut ausgelöst. Ansonsten wird das Symbol verworfen, ein neues wird vom Scanner angefordert, und die Tests beginnen von vorne. Die follow-Menge und die Markierung von Symbolen als Stopsymbole sind die nicht lokalen Informationen, welche die Fehlererholung außerhalb des `Seq`-Knotens stattfinden lassen.

```
public int recover (Scanner scanner, Parser parser, ParseException pe)
                   throws IOException {
    Set symbol = scanner.tokenSet(); // aktuelles Eingabesymbol
    for(;;) {
        if(symbol != null &&
           ((Node) nodes.elementAt(cur)).lookahead.matches(symbol))
            return RESUME;
        if(symbol != null && cur+1 < nodes.size() &&
           ((Node) nodes.elementAt(cur+1)).lookahead.matches(symbol))
            return RESUME_NEXT;
        if(symbol != null && lookahead.matches(symbol))
              return RETRY;
        if(symbol != null && follow.matches(symbol))
            return ABORT;
        if(symbol != null && parser.isStopSymbol(symbol))
            throw pe;
        if(!scanner.advance())
            throw new RuntimeException("EOF while error recovery");
        symbol = scanner.tokenSet();
    }
}
```

Man beachte, daß `cur` in dem `catch`-Block der `parse()`-Methode gesetzt wurde (siehe oben) und daher hier verwendet werden kann. Die eine `Parse`-Instanz im Parser-Baum kennt alle Stopsymbole und kann mit `isStopSymbol()` dahingehend befragt werden.

8.2.4 Fazit erster Ansatz

Möglicherweise möchte der Anwender die Fehlerbehandlung punktuell deaktivieren können. Das ist relativ einfach zu erreichen, indem die `error()`-Methode des `Goal`-Objekts so implementiert wird, daß dort die als Parameter empfangene `ParseException` erneut geworfen wird.

Alternativ könnte eine zusätzliche Syntax für die Grammatiken eingeführt werden, welche Knoten mit oder ohne Fehlerbehandlung erzeugt. Ist keine Fehlerbehandlung erwünscht, wird die `ParseException` entweder nicht abgefangen oder neu ausgelöst, und das `Goal` erhält keine `error()`-Meldung.

Selbst wenn der Anwender nicht in die automatische Fehlerbehandlung eingreifen möchte, muß er zumindest über Vererbung `error()` in den `Goal`-Klassen implementieren und dort `recover()` beim Sender aufrufen.

Pro Klasse der Knoten eines Parser-Baums gibt es verschiedene als Klassenvariablen definierte Namen für Rückgabewerte der Methode. Die Namen sind mit einem Verhalten bezüglich der Fehlerbehandlung assoziiert und sind pro Klasse definiert. Das macht die Anwendung etwas unübersichtlich. Die Namen sollten in `Node` mit einem als Konvention dokumentierten Verhalten für alle Klassen vordefiniert werden.

Nach einer erfolgreichen Erholung wird der nächste Fehler sofort wieder berichtet. Neue Fehler sollten wie bei *yacc* im Regelfall nach drei erfolgreich und in Folge erkannten Symbolen berichtet werden.

Dieser Ansatz der Fehlerbehandlung wurde nur mit `Goal`-Aktionen getestet. Er ist nicht ohne zusätzliche Implementierungsarbeit für alle, freiwählbare Aktionsarten einzusetzen und stellt daher keine allgemeine Lösung dar.

Die Entscheidung aufgrund der follow-Menge die Parsierung nach dem aktuellen Knoten fortzufahren, kann falsch sein. Die follow-Menge ist eine Vereinigung von Mengen. Je nach Aktivierung wäre eigentlich nur eine Teilmenge der kompletten follow-Menge für die Entscheidung zu beachten.

Tritt während der Parsierung einer lokalen Sequenz ein Stopsymbol als fehlerhaftes Extrasymbol in der Eingabe auf, wird dieses nicht verworfen, und die Sequenz wird fälschlicherweise abgebrochen. War dieses Symbol als einziger Fehler zuviel in der Eingabe, kann dieses Stopsymbol die Erkennung komplett in die Irre führen.

Wirth hat zwar die Verwendung der Stopsymbole aufgezeigt, aber er gibt keine Strategien an, wie man überhaupt die Stopsymbole einer Grammatik findet. Alle Literale zu verwenden, ist sicherlich kontraproduktiv, keine zu verwenden auch. Das Auffinden der Stopsymbole bleibt damit dem Gefühl und der Erfahrung des Entwicklers eines Compilers überlassen.

Das Verhalten der Fehlerbehandlung wird pro Knoten des Parser-Baums nur aufgrund der lokalen Informationen des Knotens bestimmt. Dies kann für eine gute Entscheidung zu wenig sein. Zumindest durch die Stopsymbole kommt etwas globalere Information hinzu. Könnte man aber die Informationen aller aktiven bzw. gegebenenfalls noch zu aktivierenden Knoten in die Entscheidungsfindung mit einbeziehen, bräuchte man die Stopsymbole nicht mehr. Aufgrund der zusätzlichen Information sollte der Parser sich wesentlich besser erholen können, und die Grammatik hätte keine Extrasyntax.

Die Implementierung dieses Ansatzes war aber wichtig. Ich bekam dadurch ein Gefühl für die verschiedenen Situationen und ihre Probleme. Letztendlich ist die lokale Entscheidung pro Knoten aber zu schwach, und eine globalere Sicht der Dinge fehlt.

8.3 Ein zweiter Ansatz

8.3.1 Die Idee

Wie der erste Ansatz gezeigt hat, reichen die lokalen Entscheidungen der Knoten für eine gute Fehlerbehandlung nicht aus. Auch die Stopsymbole genügen als einzige globalere Information nicht. Besser wäre eine Technik, in der die aktiven Methoden bzw. für *oops* die Knoten des Parser-Baums über das weitere Fortgehen in der Fehlerbehandlung abstimmen können.

Daher beruht die zweite und in *oops* realisierte Technik auf einer Kette von Objekten, welche die momentane Aufrufverschachtelung der `parse()`-Methoden modelliert und damit alle aktiven Knoten erreichen kann. Abbildung 8.3 zeigt für die blaue Aufrufverschachtelung des bekannten Beispiels die Objekte zur Modellierung der Aufrufverschachtelung und deren Verkettung in Rot.

Bei einem Parsierungsfehler wird in Form von Methodenaufrufen zurück von innen nach außen entlang der Kette und damit entlang der Aufrufverschachtelung gefragt, ob ein Knoten weiter außen das aktuelle Eingabesymbol akzeptiert bzw. dort eine Fehlererholung durchführen möchte. Wenn ja, so wird die Methodenaufruf-Verschachtelung bis zu diesem Knoten abgebrochen, welcher daraufhin mit der Erkennung fortfahren kann. Diese Technik und ihre Programmierung wurde in [Sch99a] erläutert.

Mit Hilfe dieses Ansatzes kann eine auf globaler Information beruhende, mächtige und automatische Fehlerbehandlung umgesetzt werden, ohne daß die Grammatik durch zusätzliche Syntax für die Fehlerbehandlung verunreinigt wird.

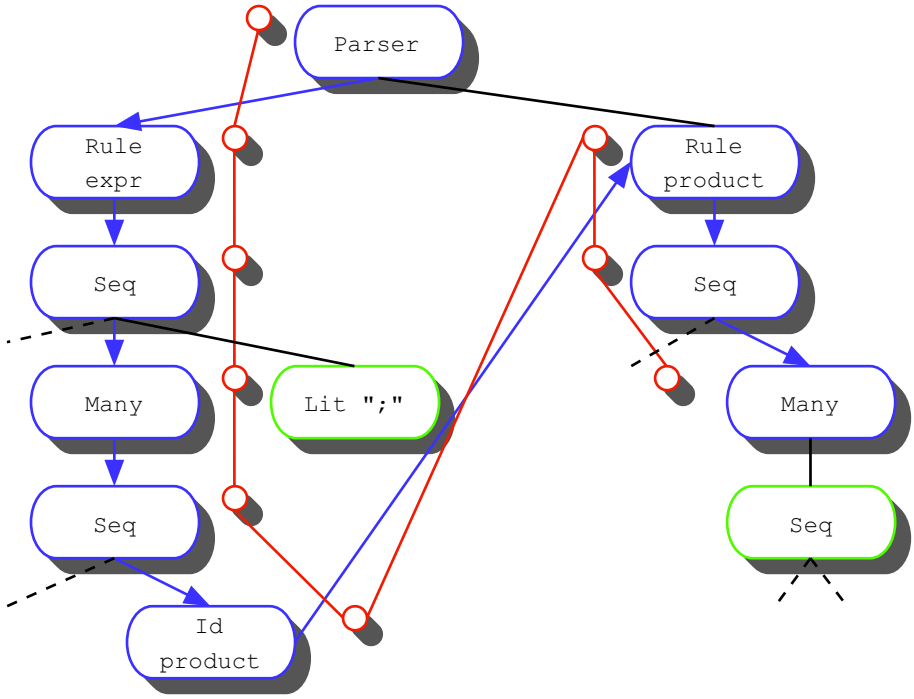

Abbildung 8.3: Die Modellierung der Aufrufverschachtelung.

8.3.2 Activation

Das `oops.parser`-Paket sammelt alle Klassen, deren Instanzen einen Baum als Repräsentation einer Grammatik bzw. als Parser über der Grammatik darstellen können. Die abstrakte Klasse `Activation` des gleichen Pakets bildet die Achse zur Fehlerbehandlung in einem Parser-Baum. `Activation` stammt von `Throwable` ab und kann damit analog zu einer Exception verwendet, d.h. ausgelöst und abgefangen, werden:

```
package oops.parser;

public abstract class Activation extends Throwable {
    protected final Activation caller;
    protected Activation (Activation caller) { this.caller = caller; }
```

In einem *oops*-Parser wird während der Parsierung die Aufrufverschachtelung der `parse()`-Methoden durch eine Kette von `Activation`-Objekten modelliert. Pro Aktivierung einer `parse()`-Methode wird in dem Körper der Methode eine `Activation` oder besser eine Instanz einer Unterklasse von `Activation` erzeugt und bei einem rekursiven Aufruf von `parse()` als Argument übergeben. Bei der Erzeugung einer `Activation` wird das `Activation`-Objekt der `parse()`-Methode, welche die aktuelle `parse()`-Methode aufgerufen hat, als `caller`-Verweis hinterlegt. Dadurch bilden während der Parsierung die `Activation`-Objekte die bereits beschriebene Kette.

Die Methode `handle()` löst die Empfänger-`Activation` der Methode als `Throwable` aus. Durch einen entsprechenden `try/catch`-Block wird damit die zu der `Activation` gehörende `parse()`-Methode wieder aktiv, und die Aufrufverschachtelung wird bis zu dieser Methoden-Aktivierung abgeschnitten. Als Argument zu `handle()` kann ein Informationsobjekt hinterlegt werden, welches später per `info()` erfragbar ist:

```
    private Object info;
    protected final void handle (Object info) throws Activation {
        this.info = info;
```

```
            throw this;
        }
        protected final Object info () { return info; }
```

Tritt ein Eingabefehler in der Parsierung auf, wird innerhalb der Fehlerbehandlung von `parse()` die zugehörige `Activation` (und optional alle anderen `Activation` der Kette) durch Aufruf der Methode `up()` gefragt, ob einer der aktiven Knoten mit der Parsierung fortfahren möchte. Die default-Implementierung fragt die `Activation` des weiter außen aktiven Knotens:

```
        protected Object up (Object info) throws Activation, NullPointerException {
            return caller.up(info);
        }
```

Pro Knotenklasse wird in einer Unterklasse von `Activation` die Methode `up()` sinnvoll überschrieben. Optional kann ein Objekt entlang der Kette übergeben werden. Entscheidet die `Activation` einer der aktiven Knoten, daß der Knoten mit der Parsierung fortfährt und somit der Punkt der Fehlererholung gefunden ist, durchbricht diese per `handle()` die Aufrufverschachtelung.

Die Knoten eines Parsers befragen zur Laufzeit übergeordnete Knoten durch `canUse()`, ob einer von ihnen das aktuelle Symbol in der Eingabe akzeptieren würde und mit diesem in der Erkennung fortfahren könnte:

```
        protected boolean canUse() { return caller.canUse(); }
```

Tritt ein Eingabefehler auf, sollte dem Anwender die Menge der aktuell akzeptablen Symbole ausgegeben werden. Die Berechnung der Menge geschieht auch durch die `Activation`-Kette, und zwar durch `expect()`. Pro Knoten des Baums berechnet die Knoten-spezifische Unterklasse von `Activation` die für den Knoten akzeptablen Symbole und fügt diese der Menge `expect` hinzu. Je nach Art des Knotens ruft dieser `expect()` wiederum weiter nach außen auf:

```
        protected void expect(Set expect) {
            if (caller != null)
                caller.expect(expect);
        }
    }
```

Die Berechnung der Menge der momentan akzeptablen Symbole soll noch einmal an dem bereits bekannten Beispiel erläutert werden und führt damit exemplarisch mögliche Methodenaufrufe entlang der `Activation`-Kette vor. Hat der *oops*-Parser für die einfachen arithmetischen Ausdrücke die Eingabe `2+3` erkannt, stellt sich die Aktivierungssituation des Parser wie in Abbildung 8.4 gezeigt dar. Der momentan aktive Knoten ist `Many` der `product`-Regel.

Folgt nun ein illegales Symbol, fordert das `Many`-Objekt durch Aufruf von `expect()` mit einer leeren Anfangsmenge als Argument die `Activation`-Instanz ihrer Aktivierung zur Berechnung der Gesamtmenge der an dieser Stelle erlaubten Symbole auf. Die `Activation` addiert die Symbole der lookahead-Menge des zugehörigen `Many`-Objekts — hier den Stern — zu der Gesamtmenge. Da der `Many`-Knoten auch die leere Eingabe akzeptiert und damit nicht zwingend erkannt werden muß, können weiter außen aktive Knoten das nächste Symbol — wie zum Beispiel ein Plus — akzeptieren. Daher fordert die `Activation` ihren Vorgänger in der Kette durch Aufruf von `expect()` mit der Gesamtmenge als Argument zur Erweiterung der Menge auf, was in Abbildung 8.4 durch die roten Pfeile symbolisiert wird.

Die Mengenberechnung erreicht schließlich die `Activation` des `Many`-Knotens der `expr`-Regel. Diese fügt das Plus der Gesamtmenge hinzu und fordert analog die `Activation` des übergeordneten `Seq`-Objekts zur Erweiterung der Menge auf. Dort endet die Mengenberechnung, da nach dem `Many` in der Sequenz das `Lit`-Objekt für das Semikolon definitiv folgen muß. Da das `Many` optional zu erkennen ist, wird der lookahead des `Lit`-Objekts, also das Semikolon, zu der Gesamtmenge addiert,

und in der nun berechneten Menge befinden sich der Stern, das Plus und das Semikolon als Menge der akzeptablen Symbole nach `2+3`.

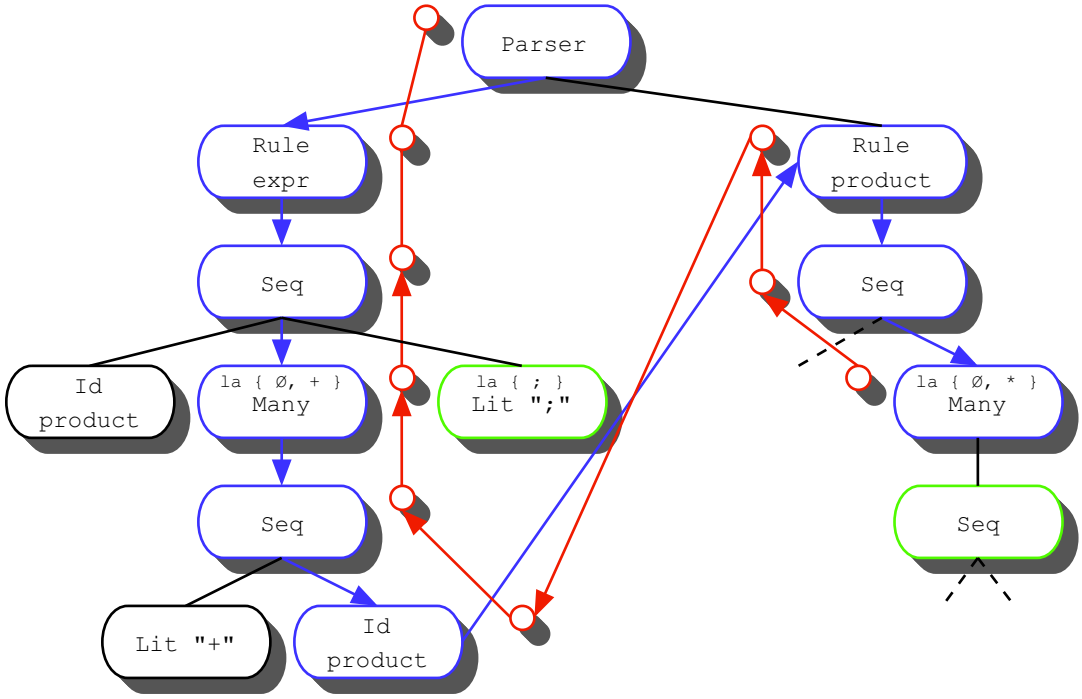

Abbildung 8.4: Die Berechnung der akzeptablen Symbole.

8.3.3 Activation-Klassen und Parsierung der Knoten

Ausgehend von der Modellierung der `parse()`-Aktivierungen durch `Activation`-Instanzen implementiert jede mögliche Klasse eines Parser-Baums eine eigene Unterklasse von `Activation`. Diese entscheidet in `up()` über eine sinnvolle und automatische Fehlerbehandlung und implementiert `canUse()` zum Test der Akzeptanz des aktuellen Symbols. Damit kann ein momentan aktiver Knoten entlang der Kette alle aktiven Elemente der Parsierung zur Entscheidungsfindung heranziehen und die, wie der erste Ansatz gezeigt hat, notwendige globale Information kommt ins Spiel.

Im Folgenden werde ich exemplarisch die `Activation`-Implementierung einiger Klassen aufzeigen:

Parser

Pro *oops*-Parser existiert eine Instanz der Klasse `Parser` als Wurzel des Parser-Baums. Die Klasse stellt für alle Knoten bzw. alle Klassen des Baums Methoden für Fehlermeldungen und zum Umgang mit dem Scanner zur Verfügung.

Ein `Parser` sammelt zum Beispiel Methoden, um das aktuelle Symbol des Scanners oder das Ende der Eingabe zu erfragen. Instanzen fast aller Klassen greifen über diese Methoden auf die Informationen des Scanners zu. Alternativ hätte sonst ein Verweis auf das `Scanner`-Objekt Argument zu `parse()` sein können:

```
public class Parser extends Node {
    ...
    protected transient Scanner scanner;
    protected boolean advance() throws IOException {
        return scanner.advance();
    }
    protected boolean atEnd() {  return scanner.atEnd(); }
```

```
        protected Set tokenSet()  {
            ...
            Object o = scanner.symbol();
            ...
            return (Set) o;
        }
```

Die Methode `error()` der Klasse `Parser` wird bei einem Eingabefehler aus `parse()` der Objekte der verschiedenen Klassen aufgerufen. So müssen nicht alle Knoten eine eigene Fehlermeldung erzeugen, und der Text ist für alle Knoten gleich. Inhalt der Fehlermeldung ist unter anderem die durch die `Activation`-Kette berechnete Menge der akzeptablen Eingabesymbole. Außerdem können Entwickler von Compilern die Methode überschreiben und dadurch ihre eigene Art von Fehlermeldungen produzieren.

```
        protected int errorOk, numberOfErrors;
        protected final int NUMBER_OF_SHIFTS_AFTER_ERROR = 3;
        protected final void error(Node n, Activation activation, Object action) {
            if (errorOk == 0) {
                numberOfErrors++;
                Set expect = new Set();
                activation.expect(expect);
                expect.removeEmpty();
                Set got = tokenSet();
                System.err.println("syntax error at "+scanner+
                    ":\n  parse node: "+ n.getClass().getName()+ ", "+n+
                    "\n  got symbol "+
                    (got == null ? ""unknown symbol"" : got.toString(this))+
                    "\n  expected one of "+ expect.toString(this));
            }
            errorOk = NUMBER_OF_SHIFTS_AFTER_ERROR;
            error(action);
        }
        public final int numberOfErrors() { return numberOfErrors; }
        protected final void shift() { if (errorOk > 0) errorOk--; }
        protected void error(Object action) { }
        ...
    }
```

Eine erneute Fehlermeldung soll analog zu *yacc* in der Regel erst nach dem erfolgreichen Akzeptieren von drei Symbolen ausgegeben werden. Daher rufen `Lit` und `Token` beim Erkennen eines Symbols beim `Parser` die Methode `shift()` auf, welche den Zähler `errorOk` für die nächste Fehlermeldung herunterzählt.

Wie bereits geschildert, dient im Fehlerfall pro Knoten-Klasse eine lokale Unterklasse von `Activation` zur Entscheidungsfindung bzw. Mengenberechnung. Die Klasse `ParserActivation` als `Activation` einer `Parser`-Instanz interessiert sich in allen Fällen lediglich für das Ende der Eingabe:

```
    public class Parser extends Node {
        ...
        protected final Rule start;
        private boolean check;
```

```
public boolean parse(final Scanner scanner, TableFactory tf) throws
    ParseException, IOException {
    if (!check)
        throw new ParseException("You try to run an unchecked parser...");

    this.scanner = scanner;
    scanner.advance();

    class ParserActivation extends Activation {
        protected ParserActivation (Activation caller) { super(caller); }
        protected Object up (Object info) throws Activation {
            if (scanner.atEnd()) handle(info);
            return info;
        }
        protected boolean canUse() { return scanner.atEnd(); }
        protected void expect(Set expect) { expect.add(Set.getEOFSet()); }
    }
```

Durch `handle()` einer `Activation` und das Abfangen der `Activation` in dem zugehörigen Parser-Knoten wird der Knoten in der Parsierung wieder aktiv. Dies entspricht dem Synthetisieren von Symbolen in der Eingabe, so daß der Knoten wieder aktiv wäre. Die Synthetisierung von Symbolen endet garantiert über einem endlichen Programm. Pro Symbol wird maximal eine Folge von Symbolen synthetisiert, und spätestens am Dateiende endet dies, da der `Parser` seine Parsierung dann beendet.

Mit einer Instanz der lokalen `Activation`-Klasse als Repräsentant der Aktivierung von `parse()` im `Parser` startet die Parsierung:

```
    final Activation self = new ParserActivation(null);

    try {
        parse(self);
        if (!scanner.atEnd())
            throw new ParseException(scanner+": trash at end");
    } catch (Activation a) {
        if ( a != self) throw new ParseException(a.toString());
    }
    return numberOfErrors == 0;
}
```

Lit, Token

`Lit`- und `Token`-Instanzen erkennen Literale oder Token als Symbole. Die Objekte sind Blätter im Parser-Baum, und `parse()` benötigt daher für diese beiden Klassen keine eigene `Activation`-Klasse:

```
protected void shift(Object action, Parser parser) { }
public void parse(Parser parser, Activation caller, Object action) throws
    ParseException, IOException, Activation {
    for(;;) {
        final Set token = parser.tokenSet();
        if (lookahead.matches(token)) {
            parser.shift();
            shift(action, parser);
            parser.advance();       // advance scanner to next symbol
            return;
        }
```

```
                parser.error(this, caller, action);
                caller.up(this);
                parser.advance();           // discard symbol
            }
        }
```

Ist das aktuelle Symbol das gesuchte Symbol, werden beim `Parser` die Methode `shift()` zum Dekrementieren des Zählers für die nächstmögliche Fehlermeldung und intern die Methode `shift(Object,Parser)` für spätere Aktionen zu erkannten Symbolen aufgerufen, und wird abschließend der Scanner zum Vorrücken zum nächsten Symbol aufgefordert.

Paßt das aktuelle Symbol nicht zum `Lit`- oder `Token`-Knoten, wird durch `error()` im `Parser` eine Meldung über den Eingabefehler ausgegeben. An dieser Stelle könnten das illegale Symbol und weitere folgende Eingabesymbole verworfen werden, bis das aktuelle Symbol zum lookahead des `Lit`- oder `Token`-Knotens paßt. Dank der `Activation`-Kette kann aber eine globalere Sicht der Dinge zur Fehlerbehandlung herangezogen werden. Analog zum Beispiel der Mengenberechnung (Abbildung 8.4) erfragt der lokale Knoten durch Aufruf von `up()` gegen das Ende der Kette, ob einer der aktiven Knoten weiter nach außen das aktuelle Symbol akzeptieren und mit diesem in der Parsierung fortfahren könnte. Falls ja, soll die Methodenverschachtelung bis zu diesem Knoten abgeschnitten werden. Daher ruft die `Activation` des Knotens, der mit der Parsierung fortfahren möchte, `handle()` auf und aktiviert damit wieder die Parsierung des Knotens.

Ist keiner der aktiven Knoten zur Fehlererholung bereit, endet `up()`, und der Scanner wird zum Vorrücken zum nächsten Symbol aufgefordert, wodurch das illegale Symbol verworfen wird. Das Verwerfen von Symbolen endet spätestens am Ende der Eingabe, da sich die `Activation` der `Parser`-Instanz dafür interessiert.

Many

Im Fehlerfall kann ein `Many`-Knoten mit der Parsierung fortfahren, wenn das aktuelle Symbol Teil der lookahead-Menge ist. Eine `ManyActivation` ruft in diesem Fall in `up()` daher `handle()` auf. Ansonsten wird der `up()`-Aufruf entlang der Kette weiter nach außen durchgereicht und bringt damit die weiter außen liegenden Knoten und deren Information ins Spiel:

```
public class Many extends Node {
    protected Node node;
    ...
    public void parse(final Parser parser, Activation caller, Object action)
        throws ParseException, IOException, Activation {
        class ManyActivation extends Activation {
            protected ManyActivation (Activation caller) { super(caller); }
            protected Object up (Object info) throws Activation {
                if(lookahead.matches(parser.tokenSet()))
                    handle(info);
                return super.up(info);
            }
```

Paßt das aktuelle Symbol zum lookahead, liefert `canUse()` als Resultat `true`. Da das zugehörige `Many` aber ein optionaler Knoten ist und nicht erkannt werden muß, wird im anderen Fall die Frage weiter nach außen geleitet:

```
            protected boolean canUse() {
                return lookahead.matches(parser.tokenSet()) ?
                    true : super.canUse();
            }
```

Aus dem gleichen Grund wird in `expect()` nicht nur der lokale lookahead zur Menge der zu erwartenden Symbole addiert, sondern über die default-Implementierung der Methode weiter nach außen aufgerufen:

```
            protected void expect(Set expect) {
                super.expect(expect); expect.add(lookahead);
            }
        }
```

Als Repräsentant der aktuellen Aktivierung von `parse()` wird ein Objekt der lokalen Klasse erzeugt:

```
        final Activation self = new ManyActivation(caller);
```

Damit kann die Parsierung starten. Paßt das aktuelle Symbol zum lookahead, wird der Unterknoten zur Parsierung aufgefordert:

```
        for(;;)
            try {
                if(lookahead.matches(parser.tokenSet())) {
                    node.parse(parser, self, action);
                    continue;
                }
                if (caller.canUse()) return;
```

Paßt das aktuelle Symbol nicht zum lookahead, gibt es zwei Möglichkeiten: Zum einen kann ein Eingabefehler vorliegen. Zum anderen kann aber auch die Parsierung des `Many` korrekt zu Ende sein, und Oberknoten haben mit der Parsierung fortzufahren. Aus diesem Grund wird als Abbruchkriterium der Parsierung des `Many`-Objekts entlang der `Activation`-Kette durch `canUse()` erfragt, ob die Parsierung weiter außen fortfährt. Falls ja, endet die Parsierung des Knotens. Auf diesen Sachverhalt wird im Folgenden noch einmal näher eingegangen.

Im anderen Fall ist ein Eingabefehler aufgetreten. Eine Fehlermeldung wird ausgegeben, und die `Activation`-Kette wird per `up()` nach einem Punkt zur Fehlererholung gefragt. Endet `up()`, so war keiner der übergeordneten Knoten an einer Erholung interessiert, da er sonst durch `handle()` seiner `Activation` wieder aktiviert worden wäre. Das falsche Symbol wird verworfen, da der Scanner zum nächsten Symbol vorrückt:

```
                parser.error(this, self, action);
                caller.up(this);
                parser.advance();
            } catch (Activation a) {
                if(a!=self) throw a;
            }
        }
    }
```

Die einfache Strategie, die Parsierung eines Wiederholungsknotens zu beenden ([Wir86]), wenn das aktuelle Eingabesymbol nicht Teil der lookahead-Menge ist, beendet die Aktivierung des Knotens und kann damit ein zusätzliches und falsches Symbol innerhalb vieler Wiederholungen nicht verwerfen und die folgenden Wiederholungen nicht erfolgreich parsieren.

Daher sollte in diesem Fall der Knoten testen, ob das aktuelle Symbol ein illegales Symbol ist oder ob die Parsierung des Knotens zu enden hat, da ein Oberknoten das Symbol akzeptieren wird.

Der einfache Ansatz, als Abbruchtest die follow-Menge des Knotens zu verwenden, ist nicht für alle Situationen korrekt. Ein Beispiel:

```
start :   many "b" many "c" ;
many  :   {[  "m" ]} ;
```

Die follow-Menge des `Many`-Knotens als rechte Seite der Regel `many` beinhaltet die Literal-Symbole `b` und `c`. Nach Eingabe der Folge

```
m m b m m b m m c
```

würde beim Abbruchtest der zweiten Aktivierung des `Many` ein Test, ob das Symbol `b` in der follow-Menge enthalten ist, wahr liefern. Ein `b` kann an dieser Stelle nach dem `Many` aber nicht akzeptiert werden. Besser wäre es, wenn das `b` verworfen wird und `Many` mit der Parsierung fortfährt.

Aus diesem Grund wird, wie gezeigt, für den Abbruchtest des Wiederholungsknotens nicht die follow-Menge verwendet, sondern durch `canUse()` wird entlang der Aktivierung gefragt, ob die Knoten weiter außen das `b` an der jeweiligen Position der Parsierung akzeptieren würden. In dem Beispiel würde die `Activation` der übergeordneten `start`-Sequenz mit `false` antworten. Damit ist `b` als ein an dieser Stelle illegales Symbol erkannt, und dank der globalen Entscheidungsfindung wird das illegale Symbol lokal verworfen.

Seq

`Seq` als Sequenz von Knoten ist aufgrund von möglichen Elementen, welche die leere Eingabe akzeptieren, die mit Abstand schwierigste Klasse in der Parsierung, in der Fehlerbehandlung bzw. in der Mengenberechnung.

Die Instanzvariable `i` der lokalen `Activation`-Klasse `SeqActivation` ist `protected` deklariert und kann damit innerhalb der Methoden einer `SeqActivation`, aber auch innerhalb von `parse()` verwendet werden. `i` wird in `parse()` als Laufvariable über die Elemente der Sequenz (`nodes`) benutzt.

```
public class Seq extends Node {
    protected Vector nodes = new Vector();
    ...
    public void parse(final Parser parser, Activation caller, Object action)
        throws ParseException, IOException, Activation {
        class SeqActivation extends Activation {
            protected int i;
            protected SeqActivation (Activation caller) {  super(caller); }
```

Wird die `Activation` einer `Seq` durch `up()` gefragt, ob die Sequenz das aktuelle, illegale Symbol akzeptieren würde, kann es verschiedene Möglichkeiten geben. Ein Beispiel:

```
start :   "a" b [  "c" ] [  "d" ] "e";
b     :   "b" ;
```

Das illegale Symbol kann Teil der lookahead-Menge des Elements sein, welches nach dem aktuell aktiven Element in der Sequenz folgt. In diesem Fall sollte die Sequenz die Fehlererholung durchführen, d.h. an dieser Stelle mit der Parsierung fortfahren. In dem Beispiel entspräche das einer Eingabe `acde` mit dem Syntaxfehler bei `c` und der Erkennung des `c` in der Sequenz nach dem aktiven Aufruf der Regel `b`.

Die Eingabe `ade` zeigt einen anderen Fall auf. Hier paßt das `d` nicht zum lookahead des in der Sequenz nächsten Elements. Da dieses Element das `c` aber optional erkennt, also die leere Eingabe akzeptiert, ist auch das diesem nachfolgende Element der Sequenz zu untersuchen. Ganz allgemein sind alle Elemente bis zum ersten nicht optionalen Element zu untersuchen.

Akzeptieren alle nachfolgenden Elemente der Sequenz die leere Eingabe oder ist die Sequenz in der Parsierung des letzten Elements, so kann die Erkennung auch in einem Oberknoten fortfahren. Ein Beispiel für den zweiten Fall ist die Eingabe `ac` zur folgenden Grammatik:

```
start :   ab "c";
ab    :   "a" b ;
b     :   "b" ;
```

Die `Activation` einer Sequenz hat all diese Fälle zu beachten. Wie bereits beschrieben, dient die Variable `i` in `parse()` als Laufvariable über die Elemente und verweist in `up()` auf das Element nach dem aktiven Element der Sequenz:

```
        protected Object up (Object info) throws Activation {
            for (int j = i; j < nodes.size(); ++j) {
                Node node = (Node) nodes.elementAt(j);
                if (node.lookahead.matches(parser.tokenSet())) {
                    i = j;
                    handle(info); // do recover
                }
                if (!node.lookahead.matchesEmpty())
                    break;
            }
            return super.up(info);
        }
```

`up()` überspringt keine nicht optionalen Elemente einer Sequenz. Vor- und Nachteile dieser Strategie werden anhand eines Beispiels im nächsten Abschnitt erläutert.

In `canUse()` wird analog das aktuelle Symbol gegen die lookahead-Mengen der nachfolgenden Elemente bis zum nächsten, nicht optionalen Element der Liste getestet. Nur wenn der Rest der Sequenz komplett optional ist oder die Sequenz bereits am Ende ist, wird `canUse()` beim Oberknoten befragt. `expect()` erweitert analog die Menge der akzeptablen Symbole:

```
        protected boolean canUse() {
            for (int j = i; j < nodes.size(); ++j) {
                Node node = (Node) nodes.elementAt(j);
                if (node.lookahead.matches(parser.tokenSet()))
                    return true;
                if (!node.lookahead.matchesEmpty())
                    return false;
            }
            return super.canUse();
        }

        protected void expect(Set expect) {
            for (int j = i; j < nodes.size(); ++j) {
                Node node = (Node) nodes.elementAt(j);
                expect.add(node.lookahead);
                if (!node.lookahead.matchesEmpty())
                    return;
            }
            super.expect(expect);
        }
    }
```

Als Repräsentant der `Seq`-Aktivierung wird eine `SeqActivation` erzeugt:

```
            final SeqActivation self = new SeqActivation(caller);
```

Die eigentliche Parsierung läuft über die Instanzvariable `i` von `self` die Elemente der Sequenz ab. Ist das aktuelle Symbol Teil der lookahead-Menge des gerade betrachtenden Elements, wird `self.i` erhöht und dieses Element zur Parsierung aufgefordert:

```
            loop: for (self.i = 0; self.i < nodes.size(); )
                try {
                    Node node = (Node) nodes.elementAt(self.i);
                    if (node.lookahead.matches(parser.tokenSet())) {
                        self.i++;
                        node.parse(parser, self, action);
                        continue;
                    }
```

Ist das Eingabesymbol nicht Teil der lookahead-Menge des aktuellen Elements, kann dies an verschiedenen Möglichkeiten liegen: Zum einen kann das aktuelle Element optional sein, und das Symbol gehört zu einem folgenden Element der Sequenz. In diesem Fall sind die unpassenden, optionalen Elemente der Liste zu überspringen. Andererseits kann der Rest der Sequenz komplett optional sein, und das Symbol ist Teil eines anderen, der Sequenz nachfolgenden bzw. übergeordneten Knotens. Hier hat `parse()` einfach erfolgreich zu enden. Ist keiner der beiden ersten Fälle richtig, so liegt ein Eingabefehler vor.

Aus diesem Grund werden in einer zweiten `for`-Schleife bis zum nächsten, nicht optionalen Element der Sequenz die nachfolgenden Elemente gegen das aktuelle Symbol getestet.

```
                    int j;
                    for (j = self.i; j < nodes.size(); j++) {
                        Node n = (Node) nodes.elementAt(j);
                        if (n.lookahead.matches(parser.tokenSet())) {
                            self.i = j + 1;
                            n.parse(parser, self, action);
                            continue loop;
                        }
                        if (!n.lookahead.matchesEmpty())
                            break;
                    }
```

Ist das Ende der Sequenz erreicht — alle nachfolgenden Elemente waren optional — und reflektiert `canUse()` gegen die `Activation`-Kette, daß das Symbol zu einem anderen Knoten gehört, endet `parse()` erfolgreich:

```
                    if (j >= nodes.size() && caller.canUse()) return;
```

Ansonsten liegt ein Eingabefehler vor, und der bereits mehrfach vorgestellte Fehlercode kommt zur Ausführung:

```
                    parser.error(this, self, action);
                    self.up(this);
                    parser.advance(); // discard symbol
                } catch (Activation a) {
                    if (a != self) throw a;
                }
            return;
        }
    }
```

Auch hier werden wieder für den Abbruchtest und für die Fehlerbehandlung durch die `Activation`-Kette die globalen Informationen zur Entscheidungsfindung herangezogen.

Opt, Some

`Opt` und `Some` verhalten sich in `parse()` bzw. in den Methoden der lokalen `Activation`-Klasse recht analog zu `Many` und werden daher hier nicht näher erläutert.

Alt

Ein `Alt` testet in den Methoden der `Activation` das aktuelle Symbol gegen den eigenen lookahead und trifft dementsprechend Entscheidungen. Der Programmtext würde an dieser Stelle nichts Neues bieten.

Id, Rule

Eine `Id` verweist auf einen anderen Knoten (`Rule` oder `Token`) und ruft in `parse()` nur bei diesem `parse()` auf. Eine `Rule` ruft in `parse()` lediglich beim Knoten der rechten Seite `parse()` auf. An einer Fehlererholung nehmen beide daher nicht aktiv teil.

8.3.4 Typische Konstrukte und andere Beispiele

Im Folgenden wird anhand einiger Grammatiken und anhand von Syntaxfehlern in der Parsierung von Programmen über den Grammatiken die Funktionalität der automatischen Fehlerbehandlung getestet. Um den Fortschritt der Parsierung zu verfolgen, wird dazu aus der jeweiligen Grammatik ein Ablaufverfolgungs-Parser entsprechend Kapitel 6.2 (“Ablaufverfolgung als Aktion”) erzeugt.

Optionale Folgen, Folgen, Listen

In [Sch85] werden mehr oder weniger mechanische Plazierungen von `error` innerhalb einer Grammatik für *yacc* für die typischen Konstrukte einer Programmiersprache (optionale Folgen, Folgen und Listen) vorgeschlagen. Im Folgenden wird die Fehlerbehandlung für diese Konstrukte getestet.

Die Grammatik für eine optionale Folge von `y`, welche durch ein `z` abgeschlossen wird, sieht wie folgt aus:

```
x: {[ "y" ]} "z" ;
```

Die Ausführung eines legalen Programms zeigt noch einmal die Ausgabe der Ablaufverfolgung. Der Scanner erkennt jedes einzelne Zeichen als eigenes Symbol und ignoriert allen Zwischenraum:

```
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=.:$JARS/oops.jar:$JARS/lolo.jar
$ java oops.RunParser optionaleFolge.ser Scanner
y
   Rule x start, rule level 3
   Lit accepts y
y
   Lit accepts y
z
   Lit accepts z
^D
   Rule x end, rule level 3
$
```

Da die Folge von `y` optional ist, sollte an einem `z` die Erkennung wieder aufsetzen:

```
$ java oops.RunParser optionaleFolge.ser Scanner
x
   Rule x start, rule level 3
syntax error at line 1, "x":
  parse node: oops.parser.Seq, (  [{  "y" }] "z" ):
  got symbol "unknown symbol"
  expected one of { "z", "y"}
   Parser error
z
   Lit accepts z
^D
   Rule x end, rule level 3
```

Wie man sieht, wird `x` als Eingabefehler erkannt, der beteiligte Knoten textuell dargestellt, `y` und `z` als an dieser Stelle zu erwartende Symbole korrekt berechnet, und die Methode `error(Object)` des Parsers gibt in der Implementierung der Methode in dieser Unterklasse einfach einen Text aus. Ein `Goal`-Parser ruft an dieser Stelle zum Beispiel beim aktuellen `Goal` die `error()`-Methode des `Goal`-Interfaces auf.

Ein weiteres Beispiel illustriert, daß weitere Syntaxfehler erst nach drei erfolgreich erkannten Symbolen erneut berichtet werden:

```
$ java oops.RunParser optionaleFolge.ser Scanner
x
   Rule x start, rule level 3
syntax error at line 1, "x":
  parse node: oops.parser.Seq, (  [{  "y" }] "z" ):
  got symbol "unknown symbol"
  expected one of { "z", "y"}
   Parser error
y x
   Lit accepts y
   Parser error
y y y x z
   Lit accepts y
   Lit accepts y
   Lit accepts y
syntax error at line 3, "x":
  parse node: oops.parser.Many, [{  "y" }]:
  got symbol "unknown symbol"
  expected one of { "z", "y"}
   Parser error
   Lit accepts z
^D
   Rule x end, rule level 3
```

Folge

Die folgende Grammatik testet analog eine nicht optionale Folge von `y` mit abschließendem `z`:

```
x: {  "y" }  "z" ;
```

Ein falsches Symbol am Ende der Folge wird korrekt übergangen:

```
$ java oops.RunParser folge.ser Scanner
y y x z
   Rule x start, rule level 3
```

```
   Lit accepts y
   Lit accepts y
syntax error at line 1, "x":
  parse node: oops.parser.Some, {  "y" } :
  got symbol "unknown symbol"
  expected one of { "z", "y"}
   Parser error
   Lit accepts z
^D
   Rule x end, rule level 3
```

Ein falsches Symbol beendet den Wiederholungsknoten für `y` nicht und kann daher korrekt übergangen werden:

```
$ java oops.RunParser folge.ser Scanner
y y x y z
   Rule x start, rule level 3
   Lit accepts y
   Lit accepts y
syntax error at line 1, "x":
  parse node: oops.parser.Some, {  "y" } :
  got symbol "unknown symbol"
  expected one of { "z", "y"}
   Parser error
   Lit accepts y
   Lit accepts z
^D
   Rule x end, rule level 3
```

Ein falsches Symbol am Anfang der Folge wird korrekt übergangen:

```
$ java oops.RunParser folge.ser Scanner
x
   Rule x start, rule level 3
syntax error at line 1, "x":
  parse node: oops.parser.Seq, (  {  "y" }  "z" ):
  got symbol "unknown symbol"
  expected one of { "y"}
   Parser error
y z
   Lit accepts y
   Lit accepts z
^D
   Rule x end, rule level 3
```

Liste

Ein anderes, typisches Konstrukt einer Programmiersprache sind Listen. Hier eine durch Komma getrennte Liste von `y`:

```
x: "y" {[  "," "y" ]} "z" ;
```

Nach einem falschen Element in der Liste wird am nächsten Komma erholt:

```
$ java oops.RunParser liste.ser Scanner
y, x, y z
   Rule x start, rule level 3
   Lit accepts y
```

```
   Lit accepts ,
syntax error at line 1, "x":
  parse node: oops.parser.Seq, (  "," "y" ):
  got symbol "unknown symbol"
  expected one of {"y"}
   Parser error
   Parser error
   Lit accepts ,
   Lit accepts y
   Lit accepts z
^D
   Rule x end, rule level 3
```

Nach einem falschen Listentrenner wird auch am nächsten Komma erholt:

```
$ java oops.RunParser liste.ser Scanner
y; y, y z
   Rule x start, rule level 3
   Lit accepts y
syntax error at line 1, ";":
  parse node: oops.parser.Seq, (  "y" [{  (  "," "y" ) }] "z" ):
  got symbol "unknown symbol"
  expected one of {"z", ","}
   Parser error
   Parser error
   Lit accepts ,
   Lit accepts y
   Lit accepts z
^D
   Rule x end, rule level 3
```

Ein Fehler am Anfang der Liste:

```
$ java oops.RunParser liste.ser Scanner
x, y, y z
   Rule x start, rule level 3
syntax error at line 1, "x":
  parse node: oops.parser.Seq, (  "y" [{  (  "," "y" ) }] "z" ):
  got symbol "unknown symbol"
  expected one of {"y"}
   Parser error
   Parser error
   Lit accepts y
   Lit accepts ,
   Lit accepts y
   Lit accepts z
^D
   Rule x end, rule level 3
```

Ein Fehler am Ende der Liste:

```
$ java oops.RunParser liste.ser Scanner
y, y, x z
   Rule x start, rule level 3
   Lit accepts y
   Lit accepts ,
   Lit accepts y
```

```
   Lit accepts ,
syntax error at line 1, "x":
  parse node: oops.parser.Seq, (  "," "y" ):
  got symbol "unknown symbol"
  expected one of {"y"}
   Parser error
   Parser error
   Lit accepts z
^D
   Rule x end, rule level 3
```

canUse()

```
start :   many "b" many "c" ;
many  :   {[  "m" ]} ;
```

Die folgende Eingabe zeigt das bereits besprochene Beispiel, daß während der zweiten Parsierung des `Many`-Knotens `canUse()` und nicht die follow-Menge für das Abbruchkriterium genutzt werden muß und wird:

```
$ java oops.RunParser canUse.ser Scanner
b m b
   Rule start start, rule level 3
   Lit accepts b
      Rule many start, rule level 6
      Lit accepts m
syntax error at line 1, "b":
  parse node: oops.parser.Many, [{  "m" }]:
  got symbol {"b"}
  expected one of {"m", "c"}
      Parser error
c
      Rule many end, rule level 6
   Lit accepts c
^D
   Rule start end, rule level 3
```

Erholung in Sequenzen

Das nächste Beispiel diskutiert die Erholung innerhalb einer Sequenz. Eine `1` oder `2` als führendes Symbol führt zur Erkennung der gleichnamigen Symbolfolge. Allerdings besteht im zweiten Fall der Anfang der Sequenz aus einer Untersequenz:

```
start :    "1" one | "2" two;
one   :      "a" "b" "c"   "d" ;
two   :    ( "a" "b" "c" ) "d" ;
```

Die Eingabe

```
1 a d
```

erholt sich erst am Dateiende, da innerhalb der Sequenz zu `one` in der Fehlerbehandlung nur optionale Elemente übersprungen werden:

```
$ java oops.RunParser testSeq.ser Scanner
1 a d
   Rule start start, rule level 3
```

```
    Lit accepts 1
       Rule one start, rule level 6
       Lit accepts a
syntax error at line 1, "d":
  parse node: oops.parser.Seq, (  "a" "b" "c" "d" ):
  got symbol { "d"}
  expected one of { "b"}
       Parser error
^D
       Parser error
```

Bei der ebenfalls fehlerhaften Eingabe

```
2 a d
```

verhält sich die Fehlerbehandlung anders:

```
$ java oops.RunParser testSeq.ser Scanner
2 a d
   Rule start start, rule level 3
   Lit accepts 2
      Rule two start, rule level 6
      Lit accepts a
syntax error at line 1, "d":
  parse node: oops.parser.Seq, (  "a" "b" "c" ):
  got symbol { "d"}
  expected one of { "b"}
      Parser error
      Lit accepts d
^D
      Rule two end, rule level 6
   Rule start end, rule level 3
```

Im zweiten Beispiel ist die Sequenz für die Symbolfolge `a b c` das erste Element einer anderen Sequenz, welche nachfolgend das `d` erkennen möchte. Die `Activation` der inneren Sequenz ist beim `d` als Eingabefehler in `up()` nicht zur Fehlererholung bereit und fragt durch `super.up()` den äußeren Sequenz-Knoten. Dieser akzeptiert das `d` und löst durch `handle()` die Fehlererholung aus.

Das Beispiel suggeriert, daß in `up()` der `Activation` eines `Seq` besser alle nachfolgenden Elemente der Sequenz gegen das aktuelle Symbol getestet werden sollten. Dies ist aber ein Trugschluß. Zum Beispiel würde damit bei der Eingabe `2 a d b c d` das erste `d` die Parsierung beim `d` am Ende der Sequenz sich erholen lassen. Dies wäre aber falsch. Das `d` ist lediglich zuviel im Programm, und nach dem `d` kann die Parsierung bereits am `b` fortgesetzt werden.

Dieses Beispiel zeigt das typische Problem einer automatischen Fehlerbehandlung. Mit einer Vorschau von nur einem oder einer endlichen Anzahl von Symbolen, müßte eine ideale Fehlerbehandlung hellsehen können. Die Fehlerbehandlung wird also nicht für alle Fälle perfekt sein. Aber der Parser erholt sich, wenn teilweise auch später als nötig, und weiterer Programmtext wird erkannt.

Ich habe mich für diesen Ansatz in der `Activation` einer `Seq` entschieden, da dieser Ansatz das Beispiel `2 a d b c d` richtig erholt. Sollte das `d` nach dem `a` doch zum Ende der Sequenz gehören, wird spätestens am Symbol nach der Sequenz die Parsierung korrekt fortgesetzt. Damit ist dieser Ansatz meines Erachtens günstiger.

Lokal kann ein in *oops* versierter Anwender — wie in der Regel `two` vorgeführt — durch den Einsatz zusätzlicher Sequenzen — also durch den Einsatz von runden Klammern in der Grammatik — die Fehlerbehandlung etwas steuern. Aus diesem Grund werden in `node()` einer `Seq` die Elemente eines Elements, welches selbst eine `Seq`-Instanz ist, nicht in die `Seq` als Optimierung eingegliedert.

Eine kleine Sprache

Folgende Grammatik beschreibt eine kleine Programmiersprache, welche die Anweisungen `cmd1`, `cmd2` und einen Block von Anweisungen besitzt. Anweisungen werden in der Sprache durch Semikolons separiert:

```
start :   stmts ;
stmts :   stmt {[  ";" stmt ]} ;
stmt  :   "cmd1" | "cmd2" | block ;
block :   "{ " stmts "} " ;
```

Das Beispiel entspricht weitgehend dem Listen-Beispiel: Nach einem falschen Kommando oder nach einer falschen Separierung — zum Beispiel durch ein Komma — erholt sich der Parser am nächsten Semikolon. Aber auch eine schließende, geschweifte Klammer am Ende eines Blocks kann zur Erholung dienen:

```
$ java oops.RunParser stmts.ser Scanner
{ cmd1; cmd3, cmd3 }
   Rule start start, rule level 3
      Rule stmts start, rule level 6
         Rule stmt start, rule level 9
            Rule block start, rule level 12
            Lit accepts {
               Rule stmts start, rule level 15
                  Rule stmt start, rule level 18
                  Lit accepts cmd1
                  Rule stmt end, rule level 18
               Lit accepts ;
syntax error at line 1, "cmd3":
  parse node: oops.parser.Seq, (  ";" stmt ):
  got symbol "unknown symbol"
  expected one of {"cmd1", "{ ", "cmd2"}
               Parser error
               Parser error
               Parser error
               Parser error
            Lit accepts }
^D
            Rule block end, rule level 12
         Rule stmt end, rule level 9
      Rule stmts end, rule level 6
   Rule start end, rule level 3
```

8.4 Zeiten

In Kapitel 4 (“Ein Parser aus Objekten”) wird in Abschnitt 4.6 (“Zeiten”) die Performance von *oops*-generierten Parsern mit von anderen Parsergeneratoren erzeugten Parsern verglichen. Resultat der Messungen ist, daß *oops*-Parser einen kleinen Geschwindigkeits-Nachteil gegenüber den schnellsten — von *JavaCC* erzeugten — Parsern haben.

Ein Grund dafür könnte die Technik der Fehlerbehandlung sein. Die Erzeugung eines `Activation`-Objekts pro Aktivierung einer `parse()`-Methode ist nicht gänzlich zu vernachlässigen. Außerdem entscheidet ein Aufruf von `canUse()` gegen die `Activation`-Kette, ob zum Beispiel ein Wiederholungsknoten abbricht. Damit wird im Fall einer Fehleingabe eine Erholung in dem Wiederholungs-Knoten bzw. ein Substituieren eines fehlenden Symbols möglich. Dies ist in *ANTLR*-

bzw. *JavaCC*-Parsern nur durch mühsame Handarbeit, d.h. durch Schreiben von Java-Programmtext an allen nötigen Stellen (wenn man sie kennt), möglich.

Die Technik der automatischen Fehlerbehandlung ist sehr mächtig, kostet dafür aber etwas an Performance...

8.5 Fazit

Der aktuelle Ansatz der Fehlerbehandlung ist — im Gegensatz zum ersten — von der Aktions-Schnittstelle unabhängig. Ein `Parser` ruft im Fehlerfall die Methode `error(Object)` mit dem Aktions-Objekt der `parse()`-Methoden als Argument auf, welche als Template-Methode von Unterklassen überschrieben werden kann. So ruft in dieser Methode der `Parser` für `Goal`- bzw. der für `Reducer`-Aktionen die Methode `error()` des jeweiligen `Goal`- bzw. `Reducer`-Interfaces auf.

Wird die Aufrufverschachtelung während des rekursiven Abstiegs durch `Activation`-Objekte modelliert, kann entlang der Kette die jeweilige `Activation` zu der Situation im zugehörigen Knoten befragt werden. Dadurch kommt die für eine gute Fehlerbehandlung notwendige, globale Information der Zustände der aktiven Knoten ins Spiel.

Im Fehlerfall kann der momentan aktive Knoten alle anderen aktiven Knoten befragen. Damit kann jeder Knoten im Fehlerfall lokal ein illegales, von keinem anderen Knoten gewünschtes Zeichen verwerfen und mit dem nächsten Zeichen lokal in seiner Parsierung fortfahren (*resume*). Hat ein *yacc*-Parser seinen Zustandsstack erst einmal abgeschnitten, ist dessen Zustand nicht mehr wiederherzustellen.

Auch in der Fehlerbehandlung ist *divide & conquer* wiederzufinden. Instanzen einzelner Klasse als Unterklasse von `Activation` entscheiden über eine Fehlerbehandlung bezüglich des zugehörigen Knotens. Damit verteilt sich der Algorithmus wieder auf kleine Teilprobleme, welche leicht zu verstehen sind. Erst das Zusammenspiel der einzelnen `Activation`-Klassen zusammen mit dem Programmtext der `parse()`-Methoden führt zur komplexen und automatischen Fehlerbehandlung.

Da eine `Activation` analog zu einer Exception ausgelöst werden kann, werden mit diesem Trick die rekursiven Aufrufe von `parse()` gezielt durchbrochen, wenn die `Activation` entscheidet, daß der zugehörige Knoten nach einem Eingabefehler mit der Parsierung fortfahren und so der Punkt der Fehlererholung sein soll. Diese würde einem `setjmp()` in C entsprechen, ist aber gekapselt in Objekte und durch `try`/`catch`-Blöcke wesentlich einfacher und gewinnbringender zu nutzen.

Da im Fehlerfall durch die `Activation`-Kette alle aktiven Knoten befragt werden können und damit die lokale Sicht erweitert werden kann, sind die Stopsymbole aus dem ersten Ansatz nicht länger nötig. Damit werden Grammatiken nicht durch eine zusätzliche Syntax verunreinigt, welche der Anwender außerdem zusätzlich zu lernen hätte. Neben der automatischen Fehlerbehandlung war dies das zweite, wichtige Designkriterium für die Fehlerbehandlung in *oops*-Parsern.

Im ersten Ansatz konnte der Anwender in den Algorithmus der Fehlerbehandlung eingreifen. Ich habe mich im zweiten Ansatz und damit für *oops*-Parser dagegen entschieden. Der Anwender hätte die Technik des Eingreifens in die Erholung zu erlernen, nur um stellenweise eine etwas bessere Fehlerbehandlung zu erhalten. Das lohnt meines Erachtens den Aufwand nicht und hält *oops* einfach.

Wie weiter oben in diesem Kapitel gezeigt, kann für jeden Ansatz der automatischen Fehlerbehandlung ein Beispiel gefunden werden, so daß die Erholung nicht optimal ist. Der Parser findet den ersten Fehler und berichtet diesen an den Benutzer. Weitere Fehler werden möglicherweise nicht alle gefunden, da der Parser zu spät mit einer erfolgreichen Parsierung der Symbolfolge fortfährt. Dies ist für mich aber definitiv kein Nachteil. Das Programm ist fehlerhaft, und der Anwender erfährt dies durch Ausgabe der ersten Fehlermeldung. Daß er unter Umständen nur einen Teil der Fehler sieht, ist sicherlich kein Problem. Er wird das Programm ausbessern, und eine erneute Parsierung findet die weiteren Fehler. Lediglich eine von Hand sehr sorgfältig konzipierte Grammatik mit zusätzlicher Syntax für die

Fehlerbehandlung wird (möglicherweise) alle Fehler finden. Dadurch wird aber die Grammatik verunreinigt und meistens auch mehrdeutig. Ob eine derart von Hand sorgfältig konzipierte Grammatik wirklich alle möglichen Fehlersituationen abdecken kann, möchte ich bezweifeln.

9 Grammatik-Notationen, Erweiterung durch Klassen

oops ist eigentlich die Klassenbibliothek, auf der Bäume für Parser als Repräsentation von Grammatiken gebaut werden. Um die Grammatik zu prüfen, wird der Baum nicht von außen, zum Beispiel über ein Visitor-Entwurfsmuster untersucht, sondern der Baum selbst hat die Fähigkeit, sich zu prüfen. Ist der Baum erzeugt, ist es egal, zu welcher Grammatik er die Repräsentation darstellt bzw. in welcher Notation die repräsentierte Grammatik aufgeschrieben worden war. Die Grammatik kann in EBNF, aber auch in jeder anderen denkbaren Notation aufgeschrieben worden sein.

Die Klassenbibliothek, auf der Bäume für Grammatiken bzw. für Parser gebaut werden, ist aufgrund des Einsatzes der Objekt-Orientierung ganz natürlich erweiterbar. Neue Knoten-Klassen können als Repräsentant weiterführender Grammatikoperationen implementiert werden. Im Zusammenspiel mit den alten Knoten-Klassen sind mögliche Parser durch Instanzen der neuen Klassen schnell um die weiterführende Funktionalität erweitert.

9.1 Grammatik-Repräsentation

oops ist eine Klassenbibliothek aus mehreren Paketen, deren Instanzen eine Grammatik als Baum repräsentieren können. Je nach Klasse der verschiedenen Objekte des Baums kann dieser die Grammatik prüfen und — zusammen mit einem Scanner als Partner — Programme über der Grammatik parsieren, aber auch vom Anwender zu spezifizierende Aktionen während der Parsierung zu erkannten Teilen der Grammatik zur Ausführung bringen. Ist der Baum als Repräsentant einer Grammatik erst einmal gebaut, ist die Notation, in der die Grammatik aufgeschrieben wurde, nicht länger von Bedeutung.

Bislang wurden in dieser Arbeit Grammatiken in EBNF formuliert. Hier eine einfache EBNF-Grammatik, welche alle Operationen beinhaltet:

```
start : ( "a" "b" ) | [ "c" ] { "d" } {[ "e" ]} ;
```

Alternativ könnte die Notation der Grammatik die Operationen aber auch analog zu regulären Ausdrücken bzw. analog zu *JavaCC* formulieren. Die gleiche Beispiel-Grammatik sieht in dieser Notation wie folgt aus:

```
start : ( "a" "b" ) | "c"? "d"+ "e"* ;
```

Der Teil einer Grammatik links von einem Fragezeichen-Operator ist optional, der Teil links von einem Plus ist beliebig oft, aber mindestens einmal, und der Teil links von einem Stern beliebig oft zu erkennen. Ein | trennt Alternativen und hat einen höheren Vorrang als die Sequenz, und die Sequenz hat einen höheren Vorrang gegenüber Plus, Fragezeichen und Stern. Runde Klammern erzwingen Vorrang.

Alternativ kann die gleiche Grammatik auch in Java-Programmtext als Notation formuliert werden:

```
public static oops.parser.Parser grammar = new oops.parser.Parser (
    new oops.parser.Rule(
        new oops.parser.Id("start"),
        new oops.parser.Alt(new oops.parser.Node[] {
            new oops.parser.Seq(new oops.parser.Node[] {
                new oops.parser.Lit("a"), new oops.parser.Lit("b")
            }), // end of first alternative
            new oops.parser.Seq (new oops.parser.Node[] {
                new oops.parser.Opt (new oops.parser.Lit("c")),
```

```
                    new oops.parser.Some(new oops.parser.Lit("d")),
                    new oops.parser.Many(new oops.parser.Lit("e"))
                }) // end of second alternative
            }) // end of Alt, rhs of start
        ) // end of Rule start
    );
```

Die Beispiel-Grammatiken zu den drei Notationen bzw. auch zu jeder weiteren, denkbaren Notation resultieren in dem gleichen Baum zur Repräsentation der Grammatik. Ist der Baum erst erzeugt, ist die frühere Notation nicht weiter wichtig. Der Baum selbst ist die “Message”, und alle nötigen Algorithmen sind im Baum verteilt.

Beschränkungen von legalen XML-Dokumenten auf eine spezielle Untermenge von XML-Dokumenten werden durch eine *Document Type Definition* (DTD) beschrieben. Eine DTD ist also eine Grammatik, welche spezielle XML-Dokumente beschreibt. Auch eine DTD ist daher eine Grammatik-Notation und stellt eine weitere — oder besser die übliche — Alternative auf dem Gebiet von Grammatiken für XML-Dokumente dar.

Als Beispiel hier eine DTD, welche den Aufbau von Personenlisten als XML-Dokumente festlegt. Ein `personen`-Element darf beliebige viele `person`-Elemente und diese müssen wiederum genau ein `vorname`- und ein `nachname`-Element beinhalten. `person`-Elemente müssen die Attribute `persnr` und `geschlecht` besitzen. Das Attribut `chef` ist optional:

```
<?xml version='1.0' encoding="ISO-8859-1" ?>
<!ELEMENT personen             (person*)>
<!ELEMENT person               (vorname,nachname)>
    <!ATTLIST person   persnr      ID #REQUIRED>
    <!ATTLIST person   chef        IDREF #IMPLIED>
    <!ATTLIST person   geschlecht  (männlich|weiblich) #REQUIRED>
<!ELEMENT vorname              (#PCDATA)>
<!ELEMENT nachname             (#PCDATA)>
```

Ein `Alt`-Knoten erkennt genau eine seiner Alternativen. Eine DTD kennt aber noch andere dem `Alt` verwandte Operationen:

In einer DTD mit `IMPLIED` markierte Attribute von Elementen müssen nicht vorkommen, dürfen es aber jeweils genau einmal. Diese mit `IMPLIED` markierten Attribute stellen also Alternativen dar, von denen beliebig viele, aber jedes nur einmal, erkannt werden dürfen. Dabei ist die Reihenfolge der Attribute aber beliebig und wird nicht durch die DTD festgelegt. Dies alles ist mit einem `Alt`-Knoten nicht darstellbar. Alle möglichen Kombinationen der Attribute müßten mit den bisherigen Parser-Knoten einzeln und konfliktfrei beschrieben werden.

Alle mit `REQUIRED` markierten Attribute müssen vorkommen und können nicht ausgelassen werden. Aber auch hier ist keine Reihenfolge der Attribute vorgeschrieben. Auch diese Situation kann mit den bislang bekannten Knoten-Klassen nur mühselig durch Aufzählung aller Kombinationen repräsentiert werden.

Damit ist die Repräsentation einer DTD durch Instanzen der bekannten Knoten-Klassen nicht wirklich praktikabel. Es fehlen Klassen, welche die beiden Situationen im Baum elegant repräsentieren. Im nächsten Abschnitt wird daher gezeigt, wie einfach dank der Zerlegung der Parser in Instanzen von Klassen und dank der Vererbung die beiden fehlenden Operationen durch neue Klassen zu modellieren sind und wie einfach damit das Parser-System um eine neue und mächtige Funktionalität erweiterbar ist.

9.2 Extending EBNF

9.2.1 Die Idee, eine neue Grammatik-Notation

Die bislang vorgestellten Klassen eines Parser-Baums von *oops* sind in der Lage EBNF-Grammatiken zu repräsentieren. Wie Kapitel 6 (“Parser-Aktionen”) gezeigt hat, können die Knoten-Klassen in Unterklassen erweitert werden, und Instanzen der Unterklassen integrieren sich ganz natürlich in mögliche Bäume und bringen so eine neue Funktionalität in die Parser. Dies wurde anhand der Klassen `Parser`, `Rule`, `Lit` und `Token` vorgeführt.

Auch Klassen wie `Seq` oder `Alt` als Repräsentation einer Sequenz bzw. einer Alternative könnten durch Unterklassen erweitert werden, doch fehlte es hier bislang an einem Beispiel, wo die Erweiterung sinnvoll wäre.

Wie die DTD für die Personenliste gezeigt hat, gibt es aber Konstrukte, welche mit BNF und damit mit EBNF nur derart mühsam zu formulieren sind, daß man dies erst gar nicht versucht. Als Beispiel die Attribute eines XML-Elements, welche in beliebiger Reihenfolge aber jeweils nur einmal auftreten dürfen. Dabei sind einige der Attribute zwingend vorgeschrieben und andere wiederum optional. In BNF bzw. EBNF würde man die Attribute rekursiv bzw. über eine Wiederholung als Liste aufsammeln und dann in den Aktionen semantisch prüfen, daß kein Attribut mehrfach aufgetreten ist und daß alle notwendigen Attribute vorhanden sind.

In *oops* ist die Erweiterung von Parsern einfach. Führt man neue Knoten-Klassen zur eleganten Repräsentation gewünschter Konstrukte ein, so verlegt man die Prüfung dorthin, wo sie hingehört, nämlich in die Syntaxprüfung: So sammelt eine Instanz der neuen Klasse `And` analog zu `Alt` viele Unterknoten, erkennt aber nicht nur einen, sondern in beliebiger Reihenfolge exakt alle Unterknoten jeweils einmal. Ein `Or`-Knoten kapselt wiederum analog zu `Alt` viele Alternativen und erkennt in beliebiger Reihenfolge mindestens eine, aber beliebig viele der Alternativen auch jeweils einmal.

Auf der anderen Seite braucht es aber auch eine Grammatik-Notation, um die alten und neuen Operationen spezifizieren zu können. Um den Aufwand zum Erlernen der neuen Notation minimal zu halten und um den Wiedererkennungswert auszunutzen, orientiert sich die folgende Notation an den bekannten, logischen Operatoren einer Programmiersprache mit den gewohnten Vorrangregeln. Die neue Notation wird im Folgenden Extended EBNF oder kurz XEBNF genannt. Das X ist dabei mit Absicht mehrdeutig zu verstehen: Es erinnert daran, daß die Ideen der neuen Konstrukte aus der XML-Welt entliehen worden sind und daß sie eine Erweiterung zu EBNF darstellt. XEBNF-Grammatiken müssen im ersten Ansatz folgender EBNF-Grammatik genügen:

```
parser  : { rule };
rule    : ID ":" or ";";
or      : xor [{ "|" xor }] ; // or
xor     : and [{ "^" and }] ; // xor
and     : seq [{ "&" seq }] ; // and
seq     : { ID | LIT | TOKEN | some | opt | "(" or ")" };
some    : "{" or "}";          // one or more
opt     : "[" or "]";          // zero or one
```

Durch ein `|` getrennte Grammatik-Teile einer XEBNF-Grammatik sind oder-verknüpft und werden im Parser-Baum durch ein Objekt der Klasse `Or` repräsentiert. Eines der Elemente **oder** ein anderes Element **oder** noch ein anderes Element werden erkannt; also mindestens eines der Elemente muß und alle Elemente dürfen jeweils einmal in der Eingabe vorkommen, wobei die Reihenfolge beliebig ist.

Durch ein `&` getrennte Grammatik-Teile sind und-verknüpft und werden im Parser durch einen `And`-Knoten repräsentiert. Ein Element **und** ein anderes Element **und** noch ein anderes Element **und** ... werden akzeptiert; also alle Elemente müssen genau einmal, in beliebiger Reihenfolge erkannt werden.

Durch ein ^ getrennte Grammatik-Teile sind xor-verknüpft und werden im Parser durch einen XOr-Knoten repräsentiert. **Entweder** das eine Element **oder** ein anderes Element wird erkannt; also genau eines der Elemente wird akzeptiert; wobei die Reihenfolge wieder egal ist. Dies entspricht der aus EBNF bekannten Verknüpfung von Alternativen durch |.

Als Beispiel eine XEBNF-Grammatik, welche die neuen Konstrukte vorführt:

```
start : and or xor ;
and   : "a1" & "a2" & "a3";  #  und-Verknuepfung von Elementen
or    : "o1" | "o2" | "o3";  # oder-Verknuepfung von Elementen
xor   : "x1" ^ "x2" ^ "x3";  #  xor-Verknuepfung von Elementen
```

Damit wäre a1 a3 a2 o2 o1 x2 ein legales Programm der Grammatik. Alle Elemente der rechten Seite der and-Regel, manche der Elemente der or-Regel und ein Element der xor-Regel sind jeweils einmal in beliebiger Reihenfolge erkannt.

Hier noch einmal tabellarisch die Randbedingungen der einzelnen Verknüpfungen:

&	alle Elemente müssen erkannt werden, die Reihenfolge der Elemente ist beliebig, jedes Element wird nur einmal erkannt.
|	manche Elemente müssen erkannt werden, die Reihenfolge der Elemente ist beliebig, jedes Element wird nur einmal erkannt.
^	genau ein Element wird erkannt, das eine Element wird nur einmal erkannt.

Untersucht man für die Konstrukte Variationen entlang der Achsen der Randbedingungen, führt dies für die und- und oder-Verknüpfung zu einer sinnvollen Erweiterung:

Modifiziert man die Bedingung der beliebigen Reihenfolge der Elemente zu einer festen Reihenfolge, so ist dies nichts anderes als eine bereits bekannte Sequenz. Eine Kombination von fester und beliebiger Reihenfolge kann als Sequenz von Elementen als Element einer und- oder oder-Verknüpfung oder als eine derartige Verknüpfung als Teil einer Sequenz dargestellt werden. Eine Variation entlang dieser Achse führt also zu keinen neuen Resultaten.

Variiert man die Bedingungen entlang der Achse, welche die Anzahl der zu erkennenden Elemente festlegt, so ist schnell zu sehen, daß eine oder-Verknüpfung, welche alle Elemente erkennen muß, einer und-Verknüpfung gleichkommt und daß optionale Elemente einer und-Verknüpfung den Zwang der Erkennung aller Elemente aufweichen. Schachtelt man ganze oder-, und- oder xor-Verknüpfungen in einen optionalen Opt-Knoten, wird auch kein Element akzeptiert.

Entlang der letzten Achse entspricht eine gewünschte, mehrfache Erkennung des einen parsierten Elements einer xor-Verknüpfung dem Schachteln aller Elemente der Verknüpfung in jeweils einem Wiederholungsknoten und ist damit bereits abgehandelt. Für und- oder oder-Verknüpfungen wird eine mögliche mehrfache Erkennung der Elemente nun aber interessant. Als Beispiel wird in der Grammatik

```
start:  "a" & { "b" } & c ;
```

das b in einem Wiederholungsknoten geschachtelt. Die Wiederholung könnte bedeuten, daß die Wiederholung ein Element der und-Verknüpfung ist und damit c a b b b eine legale Eingabe wäre, oder die Wiederholung beschreibt, daß das b-Element der Verknüpfung wiederholt, also mehr als einmal erkannt werden darf, womit a b c b eine legale Eingabe wäre. Analog gilt die gleiche Überlegung für eine oder-Verknüpfung wo für

```
start:  "a" | { "b" } | c ;
```

je nach Deutung der Wiederholung b b a oder b c b legale Eingaben wären.

Ganz allgemein ist die Idee, die Elemente einer und- oder oder-Verknüpfung nicht nur einmal zu erkennen, eine sinnvolle Erweiterung der beiden Konstrukte.

Bedeutet in

```
start:  "a" | { "b" } | c ;
```

die Wiederholung ein mehrfaches Auftreten des Elements in der oder-Verknüpfung und keine einmalige Wiederholung des `b`, so muß die zweite Variante aber syntaktisch auch spezifiziert werden können. Als Beispiel könnte analog

```
start:  "a" | ( { "b" } ) | c ;
```

eine Klammerung einer Wiederholung diese vor einer und- oder oder-Verknüpfung verbergen und daher hier das `b` einmalig wiederholen. Da dieser feine semantische Unterschied meiner Ansicht nach in komplexen Grammatiken nur schwer zu erfassen sein wird, habe ich mich für eine andere Notation in XEBNF entschieden. Ein Beispiel:

```
start : and | or ;
and   : "a1" & "a2"+ & "a3";  // a1 und a3 genau einmal, a2 mindestens einmal
or    : "o1" | "o2"* | "o3";  // o1 und o3 maximal einmal, o2 beliebig oft
```

Die Elemente einer und-Verknüpfung sind nachfolgend mit einem Plus zu versehen, wenn dieses Element der Verknüpfung mehr als einmal akzeptiert werden kann. Analog sind Elemente einer oder-Verknüpfung nachfolgend mit einem Stern für eine mehrfache Erkennung zu markieren. Das Plus und der Stern zielen dabei wieder auf die Notation von regulären Ausdrücken, wo diese beiden Operatoren auch eine “eine bis viele”- bzw. “keine bis viele”- Wiederholung ausdrücken.

Die Elemente einer oder- bzw. und-Verknüpfung dürfen sinnvollerweise nur dann mit einem Stern oder einem Plus markiert werden, wenn die Verknüpfung auch mehr als ein Element enthält. Daher ist die EBNF-Grammatik für XEBNF-Grammatiken in den Regeln dieser beiden Verknüpfungen etwas komplexer:

```
parser  : { rule };
rule    : ID ":" or ";";
or      : xor [ "*" "|" xor [ "*" ]] [{ "|" xor [ "*" ] }];
xor     : and [{ "^" and }];
and     : seq [ "+" "&" seq [ "+" ]] [{ "&" seq [ "+" ] }];
seq     : { ID | LIT | TOKEN | some | opt | "(" or ")" };
some    : "{" or "}";  // one or more
opt     : "[" or "]";  // zero or one
```

Gegen diese Notation von XEBNF spricht, daß das Plus und der Stern eine zusätzliche Syntax darstellen und bei einer Verwendung von Stern, Plus und Fragezeichen analog zu *JavaCC* zum Markieren von Wiederholungen bzw. optionalen Teilen einer Grammatik die Zeichen kollidieren. In diesem Fall könnte man aber auf den ersten dargestellten Ansatz ausweichen, wo eine Wiederholung eines Elements einer und- oder oder-Verknüpfung den Inhalt des Knotens für eine mehrfache Erkennung als Teil der Verknüpfung markiert und Klammern die Wiederholung vor der Verknüpfung verbergen.

Um *oops* bzw. die Parser-Bäume um die neuen Konstrukte zu erweitern, ist die Klassenbibliothek um drei neue Klassen zu erweitern. Wie bereits skizziert, kapselt eine `And`-Instanz eine und-, eine `Or`-Instanz eine oder- und eine `XOr`-Instanz eine xor-Verknüpfung.

9.2.2 XOr

Die Klasse `XOr` ist schnell zu implementieren. Sie stammt lediglich von `Alt` ab und übernimmt komplett deren Funktionalität (Mengenberechnung, Prüfung, Parsierung und Fehlerbehandlung), bietet aber eine Knoten-Klasse, deren Namen zur Art der Verknüpfung korrespondiert.

9.2.3 Or

Auch die Klasse `Or` stammt von `Alt` ab. Damit erbt sie unter anderem die Verwaltung der Unterknoten. Zusätzlich wird in einem `Or` für jedes mehr als einmal erkennbare Element — also in der aktuellen XEBNF-Notation jedes mit einem Stern markierte Element — ein Bit in der Instanzvariablen `special` gesetzt:

```
public class Or extends Alt {
    protected final BitSet special = new BitSet();
    public Or(Node node) { this(node, false); }
    public Or(Node node, boolean special) { add(node, special); }
    public void add(Node node, boolean special) { ... }
    ...
}
```

Alternativ zu der ausdrücklichen Markierung eines Knotens für eine wiederholte Erkennung durch den zweiten, Boolschen Parameter zu `add()` hätte ein `Or` (und analog ein `And`) diese Elemente anhand ihrer Klasse bestimmen können. So könnten alle `Many` oder `Some` als Wiederholungsknoten, wie besprochen, das mehrfache Erkennen des gekapselten Unterknotens markieren. Ich habe mich aber für den aktuellen Ansatz entschieden, da er allgemeiner ist und beide Fälle beinhaltet.

Mengenberechnung

Die Berechnung der lookahead- und follow-Menge eines `Or`-Knotens unterscheidet sich nicht von der Berechnung der beiden Mengen eines `Alt`-Knotens. In `Or` sind daher die beiden Methoden `setLookahead()` und `setFollow()` nicht überimplementiert und werden von `Alt` geerbt.

Prüfung

In der geerbten Berechnung der lookahead-Menge wird bereits geprüft, ob die lookahead-Mengen der einzelnen Elemente disjunkt sind. Darüber hinaus stellt sich für ein `Or` — analog zu `Some` — aber noch folgende mehrdeutige Situation:

```
start : or "o2" ;
or    : ( "o1" | "o2" ) ;
```

Bei der Eingabe

```
o1 o2
```

ist nicht klar, ob das `o2` in der Regel `or` oder in der Regel `start` erkannt werden soll. Das Beispiel zeigt, daß für ein `Or` die lookahead- und follow-Menge disjunkt sein sollten. In `checkLL1()` prüft `Or` diesen Umstand und fordert indirekt über `checkLL1()` der Oberklasse die Unterknoten zur Prüfung auf. Sind die lookahead- und follow-Menge nicht disjunkt, wird lediglich eine Warnung ausgegeben, und das Symbol wird als Element des `Or`-Knotens erkannt.

Auch `checkDeadLoop()` als Bestandteil des Markierungsalgorithmus zum Auffinden von unendlichen Rekursionen paßt unverändert und wird von der Oberklasse übernommen.

Parsierung und Fehlerbehandlung

Die Parsierung und die Fehlerbehandlung geschehen analog zu den bereits bekannten Klassen eines Parser-Baums. Eine Instanz der lokalen Klasse `OrActivation` repräsentiert eine Aktivierung von `parse()` eines `Or`-Knotens. In der Instanzvariablen `done` der `OrActivation` wird markiert, ob der Knoten bereits ein Element parsiert hat und damit enden könnte. Zusätzlich reflektiert ein zu `parse()` lokaler `BitSet` namens `closed` die Elemente, welche nicht mehr erkannt werden müssen bzw. dürfen:

```
public void parse(final Parser parser, Activation caller, Object action)
    throws ParseException, IOException, Activation {
    final BitSet closed = new BitSet();

    class OrActivation extends Activation {
        protected boolean done;
        protected OrActivation (Activation caller) { super(caller); }
```

`up()` und `canUse()` brauchen beide die Information, ob das aktuelle Symbol zum lookahead der noch zu erkennenden Elemente paßt. Die Bestimmung dieser Information ist daher in eine eigene Methode gekapselt:

```
        private boolean localCanUse() {
            for (int i = 0; i < nodes.size(); ++i) {
                Node node = (Node) nodes.elementAt(i);
                if (!closed.get(i) &&
                    node.lookahead.matches(parser.tokenSet()))
                    return true;
            }
            return false;
        }
```

Damit sind `up()` und `canUse()` einfach. Liefert im Fehlerfall `localCanUse()` als Wert `true`, das heißt, daß das aktuelle Symbol zum lookahead der noch zu erkennenden Elemente paßt, wird die Erholung durch `handle()` ausgelöst. Liefert `localCanUse()` als Wert `true`, endet `canUse()` auch mit `true`. Ansonsten wird die Anfrage nach außen entlang der Activation-Kette weitergereicht:

```
        protected Object up (Object info) throws Activation {
            if(localCanUse()) handle(info);  // recover
            return super.up(info);
        }
        protected boolean canUse() {
            return localCanUse() ? true: super.canUse();
        }
```

`expect()` fordert gegebenenfalls den Oberknoten zum Hinzufügen möglicher Symbole auf und addiert alle Symbole aus dem lookahead der noch zu erkennenden Elemente:

```
        protected void expect(Set expect) {
            if (done) super.expect(expect);
            for (int i = 0; i < nodes.size(); ++i)
                if (!closed.get(i))
                    expect.add(((Node) nodes.elementAt(i)).lookahead);
        }
    }
```

Wie gewohnt wird in `self` eine Instanz der lokalen `Activation` als Repräsentant der Aktivierung hinterlegt. `self.done` ist wahr, wenn der `Or`-Knoten optional ist, also übergangen werden kann:

```
            final OrActivation self = new OrActivation(caller);

            self.done = lookahead.matchesEmpty();
```

Damit kann die Parsierung starten. Paßt das aktuelle Symbol zum lookahead des `Or`-Knotens, wird über eine `for`-Schleife das zugehörige Element bestimmt. Reflektiert `closed`, daß das Element noch zu erkennen ist, startet die Parsierung des Elements. War das Element nicht mit einem Stern für wiederholte Erkennung markiert, wird nach der Parsierung des Elements in `closed` das Bit gesetzt:

```
            parse: for (;;)
                try {
                    if (lookahead.matches(parser.tokenSet()))
                        for (int i = 0; i < nodes.size(); ++i) {
                            Node node = (Node) nodes.elementAt(i);
                            if (!closed.get(i) &&
                                node.lookahead.matches(parser.tokenSet())) {
                                node.parse(parser, self, action);
                                self.done = true;
                                if (!special.get(i))
                                    closed.set(i);
                                continue parse;
                            }
                        }
```

Erreicht der Programmfluß diese Stelle, so paßt das aktuelle Symbol nicht zum lookahead des `Or`-Knotens, bzw. das zugehörige Element wurde bereits parsiert und war nicht für wiederholtes Erkennen markiert. Reflektiert `self.done`, daß die Parsierung des Knotens zu Ende gehen kann, und wird als Abbruchkriterium das aktuelle Symbol weiter außen akzeptiert, endet `parse()` erfolgreich:

```
                    if (self.done && caller.canUse()) return;
```

Endet `parse()` nicht, so paßt das aktuelle Symbol nicht zum lookahead der ausstehenden Elemente und wird auch nicht weiter außen akzeptiert. Es liegt ein Eingabefehler vor. Entlang der `Activation`-Kette werden durch `up()` alle aktiven Knoten befragt, ob sie mit dem aktuellen Symbol die Fehlererholung durchführen wollen. Falls ja, so löst die `Activation` des zugehörigen Knotens per `handle()` die Erholung aus. Endet `up()`, wird das Symbol verworfen und der Scanner zum Vorrücken um ein Symbol aufgefordert, und die `for`-Schleife führt die Tests gegen das neue Symbol von vorne aus:

```
                    parser.error(this, self, action);
                    caller.up(this);
                    parser.advance();  // discard symbol
                } catch (Activation a) {
                    if (a != self) throw a;
                }
        }
    }
```

9.2.4 And

`And` stammt von `Or` ab. Die Klasse erbt unter anderem indirekt von `Alt` die Verwaltung der Unterknoten und erbt von `Or` die Methode `add()` und die Instanzvariable `special` zum Markieren von Elementen, welche für mehrfaches Erkennen markiert sind.

Mengenberechnung

Auch für ein `And` sind die lookahead- und die follow-Menge analog zu `Alt` zu berechnen. `And` verwendet daher auch die beiden indirekt von `Alt` geerbten Methoden `setLookahead()` und `setFollow()`.

Prüfung

Die Grammatik

```
start : "a1" & [ "a2" ] & [ "a3" ] ;
```

erkennt zwingend `a1`, kann aber die optionalen Elemente als leer parsieren und damit übergehen. In der Grammatik

```
start : [ "a1" ] & [ "a2" ] & [ "a3" ] ;
```

sind aber alle Elemente optional. Damit können alle Elemente übergangen werden, das heißt, der gesamte `And`-Knoten ist optional. Darüber hinaus kann aber jeder Knoten erkannt werden, muß es aber nicht. Ein solches `And` verhält sich daher wie ein optionaler `Or`-Knoten. Die Grammatikprüfung warnt zumindest vor dieser vielleicht ungewollten Situation. Dies geschieht durch das Ersetzen der Methode `acceptEmptyElements()`, welche für einen `Alt`-Knoten bzw. damit auch für Knoten einer Unterklasse die Anzahl der optionalen Elemente prüft:

```
protected void acceptEmptyElements(int numberOfEmptyElements, Set lookahead)
    throws CheckLL1Exception{
    if (numberOfEmptyElements == nodes.size())
        System.err.println("warning, "+this+
            ":\n  All alternative have empty in lookahead."+
            "\n  This will act like an optional or expression.");
    else
        lookahead.removeEmpty();
}
```

Sind nicht alle Elemente, ist aber mindestens eines der Elemente optional, ist ein `And` im Gegensatz zu einem `Alt` als Ganzes nicht optional. Daher wird im `else`-Zweig in `acceptEmptyElements()` diesem Umstand Rechnung getragen.

Der Markierungsalgorithmus in `checkDeadLoop()` zum Auffinden von unendlichen Rekursionen hat für ein `And` alle nicht optionalen Elemente abzugehen:

```
public boolean checkDeadLoop() throws CheckLL1Exception {
    if (lookahead.matchesEmpty()) return false;
    for (int n = 0; n < nodes.size(); ++ n) {
        Node node = (Node)nodes.elementAt(n);
        if ( ! node.lookahead.matchesEmpty() && node.checkDeadLoop() )
            return true;
    }
    return false;
}
```

Lediglich durch das Ersetzen dieser beiden Methoden ist die Prüfung eines `And`-Knotens bereits realisiert.

Parsierung und Fehlerbehandlung

Die Implementierung der `parse()`-Methode der `And`-Klasse inklusive der lokalen `Activation`-Klasse agiert recht ähnlich zu dem Programmtext der `Or`-Klasse. Allerdings hat ein `And` einige

Randbedingungen mehr zu beachten. Im Gegensatz zu `Or` müssen zum Beispiel alle nicht optionalen Elemente erkannt werden. Genauer möchte ich hier aber nicht auf die Implementierung eingehen, da das nichts wesentlich Neues bringen würde.

9.3 Ein Beispiel

Analog zu dem DTD- bzw. XML-Beispiel beschreibt die folgende Grammatik auch eine Personenliste mit den gleichen Informationen bezüglich einer Person in XEBNF:

```
personen   : {[ person ]} ;
person     : "person" ( persnr & [ chef ] & geschlecht & vorname &
               nachname ) ";" ;
persnr     : "persnr" NUMBER;
chef       : "chef" NUMBER;
geschlecht : "maennlich" ^ "weiblich";
vorname    : "vorname" WORD ;
nachname   : "nachname" WORD ;
```

Da hier die Informationen im Gegensatz zu XML nicht immer umständlich in spitzen Klammern eingeschlossen sein müssen, sind Personenlisten in dieser Form wesentlich lesbarer:

```
person
    persnr 2345
    weiblich
    vorname  Berta
    nachname Boss ;

person
    nachname Untertan
    weiblich
    chef 2345
    persnr 123
    vorname  Ulla ;

person
    vorname  Knut
    chef 123
    persnr 0815
    nachname Knetch
    maennlich ;
```

Zu beachten ist, daß analog zu dem XML-Beispiel bis auf den optionalen `chef`-Eintrag alle anderen Informationen bezüglich einer Person angegeben sein müssen, wobei aber wiederum die Reihenfolge egal ist. Dies syntaktisch mit `BNF` oder `EBNF` zu prüfen, ist wohl kaum möglich.

9.4 Fazit

Das Kapitel hat die These bestätigt, daß auch im Compilerbau der Einsatz der Objekt-Orientierung zu leicht erweiterbaren Lösungen führt. Die Erweiterung von *oops* um neue, mächtige Klassen wie `And`, `Or` und `Xor` ist aufgrund der Modellierung von Parsern durch Instanzen von Knoten-Klassen keine Schwierigkeit. Dank der Vererbung übernehmen die neuen Klassen Instanzvariablen und Methoden von ihrer Oberklasse und brauchen lediglich lokal nur einige Methoden überschreiben. Natürlich sind die Klassen für verschiedene Parser wiederverwendbar.

Neben den Klassen `And`, `Or` und `Xor` könnte eine Instanz der Klasse `Iteration` einen Unterknoten und zwei Zahlen `m` und `n` besitzen. Während der Parsierung wiederholt eine `Iteration` die Parsierung

des Unterknotens zwischen `m`- und `n`-mal. Wieder kann eine mögliche Notation den regulären Ausdrücken entliehen werden:

```
start :  "x" <2,6>
```

In spitzen Klammern beschreibt diese Notation, wie oft der direkt links davon stehende Teil einer Regel wiederholt werden kann. Durch runde Klammern könnte ein unpassender Vorrang übergangen bzw. eine längere Sequenz zur Wiederholung markiert werden.

Da die neuen Klassen lediglich die Parsierung an einen oder mehrere Unterknoten delegieren, sind die Klassen mit allen vorgestellten Aktions-Mustern zu erkannten Teilen einer Grammatik kompatibel. Damit können `And`, `Or` und `Xor` sofort zum Beispiel in einem Parser mit `Goal`-Aktionen eingesetzt werden.

Die von mir gewählte Notation von XEBNF oder die einer `Iteration`-Wiederholung in der Grammatik ist nicht wichtig und kann ausgetauscht werden. Ist der Parser-Baum erst einmal gebaut, ist die Ausgangsnotation irrelevant. Allgemeiner ist die ursprüngliche Notation einer Grammatik, die ein Parser-Baum repräsentiert, nicht länger entscheidend. Als Beispiel wurde in der Vorlesung “XML — Architektur, Werkzeuge, Techniken” ([Sch01]) ein *oops*-Parsergenerator für EBNF-Grammatiken verwendet, welcher analog zu *JavaCC* einen Stern, ein Plus oder Fragezeichen zur Beschreibung der EBNF-Sprachkonstrukte verwendet. Der Baum selbst ist die “Message” und die beste, unabhängige Darstellung der Grammatik.

10 oops als Parsergenerator

oops ist eine Klassenbibliothek, auf der Bäume zur Repräsentation von Grammatiken als Parser für die Grammatiken (optional mit verschiedenen Arten von Aktionen zu erkannten Teilen einer Grammatik) gebaut werden.

oops ist aber auch ein Parsergenerator, der für EBNF- oder XEBNF-Grammatiken die zugehörigen Parser-Bäume erzeugt und prüft. Dieses Kapitel schildert die Idee und Umsetzung der Parsergeneratoren in *oops*.

Die erste Version von *oops* als Parsergenerator wurde von Axel-Tobias Schreiner im Rahmen der Vorlesung “Compilerbau mit Java” ([Sch98a]) für EBNF-Grammatiken mit Hilfe von *jay* entwickelt. Alle weiteren Versionen von *oops* wurden mit *oops* selbst erzeugt. Bei der Übersetzung von *oops* mit *oops* ergeben sich aber zwei interessante Probleme bzw. Aspekte, die erst bei der Verwendung der Objekt-Orientierung im Compilerbau sichtbar werden. Auf diese beiden Punkte und auf die Übersetzung von *oops* mit *oops* geht dieses Kapitel im Folgenden ein.

10.1 Die Idee des Parsergenerators

Eine EBNF- oder XEBNF-Grammatik wird durch einen Baum von Objekten der Klassenbibliothek repräsentiert. Ein Aufruf von `parse()` gegen den Baum startet die Erkennung von Programmen über der Grammatik. Ohne einen Parsergenerator muß die Erzeugung des Parser-Baums von Hand durch Aufruf von Konstruktoren der jeweiligen Knoten und durch Aufruf von Methoden wie `add()` geschehen. Aber nur mit einem Parsergenerator zum automatischen Erzeugen der Bäume für eine Grammatik ist die Klassenbibliothek elegant zu nutzen. Abbildung 10.1 stellt die Situation noch einmal dar:

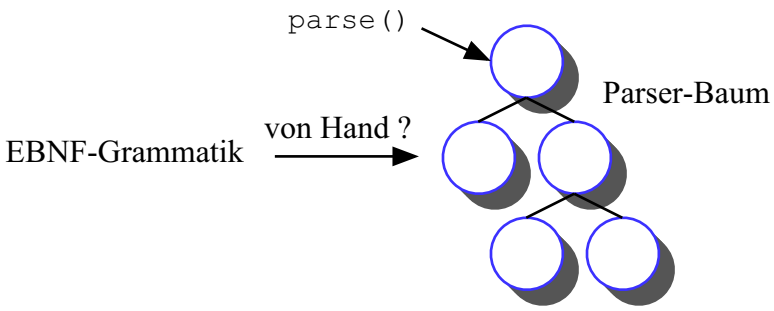

Abbildung 10.1: Von der Grammatik zum Parser-Baum.

Folgende EBNF-Grammatik beschreibt EBNF-Grammatiken. In der Grammatik steht die Terminals `ID` für einen Regel-Namen, `LIT` für ein Literal und `TOKEN` für ein Symbol als Klassifizierung:

```
parser  : { rule };             // a parser is one ore more rules.
rule    : ID ":" alt ";";       // rule is a name and a right hand side
alt     : seq [{ "|" seq }];    // alternatives
seq     : { ID | LIT | TOKEN | some | opt | "(" alt ")" }; // sequence
some    : "{ " alt "} ";        // one or more
opt     : "[ " alt "] ";        // zero or one
```

Analog beschreibt folgende XEBNF-Grammatik legale XEBNF-Grammatiken:

```
parser  : { rule };
rule    : ID ":" or ";";
```

```
or      : xor [ "*" "|" xor [ "*" ]] [{ "|" xor [ "*" ] }];
xor     : and [{ "^" and }];
and     : seq [ "+" "&" seq [ "+" ]] [{ "&" seq [ "+" ] }];
seq     : { ID ^ LIT ^ TOKEN ^ some ^ opt ^ "(" or ")" };
some    : "{ " or "}";
opt     : "[ " or "]";
```

Die beiden Grammatiken sind der Schlüssel zu den gewünschten Parsergeneratoren. Ein Parser-Baum für eine der obigen Grammatiken erkennt bei Aufruf von `parse()` Programme über der Grammatik, also wieder eine EBNF- bzw. XEBNF-Grammatik. Durch Aktionen zum Parser-Baum, die für Teile der Eingabe-Grammatik die entsprechenden Knoten des Ziel-Baums bauen, ist der Parsergenerator bereits realisiert. In der Abbildung 10.2 ist der Parsergenerator mit *oops* bezeichnet.

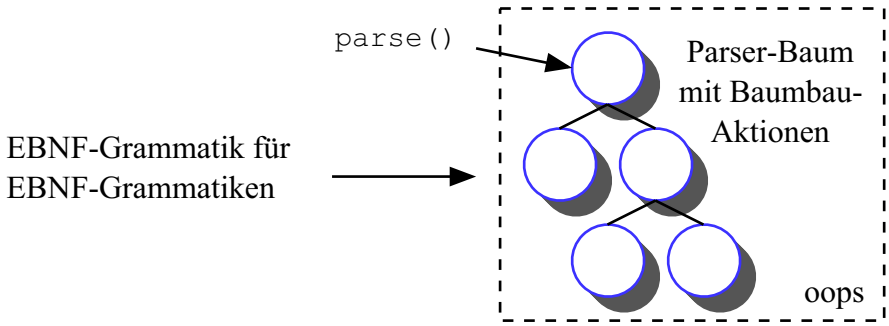

Abbildung 10.2: Von einer Grammatik für EBNF zu oops.

Der Parsergenerator in *oops* ist also selbst ein Parser-Baum, welcher als Aktionen einen Baum für die erkannten Teile der Eingabe-Grammatik erzeugt. Wie Abbildung 10.3 zeigt, leistet *oops* den gewünschten Übergang von einer Grammatik zum Parser-Baum für die Grammatik:

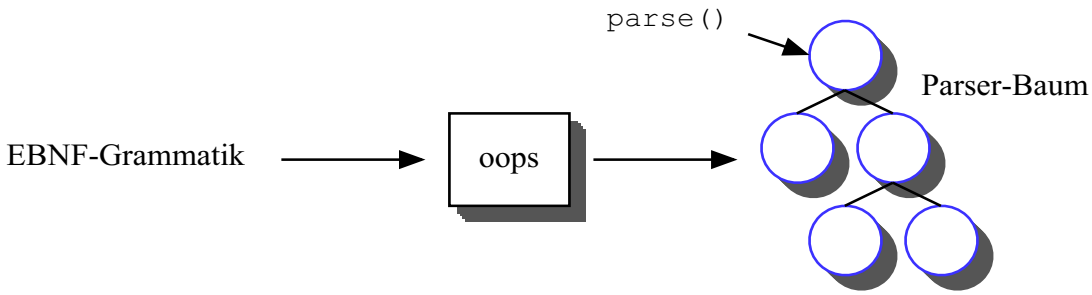

Abbildung 10.3: oops als Parsergenerator.

10.2 Übersetzen von oops mit oops

10.2.1 Standard Goal-Klassen zum Baumbau

Die Parsergeneratoren von *oops* sind selbst Parser-Bäume, welche mit `Goal`-Aktionen die Knoten-Objekte zu erkannten Teilen der Eingabe-Grammatik erzeugen.

Repräsentativ werden die `Goal`-Klassen zu den Regeln `opt`, `some`, `parser` und `seq` der obigen Grammatiken gezeigt. `opt` und `some` erzeugen einen Knoten mit einem Unterbaum, und `parser` und `seq` stehen repräsentativ für `Goal`-Klassen, die einen Knoten mit mehreren Unterknoten des Parser-Baums erzeugen.

Ich hatte mich während der Entwicklung der `Goal`-Klassen für einen defensiven Ansatz entschieden. Alle Klassen stammen direkt oder indirekt von der Klasse `base` ab. `base` leitet

`oops.goal.GoalAdapter` ab und wirft in allen drei `shift()`-Methoden eine Exception. Damit wird sichergestellt, daß nur die erwarteten `shift()`-Methoden, d.h. die in Unterklassen ersetzten, aufgerufen werden. Inzwischen könnte `base` wieder entfernt werden, und alle direkten Unterklassen könnten von `GoalAdapter` abstammen. `base` und damit alle Unterklassen von `base` erben von `GoalAdapter` die Variablen `error` und `result` bzw. die Methoden `error()` und `reduce()`.

Eine Instanz der `Goal`-Klasse `opt` zur Regel `opt`

```
opt     : "[ " alt "] ";          // zero or one
```

hat die Aufgabe, einen `oops.parser.Opt`-Knoten zu einer erkannten `opt`-Regel zu bauen. Ist bei der Parsierung kein Fehler aufgetreten, wird als indirekter `GoalAdapter` der Wert von `result` als Ergebnis von `reduce()` geliefert. In `shift(Goal,Object)` wird eine `opt`-Instanz als Werte-Objekt über den Unterbaum des zu erzeugenden Knotens unterrichtet. Neben einigen Tests ruft die Methode `build(Node)` zum Erzeugen des Resultat-Knotens für `reduce()` auf:

```
package oops.oops.goals;

public class opt extends base {
    protected void build(oops.parser.Node node) {
        result = new oops.parser.Opt(node.node());
    }

    public void shift (oops.parser.goal.Goal sender, Object node) {
        if (error |= (node == null)) return;
              if ( !(node instanceof oops.parser.Node) )
            throw new RuntimeException("illegal node class "+
                                       node.getClass().getName());
        if (result != null)
            throw new RuntimeException("second child for "+
                                       getClass().getName());
        build((oops.parser.Node) node);
    }

    public void shift (oops.parser.goal.Lit sender, Object node) {
        // ignore "[ " and "] "
    }
}
```

Als Spielregel für alle `Goal`-Instanzen gilt, daß Unterknoten vor der Verwendung per `node()` optimiert werden. Alternativ könnten die Resultat-Knoten der `reduce()`-Methoden optimiert werden.

Die `Goal`-Klasse `some` zur Regel `some`

```
some    : "{ " or "} ";
```

stammt von `opt` ab, erbt deren Methoden und ersetzt lediglich `build()`, um eine `Some`- anstelle der `Opt`-Instanz zu erzeugen:

```
public class some extends opt {
    protected void build(oops.parser.Node node) {
        result = new oops.parser.Some(node.node());
    }
}
```

`Goal`-Instanzen zu der `parser`-Regel

```
parser  : {  rule } ;
```

sammeln viele `oops.parser.Rule`-Objekte als Unterknoten eines `oops.parser.Parser`-Objekts auf. Auch andere `Goal`-Objekte, wie die der Unterklasse `xor` oder `seq` der jeweils gleichnamigen Regel, haben eine ähnliche Aufgabe. Hier werden allerdings allgemein `oops.parser.Node`-Instanzen gesammelt. Die Methode `add(Node)` ist pro Klasse für das Aufsammeln der Unterknoten verantwortlich:

```
public class parser extends base {
    protected oops.parser.Parser parser;

    protected oops.parser.Parser buildParser(oops.parser.Rule rule) {
        return new oops.parser.Parser(rule);
    }

    protected void add(oops.parser.Node node) {
        if ( !(node instanceof oops.parser.Rule) )
            throw new RuntimeException("illegal node class "+
                                       node.getClass().getName());
         if (parser == null)
            result = parser = buildParser((oops.parser.Rule) node);
        else
            try {
                parser.add((oops.parser.Rule) node);
            } catch (oops.parser.ParserBuildException pbe) {
                System.err.println(pbe);
            }
    }

      public void shift (oops.parser.goal.Goal sender, Object node) {
        if (error |= (node == null)) return;
        if ( !(node instanceof oops.parser.Node) )
            throw new RuntimeException("illegal node class "+
                                       node.getClass().getName());
        add((oops.parser.Node) node);
    }
}
```

Die `Goal`-Klasse zur Regel `seq`-Regel der Grammatik für EBNF-Grammatiken

```
seq     : { ID | LIT | TOKEN | some | opt | "(" alt ")" };
```

bzw. die der Grammatik für XEBNF-Grammatiken

```
seq     : { ID ^ LIT ^ TOKEN ^ some ^ opt ^ "(" or ")" };
```

hat die Aufgabe, viele Unterknoten in einer `oops.parser.Seq`-Sequenz zu sammeln. `seq` leitet `parser` ab und erbt damit die Methode `shift(Goal,Object)`. Die Methode `add(Node)` wird ersetzt, um neue Elemente der Sequenz hinzuzufügen.

```
public class seq extends parser {
    protected oops.parser.Seq seq;

    protected void add(oops.parser.Node node) {
        if (seq == null)
            result = seq = new oops.parser.Seq(node.node());
        else
            seq.add(node.node());
    }
```

```
        protected oops.parser.Lit buildLit(String body) {
            return new oops.parser.Lit(body);
        }

        public void shift (oops.parser.goal.Token sender, Object node) {
            if (error |= (node == null)) return;
            if ( !(node instanceof Object[]) )
                throw new RuntimeException("illegal node class "+
                                           node.getClass().getName());

            Object[] array = (Object[]) node;
            if (array.length != 2)
                throw new RuntimeException("illegal Object[], len "+array.length);

            String type = (String) array[0];
            if (type.equals("Lit")) add(buildLit((String) array[1]));
            else if (type.equals("Id")) add(new oops.parser.Id((String)array[1]));
            else throw new RuntimeException("illegal type "+type);
        }

        public void shift (oops.parser.goal.Lit sender, Object node) {
            // ignore literals like "(" and ")" in seq or
            // like "^" in subclass xor.
        }
    }
```

Die Regel für `seq` besteht auf der rechten Seite als einzige Regel neben `rule` nicht nur aus Literalen (runden Klammern) und aus Verweisen auf andere Regeln (`some` oder `opt`). `LIT` als Token-Symbol steht für ein Literal, also einen zitierten Text, `ID` für einen Verweis auf eine Regel und `TOKEN` für eine Kategorie.

Eine `seq`-Instanz erhält durch `shift(Token,Object)` Informationen über erkannte Symbole dieser drei Kategorien. Für erkannte Literale hinterlegt der Scanner als Werte-Objekt ein `Object`-Array der Form:

```
new Object[] { "Lit", "while" }
```

Das erste Element beschreibt textuell die Art des Symbols, und das zweite Element stellt den erkannten Text des Symbols dar.

In der `shift()`-Methode erzeugt ein `seq` aus dem `Object`-Array eine `oops.parser.Lit`-Instanz. Erst im Folgenden wird bei der Diskussion des zweiten Problems klar, warum der Scanner diese `Lit`-Instanz nicht sofort als Werte-Objekt erzeugt.

Die Kategorie `ID` steht für erkannte Bezeichner — ein Name als Verweis auf eine Regel oder als Name für eine Kategorie —, und der Scanner hinterlegt ein analoges Array:

```
new Object[] { "Id", "identifier" }
```

Ein `seq` erzeugt aus dieser Information ein `oops.parser.Id`-Objekt und baut es in die Sequenz ein.

Der Scanner liefert die `Object`-Arrays als Werte-Objekte, während die Eingabegrammatik erkannt wird. Da noch nicht alle Regeln bearbeitet worden sind, ist zu diesem Zeitpunkt noch nicht klar, ob ein Name auf eine Regel verweist oder ein Token als Kategorie darstellt. Bei Prüfung des erzeugten Parsers wird daher berechnet, welche `oops.parser.Id`-Instanzen auf `oops.parser.Rule` bzw. auf ein `oops.parser.Token` verweisen. Im Zuge der Prüfung werden die `Token`-Instanzen auch erst

erzeugt. Der Scanner liefert daher nie das `TOKEN`-Symbol, und dieses ist lediglich der Vollständigkeit halber Bestandteil der Grammatik.

Scanner

Der Scanner `oops.oops.Input` zum Parser für Grammatiken hat das in Kapitel 5 (“Eine objektorientierte Scanner-Schnittstelle”) erläuterte Interface `oops.scanner.Scanner` zu implementieren. Als Scanner wird ein `java.io.StreamTokenizer` benutzt, der Zwischenraum und C- und C++-Kommentare ignoriert und der das Muster

```
[a-zA-Z_.][a-zA-Z_0-9.]*
```

für Regelnamen bzw. für Kategoriennamen und Literale in Doppelhochkommata scannt.

Innerhalb von `advance()` rückt der Scanner zum nächsten Symbol in der Eingabe vor. Wie bereits beschrieben, werden als Werte-Objekte zu Literalen und Namen die 2-elementigen `Object`-Arrays hinterlegt.

```
    public boolean advance() throws IOException {
        if (atEnd()) return false;
        value = symbol = null;
        switch (st.nextToken()) {
            case StreamTokenizer.TT_EOF:  return false;
            case StreamTokenizer.TT_WORD:
                      value = new Object[] { "Id", st.sval};
                      symbol = ID; break;
            case '"': value = new Object[] { "Lit", st.sval };
                      symbol = LIT; break;
            default:  symbol = symtab.get(""+(char)st.ttype).symbol();
                      break;
        }
        return true;
    }
}
```

10.2.2 Übersetzung von oops

Mit den Grammatiken und der Scanner-Klasse kann bereits ein Erkenner für EBNF- bzw. XEBNF-Grammatiken erzeugt werden. Im folgenden Beispiel enthält die Datei `ebnf.ebnf` die EBNF-Grammatik für EBNF-Grammatiken und `xebnf.xebnf` die XEBNF-Grammatik für XEBNF-Grammatiken:

```
$ export JARS=$HOME/Promotion/jars
$ export CLASSPATH=$JARS/oops.jar
$ export PROPS=-Doops.oops.actionType=none
$ java $PROPS oops.EBNF ebnf.ebnf > ebnf.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using ebnf parser generator generated by oops on Mon Sep  3 17:25:46 MEST 2001
$ java $PROPS oops.Oops xebnf.xebnf > xebnf.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using oops generated by oops on Mon Sep  3 17:25:48 MEST 2001
$
```

Wie bereits beschrieben, wurde die erste Version des Parsergenerators für EBNF-Grammatiken mit *jay* erzeugt. Hier wird bereits `oops` selbst zum Generieren der Parsergeneratoren benutzt. Das Werkzeug `oops.EBNF` akzeptiert EBNF- und `oops.Oops` akzeptiert XEBNF-Grammatiken. Beide analysieren die Grammatik, bauen und prüfen den Parser-Baum als Repräsentation der Grammatik und abschließend serialisieren sie den Parser auf der Standardausgabe.

`ebnf.ser` und `xebnf.ser` enthalten in serialisierter Form die Parser für EBNF- bzw. XEBNF-Grammatiken. Im Folgenden erkennen beide Parser eine Grammatik als Test, ob die Parsierung von EBNF- bzw. XEBNF-Grammatiken funktioniert. Als Beispiel-Eingabe werden erneut `ebnf.ebnf` bzw. `xebnf.xebnf` verwendet:

```
$ export input=oops.oops.Input
$ java oops.RunParser ebnf.ser $input ebnf.ebnf
$ java oops.RunParser xebnf.ser $input xebnf.xebnf
$
```

Damit ist gezeigt, daß die Parser für EBNF- bzw. für XEBNF-Grammatiken ausgeführt werden und bereits Grammatiken erkennen können. Damit die Parser aber als Parsergenerator agieren, sind die `Goal`-Aktionen hinzuzubinden. Dazu werden zu `Goal`-Aktionen fähige Parser erzeugt:

```
$ export PROPS=-Doops.oops.actionType=goal  # default...
$ java $PROPS oops.EBNF ebnf.ebnf > ebnf.ser
....
$ java $PROPS oops.Oops xebnf.xebnf > xebnf.ser
....
$
```

Zusammen mit den `Goal`-Instanzen können somit bei Ausführung der Parser Grammatiken erkannt und jeweils ein Parser-Baum für diese erzeugt werden, welche aber noch mit `check()` zu prüfen sind. Daher sind noch nicht für alle Knoten der neuen Parser-Bäume die lookahead-Mengen berechnet, was erklärt, daß die Dateigröße der neuen, serialisierten Parser kleiner ist:

```
$ export goalProps=-Doops.parser.goal.DefaultGoalMakerFactory.goalPackage=\
> oops.oops.goals
$ java $goalProps oops.RunGoalParser ebnf.ser $input ebnf.ebnf > ebnf2.ser
$ java $goalProps oops.RunGoalParser xebnf.ser $input xebnf.xebnf > xebnf2.ser
$ ls -l ebnf.ser ebnf2.ser xebnf.ser xebnf2.ser
-rw-r--r--   1 bernd    staff        2995 Sep  6 13:35 ebnf.ser
-rw-r--r--   1 bernd    staff        1685 Sep  6 13:40 ebnf2.ser
-rw-r--r--   1 bernd    staff        4404 Sep  6 13:35 xebnf.ser
-rw-r--r--   1 bernd    staff        2418 Sep  6 13:40 xebnf2.ser
```

10.2.3 Standard Goal-Klassen zum Baumbau mit Aktionen

Instanzen der `Goal`-Klassen des Pakets `oops.oops.goals` bauen Parser-Bäume aus Knoten, die lediglich ein Programm über der zugrundeliegenden Grammatik erkennen können. Drei weitere Pakete `oops.oops.goals.goal`, `oops.oops.goals.reducer` und `oops.oops.goals.trace` beherbergen `Goal`-Klassen, deren Instanzen Knoten in den Baum einbauen, welche die jeweiligen Arten von Benutzer-Aktionen unterstützen.

Alle drei Pakete beinhalten die drei Klassen `parser`, `rule` und `seq`, welche alle von der gleichnamigen `Goal`-Klasse aus dem Paket `oops.oops.goals` abstammen und lediglich eine Methode der Oberklasse überschreiben. Hier als Beispiel die `Goal`-Klassen des Pakets `oops.oops.goals.reducer` zum Bau von Knoten mit `Reducer`-Aktionen:

```
public class parser extends oops.oops.goals.parser {
    protected oops.parser.Parser buildParser(oops.parser.Rule rule) {
        return new oops.parser.reducer.Parser(rule);
    }
}
public class rule extends oops.oops.goals.rule {
    protected oops.parser.Rule buildRule(oops.parser.Id id,
                                         oops.parser.Node rhs) {
        return new oops.parser.reducer.Rule(id, rhs.node());
    }
}
public class seq extends oops.oops.goals.seq {
    protected oops.parser.Lit buildLit(String body) {
        return new oops.parser.reducer.Lit(body);
    }
}
```

10.2.4 CompilerFactory-Klassen, ein erstes Problem

Der Aufruf eines in einer Datei serialisierten Grammatik-Parsers, zusammen mit den jeweiligen Aktionen zum Baumbau für Teile einer Grammatik, wurde oben vorgestellt. Die Anwendung der Parsergeneratoren ist so aber sicherlich nicht sehr elegant, und unklar bleibt, wie die Unterstützung der verschiedenen Aktions-Arten für die zu bauenden Parser-Bäume geschieht, da eine `DefaultGoalMakerFactory` dies so nicht realisiert, weil Instanzen von `Goal`-Klassen aus verschiedenen Paketen, wie zum Beispiel `oops.oops.goals` und `oops.oops.goals.reducer`, zu verwenden sind.

Die Anwendung der Parsergeneratoren ist in Form einer `oops.opi.CompilerFactory` gekapselt. Eine `EBNFCompilerFactory` als Parsergenerator für EBNF-Grammatiken erzeugt einen `Compiler`, der eine passende, eigene `GoalMakerFactory` verwendet, der die passende Scanner-Klasse kennt und eine Instanz der Scanner-Klasse erzeugt, der einen Parser für EBNF verwendet, der einen erfolgreich gebauten Parser-Baum prüft und der diesen als Resultat von `parse()` zurückliefert:

```
package oops.oops;

import ...

public class EBNFCompilerFactory extends CompilerFactory  {
    ...
    public Compiler compiler() {
        return new Compiler () {
            public Result compile(InputSource input) throws OpiException {
                try {
                    oops.parser.goal.Parser oops = getParser();
                    Input scanner = new Input(input);

                    System.err.print(getMessage());
                    oops.parser.goal.Parser.Result result =
                        oops.parse(scanner, getGoalMakerFactory(), null);

                    oops.parser.Parser parser =
                        (oops.parser.Parser) result.result;

                    if (result.error || result.result == null)
                        return new oops.opi.Compiler.Result(false, null);
```

```
                    parser.check();

                    return new oops.opi.Compiler.Result(true, parser);
                } catch (Throwable t) {
                    throw new OpiException("error while compiling", t);
                }
            }
        };
    }
}
```

Der zu verwendende Parser für EBNF wird als Resultat von `getParser()` ermittelt. Natürlich wird diese Methode von Unterklassen überschrieben. So wird die Factory für XEBNF-Grammatiken als Resultat der Methode einen Parser für XEBNF liefern.

In dem obigen Beispiel beinhaltet eine Datei den Parser in serialisierter Form. In der *jar*-Datei zu *oops* ist der Parser für EBNF unter `oops/oops/ebnf.ser` hinterlegt. Normalerweise sucht der Klassenlader auf dem Klassenpfad — und damit auch in der *jar*-Datei zu *oops* — nur nach Klassen. Die Methode `getParser(String)` sucht aber das Argument als Pfadnamen auf dem Klassenpfad — und damit auch in *jar*-Dateien —, öffnet die Datei und deserialisiert aus der Datei einen Parser:

```
public class EBNFCompilerFactory extends oops.opi.CompilerFactory {
    ...
    protected oops.parser.goal.Parser getParser() throws
        StreamCorruptedException, OptionalDataException,
        ClassNotFoundException, IOException {
        return getParser("oops/oops/ebnf.ser");
    }
    protected oops.parser.goal.Parser getParser(String resource) throws
        StreamCorruptedException, OptionalDataException,
        ClassNotFoundException, IOException { ... }
```

An dieser Stelle tritt das erste Problem auf: Alle Ideen dieser Arbeit sollen unabhängig von der zu verwendenden Programmiersprache sein. Der Trick, auf dem Klassenpfad eine Datei zu hinterlegen und aus den Daten der Datei einen Parser zu deserialisieren, ist zum Beispiel nur ähnlich auf C# anwendbar. Aber: Alternativ zu diesem Ansatz kann eine Klasse erzeugt werden, die die Daten des serialisierten Parsers als `byte`-Array enthält und als Ergebnis eines Methodenaufrufs die Daten oder besser gleich den deserialisierten Parser liefert. Diese Klasse kann dann anstelle von `ebnf.ser` verwendet werden. Das Werkzeug `oops.tools.ParserSer2JavaFile` führt dieses vor.

Der Parsergenerator erzeugt durch `getMessage()` vor der Parsierung einer Grammatik eine kurze Ausgabe. Auch diese Methode werden Unterklassen überschreiben. Das Datum der Erzeugung des Parsers ist als Inhalt der Datei `oops/oops/ebnf.date` in der *jar*-Datei zu *oops* hinterlegt. Durch `getContentFromSystemResource()` wird der Inhalt einer beliebigen Datei auf dem Klassenpfad als String ermittelt:

```
    protected String getMessage() throws IOException {
        return "Using ebnf parser generator generated by oops on "+
               getContentFromSystemResource("oops/oops/ebnf.date");
    }
    protected String getContentFromSystemResource(String resource) throws
        IOException { ... }
    ...
}
```

Während der Parsierung erzeugen `Goal`-Objekte die zu erkannten Teilen der Eingabe-Grammatik gehörenden Knoten des Parser-Baums. Eine eigene `GoalMakerFactory` erzeugt dabei je nach Wert der `oops.oops.actionType`-Property `Goal`-Instanzen aus verschiedenen Paketen:

```
public class EBNFCompilerFactory extends CompilerFactory  {
    ...
    protected oops.parser.goal.GoalMakerFactory getGoalMakerFactory()  {
        return new GoalMakerFactory();
    }
    protected static class GoalMakerFactory implements
        oops.parser.goal.GoalMakerFactory {
        protected final String packages [] = {
            "oops.oops.goals.goal.", "oops.oops.goals." };
        { getProperty(); }
        protected void getProperty() {
            String actionType = System.getProperty("oops.oops.actionType",
                                                   "goal");
            if (actionType.equals("goal"))
                packages[ 0] = "oops.oops.goals.goal.";
            else if (actionType.equals("none"))
                packages[ 0] = "";
            else if (actionType.equals("trace"))
                packages[ 0] = "oops.oops.goals.trace.";
            else if (actionType.equals("reducer"))
                packages[ 0] = "oops.oops.goals.reducer.";
            else throw new RuntimeException("unkown action type");
        }
        public GoalMaker goalMaker (final String ruleName) {
            return new GoalMaker() {
                Class goalClass = null;
                {   for (int i = 0; i < packages.length; i++)
                        try {
                            goalClass = oops.tools.Util.classForName(
                                                    packages[ i] +ruleName);
                            break;
                        } catch (Exception e) {}
                    if (goalClass == null)
                        throw new RuntimeException("No Goal for "+ruleName);
                }
                public Goal goal () {
                    try {
                        return (Goal) goalClass.newInstance();
                    } catch (Exception e) {
                        throw new RuntimeException("No Goal for "+ruleName);
                    }
                }
            };
        }
    }
    ...
}
```

Die Factory oder besser der von der Factory erzeugte `Compiler` kann mit `oops.opi.Compile` betrieben werden und erzeugt damit zu einer EBNF-Eingabegrammatik einen geprüften Parser-Baum. Wie der Test zeigt, sind die serialisierten Bytes der beiden EBNF-Parser exakt gleich.

```
$ PROPS=-Doops.opi.CompilerFactory=oops.oops.EBNFCompilerFactory
$ java $PROPS oops.opi.Compile ebnf.ebnf > ebnf2.ser
Using ebnf parser generator generated by oops on Thu Sep  6 15:42:43 MEST 2001
$ ls -l ebnf.ser ebnf2.ser
-rw-r--r--   1 bernd    staff        2995 Sep  6 14:57 ebnf.ser
-rw-r--r--   1 bernd    staff        2995 Sep  7 13:18 ebnf2.ser
$ cmp ebnf.ser ebnf2.ser
$
```

Das Werkzeug `oops.EBNF` nimmt einem das Setzen der Property `oops.opi.CompilerFactory` und den Aufruf von `oops.opi.Compile` noch ab und stellt als Kommandozeilen-Werkzeug den EBNF-Parsergenerator von *oops* dar:

```
$ java oops.EBNF ebnf.ebnf > ebnf2.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using ebnf parser generator generated by oops on Thu Sep  6 15:42:43 MEST 2001
$ ls -l ebnf.ser ebnf2.ser
-rw-r--r--   1 bernd    staff        2995 Sep  6 14:57 ebnf.ser
-rw-r--r--   1 bernd    staff        2995 Sep  7 13:19 ebnf2.ser
$
```

Analog zu EBNF ist auch der Parsergenerator zu XEBNF in einer Factory gekapselt. Diese hat aber nur noch zwei Methoden gegenüber ihrer Oberklasse zu ersetzen:

```
public class OopsCompilerFactory extends EBNFCompilerFactory  {
    protected oops.parser.goal.Parser getParser() throws
      StreamCorruptedException, OptionalDataException, ClassNotFoundException,
      IOException {
        return getParser("oops/oops/oops.ser");
    }

    protected String getMessage() throws IOException {
        return "Using oops generated by oops on "+
               getContentFromSystemResource("oops/oops/oops.date");
    }
}
```

Analog zu `oops.EBNF` verkapselt `oops.Oops` das Setzen der Factory-Property und den Aufruf von `oops.opi.Compile`. Damit stellt `oops.Oops` den Parsergenerator für XEBNF-Grammatiken dar:

```
$ PROPS=-Doops.opi.CompilerFactory=oops.oops.OopsCompilerFactory
$ java $PROPS oops.opi.Compile xebnf.xebnf > xebnf2.ser
Using oops generated by oops on Thu Sep  6 15:42:45 MEST 2001
$ ls -l xebnf.ser xebnf2.ser
-rw-r--r--   1 bernd    staff        4404 Sep  6 13:35 xebnf.ser
-rw-r--r--   1 bernd    staff        4404 Sep  7 13:27 xebnf2.ser
$ cmp xebnf.ser xebnf2.ser
$ java oops.Oops xebnf.xebnf > xebnf2.ser
This is oops version 1.10
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using oops generated by oops on Thu Sep  6 15:42:45 MEST 2001
$ cmp xebnf.ser xebnf2.ser
$
```

10.2.5 Klassenversionen, ein zweites Problem

Wie gezeigt worden ist, scheint der obige Ansatz für einen Parsergenerator, daß die Instanzen der `Goal`-Klassen zu erkannten Teilen einer Grammatik die Knoten des Parser-Baums bauen, zu funktionieren. Leider ist dies nicht immer der Fall: Der *oops*-Parser für EBNF- bzw. XEBNF-Grammatiken besteht aus Objekten des `oops.parser`- bzw. des `oops.parser.goal`-Pakets. Mit dem ersten Ansatz bauen diese Objekte — über `Goal`-Instanzen — einen Baum von Objekten der gleichen Klassen. Wenn sich nun aber eine der Klassen des `oops.parser`-Pakets geändert hat, muß eine Instanz dieser Klasse ein Objekt einer neuen Klasse des gleichen Namens erzeugen. Auf dem Klassenpfad kann aber ohne weiteres nur eine Klasse je Name gefunden werden.

Abbildung 10.4 zeigt das Problem noch einmal. In dem Baum über den alten Paketen baut die `Rule`-Instanz für die `rule`-Regel der Grammatik über ein `Goal`-Objekt ein Objekt der neuen `Rule`-Klasse (hier mit `Rule′` bezeichnet):

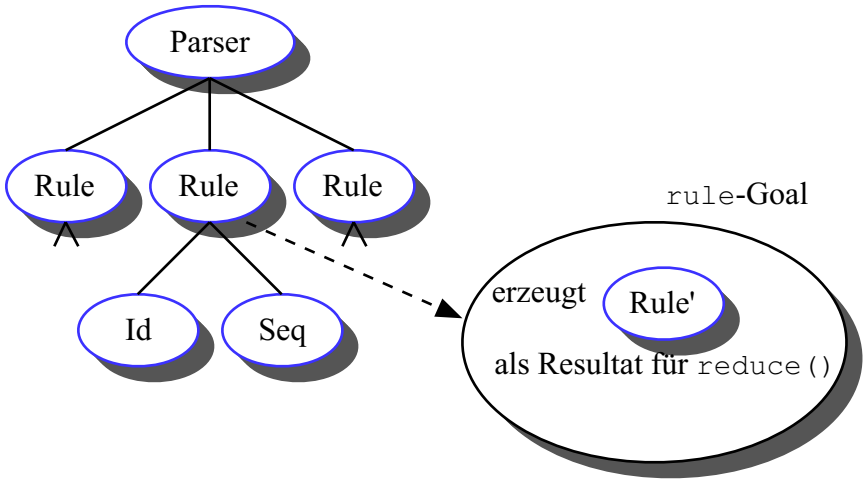

Abbildung 10.4: Unterschiedliche Rule-Klassen als Problem.

Wie eine unveröffentlichte Studie von Axel-Tobias Schreiner gezeigt hat, kann die sorgfältige Trennung der Klassenversionen durch den komplexen Einsatz von zwei weiteren Klassenladern erreicht werden:

Auf dem System-Klassenlader werden nur die Klassen gesucht, die sich zwischen den verschiedenen Versionen von *oops* nicht ändern.

Der EBNF-Parser der alten Version von *oops* besteht aus bzw. benutzt Objekte der alten Paketversionen (`oops.parser`, `oops.parser.goal` und `oops.scanner`). Durch das Ersetzen der Methode `resolveClass(ObjectStreamClass)` deserialisiert ein modifizierter `ObjectInputStream` die Klassen mit Hilfe eines eigenen Klassenladers. Dieser Klassenlader hat die *jar*-Datei der alten Paketversionen auf dem Klassenpfad.

Instanzen der durch den Klassenlader der `ObjectInputStream`-Instanz geladenen `Goal`-Klassen erzeugen zur Laufzeit des Parsers mit Hilfe eines zweiten, eigenen Klassenladers den neuen Parser aus Instanzen von Klassen der neuen Paketversionen. Dazu enthält der Klassenpfad des zweiten, eigenen Klassenladers die neuen Paketversionen.

Trotzdem habe ich mich für *oops* gegen diesen Ansatz entschieden:

Der Ansatz ist zwar technisch machbar, erfordert aber eine sehr sorgfältige Trennung der verschiedenen Klassen, der Instanzen der Klassen und der Klassenversionen. Die Technik ist meines Erachtens für einen produktiven Einsatz zu kompliziert.

Die Technik der Klassenlader hat sich von Java 1.1 zu Java 1.2 geändert und wird sich für weitere Java-Versionen vielleicht wieder ändern. Die Studie beruht aufgrund des Einsatzes der Klasse `java.net.URLClassLoader` auf Java 1.2.

Weiterhin wäre der Ansatz nicht sprachunabhängig. In anderen Sprachen wie C++, ObjectiveC oder C# gibt es das Konzept eines Klassenladers nicht bzw. kann nicht als allgemein gegeben vorausgesetzt werden. Alle Ideen dieser Arbeit sollen aber sprachunabhängig sein.

Ein zweiter Ansatz

Nur wenn *oops* eine neue Version von *oops* erzeugt und sich seit der Erzeugung der alten *oops*-Version die Klassen der Bibliothek geändert haben, tritt das Klassenversions-Problem auf. In der alltäglichen Anwendung von *oops* durch den Anwender ist das Problem also nicht gegeben, und daher ist der erste Ansatz für den allgemeinen Fall gültig. Nur für die Übersetzung von *oops* selbst muß ein anderer Ansatz gewählt werden.

Das Problem ist, daß zwei Klassen mit dem gleichen Namen in einem Java-Prozeß verwendet werden. Die Lösung besteht nun darin, zwei Prozesse zu verwenden, und je Prozeß befindet sich eine andere Version der Klassen auf dem Klassenpfad.

Der *oops*-Parser als erster Prozeß arbeitet auf der alten Version der Klassen. Die `Goal`-Instanzen bauen aber als Repräsentation der Grammatik keinen Parser-Baum, sondern eine von der aktuellen Version der Pakete unabhängige Beschreibung auf.

In dem zweiten Prozeß mit den neuen Klassen auf dem Klassenpfad wird dann die unabhängige Beschreibung in den Parser-Baum gewandelt.

Die Kommunikation zwischen beiden Prozessen fließt nur in eine Richtung. Daher kann dies einfach durch eine Datei oder besser eine Pipe und durch eine Serialisierung und Deserialisierung der unabhängigen Beschreibung realisiert werden.

Ich habe mich bei der unabhängigen Beschreibung eines Parsers für eine Grammatik für ein Konstrukt aus `Object`-Arrays und `String`-Objekten entschieden. Beide Klassen sind Teil von Java und werden sich von *oops*- zu *oops*-Version nicht ändern, und Instanzen beider Klassen sind serialisierbar.

Als Beispiel wird die Grammatik

```
program : [ a ] "x" ;
a       : "a" "a" ^ { "b" } ;
```

durch folgende Beschreibung dargestellt:

```
new Object[] {                      // parser
  "Parser",
  new Object [] {                   // rule   program : [ a ] "x" ;
    "Rule",
    new Object [] { "Id", "program" },  // rule name
    new Object [] {                 // rhs
      "Seq",
      new Object [] { "Opt", new Object [] { "Id", "a" } },
      new Object [] { "Lit", "x" }
    }
  },                                // end rule program
  new Object [] {                   // rule   a : "a" "a" ^ { "b" } ;
    "Rule",
    new Object [] { "Id", "a" },    // rule name
    new Object [] {                 // rhs
      "XOr",
```

```
            new Object [] {
              "Seq",
              new Object [] { "Lit", "a" } ,
              new Object [] { "Lit", "a" }
            },
            new Object [] { "Some", new Object [] { "Lit", "b" } }
          }
        }                                   // end rule a
      };                                    // parser end
```

Je zu erzeugender Knoten eines Parser-Baums wird ein `Object[]`-Objekt zur Beschreibung verwendet. Das Array hat immer auf der ersten Position als String den Klassennamen des Knotens (allerdings ohne den Paketnamen), und auf den weiteren Positionen folgen wiederum als `Object[]` oder als `String` die Unterknoten bzw. Informationen.

An dieser Stelle stellt sich die Frage, wieso die Beschreibung zum Beispiel nicht in Form von XML geschieht. Die Erzeugung der Beschreibung durch `Goal`-Objekte des Parsers wäre einfach, aber das Einlesen der XML-Beschreibung im zweiten Prozeß müßte durch einen XML-Parser geschehen, der aufgrund der verschiedenen Klassenversionen nicht mit *oops* realisiert werden kann. Der XML-Parser müßte also von Hand implementiert werden, oder zum Übersetzen von *oops* würde ein anderes Parser-System benötigt. Ein Parser (`oops.oops.ParserAsObjectArrayToGoalParser`) für die obige Beschreibung ist dagegen sehr einfach von Hand zu realisieren.

Zur Übersetzung von *oops* wird nun der *oops*-Parsergenerator aus Instanzen der alten Klassen der Bibliothek ausgeführt. Als `Goal`-Objekte werden Instanzen der Klassen des `oops.oops.bootgoals`-Pakets verwendet, die die unabhängige Beschreibung aufbauen. Die Beschreibung wird, wie erklärt, auf der Standardausgabe serialisiert ausgegeben. In dem zweiten Prozeß wird mit den neuen Klassen der Pakete auf dem Klassenpfad durch Ausführung von `ParserAsObjectArrayToGoalParser` diese Beschreibung eingelesen, deserialisiert, daraus der neue Parser — bestehend aus Instanzen der neuen Klassen der Bibliothek — erzeugt und dieser auf der Standardausgabe wieder serialisiert:

```
$ export goalProps=-Doops.parser.goal.DefaultGoalMakerFactory.goalPackage=\
> oops.oops.bootgoals
$ java $goalProps oops.RunGoalParser ebnf.ser $input ebnf.ebnf > x
$ ls -l x
-rw-r--r--   1 bernd    staff        1163 Sep 10 11:57 x
$ java oops.oops.ParserAsObjectArrayToGoalParser < x > ebnf2.ser
$ ls -l ebnf.ser ebnf2.ser
-rw-r--r--   1 bernd    staff        2995 Sep 10 11:20 ebnf.ser
-rw-r--r--   1 bernd    staff        2995 Sep 10 11:57 ebnf2.ser
$ cmp xebnf.ser xebnf2.ser
$
```

Natürlich ist die Erzeugung der unabhängigen Beschreibung für EBNF bzw. XEBNF wieder (durch `oops.oops.EBNFBootCompilerFactory` und `oops.oops.OopsBootCompilerFactory`) in Factories gekapselt, welche wiederum direkt oder indirekt von `oops.oops.EBNFCompilerFactory` abstammen.

Dieser Ansatz ist sprachunabhängig. In jeder objekt-orientierten Sprache wird es Arrays und String-Objekte geben. Als einzige Bedingung müssen Objekte dieser Klassen serialisierbar sein. Sollte dies zum Beispiel für Arrays nicht der Fall sein, können diese aber auch durch serialisierbare Container-Objekte ersetzt werden.

Für alle `Goal`-Pakete — `oops.oops.bootgoals`, `oops.oops.goals`, `oops.oops.goals.goal`, `oops.oops.goals.reducer` und `oops.oops.goals.trace` — wird der gleiche Scanner verwendet. Bei der Übersetzung von *oops* darf der Scanner im ersten Prozeß keinen Konflikt bezüglich

der Klassenversionen von `Lit` bzw. `Id` erzeugen. Daher liefert der Scanner für Namen und Literale immer ein `Object[]` -Objekt als Werte-Objekt.

10.3 T-Diagramme

Ein *oops*-Parser besteht aus zwei Arten von Quellen: Zum einen liegen die Aktionen (zum Beispiel als `Goal`-Klassen) und der Scanner als Programm-Dateien einer Programmiersprache (zum Beispiel Java) vor. Zum anderen beschreibt eine Grammatik (zum Beispiel in EBNF) die Syntax der vom Parser akzeptierten Programme. Wie Abbildung 10.5 noch einmal anhand von T-Diagrammen zeigt, besteht die Übersetzung eines Parser damit aus dem Aufruf des Java-Compilers für die Java-Quellen und aus dem Aufruf von *oops* zur Erzeugung des serialisierten Parsers:

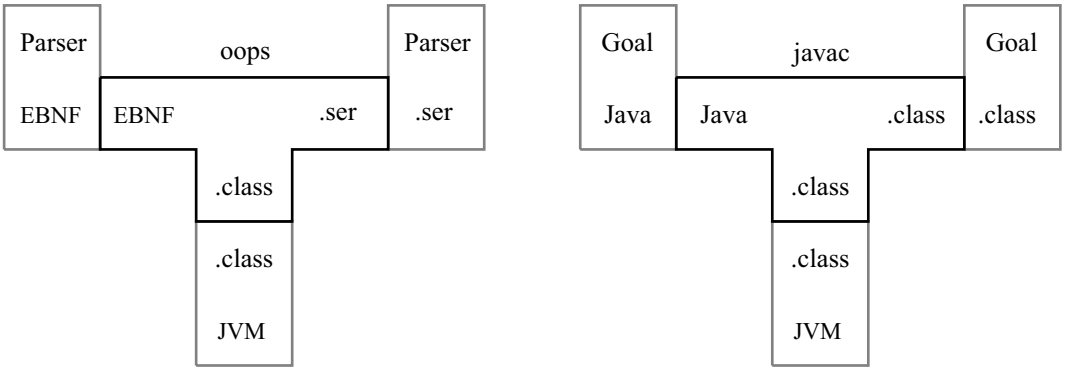

Abbildung 10.5: T-Diagramme einer Parser-Übersetzung.

Soll mit *oops* eine neue Version von *oops* übersetzt werden, so ist aufgrund des Klassenversions-Problems der T-Block zum Erzeugen des serialisierten Parsers aus der Grammatik in zwei unabhängige T-Blöcke zu zerlegen. Der erste T-Block generiert über den alten Parser-Klassen aus der Grammatik die unabhängige Beschreibung, und der zweite Block transferiert diese Beschreibung über den neuen Parser-Klassen in den serialisierten Parser-Baum. In Abbildung 10.6 sind die alte *oops*-Version und deren Klassen des Parser-Baums jeweils mit dem Index 1 und die neue *oops*-Version und die neuen Parser-Klassen mit dem Index 2 versehen:

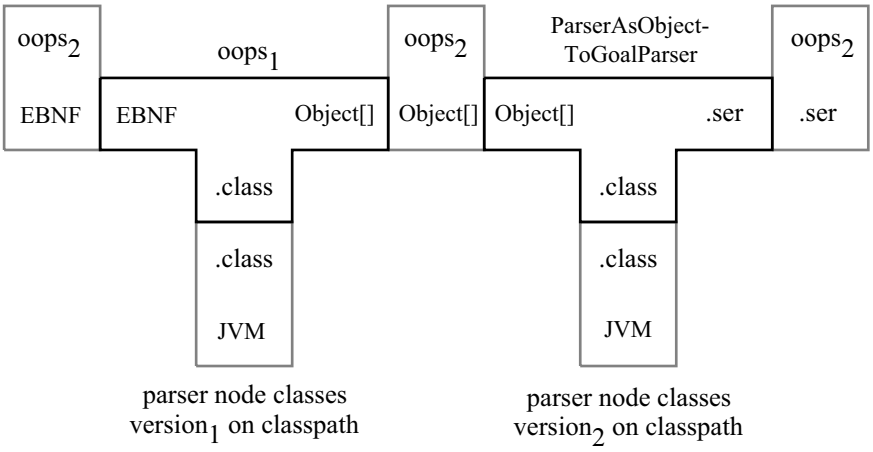

Abbildung 10.6: T-Diagramme einer oops-Übersetzung.

10.4 Fazit

Das Kapitel zeigt, daß eine Grammatik, welche Grammatiken beschreibt, und Aktionen, die zu Teilen einer erkannten Grammatik die zugehörigen Knoten des Parser-Baums bauen, zusammen mit einem

passenden Scanner bereits einen Parsergenerator für die durch die Grammatik beschriebenen Grammatiken darstellen. Vorgeführt wurden Parsergeneratoren für EBNF und XEBNF. Parsergeneratoren für jede beliebige andere Notation — wie zum Beispiel eine der diskutierten, alternativen XEBNF-Notationen — sind aber durch eine passende Grammatik und die zugehörigen `Goal`-Klassen schnell zu erschaffen. Als Beispiel wurde in der Vorlesung "XML — Architektur, Werkzeuge, Techniken" ([Sch01]) ein Parsergenerator für die in Kapitel 9.1 ("Grammatik-Repräsentation") vorgestellte und sich an regulären Ausdrücken anlehnende Grammatik-Notation verwendet.

Dadurch, daß dem Anwender durch `oops.EBNF` bzw. durch `oops.Oops` Klassen als Kommandozeilen-Werkzeuge oder auch als Methoden der Klassen zum Erzeugen von Parsern zur Verfügung stehen und dadurch, daß die Parsergeneratoren als `CompilerFactory` benutzt werden können, sind die Parsergeneratoren vom Anwender leicht zu nutzen.

Allerdings treten aufgrund der Verwendung der Objekt-Orientierung zwei interessante Probleme auf:

oops erzeugt neue Parser in Form von Bytes eines serialisierten Parsers, welche damit wiederverwendbar sind. Vor der Verwendung müssen die Bytes aber deserialisiert und überhaupt gefunden werden, was für den Grammatik-Parser der Parsergeneratoren in einer *jar*-Datei für *oops* so ohne weiteres nicht möglich ist. Die Factories sind in der Lage, auch Dateien auf dem Klassenpfad zu finden. Allgemein löst das Hinterlegen der Bytes als `byte`-Array in einer Klasse mit entsprechenden Methoden das Problem.

Zur Übersetzung einer neuen *oops*-Version erzeugen Objekte eines Pakets — über `Goal`-Objekte als Umweg — Objekte des gleichen Pakets. Haben sich Klassen in dem Paket geändert, sollen Instanzen einer älteren Klassenversion ein Objekt der neuen Klassenversion erzeugen. Nur mit einer komplizierten Klassenlader-Technik, die darüber hinaus auch nicht sprachunabhängig ist, ist das Problem zu umgehen. Die Erzeugung einer unabhängigen Beschreibung in einem ersten Prozeß und die Umsetzung der unabhängigen Beschreibung in den Parser-Baum über der neuen Klassenversion lösen das Problem einfach und sprachunabhängig.

Würde die Beschreibung nicht aus Objekten, sondern aus echtem Text bestehen, könnte sogar zum Beispiel eine Java-Version von *oops* die Beschreibung erzeugen und eine C#-Version von *oops* den Parser-Baum bauen. Damit wäre aber der Parser der unabhängigen Beschreibung komplexer.

Das Kapitel zeigt noch einmal, daß der Einsatz des Factory-Entwurfsmusters an den wesentlichen Stellen den Einsatz von *oops* sehr flexibel macht. Die beiden `CompilerFactory`-Klassen zum Kapseln der Parsergeneratoren oder die im Parsergenerator verwendete `GoalMakerFactory` sind Beispiele der These.

Die Parsergeneratoren sind ein gutes Beispiel für die Vorteile der Trennung von Grammatiken und Aktionen und für den Einsatz des `Goal`-Aktionsmusters:

Dadurch, daß `Goal`-Klassen spezielle Aktionen in Methoden verlegen, können Unterklassen durch Überschreiben dieser Methode das Verhalten der `Goal`-Instanzen leicht abändern. So leitet als Beispiel `oops.oops.goals.goal.seq` die Klasse `oops.oops.goals.seq` ab und erzeugt in `buildLit()` eine Instanz der Klasse `oops.parser.goal.Lit` anstelle von `oops.parser.Lit`.

Weiterhin werden hier die gleichen `Goal`-Klassen von verschiedenen Parsern verwendet. Sowohl EBNF- als auch der XEBNF-Parser teilen sich alle `Goal`-Klassen. Kein Duplizieren des Aktions-Programmtexts ist nötig. Exakt die gleichen Aktions-Klassen werden einfach von beiden Parsern verwendet.

11 Resümee und Ausblick

11.1 Resümee

Inhalt der Arbeit ist es, den Einsatz der Objekt-Orientierung im Compilerbau zu untersuchen. Die Arbeit hat gezeigt, daß die objekt-orientierte Programmierung auch auf diesem Gebiet gewinnbringend einzusetzen ist. Die in Kapitel 2.4 aufgestellten Thesen wurden vielfach bestätigt:

Wiederverwendung

Die Wiederverwendung von Klassen, aber auch die von Objekten, vereinfacht die Entwicklung und die Verwendung von Software auch auf dem Gebiet des Compilerbaus. Dies wurde in der Arbeit unter anderem durch folgende Beispiele gezeigt:

lolo, siehe Kapitel 3 (“Lexikalische Analyse”), beinhaltet neben den Klassen, welche das Framework eines *lolo*-Scanners ausmachen, eine Bibliothek fertiger Erkenner-Klassen für den Einsatz im Compilerbau. Alle Klassen sind in Form von Objekten für verschiedene Scanner wiederverwendbar. Existente Scanner können sogar serialisiert und damit von Ausführung zu Ausführung neu benutzt werden.

Wie in Kapitel 4 (“Ein Parser aus Objekten”) und in Kapitel 9 (“Grammatik-Notationen, Erweiterung durch Klassen”) gezeigt wird, kann eine beliebige EBNF- bzw. XEBNF-Grammatik als Baum von Objekten repräsentiert werden. Die Knoten-Klassen sind sogar unabhängig von der durch die Instanzen der Klassen repräsentierten Grammatik. Sie sind als Klassen allgemein für unterschiedliche Grammatiken, ja sogar für verschiedene Grammatik-Notationen, wiederverwendbar.

Der Baum von Objekten als Repräsentation einer Grammatik ist serialisierbar. Damit kann der Parser-Baum selbst wiederverwendet werden und muß nicht neu erzeugt und geprüft werden. In Java kann damit der Parser, ohne neu erzeugt zu werden, direkt auf verschiedenen Rechnern und Systemen zum Einsatz kommen.

Entwurfsmuster

Wie die Arbeit gezeigt hat, bringt der Einsatz verschiedener typischer Entwurfsmuster der Objekt-Orientierung auch den Compilerbau voran. Beispiele:

Das Factory-Entwurfsmuster verschiebt die Konstruktion von Objekten in den Verantwortungsbereich der Factory. Diese erzeugt die Objekte und entscheidet damit über die genaue Klasse der Objekte. Ist die Klasse der benutzten Factory vom Anwender auswählbar, so kann er eigene Factory-Klassen ins Spiel bringen und damit über die Klassen der Objekte entscheiden. Er kann dadurch zur Laufzeit in den Algorithmus eingreifen.

Als Beispiel werden in Kapitel 6 (“Parser-Aktionen”) `Goal`-Instanzen zur Ausführung von Benutzer-Aktionen zur Laufzeit eines *oops*-Parser von einer Factory erzeugt. Standardimplementierungen verschiedener Factories halten die Verwendung des Entwurfsmusters einfach. Der Anwender des Systems kann aber durchaus eine eigene Factory-Klasse schreiben und so — im Gegensatz zu den Standardimplementierungen — zum Beispiel pro Regel oder für alle Regeln immer das gleiche `Goal`-Objekt liefern.

Weiterhin wurde in Kapitel 5 (“Eine objekt-orientierte Scanner-Schnittstelle”) das Factory-Muster zur Auswahl der Tabellen eines Scanners oder in Kapitel 7 (“Die opi-Schnittstelle”) in der *opi*-API eingesetzt, was zum Beispiel im zweiten Fall die Werkzeuge `oops.EBNF` und `oops.Oops` als Schnittstelle zum Parsergenerator durch nur wenige Zeilen Programmtext realisierbar macht.

Das Template-Entwurfsmuster bereitet in Form von Methoden, welche von einem Algorithmus zu bestimmten Zwecken aufgerufen werden (*Hollywood-Prinzip*), eine Variation des Algorithmus durch das Überschreiben der Template-Methoden vor.

Durch diesen Ansatz wurden zum Beispiel in Kapitel 6 (“Parser-Aktionen”) die *oops*-Parser, welche anfänglich lediglich zur reinen Parsierung in der Lage waren, um drei unterschiedliche Arten von Benutzer-Aktionen zu erkannten Teilen einer Grammatik erweitert. Durch weitere Unterklassen können aber beliebige Aktionsarten dem Parser aufgezwungen werden.

divide & conquer

Die objekt-orientierte Programmierung ist ein Automatismus für *divide & conquer*, was auch im Compilerbau gewinnbringend genutzt werden kann. Wie gezeigt wurde, zerfallen selbst komplexe Algorithmen des Compilerbaus in kleine, einfache Teilprobleme und sind dadurch in der Lehre für Studierende leicht zu verstehen. Beispielhaft wurde dies an folgenden Stellen untersucht:

In Kapitel 3 (“Lexikalische Analyse”) arbeitet ein *lolo*-Scanner als Gesamtheit nach dem *divide & conquer*-Prinzip. Ein `Scanner`-Objekt delegiert an eine `Input`-Instanz die Verwaltung und Beschaffung der Eingabezeichen und delegiert das Erkennen der einzelnen Symbole an viele Objekte von `Scan`-Unterklassen.

Wie Kapitel 4 (“Ein Parser aus Objekten”) gezeigt hat, ist selbst die Modellierung eines so komplexen Systems wie ein Parser in Form von Objekten mit zugehörigen Methoden kein Problem und ein Beispiel für *divide & conquer*. Die Objekte der Parser-Klassen teilen die Verantwortung untereinander auf und entscheiden lokal über den Aufruf von Methoden bei anderen Objekten. Durch diesen Ansatz lösen sich die Berechnungen der lookahead- und follow-Mengen und die Prüfung und Parsierung der Grammatik in pro Klasse kleine Teilprobleme auf. Selbst der klassisch eher schwer zu begreifende Algorithmus zur Berechnung der follow-Menge ist dadurch leicht zu verstehen.

Auch der in Kapitel 8 (“Fehlerbehandlung durch Objekte und Exceptions”) vorgestellte Algorithmus zur Fehlerbehandlung in einem Parser nach der Technik des rekursiven Abstiegs ist ein weiteres Beispiel für *divide & conquer* im Compilerbau. Eine Kette von Objekten modelliert die Aufrufverschachtelung des rekursiven Abstiegs nach, und im Fehlerfall entscheidet ein Objekt der Kette lediglich lokal über eine mögliche Fehlererholung. Die Kette als Ganzes stellt den komplexen und auf globaler Information bestehenden Algorithmus dar.

Erweiterbarkeit, Vererbung

Ein weiterer Vorteil der Objekt-Orientierung, daß objekt-orientiert konzipierte Software leicht zu erweitern ist, wurde im Rahmen dieser Arbeit auch für den Einsatz der Objekt-Orientierung im Compilerbau aufgezeigt. Unterklassen existenter Klassen erweitern objekt-orientierte Systeme äußerst einfach, da die neuen Klassen die Instanzvariablen und Methoden der Oberklasse erben und da sie die Oberklasse lediglich zu spezialisieren oder zu erweitern brauchen. Die Arbeit hat dies unter anderem an folgenden Stellen aufgezeigt:

Das *lolo*-Scannersystem aus Kapitel 3 (“Lexikalische Analyse”) ist leicht und sehr elegant erweiterbar. Neue Erkenner-Klassen müssen lediglich von der abstrakten Basisklasse `Scan` abstammen. Dank der Vererbung können neue Klassen aber auch bestehende Erkenner-Klassen ableiten, was spezialisierte Erkenner leicht zu implementieren macht. Instanzen der neuen Klassen nehmen ganz natürlich wie alle `Scan`-Objekte an dem Wettbewerb eines *lolo*-Scanners teil.

Auch *oops*-Parser sind, wie in Kapitel 4 (“Ein Parser aus Objekten”) gezeigt wurde, leicht erweiterbar. So stammt als Beispiel die Klasse `Some` von `Many` ab, übernimmt Instanzvariablen und Methodenimplementierungen und erweitert die Oberklasse und damit mögliche Parser schnell um die neue Funktionalität. Als weiteres Beispiel wurden *oops*-Parser in Kapitel 6 (“Parser-Aktionen”) um drei unterschiedliche Arten von Benutzer-Aktionen erweitert.

Die Klassen `And`, `Or` und `XOr` aus Kapitel 9 (“Grammatik-Notationen, Erweiterung durch Klassen”) haben darüber hinaus gezeigt, daß neue Knoten-Klassen die Parser sogar um neue Konstrukte der Erkennung erweitern können, die in BNF oder EBNF gar nicht oder kaum dargestellt werden können.

11.2 Fazit

Die Arbeit hat gezeigt, daß die typischen Vorteile der Objekt-Orientierung auch im Compilerbau wiederzuentdecken sind. Der Einsatz der Objekt-Orientierung bringt auch hier eine klarere Strukturierung und einen Zugewinn an Möglichkeiten, Flexibilität und Mächtigkeit.

Alle hier vorgestellten Ideen und Ansätze sind allgemein verwendbar und sprachunabhängig, d.h. nicht auf Java bezogen. So wurden das Scanner-System *lolo* und das Parser-System *oops* zum Beispiel nach C# portiert, und andere Entwickler haben bereits Interesse an einer Portierung von *oops* nach Ruby ([Ruby]) bekundet.

11.3 Ausblick

Die in dieser Arbeit vorgestellten Ideen untersuchen den Einsatz der Objekt-Orientierung nicht für alle Phasen des Compilerbaus. Auch bleiben für die betrachteten Phasen durchaus interessante Fragen offen, welche es sich lohnen würde, in einer weitergehenden Untersuchung zu bearbeiten. Daher werden im Folgenden kurz mögliche, interessante Ansätze einer weiteren Forschung bezüglich des Einsatzes der Objekt-Orientierung im Compilerbau skizziert.

Thread-sichere lolo-Scanner und oops-Parser

lolo-Scanner und *oops*-Parser sind momentan nicht Thread-sicher, d.h., zwei verschiedene Threads können nicht gleichzeitig den Programmtext des gleichen Scanners oder des gleichen Parsers nutzen.

Ein Anwender von *lolo* sollte nie die Methoden der gekapselten Erkenner-Instanzen selbst aufrufen, sondern wird immer das `Scanner`-Objekt zum Finden des nächsten Symbols auffordern. Genauso wird der Anwender immer beim `Parser`-Objekt und nicht bei den einzelnen Knoten des Parser-Baums die Prüfung oder die Parsierung per Methodenaufruf aktivieren.

Daher sollte es reichen, wenn die `Scanner`-Klasse und die `Parser`-Klasse Thread-sicher sein würden. Da der Aufruf von synchronisierten Methoden aber länger dauert als die von nicht synchronisierten, sollten die Klassen nur auf Wunsch Thread-sicher agieren. Dies könnte analog zu der Klasse `Collections` aus dem `java.util`-Paket geschehen, welche über Klassenmethoden nicht Thread-sichere `Collection`-Klassen in eine Instanz einer inneren, Thread-sicheren Klasse einwickelt und diese als Resultat liefert.

Für die `Scanner`-Klasse sähe das dann in etwa wie folgt aus:

```
public class Scanner {
    ... // old code
    private static class SynchronizedScanner extends Scanner {
        private final Scanner scanner;
        SynchronizedScanner(Scanner scanner) { this.scanner = scanner; }
        public synchronized Scan scan(Input input) throws ... {
            return scanner.scan(input);
        }
        ... // all synchronized methods like add(), remove(), ...
    }
    public static Scanner synchronizedScanner(Scanner scanner) {
        return new SynchronizedScanner(scanner);
    }
}
```

Die Klassenmethode `synchronizedScanner()` bekommt einen `Scanner` als Argument und liefert als Resultat einen Stellvertreter des Arguments, der als Instanz der inneren `Scanner`-Unterklasse `SynchronizedScanner` Thread-sicher ist. Analog wären *oops*-Parser optional in Thread-sichere Stellvertreter zu wandeln.

Kombination von Erkenner-Instanzen des lolo-Systems

Das *oolex*-System besitzt gegenüber dem *lolo*-System die Erkenner-Klassen `Alt`, `Opt`, `Seq` und `Loop`, deren Instanzen sich zur Kombination von Objekten existierender Erkenner-Klassen eignen. Wie *oolex* allerdings gezeigt hat, müssen dazu in den Kombinations-Objekten alle möglichen Wege der Erkennung der gekapselten Objekte parallel betrachtet werden.

In *oolex* wurde die parallele Betrachtung durch das (vielfache) Klonen der gekapselten Erkenner und deren parallelen Verwendung gelöst. Dies macht das System insgesamt aber sehr langsam, da das häufige Klonen von Erkenner-Objekten und die parallele Verarbeitung sehr zeitaufwendig sind.

Ein alternativer Ansatz könnte einen besseren Kombinations-Ansatz für *lolo* erforschen. Anstelle der parallelen Betrachtung vieler gekapselter, geklonter Erkenner könnte der neue Ansatz, neben allen in der aktuellen Erkennungsrunde aufgetretenen Zeichen, alle möglichen Kombinationen der Erkennung in einer inneren Struktur notieren und diese nacheinander auf die gekapselten Erkenner anwenden.

Die Kombinations-Klassen würden das System in der Anwendung sehr einfach, aber mächtig erweitern. Einfache Erkenner könnten analog zu regulären Ausdrücken zu komplexeren Objekten verknüpft werden. Dies erspart die Implementierung neuer, eigener Erkenner-Klassen.

Die Untersuchung wäre aus akademischer Sicht sicherlich spannend. Die Performance dürtfe aber auch hier eher schlecht sein, da der Unterhalt der Struktur und die Betrachtung aller Möglichkeiten der Erkennung der gekapselten Instanzen Zeit kosten. Allerdings würde der Zeitaufwand für das Klonen der Erkenner entfallen. Da die Verwendung der Kombinations-Klassen optional wäre, könnten diese aber zumindest für ein *rapid prototyping* genutzt werden.

Erweiterung von oops-Parsern um neue Knoten-Klassen

Wie in Kapitel 9.2 (“Extending EBNF”) gezeigt wurde, ist die Erweiterung von *oops*-Parsern durch neue Knoten-Klassen relativ einfach. Instanzen der neuen Knoten-Klassen integrieren sich ganz natürlich in den Parser-Baum und erweitern so die Funktionalität der Parser.

Ziel einer weiteren Forschung könnte es sein zu untersuchen, welche weitere Knoten-Klassen die Parser um welche Funktionalität ausbauen. Wie bereits beschrieben, könnte eine Instanz einer Klasse `Iteration` einen Unterknoten und zwei Zahlen `m` und `n` besitzen. Während der Parsierung wiederholt eine `Iteration` die Parsierung des Unterknotens zwischen `m`- und `n`-mal. Eine mögliche Grammatik-Notation könnte den regulären Ausdrücken entliehen werden:

```
rule :  ( "x" "y" ) <2,6>
```

In spitzen Klammern beschreibt diese Notation, wie oft der direkt links davon stehende Teil einer Regel wiederholt werden kann. Wie in dem Beispiel gezeigt, kann eine Klammerung den Bereich der Wiederholung auf mehrere Elemente der Regel erweitern.

Doch welche weiteren, sinnvollen Knoten-Klassen kann es geben? Welche Programme können damit parsiert werden, die ohne die neuen Knoten gar nicht oder nur sehr schwer zu erkennen sind? Welche Grammatiken entsprechen den neuen Möglichkeiten?

Konfigurationsdateien für lolo-Scanner

Das in Kapitel 3 (“Lexikalische Analyse”) vorgestellte objekt-orientierte *lolo*-Scannersystem kapselt viele Symbolerkenner-Objekte in einem Scanner als Modellierung eines Raums, in dem die Objekte in einem Wettbewerb um die Erkennung des nächsten Symbols stehen.

Momentan muß der Anwender des Systems die Erkenner-Instanzen programmatisch in Form von Java-Programmtext durch Auswahl des gewünschten Konstruktors erzeugen und über Methodenaufrufe zum Raum hinzufügen. Der fertige Scanner kann dann serialisiert und zu verschiedenen Zeitpunkten für verschiedene Anlässe wiederverwendet werden.

In einem Scanner-System wie *lex* wird der Scanner in Form von Text, d.h. durch reguläre Ausdrücke, beschrieben. *lex* generiert aus der Beschreibung eine Funktion, welche die beschriebenen Muster erkennt. Etwas Vergleichbares wäre vielleicht auch für *lolo* wünschenswert. Eine textuelle Konfigurationsdatei könnte die Elemente des Scanners beschreiben. Ein mögliches Beispiel:

```
Skip
    lolo.scans.JavaWhitespace, lolo.scans.JavadocComment;

lolo.scans.Char,
lolo.scans.Char char 'c',
lolo.scans.Word String "hello world!",
lolo.scans.Set String "+-/*" boolean true;
```

Inhalt einer Konfigurationsdatei wären in diesem Ansatz durch Kommata getrennte und durch ein Semikolon terminierte Liste von Beschreibungen der Erkenner-Objekte. Pro Beschreibung wählt der erste Eintrag (wie hier zum Beispiel `lolo.scans.Word`) als Text die Klasse des Erkenner-Objekts aus, und die weiteren Elemente der Beschreibung bestimmen den Konstruktor und die Argumente für die Konstruktion. Ein Scanner hat möglicherweise Symbole zu verwerfen, und daher sammelt eine optionale `Skip`-Sektion Beschreibungen von Erkennern für zu ignorierende Symbole (wie hier zum Beispiel `lolo.scans.JavaWhitespace`). Da diese Beschreibung keine Aktionen enthält, ist sie wieder sprachunabhängig und könnte von *lolo*-Implementierungen verschiedener Sprachen verwendet werden.

Ein Scannergenerator, der zum Beispiel mit *oops* realisiert werden könnte, liest diese Beschreibung, erzeugt die Erkenner-Instanzen, fügt diese zum Scanner hinzu und serialisiert am Ende den konfigurierten Scanner für eine spätere Verwendung. Vorteil dieses Ansatzes wäre es, daß der Anwender keinen Java-Programmtext zum Erzeugen eines Scanners schreiben müßte, ihm aber trotzdem die ganze Macht von *lolo* zur Verfügung stehen würde.

Die Idee birgt Probleme, die in einer Betrachtung näher untersucht werden müssen: Soll die Konfigurationsdatei Unicode unterstützen, muß das Encoding für zumindest `String`-Objekte und einzelne `char`-Werte festgelegt sein. Außerdem bleibt zu untersuchen, ob der einfache Ansatz auch allgemein für alle möglichen Parameter-Typen der Erkenner-Konstruktoren zu realisieren ist. Man denke nur an eine Instanz eines mehrdimensionalen Arrays als Argument eines Konstruktors. Vielleicht wäre eine Beschreibung in Form einer durch eine DTD wohldefinierte XML-Datei ein besserer Ansatz. Zumindest das in der XML-Datei verwendete Encoding wäre in der Datei selbst dokumentiert.

Zunächst unklar und damit auch weiter zu untersuchen bleibt, wie vom Benutzer zu definierende Aktionen elegant spezifiziert werden können. Der von dem Generator erzeugte Scanner könnte eine Schnittstelle in Form einer Methode unterstützen, welche unter dem Klassennamen eines Erkenners diesen als Verweis liefert. Damit können verschiedene Aktionen nachträglich zur Laufzeit angeschlossen werden. Als weitere Idee könnte der Programmtext der Aktionen Bestandteil der Konfigurationsdatei sein. In diesem Fall müßte der Generator wie bei *lex* Programmtext, d.h. hier Klassen, erzeugen, übersetzen und Instanzen davon kreieren. Außerdem wäre die Beschreibung nicht mehr sprachunabhängig. Meines Erachtens bietet sich ein Ansatz analog zu der `Goal`-Schnittstelle von

oops am besten an. Zur Laufzeit des Scanners werden die `Action`-Objekte der Erkenner von einer Factory erzeugt. Durch Standard-Implementierungen der Factory sollte dies dem Anwender leicht gemacht werden. Da er aber eigene Factory-Klassen schreiben kann, steht ihm die Art der Auswahl der Aktions-Objekte frei.

Automatischer Scanner-Anschluß an oops-Parser

Wie eine eigene, unveröffentlichte Studie gezeigt hat, kann die Konfigurationsdatei für *lolo*-Scanner auch für einen automatischen Anschluß eines *lolo*-Scanners an einen *oops*-Parser genutzt werden.

Bislang hat ein Anwender eines *oops*-Parsers den Scanner selbst zu implementieren. Durch eine Konfiguration des Scanners über die textuelle Beschreibung kann dies aber wesentlich schneller geschehen. Als Aktionen zu erkannten Symbolen hinterlegt der Scanner das Identifizierungs-Objekt für den Parser und als Werte-Objekt die für das Symbol erkannte Zeichenfolge als `String`-Objekt.

Die Verbindung zwischen der Symbol-Beschreibung in der Grammatik und den Erkenner-Instanzen des Scanners könnte in etwa so aussehen:

```
exprs   : [{ [ sum ] ";" }];
sum     : product [{ ("+" | "-") product }];
product : term [{ ("*" | "/") term }];
term    : lolo.scans.Flt | "(" sum ")";
```

Literale beschreiben `lolo.scans.Word`-Erkenner, und Token sind analog zu der Klasse des zugehörigen Erkenner-Objekts benannt. Da in der Konfigurationsdatei die zu ignorierenden Symbole beschrieben sind, braucht der Anwender für die Parsierung keinen Programmtext mehr zu schreiben, da sowohl der Parser als auch der Scanner generiert werden. In den Aktionen des Parsers kann der Anwender dann die `String`-Objekte der erkannten Symbole weiterverarbeiten.

Semantische Analyse, Interpreterbaum, Code-Generierung

Die Arbeit hat unter anderem den Einsatz der Objekt-Orientierung in der lexikalischen und syntaktischen Analyse, in dem Zusammenspiel der beiden Phasen, für unterschiedliche Arten von Benutzer-Aktionen, für die Parsergenerierung und für die Fehlerbehandlung untersucht. Doch die objekt-orientierte Programmierung kann auch auf den anderen Phasen eines Compilers angewendet werden.

Im Rahmen der Vorlesung “Compilerbau mit Java” hat Axel-Tobias Schreiner anhand des *CompilerKits* ([Sch98b]) gezeigt, daß ein erkanntes Programm einer Sprache nicht nur wieder als Baum von Objekten repräsentiert (Parse-Baum), sondern daß dieser Baum mit seinen Methoden zur semantischen Analyse des Programms herangezogen werden kann.

Ziel des *CompilerKits* war eine Klassenbibliothek, deren Klassen unter anderem die Variablen, die Kontrollstrukturen, die Funktionen und auch die verschiedenen Daten-Typen eines Programms modellieren. Die Klassen können zur Abbildung von Programmen unterschiedlicher Quell-Sprachen wiederverwendet werden und sollen leicht erweitert werden können.

Das *CompilerKit* muß modifizierbar und erweiterbar sein. Zum Beispiel sind die Regeln für das implizite Umwandeln von Werten von Sprache zu Sprache verschieden. Daher werden die Umwandlungsregeln pro Typen-Klasse in einer Klassenvariablen als Objekt einer speziellen Klasse hinterlegt, welches der Anwender bei Bedarf austauschen kann, um so die Regeln seiner Sprache anzupassen.

Die Instanzen eines Baums des *CompilerKits* werden durch die Methode `sem()` zur semantischen Analyse des repräsentierten Programms aufgefordert. So testet ein Addier-Knoten eines Ausdrucks, ob die Operanden den gleichen Typ haben, ob für diese Typen eine Addition erlaubt ist und ob gegebenenfalls der Wert der Typen implizit gewandelt werden kann.

Resultat der erfolgreichen semantischen Prüfung ist ein Baum einer zweiten Klassen-Bibliothek des *CompilerKits*. Instanzen der Klassen der zweiten Bibliothek repräsentieren das gleiche Programm, sind aber zur Interpretation des semantisch geprüften Programms in der Lage und beinhalten zum Beispiel gegebenenfalls einen von der semantischen Analyse erzeugten Umwandlungsknoten für implizite Umwandlungen.

Das Resultat des *CompilerKits* ist eine klare Trennung von Frontend und Backend eines Compilers, welche sich mit der Repräsentierung der Sprache bzw. der Analyse und Interpretation des Programms beschäftigen. Die Repräsentierung ändert man in den Quellen des Parsers und des Scanners. Die Analyse und Interpretation können dank des *CompilerKits* weitgehend wiederverwendet werden.

Das *CompilerKit* enthält noch keine Unterstützung für Strukturen und damit auch keine Unterstützung für objekt-orientierte Sprachen. Eine weitere Forschung könnte das *CompilerKit* dahingehend erweitern. Das Resultat der semantischen Analyse könnte auch ein Baum von Objekten sein, welcher zur Optimierung und zur Code-Generierung genutzt werden kann. Welche Vorteile die Objekt-Orientierung an dieser Stelle bringt, bleibt weiter offen.

LL(k)- und LR-Parser aus Objekten

Die Arbeit hat gezeigt, daß Objekte als Repräsentation einer Grammatik zur Berechnung der Symbolmengen, zur Prüfung der Grammatik und zur Parsierung von Programmen über der Grammatik in der Lage sind. Die vorgestellte Parsierung verwendet die Technik des rekursiven Abstiegs mit einem Symbol Vorschau, erlaubt also LL(1)-Grammatiken.

Interessant für eine weiterführende Arbeit wäre es zu untersuchen, ob und wie leicht dieser objekt-orientierte Ansatz für Parser auch auf LL(k)-Grammatiken ausgeweitet werden kann. Ist die Berechnung der Symbolmengen für eine größere Symbol-Vorschau genau so einfach, und wie sehen die Algorithmen zum Prüfen der Grammatiken bzw. zum Parsieren von Programmen aus?

Die Technik des rekursiven Abstiegs bot sich als natürliche Umsetzung einer Grammatik in Programmtext für eine erste Untersuchung, ob Parser elegant durch Objekte dargestellt werden können, an. LR(k)-Grammatiken und damit deren Parser sind aber mächtiger als LL(k)-Grammatiken und deren Parser. Ein nächster Schritt könnte daher eine Repräsentation eines LR(1)-Parsers aus Objekten untersuchen. Bietet dieser Ansatz Vorteile gegenüber dem klassischen, tabellengetriebenen Ansatz? Welche Aufgabe kommt dabei den einzelnen Objekten zu?

Vererbung von Grammatiken

Weiterführende Überlegungen könnten Vererbung für Grammatiken untersuchen. Als Beispiel eine `add` benannte Grammatik:

```
grammar add;
expr:  term {[  "+" term ]}  ;
term:  NUMBER;
```

Analog zu der Syntax von Java könnte für eine Grammatik eine Obergrammatik mit `extends` angegeben werden:

```
grammar mul extends add;
expr:  product {[  "+" product ]}  ;
product:  term {[  "*" term ]}  ;
```

In dem Beispiel erweitert die Grammatik `mul` die Grammatik `add`, ersetzt die Regel `expr` der Oberklasse, führt `product` als neue Regel ein und erbt `term`.

Als Fragen bleiben unter anderem offen: Welche Vorteile bietet eine Vererbung für Grammatiken? Als Beispiel könnte dies für die erzeugten Parser oder für Benutzer-Aktionen der Parser untersucht werden. Was bedeutet hier eine Mehrfachvererbung?

Zumindest eine objekt-orientierte `AST`-Unterstützung sollte dieser Ansatz möglich machen, da die Entwicklungsschritte anhand der Grammatik-Vererbungshierarchie sichtbar sind.

12 Anhang

12.1 Bibliographie

[Aho83] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, *Data structures and algorithms*, Addison Wesley, 1983.

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilerbau*, Bell Laboratories, 1986/87.

[Amm78] Urs Ammann, Error recovery in recursive descent parsers, Report Nr. 25, Institut für Informatik der ETH, Zürich, 1978.

[Antlr] Homepage ANTLR (ANother Tool for Language Recognition).
`http://www.antlr.org`

[App97] Andrew W. Appel, *Modern compiler implementation in Java*, Cambridge University Press, 1997.
`http://www.cs.princeton.edu/~appel/modern/java`

[App01] Apple Computer Inc., Mac OS X homepage.
`http://www.apple.com/macosx/`

[Ben99] Brent W. Benson, Reflections: Inner Classes: Closures for the Masses, *SIGPLAN Notices of ACM*, 02/1999.

[Bor79] Richard Bornat, *Understanding and writing compilers*, The MacMillan Press, 1979.

[Bud02] Timothy A. Budd, *An Introduction to Object-Oriented Programming*, 3rd Edition, Addison Wesley, 2002.

[Cox91] Brad J. Cox, Andrew J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach*, 2nd edition, Addison-Wesley, 1991.

[Cup] Homepage Cup.
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[DIN66261] DIN 66261, Sinnbilder für Struktogramme nach Nassi-Shneiderman.

[Don91] Ch. Donnelly, R. M. Stallman, BISON the YACC-compatible parser generator. Technical report, Free Software Foundation, 1991.

[Ell01] Frank Eller, *C# lernen*, Addison Wesley, 2001.

[Ens00] Oliver Enseling, Build your own languages with JavaCC, JavaWorld, 2000.
`http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html`

[Fla97] David Flanagan, Java in a Nutshell, 2nd Edition, O'Reilly, 1997.

[Gag98a] Étienne M. Gagnon, *SableCC — an object-oriented compiler framework*, School of Computer Science, McGill University, Montreal, 1998.

[Gag98b] Étienne M. Gagnon, *SableCC — an object-oriented compiler framework*, 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (*TOOLS*-98).

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns — Elements of Reusable Object-Orientated Software*, Addison-Wesley, 1995.

[Gol83] Adele Goldberg and David Robson, *Smalltalk-80 the language and its implementation*, Xerox PARC Research Center, Addison-Wesley, 1983.

[JavaCC] Homepage Java Compiler Compiler (JavaCC). `http://www.webgain.com/products/java_cc`

[Javadoc] Homepage Javadoc 1.3. `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/index.html`

[JAXP] Homepage JAXP (Java API for XML Processing). `http://java.sun.com/xml/jaxp/index.html`

[Jay] Homepage jay. `http://www.inf.uos.de/bernd/jay`

[JLex] Homepage JLex. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[Joh75] S. C. Johnson. YACC — Yet Another Compiler-Compiler. Computer Science Technical Report 1975, Bell Laboratories, Murray Hill, NJ, 1975.

[Knu64] Donald E. Knuth, Backus-Normal-Form vs Backus-Naur-Form, *Communications of the ACM*, 7(12);735-736, 1964.

[Kra02] Jan Kraneis, *oops — Ein objekt-orientierter Parser-Generator für C#*, Bachelorarbeit, Universität Osnabrück, 2002.

[Krü00] Guido Krüger, *Go To Java 2*, 2. Auflage, Addison-Wesley, 2000.

[Küh99] Bernd Kühl, Axel-Tobias Schreiner, jay — ein yacc für Java, *iX* 10/99, Heise Verlag, Germany.

[Küh00a] Bernd Kühl, Axel-Tobias Schreiner, An object-oriented LL(1) parser generator, *SIGPLAN Notices of ACM*, 12/2000.

[Küh00b] Bernd Kühl, *oolex* — Lexikalische Analyse mit Objekten, IFC-Seminar Sommersemester 2000, Mai 2000, Universität Osnabrück, Germany. `http://www.inf.uos.de/bernd/talks/oolex-ifc00/pdf/talk.pdf`

[Küh00c] Bernd Kühl, Axel-Tobias Schreiner, *Lex für Java*, iX, Heft 03/00, Verlag Heinz Heise.

[Küh01a] Bernd Kühl, oops — Ein Parser aus Objekten, Forschungsseminar Sommersemester 2001, Mai 2001, Humboldt-Universität Berlin. `http://www.inf.uos.de/bernd/talks/berlin01/talk.pdf`

[Küh01b] Bernd Kühl, Axel-Tobias Schreiner, Objects for Lexical Analysis, submitted to *SIGPLAN Notices of ACM*.

[Les75] M. E. Lesk, Lex - a lexical analyzer generator, Tech. Rep. Computing Science, Technical Report 39, Bell Laboratories, 1975.

[LoloDoc] Dokumentation der Pakete von lolo. `http://www.inf.uos.de//bernd/lolo/staff/doc/index.html`

[Lolo] Homepage lolo. `http://www.inf.uos.de/bernd/lolo`

[Met01a] Steven John Metsker, *Building Parsers with Java*, Addison-Wesley, 2001. http://cseng.aw.com/book/0,3828,0201719622,00.html

[Met01b] Steven John Metsker, Tutorial Building Parsers with Java, *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2001.

[Mic01] Microsoft Corporation, C# Language Specification, Version 0.28, 2001. http://msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp

[MifDoclet] Homepage MIF Doclet. http://java.sun.com/j2se/javadoc/mifdoclet/index.html

[Nas73] I. Nassi, B. Shneiderman, Flowchart techniques for structured programming, *SIGPLAN Notices of ACM*, 8/1973.

[Nau63] Peter Naur, Revised report on the algorithmic language Algol 60, *Communications of the ACM*, 6(1):1-17, 1963.

[Oba01] Dare Obasanjo, *A comparison of Microsoft's C# programming language to Sun microsystem's java programming language*, 2001. http://www.25hoursaday.com/CsharpVsJava.html

[Oolex] Homepage oolex. http://www.inf.uos.de/bernd/oolex

[OopsDoc] Dokumentation der Pakete von oops. http://www.inf.uos.de/bernd/oops/staff/javadoc_oops1.1/index.html

[Oops] Homepage oops. http://www.inf.uos.de/bernd/oops

[Par93] Terence Parr, *Practical LL(k) and LR(k) for k>1*, PhD thesis, Purdue University, 1993.

[Pax95] V. Paxson, *Flex - Fast lexical analyzer generator*, Lawrence Berkeley Laboratory, 1995. ftp://ftp.ee.lbl.gov/flex-2.5.4.tar.gz

[Rig96] Roger Riggs, Jim Waldo, Ann Wollrath, Pickling State in the Java(tm) System, *USENIX Conference on Object-Oriented Technologies*, 1996.

[Ruby] Homepage Ruby. http://www.ruby-lang.org/en/index.html

[SableCC] Homepage SableCC. http://www.sablecc.org

[SAX] Homepage SAX. http://www.saxproject.org

[Sch85] Axel-Tobias Schreiner, George H. Friedman, *Compiler bauen mit Unix*, Carl Hanser Verlag, 1985.

[Sch92] Franz Josef Schmitt, *Praxis des Compilerbaus*, Hanser Verlag, 1992.

[Sch94] Axel-Tobias Schreiner, *Objekt-Orientierte Programmierung mit ANSI-C*, Carl Hanser Verlag, 1994.

[Sch98a] Axel-Tobias Schreiner, Vorlesung “Compilerbau mit Java”, Fachbereich Informatik/Mathematik, Universität Osnabrück, Wintersemester 1998/99. http://www.vorlesungen.uos.de/informatik/compilerbau98

[Sch98b] Axel-Tobias Schreiner, *Compilerkit*, aus Wiederverwendung und Compiler, Vorlesung “Compilerbau mit Java”, Fachbereich Informatik/Mathematik, Universität Osnabrück, Wintersemester 1998/99.
http://www.vorlesungen.uos.de/informatik/compilerbau98/html/skript/04_mixed.htmld/index.html

[Sch99a] Axel-Tobias Schreiner, Bernd Kühl, Ausnahmezustand, *iX*, Heft 11/99, Verlag Heinz Heise. http://www.ix.de/ix/artikel/1999/11/194/

[Sch99b] Axel-Tobias Schreiner, Bernd Kühl, *Object-oriented Compiler Construction*, Human and Computer '99, 1999, University of Aizu, Japan.
http://www.inf.uos.de/bernd/Vortraege/aizu_paper.pdf

[Sch01] Axel-Tobias Schreiner, Vorlesung “XML — Architektur, Werkzeuge, Techniken”, Sommersemester 2001, Fachbereich Informatik/Mathematik, Universität Osnabrück.
http://www.vorlesungen.uos.de/informatik/xml01

[Wir86] N. Wirth, *Compilerbau — Eine Einführung*, 4. Auflage, Teubner Studienbücher, 1986.

12.2 Abgrenzung

Die erste Version von *oops* wurde von Axel-Tobias Schreiner im Rahmen der Vorlesung “Compilerbau mit Java” ([Sch98a]) entwickelt. Mit Hilfe von *oops* wurde die Idee der Repräsentierung von Grammatiken und Parser durch generische Objekte und wurden die Algorithmen zum Berechnen der follow- und lookahead-Mengen und die Technik der Parsierung durch rekursiven Abstieg den Studierenden erläutert.

Ich habe, ausgehend von dieser Version, an der Idee der Repräsentierung von Grammatiken und Parser durch generische Objekte weitergearbeitet:

Einige Fehler in den Algorithmen wurden von mir gefunden und verbessert. So stammt u.a. von mir der Algorithmus zum Finden von unendlichen Rekursionen innerhalb einer Grammatik.

Die automatische Fehlererholung wurde von mir entwickelt und den Klassen eines Parsers hinzugefügt. Die Technik der Fehlererholung basiert auf dem Artikel “Ausnahmezustand” [Sch99a].

Die Idee des Hinzufügens von Aktionen durch Ableiten der parsierenden Baumklassen ist neu und wurde von mir mit den Trace-, `Reducer`- und `Goal`-Aktionen vorgeführt.

Ich habe die Scanner-Schnittstelle durch die jetzige Version inklusive der `SymTab`-Symboltabelle ersetzt.

Von Sun's JAXP angeregt, habe ich das Factory-Muster an vielen Stellen in *oops* (`oops.opi.CompilerFactory`, `oops.parser.goal.GoalMakerFactory`, ...) verwendet und eine Klasse `oops.opi.InputSource` zur Repräsentation von Eingabequellen eingeführt.

Die Erweiterung der Funktionalität eines Parsers durch Klassen wie `Xor` und `And` sollte zeigen, daß der Einsatz der Objekt-Orientierung durch einfache Mittel sehr schnell zu mächtigen Erweiterungen führt.

Viele Ideen der Arbeit entstanden in den stetigen und immer sehr fruchtbaren Diskussionen mit meinem Betreuer Axel-Tobias Schreiner.

Das Kapitel 2.2 (“Objekt-Orientierung”) wurde inspiriert durch das Kapitel “Classes and Objects in Java” in [Fla97] und durch Passagen aus [Krü00].

Das Kapitel 2.1 (“Compilerphasen und deren klassische Implementierung”) wurde inspiriert durch Passagen aus [Aho86], [Wir86], [Sch98a] und [Sch92].

12.3 Mein Dank gilt ...

... Herrn Professor Dr. Axel-Tobias Schreiner für die sehr persönliche Betreuung der Arbeit und seine immer ideenreichen und wertvollen Anregungen.

... Herrn Professor Christoph Polze für die Gelegenheit, außerhalb der Universität Osnabrück über die Inhalte meiner Arbeit vortragen zu können.

... Frau Dr. Ute Schmid für ihre Hilfsbereitschaft und Unterstützung bezüglich vieler Aspekte meiner Arbeit.

... Frau Gerda Holmann, Frau Astrid Heinze, Herrn Stephan Jätzold und meiner Schwester Marion für das geduldige und humorvolle Korrekturlesen der Arbeit.

... Jan Kraneis für die Portierung von *oops* und *lolo* nach C# und für seine Hilfe bei dem Tutorial und dem Manual von *oops*.

... Frau Carol Schreiner für die besten Spinattaschen der ganzen Ramsau.

... allen Freunden und Verwandten, die mich in der Promotionszeit nachhaltig unterstützt haben.

12.4 Paketübersicht

Die folgenden Seiten führen die Klassen der in dieser Arbeit vorgestellten Pakete und unter anderem deren Variablen, Konstruktoren und Methoden tabellarisch auf, wobei aber nur die `public` oder `protected` deklarierten Elemente der Klassen dokumentiert sind.

Die Seiten sind in einer Rohversion durch Einsatz des Werkzeugs *javadoc* ([Javadoc]) und dem *MIF Doclet* ([MifDoclet]) generiert und daraufhin von Hand editiert worden. Die ausführliche von *javadoc* erzeugte HTML-Dokumentation zu den Paketen ist auf den Homepages von *lolo* ([Lolo], [LoloDoc]) und *oops* ([Oops], [OopsDoc]) einzusehen.

Die Pakete:

Package lolo

Description

The package `lolo` provides a framework for the competition.

Class Summary	
Interfaces	
`Scan.Action`	Action code for a recognized symbols is represented by `Action` objects.
Classes	
`Input`	A `Scanner` gets the characters from an `Input` instance.
`Scan`	The abstract base class for recognizer classes.
`Scan.State`	A simple class to mark the result for `nextChar()`.
`Scanner`	A `Scanner` collects `Scan` objects to maintain a lexical scanner.
Exceptions	
`Scanner.IllegalCharacterException`	An `IllegalCharacterException` is thrown, when no `Scan` instance matches the current character.

Input

Declaration

```
public class Input
```

Description

A `Scanner` gets the characters from an `Input` instance.

Member Summary	
Fields	
`protected char[]`	`buffer`
`protected int`	`filled`
`protected int`	`mark`
`protected int`	`next`
`protected final java.io.Reader`	`reader`
`protected int`	`startOfSymbol`
Constructors	
	`Input(Reader, int)`
Methods	
`static void`	`main(String[])`
`void`	`mark()`
`int`	`next()`
`void`	`pushBackToMarked()`
`protected void`	`setStartSymbol()`
`String`	`toString()`
`void`	`unmark()`
`protected void`	`unsetSymbol()`

Scan

Declaration

```
public abstract class Scan implements java.io.Serializable
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `lolo.scans.Scan`

Description

The abstract base class for recognizer classes.

Member Summary	
Nested Classes	
static interface class	Scan.Action
static class	Scan.State
Fields	
protected transient lolo.Scan.Action	action
protected boolean	ignore
Constructors	
	Scan()
Methods	
void	action(char[], int, int)
abstract boolean	equals(Object)
Scan.Action	getAction()
boolean	getIgnore()
abstract int	hashCode()
abstract Scan.State	nextChar(char)
abstract void	reset()
Scan	setAction(Scan.Action)
Scan	setIgnore(boolean)

Scan.Action

Declaration

```
public static interface Scan.Action
```

All Known Subinterfaces: `lolo.test.Test.Action`

Enclosing Interface: `lolo.Scan`

Description

Action code for a recognized symbols is represented by `Action` objects.

Member Summary	
Methods	
void	action(Scan, char[], int, int)

Scan.State

Declaration

```
public static class Scan.State implements java.io.Serializable
```

All Implemented Interfaces: `java.io.Serializable`

Enclosing Class: `lolo.Scan`

Description

A simple class to mark the result for `nextChar()`.

Member Summary	
Fields	
`boolean`	`found`
`boolean`	`more`
Constructors	
	`Scan.State()`
Methods	
`Scan.State`	`set(boolean, boolean)`
`String`	`toString()`

Scanner

Declaration

```
public class Scanner implements java.io.Serializable
```

All Implemented Interfaces: `java.io.Serializable`

Description

A `Scanner` collects `Scan` objects to maintain a lexical scanner. The characters are read from an `Input` object. A `Scanner` can be serialized to be reused.

Member Summary	
Nested Classes	
`static class`	`Scanner.IllegalCharacterException`
Fields	
`boolean`	`debug`
`protected int`	`max`
`protected int`	`next`
`protected boolean`	`packed`
`protected lolo.Scan[]`	`scans`
`transient lolo.Scan[]`	`table`
`protected boolean`	`unicode`
Constructors	
	`Scanner()`
	`Scanner(Scan)`
	`Scanner(Scan[])`
Methods	

Member Summary	
boolean	add(Scan)
boolean	add(Scan[])
boolean	contains(Scan)
protected void	enter(int, Scan)
boolean	getUnicode()
void	pack()
boolean	packed()
boolean	remove(Scan)
Scan	scan(Input)
void	setUnicode(boolean)

Scanner.IllegalCharacterException

Declaration

```
public static class Scanner.IllegalCharacterException extends java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Enclosing Class: `lolo.Scanner`

Description

An `IllegalCharacterException` is thrown, when no `Scan` instance matches the current character.

Member Summary	
Fields	
protected char	ch
Constructors	
	Scanner.IllegalCharacterException(char, String)
Methods	
char	getChar()
String	toString()

Package lolo.scans

Description

The package `lolo.scans` contains recognizers for typical programming language symbols: identifiers, numbers, strings, comments, whitespace, ...

Class Summary	
Classes	
`CComment`	A recognizer class to scan for (optional nested) C comments.
`Char`	A simple class to scan a single character.
`Flt`	A class to scan for floating point numbers.
`HashComment`	A recognizer class to scan for hash comments: all characters from `#` up to EOL.
`Int`	A class to scan for integer numbers.
`JavadocComment`	A recognizer class to scan for javadoc comments.
`JavaIdentifier`	Instances of this class scan for Java identifiers and use the methods `java.lang.Character.isJavaIdentifierStart()` and `java.lang.Character.isJavaIdentifierPart()` to check the characters.
`JavaWhitespace`	Instances of this class scan for one or more Java whitespace characters.
`QuotedChar`	A class to scan for one quoted character.
`QuotedText`	A class to scan for quoted text.
`Set`	A recognizer class to scan for a character from a set of characters.
`SetMN`	A recognizer class to scan for many characters from a set of characters.
`SimpleIdentifier`	Instances of this class scan for `[ _a-zA-Z][ _a-zA-Z0-9]*`.
`SimpleWhitespace`	Instances of this class scan for one or more characters from ASCII `0x00` - `0x20`.
`SlashSlashComment`	A recognizer class to scan for C++ comments: all characters from `//` up to the end of line.
`Word`	A recognizer class to scan for a string (a word).
`XMLStartOfElement`	A recognizer to scan for `<aElementName`.

CComment

Declaration

```
public class CComment extends lolo.scans.SlashSlashComment
```

All Implemented Interfaces: `java.io.Serializable`

Description

A recognizer class to scan for (optional nested) C comments. By default `CComment` will be ignored.

Member Summary	
Fields	
protected static final java.lang.String	chars
protected final boolean	nested
protected static final int[][]	newState
protected transient int	numberOpenComments
Constructors	
	CComment()
	CComment(boolean)
Methods	
Scan.State	nextChar(char)
void	reset()
String	toString()

Char

Declaration

```
public class Char extends lolo.scans.Set
```

All Implemented Interfaces: `java.io.Serializable`

Description

A simple class to scan a single character.

Member Summary	
Constructors	
	Char()
	Char(char)
Methods	
String	toString()

Flt

Declaration

```
public class Flt extends lolo.scans.Scan
```

All Implemented Interfaces: `java.io.Serializable`

Description

A class to scan for floating point numbers. A leading sign (+ or -) can be switched on or off.

Member Summary	
Fields	
protected boolean	allowLeadingSign
protected static final java.lang.String	chars
protected static final int[][]	newState
protected transient boolean	reset
protected transient int	state
protected final lolo.Scan.State	stateObject
Constructors	
	Flt()
	Flt(boolean)
Methods	
boolean	equals(Object)
boolean	getAllowLeadingSign()
int	hashCode()
Scan.State	nextChar(char)
void	reset()
String	toString()

HashComment

Declaration

```
public class HashComment extends lolo.scans.SlashSlashComment
```

All Implemented Interfaces: java.io.Serializable

Description

A recognizer class to scan for hash comments: all characters from # up to EOL. By default these symbols will be ignored.

Member Summary	
Fields	
protected static final java.lang.String	chars
protected static final int[][]	newState
Constructors	
	HashComment()
Methods	
Scan.State	nextChar(char)

Int

Declaration

```
public class Int extends lolo.scans.SetMN
```

All Implemented Interfaces: `java.io.Serializable`

Description

A class to scan for integer numbers.

Member Summary	
Constructors	
	`Int()`
	`Int(int)`
Methods	
`String`	`toString()`

JavadocComment

Declaration

```
public class JavadocComment extends lolo.scans.SlashSlashComment
```

All Implemented Interfaces: `java.io.Serializable`

Description

A recognizer class to scan for javadoc comments. By default these symbols will be ignored.

Member Summary	
Fields	
`protected static final java.lang.String`	`chars`
`protected static final int[][]`	`newState`
Constructors	
	`JavadocComment()`
Methods	
`Scan.State`	`nextChar(char)`

JavaIdentifier

Declaration

```
public class JavaIdentifier extends lolo.scans.SimpleIdentifier
```

All Implemented Interfaces: `java.io.Serializable`

Description

Instances of this class scan for Java identifiers and use the methods `java.lang.Character.isJavaIdentifierStart()` and `java.lang.Character.isJavaIdentifierPart()` to check the characters.

Member Summary	
Constructors	
	`JavaIdentifier()`
Methods	
`protected boolean`	`testPart(char)`
`protected boolean`	`testStart(char)`

JavaWhitespace

Declaration

`public class` **`JavaWhitespace`** `extends lolo.scans.SimpleWhitespace`

All Implemented Interfaces: `java.io.Serializable`

Description

Instances of this class scan for one or more Java whitespace characters. The test is done by the `java.lang.Character.isWhitespace()` method. By default these symbols will be ignored.

Member Summary	
Constructors	
	`JavaWhitespace()`
Methods	
`protected boolean`	`isWhitespace(char)`

QuotedChar

Declaration

`public class` **`QuotedChar`** `extends lolo.scans.Scan`

All Implemented Interfaces: `java.io.Serializable`

Description

A class to scan for one quoted character.

Member Summary	
Fields	
`protected transient boolean`	`end`
`protected transient boolean`	`escape`
`protected final char[]`	`found`
`protected transient boolean`	`middle`
`protected final char`	`quote`
`protected transient boolean`	`reset`
`protected transient boolean`	`start`

Member Summary	
protected final lolo.Scan.State	stateObject
Constructors	
	QuotedChar()
	QuotedChar(char)
Methods	
void	action(char[], int, int)
boolean	equals(Object)
protected char	getQuote()
int	hashCode()
Scan.State	nextChar(char)
void	reset()
String	toString()

QuotedText

Declaration

```
public class QuotedText extends lolo.scans.Scan
```

All Implemented Interfaces: java.io.Serializable

Description

A class to scan for quoted text. The quoted text can span more then one line.

Member Summary	
Fields	
protected transient char[]	help
protected transient boolean	inside
protected transient char	last
protected final char	quote
protected transient boolean	reset
protected final lolo.Scan.State	stateObject
Constructors	
	QuotedText()
	QuotedText(char)
Methods	
void	action(char[], int, int)
boolean	equals(Object)
protected char	getQuote()
int	hashCode()
Scan.State	nextChar(char)
void	reset()
String	toString()

Set

Declaration

```
public class Set extends lolo.scans.Scan
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `Char, SetMN`

Description

A recognizer class to scan for a character from a set of characters.

Member Summary	
Fields	
protected final boolean	inside
protected transient boolean	reset
protected final java.lang.String	set
protected final lolo.Scan.State	stateObject
Constructors	
	Set(String)
	Set(String, boolean)
Methods	
boolean	equals(Object)
int	hashCode()
Scan.State	nextChar(char)
void	reset()
String	toString()

SetMN

Declaration

```
public class SetMN extends lolo.scans.Set
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `Int`

Description

A recognizer class to scan for many characters from a set of characters.

Member Summary	
Fields	
protected transient int	jog
protected final int	m
protected final int	n

Member Summary	
Constructors	
	`SetMN(String, boolean, int, int)`
Methods	
`boolean`	`equals(Object)`
`int`	`hashCode()`
`Scan.State`	`nextChar(char)`
`void`	`reset()`
`String`	`toString()`

SimpleIdentifier

Declaration

```
public class SimpleIdentifier extends lolo.scans.Scan
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `JavaIdentifier`

Description

Instances of this class scan for `[ _a-zA-Z][ _a-zA-Z0-9]*`.

Member Summary	
Fields	
`protected transient boolean`	`reset`
`protected transient boolean`	`start`
`protected final lolo.Scan.State`	`stateObject`
Constructors	
	`SimpleIdentifier()`
Methods	
`boolean`	`equals(Object)`
`int`	`hashCode()`
`Scan.State`	`nextChar(char)`
`void`	`reset()`
`protected boolean`	`testPart(char)`
`protected boolean`	`testStart(char)`
`String`	`toString()`

SimpleWhitespace

Declaration

```
public class SimpleWhitespace extends lolo.scans.Scan
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `JavaWhitespace`

Description

Instances of this class scan for one or more characters from ASCII `0x00` - `0x20`. By default these symbols will be ignored.

Member Summary	
Fields	
protected transient boolean	reset
protected final lolo.Scan.State	stateObject
Constructors	
	SimpleWhitespace()
Methods	
boolean	equals(Object)
int	hashCode()
protected boolean	isWhitespace(char)
Scan.State	nextChar(char)
void	reset()
String	toString()

SlashSlashComment

Declaration

```
public class SlashSlashComment extends lolo.scans.Scan
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `CComment, HashComment, JavadocComment`

Description

A recognizer class to scan for C++ comments: all characters from `//` up to the end of line. By default these symbols will be ignored.

Member Summary	
Fields	
protected static final java.lang.String	chars
protected static final int[][]	newState
protected transient boolean	reset
protected transient int	state
protected final lolo.Scan.State	stateObject
Constructors	
	SlashSlashComment()
Methods	
boolean	equals(Object)
int	hashCode()

Member Summary	
Scan.State	nextChar(char)
void	reset()
String	toString()

Word

Declaration
`public class **Word** extends lolo.scans.Scan`

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `XMLStartOfElement`

Description

A recognizer class to scan for a string (a word).

Member Summary	
Fields	
protected transient int	index
protected transient boolean	reset
protected final lolo.Scan.State	stateObject
protected final char[]	word
Constructors	
	Word(String)
Methods	
boolean	equals(Object)
int	hashCode()
Scan.State	nextChar(char)
void	reset()
String	toString()

XMLStartOfElement

Declaration
`public class **XMLStartOfElement** extends lolo.scans.Word`

All Implemented Interfaces: `java.io.Serializable`

Description

A recognizer to scan for `<aElementName`.

Member Summary	
Fields	
protected boolean	start
Constructors	

Member Summary

	XMLStartOfElement(String)
Methods	
boolean	equals(Object)
int	hashCode()
Scan.State	nextChar(char)
void	reset()
String	toString()

Package lolo.test

Description

The package `lolo.test` contains some tests.

Class Summary	
Interfaces	
`Test.Action`	
Classes	
`PrintTypicalCompilerSymbols`	Used by all `TypicalCompilerScannerBy...` scanners to print the recognized symbols..
`Test`	An example to test all features (Serialization, removing symbols, Unicode scanners, Action objects, non serializable Action objects, ...) and classes.
`TimeUtil`	Used by all `TypicalCompilerScannerBy...` scanners to print a time in a human readable form.
`TypicalCompilerScannerByJLex`	An example to meassure the scanning time using JLex for typical symbols in compiler construction.
`TypicalCompilerScannerByLolo`	An example to meassure the scanning time using lolo for typical symbols in compiler construction.
`TypicalCompilerScannerByOOlex`	An example to meassure the scanning time using oolex for typical symbols in compiler construction.

PrintTypicalCompilerSymbols

Declaration

`public class` **`PrintTypicalCompilerSymbols`**

Description

Used by all `TypicalCompilerScannerBy...` scanners to print the recognized symbols.

Member Summary	
Fields	
`static boolean`	`print`
Constructors	
	`PrintTypicalCompilerSymbols()`
Methods	
`static void`	`character(char)`
`static void`	`comment(String)`
`static void`	`floating(String)`
`static void`	`identifier(String)`
`static void`	`integer(String)`
`static void`	`quotedChar(char)`

Member Summary	
static void	quotedText(String)
static void	whitespace(String)
static void	word(String)

Test

Declaration
```
public class Test
```

Description

An example to test all features (Serialization, removing symbols, Unicode scanners, Action objects, non serializable Action objects, ...) and classes.

Member Summary	
Nested Classes	
protected static interface class	Test.Action
Constructors	
	Test()
Methods	
static void	main(String[])

Test.Action

Declaration
```
protected static interface Test.Action extends lolo.Scan.Action, java.io.Serializable
```

All Superinterfaces: `lolo.Scan.Action, java.io.Serializable`

Enclosing Interface: `lolo.test.Test`

TimeUtil

Declaration
```
public class TimeUtil
```

Description

Used by all `TypicalCompilerScannerBy...` scanners to print a time in a human readable form.

Member Summary	
Constructors	
	TimeUtil()
Methods	
static void	printMilliSecons(String, long, PrintStream)

TypicalCompilerScannerByJLex

Declaration
```
public class TypicalCompilerScannerByJLex
```

Description

An example to meassure the scanning time using JLex for typical symbols in compiler construction.

Member Summary		
Constructors		
		TypicalCompilerScannerByJLex(InputStream)
		TypicalCompilerScannerByJLex(Reader)
Methods		
	static void	main(String[])
	boolean	yylex()

TypicalCompilerScannerByLolo

Declaration
```
public class TypicalCompilerScannerByLolo
```

Description

An example to meassure the scanning time using lolo for typical symbols in compiler construction.

Member Summary		
Constructors		
		TypicalCompilerScannerByLolo()
Methods		
	static void	main(String[])

TypicalCompilerScannerByOOlex

Declaration
```
public class TypicalCompilerScannerByOOlex
```

Description

An example to meassure the scanning time using oolex for typical symbols in compiler construction.

Member Summary		
Constructors		
		TypicalCompilerScannerByOOlex()
Methods		
	static void	main(String[])

Package oops

Description

The package `oops` provides a framework of tools to generate parsers from a grammar specification or to run different parsers.

Class Summary	
Classes	
`EBNF`	Runs *oops* for EBNF grammars to generate a parser for the grammar.
`Oops`	Runs *oops* for XEBNF grammars to generate a parser for the grammar.
`RunGoalParser`	Runs a serialized `oops.parser.goal.Parser` from the command line:
`RunParser`	Runs a serialized `oops.parser.Parser` from the command line:
`RunReducerParser`	Runs a serialized `oops.parser.reducer.Parser` from the command line:

EBNF

Declaration

`public class` **`EBNF`**

Direct Known Subclasses: `Oops`

Description

Runs *oops* for EBNF grammars to generate a parser for the grammar. This class just uses an `oops.oops.EBNFCompilerFactory` object as `oops.opi.CompilerFactory` and then uses the `oops.opi.Compile` class.

An instance of `oops.oops.EBNFCompilerFactory` builds an `oops.opi.Compiler` which generates different parser regarding to the system property `oops.oops.actionType`:

```
oops.oops.actionType=none
```

The parser is build from instances of classes from the `oops.parser` package. So the parser just parses and does no action for parsed input.

```
oops.oops.actionType=trace
```

The parser is build from instances of classes from the `oops.parser.trace` and the `oops.parser` packages. So the parses prints a trace while parsing the input.

```
oops.oops.actionType=goal
```

The parser is build from instances of classes from the `oops.parser.goal` and the `oops.parser` packages. So the parses use `oops.parser.goal.Goal` objects for action code. If the property isn't set, this is the default.

```
oops.oops.actionType=reducer
```

The parser is build from instances of classes from the `oops.parser.reducer` and the `oops.parser` packages. So the parses use `oops.parser.reducer.Reducer` objects for action code.

Member Summary	
Fields	
protected static final java.lang.String	ls
static final java.lang.String	version
Constructors	
	EBNF()
Methods	
Object	compile()
Object	compile(InputSource)
static void	main(String[])
protected void	setProperty()

Oops

Declaration

```
public class Oops extends oops.EBNF
```

Description

Runs *oops* for XEBNF grammars to generate a parser for the grammar. This class just uses an `oops.oops.OopsCompilerFactory` object as `oops.opi.CompilerFactory` and then uses the `oops.opi.Compile` class.

An instance of `oops.oops.OopsCompilerFactory` builds an `oops.opi.Compiler` which generates different parser regarding to the system property `oops.oops.actionType`:

```
oops.oops.actionType=none
```

The parser is build from instances of classes from the `oops.parser` package. So the parser just parses and does no action for parsed input.

```
oops.oops.actionType=trace
```

The parser is build from instances of classes from the `oops.parser.trace` and the `oops.parser` packages. So the parses prints a trace while parsing the input.

```
oops.oops.actionType=goal
```

The parser is build from instances of classes from the `oops.parser.goal` and the `oops.parser` packages. So the parses use `oops.parser.goal.Goal` objects for action code. If the property isn't set, this is the default.

```
oops.oops.actionType=reducer
```

The parser is build from instances of classes from the `oops.parser.reducer` and the `oops.parser` packages. So the parses use `oops.parser.reducer.Reducer` objects for action code.

Member Summary	
Constructors	
	Oops()
Methods	
Object	compile()
Object	compile(InputSource)
static void	main(String[])
protected void	setProperty()

RunGoalParser

Declaration

`public class` **`RunGoalParser`**

Description

Runs a serialized `oops.parser.goal.Parser` from the command line:

```
java oops.RunGoalParser [-g GoalMakerFactory] [-t TableFactory] parser.ser
                        scannerClass [<] source [> tree.ser]
```

`parser.ser` and `source` can be a file name or an URL. `parser.ser` points to the serialized parser and `source` is the input for the deserialized parser. An `oops.opi.InputSource` is created from `source` and an `oops.scanner.ValueScanner` from the class named `scannerClass` is created with the `InputSource` as the argument to the constructor of the scanner class.

Optional the options `-g` and `-t` choice a factory class name. The factory objects are created with the paramterless constructor.

Member Summary	
Constructors	
	`RunGoalParser()`
Methods	
`static void`	`main(String[])`

RunParser

Declaration

`public class` **`RunParser`**

Description

Runs a serialized `oops.parser.Parser` from the command line:

```
java oops.RunParser [-t TableFactory] parser.ser scannerClassName [<] source
```

`parser.ser` and `source` can be a file name or an URL. `parser.ser` points to the serialized parser and `source` is the input for the deserialized parser. An `oops.opi.InputSource` is created for `source` and an `oops.scanner.Scanner` from the class named `scannerClass` is created with the `InputSource` as the argument to the constructor of the scanner class.

Optional the options `-t` choice a factory class name. The factory object is created with the parameterless constructor.

Member Summary	
Constructors	
	`RunParser()`
Methods	
`static void`	`main(String[])`

RunReducerParser

Declaration

`public class` **`RunReducerParser`**

Description

Runs a serialized `oops.parser.reducer.Parser` from the command line:

```
java oops.RunReducerParser [-r ReducerMakerFactory] [-t TableFactory] parser.ser
                           scannerClass [<] source [> tree.ser]
```

`parser.ser` and `source` can be a file name or an URL. `parser.ser` points to the serialized parser and `source` is the input for the deserialized parser. An `oops.opi.InputSource` is created from `source` and an `oops.scanner.ValueScanner` from the class named `scannerClass` is created with the `InputSource` as the argument to the constructor of the scanner class.

Optional the options `-r` and `-t` choice a factory class name. The factory objects are created with the paramterless constructor.

Member Summary		
Constructors		
		`RunReducerParser()`
Methods		
	`static void`	`main(String[])`

Package oops.scanner

Description

The package `oops.scanner` provides a framework of interfaces and interface implementations to handle the scanner (lexical analyse) part of an *oops* parser.

Class Summary	
Interfaces	
`Scanner`	Describes what a scanner (the lexical analyse part) for an *oops*-generated parser must do.
`SymTab`	A symbol table interface.
`SymTab.Handle`	A `Handle` instance knows how and where to enter information (a symbol and a value object), it can be asked for the two information objects, it can remove the entry for the `Handle` in the `SymTab`, ...
`TableFactory`	A `TableFactory` is used by an *oops* parser to create the two tables for the scanner.
`ValueScanner`	Extends the `Scanner` interface to provide a method to retrieve a value object for the current scanned symbol.
Classes	
`DebuggerTableFactory`	An implementation of the `oops.scanner.TableFactory` interface.
`DefaultSymTab`	A default implementation of the `SymTab` interface.
`DefaultTableFactory`	A default implementation of the `TableFactory` interface.

DebuggerTableFactory

Declaration

```
public class DebuggerTableFactory extends oops.scanner.DefaultTableFactory
```

All Implemented Interfaces: `java.io.Serializable, TableFactory`

Description

An implementation of the `oops.scanner.TableFactory` interface. The created objects print some information inside the `get()` and `put()` methods.

Member Summary	
Constructors	
	`DebuggerTableFactory()`
Methods	
`SymTab`	`symbolTable()`
`Hashtable`	`tokenTable()`

DefaultSymTab

Declaration

```
public class DefaultSymTab implements oops.scanner.SymTab
```

All Implemented Interfaces: `java.io.Serializable, SymTab`

Description

A default implementation of the `SymTab` interface. Inside a `DefaultSymTab` a `java.util.Hashtable` is used to store the information. Because closure objects are used as value objects inside the hash table only one lookup is needed for all `Handle` operations.

Member Summary	
Fields	
protected final java.util.Hashtable	symtab
Constructors	
	DefaultSymTab()
Methods	
SymTab.Handle	get(String)
Enumeration	keys()
static void	main(String[])
String	toString()

DefaultTableFactory

Declaration

```
public class DefaultTableFactory implements oops.scanner.TableFactory
```

All Implemented Interfaces: `java.io.Serializable, TableFactory`

Direct Known Subclasses: `DebuggerTableFactory`

Description

A default implementation of the `TableFactory` interface.

Member Summary	
Constructors	
	DefaultTableFactory()
Methods	
SymTab	symbolTable()
Hashtable	tokenTable()

Scanner

Declaration

```
public interface Scanner
```

All Known Subinterfaces: `ValueScanner`

Description

Describes what a scanner (the lexical analyse part) for an *oops*-generated parser must do.

Member Summary	
Methods	
`boolean`	`advance()`
`boolean`	`atEnd()`
`void`	`scan(SymTab, Hashtable)`
`Object`	`symbol()`

SymTab

Declaration

```
public interface SymTab extends java.io.Serializable
```

All Superinterfaces: `java.io.Serializable`

All Known Implementing Classes: `DefaultSymTab`

Description

A symbol table interface.

Member Summary	
Nested Classes	
`static interface class`	`SymTab.Handle`
Methods	
`SymTab.Handle`	`get(String)`
`Enumeration`	`keys()`

SymTab.Handle

Declaration

```
public static interface SymTab.Handle
```

Enclosing Interface: `oops.scanner.SymTab`

Description

A `Handle` instance knows how and where to enter information (a symbol and a value object), it can be asked for the two information objects, it can remove the entry for the `Handle` in the `SymTab`, ...

A `Handle` should act as a closure to avoid more than one table lookup.

Member Summary	
Methods	
void	enter(Object, Object)
boolean	isEntered()
String	key()
void	remove()
Object	symbol()
Object	value()

TableFactory

Declaration

```
public interface TableFactory extends java.io.Serializable
```

All Superinterfaces: `java.io.Serializable`

All Known Implementing Classes: `DefaultTableFactory`

Description

A `TableFactory` is used by an *oops* parser to create the two tables for the scanner.

Member Summary	
Methods	
SymTab	symbolTable()
Hashtable	tokenTable()

ValueScanner

Declaration

```
public interface ValueScanner extends oops.scanner.Scanner
```

All Superinterfaces: `Scanner`

All Known Implementing Classes: `oops.oops.Input`

Description

Extends the `Scanner` interface to provide a method to retrieve a value object for the current scanned symbol.

Member Summary	
Methods	
Object	value()

Package oops.opi

Description

The package `oops.opi` (*oops* parser interface) provides a framework of interfaces and classes to create and run a parser or a compiler and to model an input source.

Class Summary	
Interfaces	
`Compiler`	An interface to run a compiler with an `InputSource` as input source.
`Parser`	An interface to run a parser with an `InputSource` as input source.
Classes	
`Compile`	Runs an `oops.opi.Compiler` in different ways and from different sources.
`Compiler.Result`	`Result` objects holds the result of a compilation and reflects wether no syntax error occured while parsing.
`CompilerFactory`	A `CompilerFactory` object knows how to create a `Compiler`.
`InputSource`	`InputSource` intends to represent a variety of input technologies in a uniform fashion.
`Parse`	Runs an `oops.opi.Parserr` in different ways and from different sources.
`ParserFactory`	A `ParserFactory` object knows how to create a `Parser`.
Exceptions	
`OpiException`	The execption for all opi operations.

Compile

Declaration

```
public class Compile
```

Description

Runs an `oops.opi.Compiler` in different ways and from different sources. Optional the `oops.opi.Compiler` is created.

Member Summary	
Constructors	
	`Compile()`
Methods	
`Compiler.Result`	`compile()`
`Compiler.Result`	`compile(InputSource)`
`static void`	`main(String[])`

Compiler

Declaration
`public interface` **`Compiler`**

Description

An interface to run a compiler with an `InputSource` as input source. Usually a `Compiler` is created by a `CompilerFactory`.

Member Summary	
Nested Classes	
`static class`	`Compiler.Result`
Methods	
`Compiler.Result`	`compile(InputSource)`

Compiler.Result

Declaration
`public static class` **`Compiler.Result`**

Enclosing Class: `oops.opi.Compiler`

Description

`Result` objects holds the result of a compilation and reflects wether no syntax error occured while parsing.

Member Summary	
Fields	
`boolean`	`parseOk`
`java.lang.Object`	`result`
Constructors	
	`Compiler.Result(boolean, Object)`

CompilerFactory

Declaration
`public abstract class` **`CompilerFactory`**

Description

A `CompilerFactory` object knows how to create a `Compiler`. The `CompilerFactory` class creates `CompilerFactory` objects from a class which name is reflected by a property.

Member Summary	
Constructors	
	`CompilerFactory()`
Methods	
`abstract Compiler`	`compiler()`
`static CompilerFactory`	`newInstance()`

InputSource

Declaration

`public class` **`InputSource`**

Description

`InputSource` intends to represent a variety of input technologies in a uniform fashion. The constructors permit an application to represent it's means of input. Methods like `getReader()` allow a scanner to choose a convenient access.

Member Summary	
Fields	
`protected java.lang.String`	`encoding`
`protected final java.io.InputStream`	`is`
`protected java.io.Reader`	`reader`
Constructors	
	`InputSource(InputStream)`
	`InputSource(InputStream, String)`
	`InputSource(Reader)`
	`InputSource(String)`
	`InputSource(String, String)`
	`InputSource(URL)`
	`InputSource(URL, String)`
Methods	
`String`	`getEncoding()`
`InputStream`	`getInputStream()`
`Reader`	`getReader()`
`void`	`setEncoding(String)`

OpiException

Declaration

`public class` **`OpiException`** `extends java.lang.Exception`

All Implemented Interfaces: `java.io.Serializable`

Description

The execption for all opi operations. An `OpiException` optional nests another `Throwable` object.

Member Summary	
Fields	
java.lang.Throwable	nest
Constructors	
	OpiException()
	OpiException(String)
	OpiException(String, Throwable)
Methods	
String	getMessage()
Throwable	getNestedThrowable()
void	printStackTrace()
void	printStackTrace(PrintStream)
void	printStackTrace(PrintWriter)

Parse

Declaration

```
public class Parse
```

Description

Runs an `oops.opi.Parserr` in different ways and from different sources. Optional the `oops.opi.Parserr` is created.

Member Summary	
Constructors	
	Parse()
Methods	
static void	main(String[])
boolean	parse()
boolean	parse(InputSource)

Parser

Declaration

```
public interface Parser
```

Description

An interface to run a parser with an `InputSource` as input source. Usually a `Parser` is created by a `ParserFactory`.

Member Summary	
Methods	
boolean	parse(InputSource)

ParserFactory

Declaration

```
public abstract class ParserFactory
```

Description

A `ParserFactory` object knows how to create a `Parser`. The `ParserFactory` class creates `ParserFactory` objects from a class which name is reflected by a property.

Member Summary	
Constructors	
	`ParserFactory()`
Methods	
`static ParserFactory`	`newInstance()`
`abstract Parser`	`parser()`

Package oops.tools

Description

The package `oops.tools` provides a framework of useful tools to dump a parser, to find a class by name, to encapsulate a serialized *oops* parser inside a class, ...

Class Summary	
Classes	
`Dump`	A tool to dump a serialized `oops.parser.Parser` to `System.err`.
`ParserSer2JavaFile`	`main()` reads a serialized `oops.parser.Parser` from `System.in` and writes on `System.out` a Java source file for a generated class.
`PrintSerializedObject`	A simple tool to print a serialized object (tree) to `System.err`.
`Util`	A collection of useful methods.

Dump

Declaration

`public class` **`Dump`**

Description

A tool to dump a serialized `oops.parser.Parser` to `System.err`.

Usage:

```
java oops.tools.Dump [parser.ser]
```

`parser.ser` is an URL or a file name referencing the serialized parser. If `parser.ser` is missing, the bytes are read from `System.in`.

Member Summary		
Methods		
	`static void`	`dump(Node)`
	`static void`	`main(String[])`

ParserSer2JavaFile

Declaration

`public class` **`ParserSer2JavaFile`**

Description

`main()` reads a serialized `oops.parser.Parser` from `System.in` and writes on `System.out` a Java source file for a generated class. The generated class stores the serialized *oops* parser in a `byte` array and delivers the *oops* parser thru `genParser()` or `main()`. See the description of the `main()` for a more in detail explanation.

Member Summary	
Methods	
static void	main(String[])

PrintSerializedObject

Declaration

```
public class PrintSerializedObject
```

Description

A simple tool to print a serialized object (tree) to `System.err`.

Usage:

```
java oops.tools.PrintSerializedObject [object.ser]
```

`object.ser` is an URL or a file name referencing the serialized bytes. If `object.ser` is missing, the bytes are read from `System.in`.

Member Summary	
Methods	
static void	main(String[])

Util

Declaration

```
public class Util
```

Description

A collection of useful methods.

Member Summary	
Methods	
static boolean	booleanSystemProperty(String, boolean)
static Class	classForName(String)
static URL	getSystemResource(String, Object)
static InputSource	inputSourceFromString(String)

Package oops.parser

Description

The package `oops.parser` provides a framework to model a grammar with objects of the framework classes. The object tree for a grammar is able to check the grammar for LL(1) and is able to parse (with an `oops.scanner.Scanner` as lexical part) a program over the grammar.

Class Summary	
Classes	
`Activation`	The base class for error recovery objects for `parse()` methods.
`Alt`	Represents alternatives:
`And`	Represents and elements:
`Id`	Represents identifier.
`Lit`	Rrepresents explicit, quoted string.
`Many`	Represents zero or more occurrences:`[{ node }]`.
`Node`	Describes what each node in an *oops*-generated parser tree can do.
`Opt`	Represents zero or one occurrence:
`Or`	Represents or elements:
`Parser`	Represents start symbol of an *oops*-generated parser:
`Rule`	Represents grammar rule: .
`Seq`	Represents a sequence of nodes.
`Set`	A `Set` manages symboles as one-elements sets, manages lookahead and manages follow sets.
`Some`	Represents one or more occurrences:
`Token`	Represents class of terminal symbols.
`XOr`	Represents xor elements:
Exceptions	
`CheckLL1Exception`	A `CheckLL1Exception` is thrown when a grammar represented by a parser tree is not LL(1).
`ParseException`	Thrown while parsing: trash at end, the parser is unchecked, ...
`ParserBuildException`	Thrown will building a parser tree; e.g.

Activation

Declaration

```
public abstract class Activation extends java.lang.Throwable
```

All Implemented Interfaces: `java.io.Serializable`

Description

The base class for error recovery objects for `parse()` methods.

Member Summary	
Fields	
protected final oops.parser.Activation	caller
Constructors	
protected	Activation(Activation)
Methods	
protected boolean	canUse()
protected void	expect(Set)
Throwable	fillInStackTrace()
protected final void	handle(Object)
protected final Object	info()
protected String	stack()
protected Object	up(Object)

Alt

Declaration

`public class **Alt** extends oops.parser.Node`

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `Or, XOr`

Description

Represents alternatives.

Member Summary	
Fields	
protected java.util.Vector	nodes
protected char	opChar
Constructors	
protected	Alt()
	Alt(Node)
	Alt(Node[])
Methods	
protected void	acceptEmptyElements(int)
void	add(Node)
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
int	degree()
Node	node()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)

Member Summary	
Set	setLookahead(Parser)
Object	sub(int)
String	toString()

And

Declaration
public class **And** extends oops.parser.Or

All Implemented Interfaces: jag.Node, java.io.Serializable

Description

Represents and elements.

Member Summary	
Constructors	
	And(Node)
	And(Node, boolean)
Methods	
protected void	acceptEmptyElements(int)
protected boolean	checkDeadLoop()
void	parse(Parser, Activation, Object)

CheckLL1Exception

Declaration
public class **CheckLL1Exception** extends java.lang.Exception

All Implemented Interfaces: java.io.Serializable

Description

A CheckLL1Exception is thrown when a grammar represented by a parser tree is not LL(1).

Member Summary	
Constructors	
	CheckLL1Exception()
	CheckLL1Exception(String)

Id

Declaration
public class **Id** extends oops.parser.Node

All Implemented Interfaces: jag.Node, java.io.Serializable

Description

Represents identifier.

Member Summary	
Fields	
protected java.lang.String	name
protected oops.parser.Node	peer
Constructors	
	Id(String)
Methods	
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
String	getName()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)
Set	setLookahead(Parser)
String	toString()

Lit

Declaration

```
public class Lit extends oops.parser.Node
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `oops.parser.goal.Lit, oops.parser.reducer.Lit, oops.parser.trace.Lit`

Description

Rrepresents explicit, quoted string.

Member Summary	
Fields	
protected java.lang.String	body
Constructors	
	Lit(String)
Methods	
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
String	getBody()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)
Set	setLookahead(Parser)
protected void	shift(Object, Parser)
String	toString()

Many

Declaration

```
public class Many extends oops.parser.Node
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `Opt, Some`

Description

Represents zero or more occurrences. This is the base class for `Some` and for `Opt`.

Member Summary	
Fields	
protected oops.parser.Node	node
Constructors	
	Many(Node)
Methods	
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
int	degree()
Node	node()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)
Set	setLookahead(Parser)
Object	sub(int)
String	toString()

Node

Declaration

```
public abstract class Node implements java.io.Serializable, jag.Node
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `Alt, Id, Lit, Many, Parser, Rule, Seq, Token`

Description

Describes what each node in an *oops*-generated parser tree can do.

Member Summary	
Fields	
protected transient oops.parser.Set	follow
protected oops.parser.Set	lookahead
Constructors	

Member Summary	
	Node()
Methods	
void	add(Node)
protected boolean	checkDeadLoop()
protected void	checkLL1(Parser)
protected abstract	collectLitsAndIds(Hashtable, Hashtable)
void	
int	degree()
final Set	getLookahead()
Node	node()
void	parse(Parser, Activation, Object)
protected abstract Set	setFollow(Parser, Set)
protected abstract Set	setLookahead(Parser)
Object	sub(int)

Opt

Declaration

```
public class Opt extends oops.parser.Many
```

All Implemented Interfaces: jag.Node, java.io.Serializable

Description

Represents zero or one occurrence.

Member Summary	
Constructors	
	Opt(Node)
Methods	
Node	node()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)
String	toString()

Or

Declaration

```
public class Or extends oops.parser.Alt
```

All Implemented Interfaces: jag.Node, java.io.Serializable

Direct Known Subclasses: And

Description

Represents or elements.

Member Summary	
Fields	
protected	repeat
java.lang.String	
protected final	special
java.util.BitSet	
Constructors	
	Or(Node)
	Or(Node, boolean)
Methods	
void	add(Node, boolean)
void	checkLL1(Parser)
void	parse(Parser, Activation, Object)
String	toString()

ParseException

Declaration

```
public class ParseException extends java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown while parsing: trash at end, the parser is unchecked, ...

Member Summary	
Constructors	
	ParseException()
	ParseException(String)

Parser

Declaration

```
public class Parser extends oops.parser.Node
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `oops.parser.goal.Parser, oops.parser.reducer.Parser, oops.parser.trace.Parser`

Description

Represents the root of an *oops*-generated parser.

Member Summary	
Fields	
protected int	errorOk
protected final int	NUMBER_OF_SHIFTS_AFTER_ERROR
protected transient	rn2r
java.util.Hashtable	

Member Summary	
protected java.util.Vector	rules
protected transient oops.scanner.Scanner	scanner
protected final oops.parser.Rule	start
protected java.util.Vector	symbolNumbers
protected oops.scanner.SymTab	symtab
protected transient java.util.Hashtable	tn2t
protected java.util.Hashtable	tokens
Constructors	
	Parser(Rule)
	Parser(Rule[])
Methods	
void	add(Rule)
protected boolean	advance()
protected boolean	atEnd()
void	check()
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
protected SymTab	copySymTab(TableFactory)
protected Hashtable	copyTokens(TableFactory)
protected Token	createToken(String, Set)
int	degree()
protected void	error(Node, Activation, Object)
protected void	error(Object)
protected Set	getLitSet(String)
protected Node	getPeer(String)
protected TableFactory	getTableFactoryFromProperty()
protected String	getToken(int)
final int	numberOfErrors()
protected void	parse(Activation)
boolean	parse(Scanner, TableFactory)
Set	setFollow(Parser, Set)
Set	setLookahead(Parser)
protected void	setSets()
protected final void	shift()
Object	sub(int)
protected Set	tokenSet()
String	toString()

ParserBuildException

Declaration

```
public class ParserBuildException extends java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown will building a parser tree; e.g. more then one rule with the same name.

Member Summary	
Constructors	
	`ParserBuildException()`
	`ParserBuildException(String)`

Rule

Declaration

```
public class Rule extends oops.parser.Node
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `oops.parser.goal.Rule, oops.parser.reducer.Rule, oops.parser.trace.Rule`

Description

Represents grammar a rule.

Member Summary	
Fields	
`protected transient boolean`	`followChanged`
`protected transient boolean`	`inProgress`
`protected transient boolean`	`marked`
`protected java.lang.String`	`nt`
`protected oops.parser.Node`	`rhs`
Constructors	
	`Rule(Id, Node)`
Methods	
`protected boolean`	`checkDeadLoop()`
`void`	`checkLL1(Parser)`
`protected void`	`collectLitsAndIds(Hashtable, Hashtable)`
`int`	`degree()`
`protected boolean`	`followChanged()`
`String`	`getNt()`
`void`	`parse(Parser, Activation, Object)`
`void`	`setFollow(Parser)`
`Set`	`setFollow(Parser, Set)`
`void`	`setFollow(Set)`
`Set`	`setLookahead(Parser)`
`Object`	`sub(int)`
`String`	`toString()`
`String`	`toString(Parser)`

Seq

Declaration

```
public class Seq extends oops.parser.Node
```

All Implemented Interfaces: jag.Node, java.io.Serializable

Description

Represents a sequence of nodes.

Member Summary	
Fields	
protected java.util.Vector	nodes
Constructors	
	Seq(Node)
	Seq(Node[])
Methods	
void	add(Node)
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
int	degree()
Node	node()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)
Set	setLookahead(Parser)
Object	sub(int)
String	toString()

Set

Declaration

```
public class Set implements java.io.Serializable
```

All Implemented Interfaces: java.io.Serializable

Description

A `Set` manages symboles as one-elements sets, manages lookahead and manages follow sets.

Member Summary	
Fields	
protected boolean	empty
protected oops.parser.Set.Bit-Set	set
protected int	token
Constructors	
protected	Set()
	Set(int)

Member Summary	
	Set(Set)
Methods	
boolean	accepts(Set)
boolean	add(Set)
void	addEmpty()
protected void	clear()
boolean	equals(Object)
static Set	getEOFSet()
int	hashCode()
boolean	isEmpty()
boolean	matches(Set)
boolean	matchesEmpty()
boolean	remove(Set)
void	removeEmpty()
String	toString()
String	toString(Parser)

Some

Declaration

```
public class Some extends oops.parser.Many
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Represents one or more occurrences.

Member Summary	
Constructors	
	Some(Node)
Methods	
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
Node	node()
void	parse(Parser, Activation, Object)
Set	setLookahead(Parser)
String	toString()

Token

Declaration

```
public class Token extends oops.parser.Node
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Direct Known Subclasses: `oops.parser.goal.Token, oops.parser.reducer.Token, oops.parser.trace.Token`

Description

Represents class of terminal symbols.

Member Summary	
Fields	
protected java.lang.String	name
Constructors	
	Token(String, Set)
Methods	
protected boolean	checkDeadLoop()
void	checkLL1(Parser)
protected void	collectLitsAndIds(Hashtable, Hashtable)
String	getName()
void	parse(Parser, Activation, Object)
Set	setFollow(Parser, Set)
Set	setLookahead(Parser)
protected void	shift(Object, Parser)
String	toString()

XOr

Declaration

```
public class XOr extends oops.parser.Alt
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Represents xor elements.

Member Summary	
Constructors	
	XOr(Node)

Package oops.parser.trace

Description

The package `oops.parser.trace` collects subclasses of classes from the package `oops.parser` which prints a trace while parsing.

Class Summary	
Classes	
`Lit`	Extends `oops.parser.Lit` to print a trace while parsing.
`Parser`	Extends `oops.parser.Parser` to print a trace while parsing.
`Rule`	Extends `oops.parser.Rule` to print a trace while parsing.
`Token`	Extends `oops.parser.Token` to print a trace while parsing.

Lit

Declaration

```
public class Lit extends oops.parser.Lit
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Lit` to print a trace while parsing.

Member Summary	
Constructors	
	`Lit(String)`
Methods	
`protected void`	`shift(Object, Parser)`

Parser

Declaration

```
public class Parser extends oops.parser.Parser
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Parser` to print a trace while parsing.

Member Summary	
Constructors	
	`Parser(Rule)`
Methods	

Member Summary	
protected Token	createToken(String, Set)
protected void	error(Object)
protected void	parse(Activation)

Rule

Declaration
`public class **Rule** extends oops.parser.Rule`

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Rule` to print a trace while parsing.

Member Summary	
Fields	
protected static final int	INCR
Constructors	
	Rule(Id, Node)
Methods	
protected void	endRule(Integer)
void	parse(Parser, Activation, Object)
protected static void	printIndent(Integer)
protected Integer	startRule(Integer)

Token

Declaration
`public class **Token** extends oops.parser.Token`

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Token` to print a trace while parsing.

Member Summary	
Constructors	
	Token(String, Set)
Methods	
protected void	shift(Object, Parser)

Package oops.parser.goal

Description

The package `oops.parser.goal` collects subclasses of classes from the package `oops.parser` which adds one way to run user written action code for recognized rules and symbols. For another way to add action code see the `oops.parser.reducer` package.

While parsing a `GoalMakerFactory` creates for every rule a `GoalMaker`. For every activation of the corresponding rule the `GoalMaker` delivers a `Goal` object which methods do to action code.

The package comes with default implementation for `GoalMakerFactory` and for `Goal`.

Class Summary	
Interfaces	
`Goal`	Describes what each nonterminal must be able to do during parsing.
`GoalMaker`	While parsing for every activation of a rule a `GoalMaker` is asked to retrieve a `Goal` to do the action code.
`GoalMakerFactory`	While parsing a `GoalMakerFactory` is called once for every rule to create a `GoalMaker`.
Classes	
`DebuggerGoalMakerFactory`	A default `GoalMakerFactory` implementation.
`DefaultGoalMakerFactory`	A default `GoalMakerFactory` implementation.
`GoalAdapter`	A trivial `Goal` implementation.
`GoalDebugger`	A trivial `Goal` implementation.
`HashtableGoalMakerFactory`	A default `GoalMakerFactory` implementation.
`Lit`	A subclass of `oops.parser.Lit` which adds the ability to run `Goal` action while parsing.
`NopGoal`	A no operation (Nop) `Goal` implementation which implements all `shift()` and the `error()` methods with an empty body.
`NopGoalMakerFactory`	A `GoalMakerFactory` which creates `GoalMaker` instances which creates `NopGoal` objects.
`Parser`	A subclass of `oops.parser.Parser` which adds the ability to run `Goal` action while parsing.
`Parser.Result`	A `Result` object is the result of a parser run.
`Rule`	A subclass of `oops.parser.Rule` which adds the ability to run `Goal` action while parsing.
`Token`	A subclass of `oops.parser.Token` which adds the ability to run `Goal` action while parsing.

DebuggerGoalMakerFactory

Declaration

```
public class DebuggerGoalMakerFactory implements oops.parser.goal.GoalMakerFactory
```

All Implemented Interfaces: `GoalMakerFactory`

Description

A default `GoalMakerFactory` implementation. The produced `GoalMaker` objects for rules try to use a `Goal` class which name is equal to the rule name (optional adding a package name). If such a named class is found, a object from the class is created using the paramterless constructor, else a `GoalDebugger` is used.

Three system properties control the behaviour of the factories:

`oops.parser.goal.DebuggerGoalMakerFactory.goalPackage`

Sets optional a package name for the searched `Goal` classes.

`oops.parser.goal.DebuggerGoalMakerFactory.verbose`

If the property text is `true`, then a message is printed to `System.err` when no suitable `Goal` class can be found and a `GoalDebugger` will be used. The default is `true`.

`oops.parser.goal.DebuggerGoalMakerFactory.verbose2`

If the property text is `true`, then messages are printed to `System.err` while searching the `Goal` classes, when no suitable `Goal` class can be found and when a `GoalDebugger` will be used. The default is `false`.

Member Summary	
Fields	
protected	goalPackage
java.lang.String	
protected boolean	verbose
protected boolean	verbose2
Constructors	
	DebuggerGoalMakerFactory()
Methods	
GoalMaker	goalMaker(String)

DefaultGoalMakerFactory

Declaration

```
public class DefaultGoalMakerFactory implements oops.parser.goal.GoalMakerFactory
```

All Implemented Interfaces: `GoalMakerFactory`

Description

A default `GoalMakerFactory` implementation. The produced `GoalMaker` objects for rules try to use a `Goal` class which name is equal to the rule name (optional adding a package name). If such a named class is found, a object from the class is created using the paramterless constructor, else a `GoalAdapter` is used.

Three system properties control the behaviour of the factories:

`oops.parser.goal.DefaultGoalMakerFactory.goalPackage`

Sets optional a package name for the searched `Goal` classes.

`oops.parser.goal.DefaultGoalMakerFactory.verbose`

If the property text is `true`, then a message is printed to `System.err` when no suitable `Goal` class can be found and a `GoalAdapter` will be used. The default is `true`.

`oops.parser.goal.DefaultGoalMakerFactory.verbose2`

If the property text is `true`, then messages are printed to `System.err` while searching the `Goal` classes, when no suitable `Goal` class can be found and when a `GoalAdapter` will be used. The default is `false`.

Member Summary	
Fields	
protected	goalPackage
java.lang.String	
boolean	verbose
boolean	verbose2
Constructors	
	DefaultGoalMakerFactory()
Methods	
String	getGoalPackage()
GoalMaker	goalMaker(String)
void	setGoalPackage(String)

Goal

Declaration

```
public interface Goal
```

All Known Implementing Classes: `NopGoal, GoalAdapter`

Description

Describes what each nonterminal must be able to do during parsing.

Member Summary	
Methods	
void	error()
Object	reduce()
void	shift(Goal, Object)
void	shift(Lit, Object)
void	shift(Token, Object)

GoalAdapter

Declaration

```
public class GoalAdapter implements oops.parser.goal.Goal
```

All Implemented Interfaces: `Goal`

Direct Known Subclasses: `GoalDebugger`

Description

A trivial `Goal` implementation. The first non `null` value object is stored and `reduce()` returns the stored value object or `null`.

Member Summary	
Fields	
protected boolean	error
protected	result
java.lang.Object	
Constructors	
	GoalAdapter()
Methods	
void	error()
Object	reduce()
void	shift(Goal, Object)
void	shift(Lit, Object)
void	shift(Token, Object)

GoalDebugger

Declaration

```
public class GoalDebugger extends oops.parser.goal.GoalAdapter
```

All Implemented Interfaces: `Goal`

Description

A trivial `Goal` implementation. The first non `null` value object is stored and `reduce()` returns the stored value object or `null`. Every method prints a trace to `System.err`.

Member Summary	
Fields	
protected final int	count
protected final	name
java.lang.String	
Constructors	
	GoalDebugger()
	GoalDebugger(String)
Methods	
void	error()
Object	reduce()
void	shift(Goal, Object)
void	shift(Lit, Object)
void	shift(Token, Object)
String	toString()

GoalMaker

Declaration

```
public interface GoalMaker
```

Description

While parsing for every activation of a rule a `GoalMaker` is asked to retrieve a `Goal` to do the action code.

Member Summary	
Methods	
Goal	goal()

GoalMakerFactory

Declaration
public interface **GoalMakerFactory**

All Known Implementing Classes: DebuggerGoalMakerFactory, oops.oops.EBNFCompilerFactory.GoalMakerFactory, HashtableGoalMakerFactory, NopGoalMakerFactory, DefaultGoalMakerFactory

Description

While parsing a GoalMakerFactory is called once for every rule to create a GoalMaker.

Member Summary	
Methods	
GoalMaker	goalMaker(String)

HashtableGoalMakerFactory

Declaration
public class **HashtableGoalMakerFactory** implements oops.parser.goal.GoalMakerFactory

All Implemented Interfaces: GoalMakerFactory

Description

A default GoalMakerFactory implementation. A HashtableGoalMakerFactory maps in a table rule names to Goal class names. The produced GoalMaker objects for rules try to find a Goal class name for a rule name in the table. If such a Goal class is found, a object from the class is created using the paramterless constructor, else a GoalAdapter or a GoalDebugger is used.

Two system properties control the behaviour of the factories:

oops.parser.goal.HashtableGoalMakerFactory.debug

If the value is true objects of GoalDebugger are used as default Goal objects, else GoalAdapter instances are used.

oops.parser.goal.HashtableGoalMakerFactory.verbose

Sets a verbose action. The factories dump some trace while working.

Member Summary	
Fields	
boolean	debug
protected java.util.HashMap	map
boolean	verbose
Constructors	
	HashtableGoalMakerFactory()

Member Summary	
Methods	
Object	add(String, String)
GoalMaker	goalMaker(String)

Lit

Declaration
`public class` **`Lit`** `extends oops.parser.Lit`

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

A subclass of `oops.parser.Lit` which adds the ability to run `Goal` action while parsing.

Member Summary	
Constructors	
	Lit(String)
Methods	
protected void	shift(Object, Parser)

NopGoal

Declaration
`public class` **`NopGoal`** `implements oops.parser.goal.Goal`

All Implemented Interfaces: `Goal`

Description

A no operation (Nop) `Goal` implementation which implements all `shift()` and the `error()` methods with an empty body. `error()` just returns `null`.

Member Summary	
Constructors	
	NopGoal()
Methods	
void	error()
Object	reduce()
void	shift(Goal, Object)
void	shift(Lit, Object)
void	shift(Token, Object)

NopGoalMakerFactory

Declaration
`public class` **`NopGoalMakerFactory`** `implements oops.parser.goal.GoalMakerFactory`

All Implemented Interfaces: `GoalMakerFactory`

Description

A `GoalMakerFactory` which creates `GoalMaker` instances which creates `NopGoal` objects.

Member Summary	
Constructors	
	`NopGoalMakerFactory()`
Methods	
`GoalMaker`	`goalMaker(String)`

Parser

Declaration

```
public class Parser extends oops.parser.Parser
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

A subclass of `oops.parser.Parser` which adds the ability to run `Goal` action while parsing.

Member Summary	
Nested Classes	
`static class`	`Parser.Result`
Fields	
`protected transient java.lang.Object`	`result`
Constructors	
	`Parser(Rule)`
Methods	
`protected Token`	`createToken(String, Set)`
`protected void`	`error(Object)`
`protected void`	`parse(Activation)`
`Parser.Result`	`parse(ValueScanner, GoalMakerFactory, TableFactory)`
`protected Object`	`value()`

Parser.Result

Declaration

```
public static class Parser.Result
```

Enclosing Class: `oops.parser.goal.Parser`

Description

A `Result` object is the result of a parser run.

Member Summary	
Fields	
boolean	error
java.lang.Object	result
Methods	
String	toString()

Rule

Declaration

```
public class Rule extends oops.parser.Rule
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

A subclass of `oops.parser.Rule` which adds the ability to run `Goal` action while parsing.

Member Summary	
Fields	
protected transient oops.parser.goal.GoalMaker	gm
Constructors	
	Rule(Id, Node)
Methods	
void	parse(Parser, Activation, Object)
void	setGoalMaker(GoalMakerFactory)

Token

Declaration

```
public class Token extends oops.parser.Token
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

A subclass of `oops.parser.Token` which adds the ability to run `Goal` action while parsing.

Member Summary	
Constructors	
	Token(String, Set)
Methods	
protected void	shift(Object, Parser)

Package oops.parser.reducer

Description

The package `oops.parser.reducer` collects subclasses of classes from the package `oops.parser` which adds one way to run user written action code for recognized rules. For the second way to add action code see the `oops.parser.goal` package.

Class Summary	
Interfaces	
`Reducer`	A `Reducer` is called at the end of a rule to reduce the rule and to produce a result for the rule.
`ReducerLit`	A `ReducerLit` is called at the end of a rule to reduce the rule and to produce a result for the rule.
`ReducerMaker`	While parsing for every activation of a rule a `ReducerMaker` is asked to retrieve a `Reducer` or a `ReducerLit` to do the action code.
`ReducerMakerFactory`	While parsing a `ReducerMakerFactory` is called once for every rule to create a `ReducerMaker`.
Classes	
`DebuggerLitReducerMakerFactory`	A default `ReducerMakerFactory` implementation.
`DebuggerReducerMakerFactory`	A default `ReducerMakerFactory` implementation.
`DefaultLitReducerMakerFactory`	A default `ReducerMakerFactory` implementation.
`DefaultReducerMakerFactory`	A default `ReducerMakerFactory` implementation.
`DefaultReducerMakerFactory.ReducerMaker`	A default `ReducerMaker` implementation.
`Lit`	Extends `oops.parser.Lit` to support `Reducer` and `ReducerLit` action.
`Parser`	Extends `oops.parser.Parser` to support `Reducer` and `ReducerLit` action.
`Parser.Result`	A `Result` object is the result of a parser run.
`ReducerAdapter`	A trivial `Reducer` implementation.
`ReducerDebugger`	A trivial `Reducer` implementation.
`ReducerLitAdapter`	Extends `ReducerAdapter` by adapting the `ReducerLit` interface to collect value objects for literal symbols.
`ReducerLitDebugger`	Extends `ReducerDebugger` by adapting the `ReducerLit` interface to collect value objects for literal symbols.
`Rule`	Extends `oops.parser.Rule` to support `Reducer` and `ReducerLit` action.
`Token`	Extends `oops.parser.Token` to support `Reducer` and `ReducerLit` action.

DebuggerLitReducerMakerFactory

Declaration

```
public class DebuggerLitReducerMakerFactory extends
    oops.parser.reducer.DefaultReducerMakerFactory
```

All Implemented Interfaces: `ReducerMakerFactory`

Description

A default `ReducerMakerFactory` implementation. See the method description for more information.

Member Summary		
Constructors		
		`DebuggerLitReducerMakerFactory()`
Methods		
	`ReducerMaker`	`reducerMaker(String)`

DebuggerReducerMakerFactory

Declaration

```
public class DebuggerReducerMakerFactory extends oops.parser.reducer.DefaultReducerMakerFactory
```

All Implemented Interfaces: `ReducerMakerFactory`

Description

A default `ReducerMakerFactory` implementation. See the method description for more information.

Member Summary		
Constructors		
		`DebuggerReducerMakerFactory()`
Methods		
	`ReducerMaker`	`reducerMaker(String)`

DefaultLitReducerMakerFactory

Declaration

```
public class DefaultLitReducerMakerFactory extends oops.parser.reducer.DefaultReducerMakerFactory
```

All Implemented Interfaces: `ReducerMakerFactory`

Description

A default `ReducerMakerFactory` implementation. See the method description for more information.

Member Summary	
Constructors	
	`DefaultLitReducerMakerFactory()`
Methods	
`ReducerMaker`	`reducerMaker(String)`

DefaultReducerMakerFactory

Declaration

`public class` **`DefaultReducerMakerFactory`** `implements oops.parser.reducer.ReducerMakerFactory`

All Implemented Interfaces: `ReducerMakerFactory`

Direct Known Subclasses: `DebuggerLitReducerMakerFactory, DebuggerReducerMakerFactory, DefaultLitReducerMakerFactory`

Description

A default `ReducerMakerFactory` implementation. See the method description for more information.

Member Summary	
Nested Classes	
`protected static class`	`DefaultReducerMakerFactory.ReducerMaker`
Constructors	
	`DefaultReducerMakerFactory()`
Methods	
`ReducerMaker`	`reducerMaker(String)`

DefaultReducerMakerFactory.ReducerMaker

Declaration

`protected static class` **`DefaultReducerMakerFactory.ReducerMaker`** `implements oops.parser.reducer.ReducerMaker`

All Implemented Interfaces: `ReducerMaker`

Enclosing Class: `oops.parser.reducer.DefaultReducerMakerFactory`

Description

A default `ReducerMaker` implementation. An instance tries to use a `Reducer` or `ReducerLit` class which name is equal to the rule name (optional adding a package name). If such a named class is found, a object from the class is created using the paramterless constructor, else a default class is used.

Three system properties control the behaviour of the factories:

`oops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.goalPackage`

Sets optional a package name for the searched `Reducer` or `ReducerLit` classes.

`oops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.verbose`

If the property text is `true`, then a message is printed to `System.err` when no suitable `Reducer` or `ReducerLit` class can be found and a default class is used. The default is `true`.

oops.parser.reducer.DefaultReducerMakerFactory.ReducerMaker.verbose2

If the property text is `true`, then messages are printed to `System.err` while searching the `Reducer` or `ReducerLit` classes, when no suitable named class can be found and when a default class is used. The default is `false`.

Member Summary	
Fields	
protected final java.lang.Class	defaultClass
protected java.lang.String	goalPackage
protected final java.lang.String	name
protected java.lang.Class	reducerClass
boolean	verbose
boolean	verbose2
Constructors	
protected	DefaultReducerMakerFactory.ReducerMaker(String, Class)
Methods	
protected void	readProperties()
Reducer	reducer()

Lit

Declaration

```
public class Lit extends oops.parser.Lit
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Lit` to support `Reducer` and `ReducerLit` action.

Member Summary	
Constructors	
	Lit(String)
Methods	
protected void	shift(Object, Parser)

Parser

Declaration

```
public class Parser extends oops.parser.Parser
```

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Parser` to support `Reducer` and `ReducerLit` action.

Member Summary	
Nested Classes	
static class	Parser.Result
Fields	
protected transient java.lang.Object	result
Constructors	
	Parser(Rule)
Methods	
protected Token	createToken(String, Set)
protected void	error(Object)
protected void	parse(Activation)
Parser.Result	parse(ValueScanner, ReducerMakerFactory, TableFactory)
protected Object	value()

Parser.Result

Declaration

```
public static class Parser.Result
```

Enclosing Class: `oops.parser.reducer.Parser`

Description

A `Result` object is the result of a parser run.

Member Summary	
Fields	
boolean	error
java.lang.Object	result
Methods	
String	toString()

Reducer

Declaration

```
public interface Reducer
```

All Known Subinterfaces: `ReducerLit`

All Known Implementing Classes: `ReducerAdapter, ReducerDebugger`

Description

A `Reducer` is called at the end of a rule to reduce the rule and to produce a result for the rule.

Member Summary	
Methods	
void	error()
Object	reduce(Object[])

ReducerAdapter

Declaration

```
public class ReducerAdapter implements oops.parser.reducer.Reducer
```

All Implemented Interfaces: `Reducer`

Direct Known Subclasses: `ReducerLitAdapter`

Description

A trivial `Reducer` implementation. `reduce()` returns an array with all value objects or `null`.

Member Summary	
Fields	
protected boolean	error
Constructors	
	ReducerAdapter()
Methods	
void	error()
Object	reduce(Object[])

ReducerDebugger

Declaration

```
public class ReducerDebugger implements oops.parser.reducer.Reducer
```

All Implemented Interfaces: `Reducer`

Direct Known Subclasses: `ReducerLitDebugger`

Description

A trivial `Reducer` implementation. `reduce()` returns an array with all value objects or `null`. Every method prints a trace to `System.err`.

Member Summary	
Fields	
protected int	count
protected static int	counts
protected boolean	error
Constructors	
	ReducerDebugger()
Methods	
void	error()
static void	printObjectArray(Object[])
Object	reduce(Object[])
String	toString()

ReducerLit

Declaration

```
public interface ReducerLit extends oops.parser.reducer.Reducer
```

All Superinterfaces: `Reducer`

All Known Implementing Classes: `ReducerLitDebugger, ReducerLitAdapter`

Description

A `ReducerLit` is called at the end of a rule to reduce the rule and to produce a result for the rule. In opposite to `Reducer` also the value objects of literal symbols are part of the argument array to `reduce()`.

ReducerLitAdapter

Declaration

```
public class ReducerLitAdapter extends oops.parser.reducer.ReducerAdapter implements
    oops.parser.reducer.ReducerLit
```

All Implemented Interfaces: `Reducer, ReducerLit`

Description

Extends `ReducerAdapter` by adapting the `ReducerLit` interface to collect value objects for literal symbols.

Member Summary
Constructors
`ReducerLitAdapter()`

ReducerLitDebugger

Declaration

```
public class ReducerLitDebugger extends oops.parser.reducer.ReducerDebugger implements
    oops.parser.reducer.ReducerLit
```

All Implemented Interfaces: `Reducer, ReducerLit`

Description

Extends `ReducerDebugger` by adapting the `ReducerLit` interface to collect value objects for literal symbols.

Member Summary
Constructors
`ReducerLitDebugger()`

ReducerMaker

Declaration
`public interface` **`ReducerMaker`**

All Known Implementing Classes: `DefaultReducerMakerFactory.ReducerMaker`

Description

While parsing for every activation of a rule a `ReducerMaker` is asked to retrieve a `Reducer` or a `ReducerLit` to do the action code.

Member Summary
Methods
`Reducer` `reducer()`

ReducerMakerFactory

Declaration
`public interface` **`ReducerMakerFactory`**

All Known Implementing Classes: `DefaultReducerMakerFactory`

Description

While parsing a `ReducerMakerFactory` is called once for every rule to create a `ReducerMaker`.

Member Summary
Methods
`ReducerMaker` `reducerMaker(String)`

Rule

Declaration
`public class` **`Rule`** `extends oops.parser.Rule`

All Implemented Interfaces: `jag.Node, java.io.Serializable`

Description

Extends `oops.parser.Rule` to support `Reducer` and `ReducerLit` action.

Token

Member Summary		
Fields		
protected transient oops.parser.reducer.ReducerMaker		rm
Constructors		
		Rule(Id, Node)
Methods		
	void	parse(Parser, Activation, Object)
	void	setReducerMaker(ReducerMakerFactory)

Declaration

```
public class Token extends oops.parser.Token
```

All Implemented Interfaces: jag.Node, java.io.Serializable

Description

Extends `oops.parser.Token` to support `Reducer` and `ReducerLit` action.

Member Summary		
Constructors		
		Token(String, Set)
Methods		
protected	void	shift(Object, Parser)

Package oops.oops

Description

The package `oops.oops` provides the EBNF and XEBNF parser generators.

Class Summary	
Classes	
`EBNFBootCompilerFactory`	A `CompilerFactory` used to build a new version of *oops* by *oops*.
`EBNFCompilerFactory`	Runs `oops` as parser generator for EBNF grammar.
`EBNFCompilerFactory.GoalMakerFactory`	A local `GoalMakerFactory` to build the right `Goal` instances to create the parser from the wanted classes.
`Input`	A Scanner to run the oops parser generators.
`OopsBootCompilerFactory`	A `CompilerFactory` used to build a new version of *oops* by *oops*.
`OopsCompilerFactory`	Runs `oops` as parser generator for XEBNF grammar.

EBNFBootCompilerFactory

Declaration

```
public class EBNFBootCompilerFactory extends oops.oops.EBNFCompilerFactory
```

Direct Known Subclasses: `OopsBootCompilerFactory`

Description

A `CompilerFactory` used to build a new version of *oops* by *oops*. Normal users of *oops* will not use this class.

Member Summary	
Constructors	
	`EBNFBootCompilerFactory()`
Methods	
`Compiler`	`compiler()`
`protected GoalMakerFactory`	`getGoalMakerFactory()`
`protected String`	`getMessage()`

EBNFCompilerFactory

Declaration

```
public class EBNFCompilerFactory extends oops.opi.CompilerFactory
```

Direct Known Subclasses: `EBNFBootCompilerFactory, OopsCompilerFactory`

Description

Runs `oops` as parser generator for EBNF grammar. The produced `oops.opi.Compiler` uses the system property `oops.oops.actionType` to build different parsers:

```
oops.oops.actionType=none
```

The parser is build from instances of classes from the `oops.parser` package. So the parser just parses and does no action for parsed input.

```
oops.oops.actionType=trace
```

The parser is build from instances of classes from the `oops.parser.trace` and the `oops.parser` packages. So the parses prints a trace while parsing the input.

```
oops.oops.actionType=goal
```

The parser is build from instances of classes from the `oops.parser.goal` and the `oops.parser` packages. So the parses use `oops.parser.goal.Goal` objects for action code. If the property isn't set, this is the default.

```
oops.oops.actionType=reducer
```

The parser is build from instances of classes from the `oops.parser.reducer` and the `oops.parser` packages. So the parses use `oops.parser.reducer.Reducer` objects for action code.

A simple way to use this class is the class `oops.Compile` or better `oops.EBNF`.

Member Summary	
Nested Classes	
`protected static class`	`EBNFCompilerFactory.GoalMakerFactory`
Constructors	
	`EBNFCompilerFactory()`
Methods	
`Compiler`	`compiler()`
`protected String`	`getContentFromSystemResource(String)`
`protected GoalMaker-Factory`	`getGoalMakerFactory()`
`protected String`	`getMessage()`
`protected Parser`	`getParser()`
`protected Parser`	`getParser(String)`

EBNFCompilerFactory.GoalMakerFactory

Declaration

```
protected static class EBNFCompilerFactory.GoalMakerFactory implements
   oops.parser.goal.GoalMakerFactory
```

All Implemented Interfaces: `oops.parser.goal.GoalMakerFactory`

Enclosing Class: `oops.oops.EBNFCompilerFactory`

Description

A local `GoalMakerFactory` to build the right `Goal` instances to create the parser from the wanted classes. Only used in `EBNFCompilerFactory` and subclasses.

Member Summary	
Fields	
protected final java.lang.String[]	packages
Constructors	
protected	EBNFCompilerFactory.GoalMakerFactory()
Methods	
protected void	getProperty()
GoalMaker	goalMaker(String)

Input

Declaration

```
public class Input implements oops.scanner.ValueScanner
```

All Implemented Interfaces: `oops.scanner.Scanner, oops.scanner.ValueScanner`

Description

A Scanner to run the oops parser generators.

Member Summary	
Fields	
protected java.lang.Object	ID
protected java.lang.Object	LIT
protected final java.io.StreamTokenizer	st
protected java.lang.Object	symbol
protected oops.scanner.SymTab	symtab
protected java.lang.Object	value
Constructors	
	Input(InputSource)
Methods	
boolean	advance()
boolean	atEnd()
void	scan(SymTab, Hashtable)
Object	symbol()
String	toString()
Object	value()

OopsBootCompilerFactory

Declaration

```
public class OopsBootCompilerFactory extends oops.oops.EBNFBootCompilerFactory
```

Description

A `CompilerFactory` used to build a new version of *oops* by *oops*. Normal users of *oops* will not use this class.

Member Summary	
Constructors	
	`OopsBootCompilerFactory()`
Methods	
`protected String`	`getMessage()`
`protected Parser`	`getParser()`

OopsCompilerFactory

Declaration

```
public class OopsCompilerFactory extends oops.oops.EBNFCompilerFactory
```

Description

Runs `oops` as parser generator for XEBNF grammar. The produced `oops.opi.Compiler` uses the system property `oops.oops.actionType` to build different parsers:

```
oops.oops.actionType=none
```

The parser is build from instances of classes from the `oops.parser` package. So the parser just parses and does no action for parsed input.

```
oops.oops.actionType=trace
```

The parser is build from instances of classes from the `oops.parser.trace` and the `oops.parser` packages. So the parses prints a trace while parsing the input.

```
oops.oops.actionType=goal
```

The parser is build from instances of classes from the `oops.parser.goal` and the `oops.parser` packages. So the parses use `oops.parser.goal.Goal` objects for action code. If the property isn't set, this is the default.

```
oops.oops.actionType=reducer
```

The parser is build from instances of classes from the `oops.parser.reducer` and the `oops.parser` packages. So the parses use `oops.parser.reducer.Reducer` objects for action code.

A simple way to use this class is the class `oops.Compile` or better `oops.Oops`.

Member Summary	
Constructors	
	`OopsCompilerFactory()`
Methods	
`protected String`	`getMessage()`
`protected Parser`	`getParser()`