S T U D I E N A R B E I T

Berufsakademie Stuttgart - Außenstelle Horb -
Staatliche Studienakademie

Fachrichtung Informationstechnik

Design einer objektorientierten Tokenübergabe für den Parsergenerator "jay"

Juni 2004

Eingereicht von:

Holger Haas
Hermann-Hesse-Str. 19

72160 Nordstetten

Firma:

InterCom Consulting
Brödelestr. 2

72160 Horb a. Neckar

Betreuer:
Prof. Dr. Martin Plümicke

Zusammenfassung

Vor allem im Compilerbau ist es fast unumgänglich, Werkzeuge zur automatischen Generierung von Scannern und Parsern einzusetzen. Ein solches Tool ist z.B. der Parsergenerator jay. Jay ist eine für Java-Projekte weiterentwickelte Variante des C-Parsergenerator “yacc“. Da yacc für C-Projekte entwickelt wurde, sind leider keine objektorientierten Konzepte möglich. Auch bei jay übergibt der Scanner wie bei yacc Integer-Werte und keine Objekte an den Parser.

Im Rahmen dieser Studienarbeit wurde jay so modifiziert, dass die Tokenübergabe über Objekte erfolgen kann. Dabei wurden Änderungen sowohl am Quellcode als auch an der Interpreter-Vorlage “skeleton“ vorgenommen.

Ein weiteres Problem stellte die Schnittstelle zwischen Scanner und Parser dar. Diese wird durch ein inneres Interface in der von jay generierten Parser-Klasse bereitgestellt. Dieses Interface muss vom Benutzer von Hand implementiert werden. Die Implementierung des Interfaces führte bei Studentenprojekten oft zur Verwirrung.

Aus diesem Grund wurden ebenfalls im Rahmen dieser Studienarbeit die Quellen des Scannergenerators “JLex“ geändert. Der von JLex erzeugte Scanner implementiert nun automatisch das von jay geforderte Interface. Der Vorgang ist für den Benutzer transparent.

Erklärung:

Ich habe die Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Ort	Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

$\perp$	Zeichen für “undefiniert“ (Sprich: “bottom“)
AI	“Artificial Intelligence“, Künstliche Intelligenz
BNF	“Bachus-Naur-Form“, Schreibweise für Grammatiken
EBNF	“Extended Bachus-Naur-Form“, Erweiterung der BNF (z.B. um optionale Elemente)
EOF	“End of File“, Dateiende
EOL	“End of Line“, Zeilenende (meist durch “Newline“ markiert)
TOS	“Top of Stack“, oberstes Element eines Stacks.
yacc	“Yet Another Compiler Compiler“, Parsergenerator der Berkeley University, Original von ATT

1 Ausgangssituation und Motivation

Das Tool “jay“, mit dessen Hilfe sich aus einer gegebenen Grammatik und einer Steuerdatei ein Parser generieren lässt, ist eine Weiterentwicklung des C-Parsergenerator “yacc“ und in C-Code verfasst. Jay generiert allerdings einen Parser in Javacode. Da yacc aus der C-Welt kommt, sind dort keine objektorientierten Konzepte möglich. Diese Schwäche ist leider auch an jay weitervererbt worden. So wurde die Übergabe der Token vom Scanner an den Parser bisher wie beim Vorgänger über Integer-Werte gelöst.

Die bisherige Übergabe der Token erfolgt über ein Inner Interface in der erzugten Parser-Klasse. Dieses Interface enthält die Methoden

“advance()“ , welche über einen Rückgabewert anzeigt, ob ein weiteres Token eingelesen werden kann,

“token ()“ , diese Methode gibt einen Integer-Wert, der das nächste Token repräsentiert, zurück, und

“value ()“ , die Methode, über die man den Wert des Tokens erhält

Das Interface muss vom Anwender von Hand implementiert werden.

Ideal wäre ein Verzicht auf das Interface im zukünftigen Design. Es wird jedoch benötigt, um den Scanner flexibel implementieren zu können und um vom Scanner aus auf im Parser deklarierte Objekte zugreifen zu können, ohne auf ein “Henne und Ei“-Problem zu stoßen.

Die Aufgabe besteht also einerseits darin, das Interface für den Anwender unsichtbar zu implementieren. Dazu ist es nötig, einen Scannergenerator dementsprechend anzupassen. Weiterhin muss der Parsergenerator jay so modifiziert werden, dass er Token-Objekte an Stelle von Integer-Repräsentationen akzeptiert.

Im Rahmen dieser Studienarbeit soll ein Design für eine optimierte Tokenübergabe gefunden werden.

2 Grundlagen und Funktionsweise eines Parsers

2.1 Was ist ein Parser ?

Ein *Parser* (von engl. “to parse“ = grammatisch zergliedern) ist ein Programm zur syntaktischen Analyse eines Eingabetextes. Der Text wird dabei auf der Grundlage einer Grammatik analysiert und zerlegt, er wird *geparst*. Die Anwendungsgebiete eines Parser sind dabei vielfältig, so kann ein Parser z.B. eingesetzt werden, um die Eingaben eines Benutzers in Form geschriebener Worte zur Steuerung eines Spiels oder Anwendungsprogramms richtig zu interpretieren (Berühmtes Beispiel: Josephs Weizenbaums ELIZA, ein AI-Programm, das eine Unterhaltung mit einem menschlichen Gegenüber simuliert). Auch in der Spracherkennung lässt sich ein Parser einsetzen, um in einem Satz Subjekt, Prädikat, Objekt, usw. zu erkennen.

Die Grammatik einer natürlichen, gesprochenen Sprache ist allerdings nicht eindeutig und teilweise sogar wiedersprüchlich. Für Maschinen ist sie nur annäherungsweise nachvollziehbar. Die Hauptanwendungsgebiete von Parsern beschränken sich deshalb auf die Analyse von strukturierten Texten, die in einer formalen Sprache verfasst wurden. Formale Sprachen folgen einer Grammatik, die mathematisch exakt definiert wurde. Beispiele hierfür sind Beschreibungssprachen wie LaTeXoder HTML, die vor allem unter Linux üblichen Konfigurationsdateien im Textformat oder Programmiersprachen. Diese Arbeit befasst sich mit Parsern, die hauptsächlich im Compilerbau Anwendung finden.

Die einzelnen Aufgaben eines Compilers werden meist in einzelne Module ausgelagert. Die Vorteile dieses Verfahrens liegen einerseits in der besseren Übersichtlichkeit, was die Entwicklung und Fehlersuche vereinfacht. Ein weiterer Vorteil ist die Flexibilität, so lassen sich einzelne Teile austauschen, um den Compiler für andere Aufgaben anzupassen.

Der Compilevorgang lässt sich grob in zwei Phasen unterteilen: Die *Analysephase*, in der der Quelltext analysiert wird, und die *Synthesephase*, in welcher der Maschinencode generiert und ausgegeben wird. An der Analysephase wiederum arbeiten zwei Programmteile: der *Scanner* und der Parser. Der Scanner übernimmt dabei das zeichenweise Einlesen des Textes und die Bildung von “Wörtern“, diese werden auch *Token* genannt. Die erkannten Token oder eventuelle Fehlermeldungen werden an den Parser übergeben, welcher prüft, ob der Text bezüglich der Grammatik korrekt ist. Durch den Parsevorgang wird aus dem Text weiterhin ein *Syntaxbaum* aufgebaut. Dieser stellt eine interne Repräsentation des Textes dar, die wiederum von einem weiteren Programmteil des Compilers interpretiert

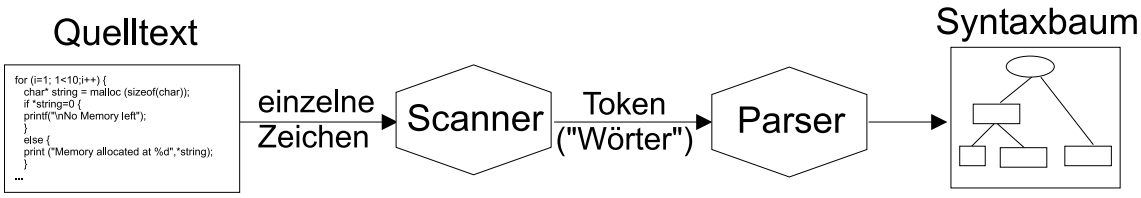

Abbildung 2.1: Verarbeitung eines Quelltextes

while, do, if, else, . . .
<, <=, >, >=, !, . . .
a, zahl, r_name, pointer5, . . .

Abbildung 2.2: Gültige Lexeme in C

werden kann.

2.2 Scanner und lexikalische Analyse

2.2.1 Aufgabe des Scanners

Der *Scanner* übernimmt die Aufgabe der *lexikalischen Analyse*, also das Bilden von "Wörtern" aus einem Quelltext. Der Quelltext wird dabei vom Scanner zeichenweise eingelesen und einzelne Zeichen nach vorgegebenen Regeln in syntaktisch atomare Einheiten, sogenannte *Token*, gruppiert. Die lexikalische Analyse erfolgt dabei linear: Der Scanner konsumiert einen Zeichenstrom und produziert einen Tokenstrom, das Bilden von geschachtelten Strukturen ist dadurch im Gegensatz zur nachfolgenden syntaktischen Analyse nicht möglich.

2.2.2 Lexeme und Reguläre Ausdrücke

Der vom Scanner einzulesende Quelltext stellt zunächst einmal nur einen Strom einzelner Zeichen dar. Die Aufgabe des Scanners besteht nun darin, aus diesem Zeichenstrom gültige Zeichenketten, die *Lexeme*, herauszulesen. Bei ungültigen Zeichenketten soll ein Fehler gemeldet, irrelevante Zeichenketten wie Leerzeichen und Kommentare sollen übergangen werden. Die Menge der gültigen Zeichenketten wird vorher definiert, so umfasst sie im Falle der Programmiersprache C Schlüsselwörter, Operatoren, Bezeichner etc, aber z.B. keine Unicodezeichen, die nicht im ASCII-Zeichensatz enthalten sind.

Um die Menge der gültigen Lexeme definieren zu können, könnte man versuchen, alle aufzuschreiben. Da z.B. die Menge aller Schlüsselwörter in C endlich ist, führt dieser

Zeichen	**Interpretation**
α (außer Metasymbol)	α
RegExp1RegExp2	Sequenz: matcht jede Zeichenkette, auf die jeder der beiden Ausdrücke nacheinander und in der angegebenen Reihenfolge passt.
RegExp1 | RegExp2	Alternative: matcht jede Zeichenkette, auf die einer der beiden Ausdrücke passt.
(RegExp)	Gruppierung, nachfolgende Operatoren beziehen sich auf die gesamte Gruppe
.	matcht jedes Zeichen außer *newline*
RegExp*	RegExp kommt 0 oder mehrmals vor
RegExp+	RegExp kommt mindestens einmal vor
RegExp?	RegExp kommt 0 oder einmal vor
[α-β]	Alternativenliste, matcht ein Zeichen zwischen α und β Beispiel: [a-d] matcht a,b,c und d.
[^α-β]	Negative Alternativenliste, matcht alle Zeichen, die nicht in der Liste vorkommen

Abbildung 2.3: Reguläre Ausdrücke für Jlex

Ansatz hier auch zum Erfolg. Anders sieht es dagegen bei der Menge aller Integer oder Bezeichner aus. Um auch solche unendlichen Mengen zu erfassen bedarf es eines entsprechenden Formalismus. Bewährt hat sich für die lexikalische Analyse der Formalismus der *Regulären Ausdrücke*. Ein Regulärer Ausdruck passt auf eine bestimmte Menge von Zeichenketten. Man spricht in diesem Fall auch von einem *Match*, von *Pattern Matching* oder neudeutsch vom *matchen*.

Ein Nachteil der Regulären Ausdrücke ist die Tatsache, dass ihre strukturelle Beschaffenheit zwar eindeutig ist, aber zahlreiche Variationen ihrer Notation existieren. Das folgende Beispiel soll einen Überblick über die Notation geben, die auch der Scannergenerator Jlex verwendet. Dabei bedeutet *RegExp* eine Regulären Ausdruck und α ein einzelnes Zeichen. Den Zeichen ? * + | ( ) ^ $. [ ] { } “ \ kommt dabei eine besondere Bedeutung zu. Bei ihnen handelt es sich um sogenannte *Metasymbole*. Abbildung 2.3 gibt einen Überblick über die wichtigsten Ausdrücke und Konstruktionen und ist bei weitem nicht vollständig.

Ein Problem ergibt sich allerdings, wenn man ein Metasymbol matchen will. Dieses wird nämlich nicht als normales Zeichen interpretiert, sondern wie im Falle von ‘*‘ z.B. als Operator. Die Lösung dieses Problems besteht im Voranstellen eines *Entwerters*, der den Metacharakter eines Zeichens aufhebt. Meist wird dazu der Backslash ‘\‘ benutzt. Der Backslash selbst wird durch ‘\\‘ dargestellt. Der Name Entwerter ist allerdings nicht sehr geläufig, in der Literatur ist meist von *Escapezeichen* bzw. *Escapesequenzen* die Rede.

Reguläre Ausdrücke stellen ein sehr mächtiges Werkzeug bei der Suche nach Mustern dar. Allerdings ist auch ihre Mächtigkeit begrenzt. Die Beschränkung liegt in der Tatsa-

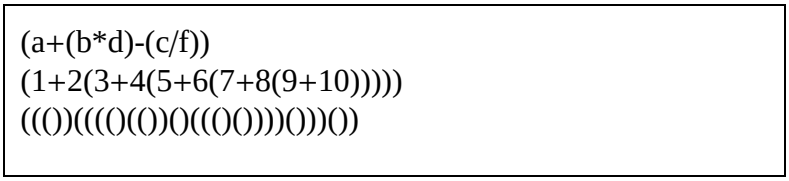

Abbildung 2.4: Balancierte Klammergebirge

che, dass die Operatoren zur *Iteration*, also + * ? keine Möglichkeit haben, die Anzahl der Matches von Teilausdrücken zu speichern. Ein oft verwendetes Beispiel ist das *balancierte Klammergebirge* (Abbildung 2.4): Es besteht aus einer beliebigen Anzahl öffnender und schließender Klammern, aber die Anzahl öffnender und schließender Klammern ist gleich groß. Das balancierte Klammergebirge lässt sich mit einem regulären Ausdruck nicht erfassen.

2.2.3 Mehrdeutigkeiten, "Principle of the longest Match"

Die Mengen von Zeichenketten, auf die zwei oder mehrere Reguläre Ausdrücke passen, können sich überschneiden. Zum Beispiel könnte man den Regulären Ausdruck für ein Schlüsselwort in C wie folgt angeben:

$$for \mid if \mid while \mid \ldots$$

und für einen Bezeichner (der in diesem Beispiel keine Sonderzeichen wie den Unterstrich enthält):

$$[a-zA-Z][a-zA-Z0-9]+$$

Allerdings matchen beide Ausdrücke auf die Zeichenkette 'while'. Es muss also bei der Deklaration der Regulären Ausdrücke darauf geachtet werden, dass spezifischere Ausdrücke zuerst abgearbeitet werden. Matcht ein Regulärer Ausdruck, werden keine weiteren mehr angewandt.

Soll nun mit oben aufgeführter Regel ein Bezeichner gefunden werden, ist zunächst nicht klar, wie viele Zeichen der Bezeichner umfasst. Heißt der Bezeichner z.B. 'Variable123' wie in Abbildung 2.5, so matcht der Reguläre Ausdruck sowohl auf 'V' alleine, als auch auf den gesamten Namen, sowie alle Zwischenschritte. Das Problem wird dadurch gelöst, dass immer die längste zum Ausdruck passende Zeichenkette ausgewählt wird. Dieses Verfahren wird "Principle of the longest Match" oder auch "Maximum Munch" genannt. (In der Perl-Welt einfach nur "Greed", Gier).

RegExp = [a-zA-Z][a-zA-Z0-9]+
⇒ V
⇒ Va
⇒ Var
⇒ Vari
⇒ ...
⇒ Variable12
⇒ **Variable123**

Abbildung 2.5: Principle of the longest Match

Schlüsselwörter:	while, do, if, else, ...
Operatoren:	<, <=, >, >=, !, ...
Bezeichner:	a, zahl, r_name, pointer5, ...
Ganze Zahlen:	1, 17, 5, -3, ...
Fließkommazahlen:	0.3, .789, -12.5, ...

Abbildung 2.6: Tokenklassen in C

2.2.4 Klassifizierung von Lexemen: Token

Durch die Definition der Lexeme kann der Scanner mit Hilfe entsprechender Regulärer Ausdrücke nun die Gültigkeit von Zeichenketten erkennen, die erkannten Zeichenketten aus dem Zeichenstrom separieren und an die nächste verarbeitende Instanz (den Parser) weitergeben. Für die nachfolgende semantische Analyse ist es jedoch notwendig, zusätzliche Informationen über die Bedeutung der Zeichenkette zu erlangen.

Aus diesem Grund werden deshalb Lexeme mit ähnlicher Bedeutung zu sogenannte *Token* zusammengefasst. Durch diese Maßnahme erhält man eine übersichtliche lexikalische Struktur. Während zum Beispiel die Menge aller möglichen Bezeichner einer Programmiersprache extrem groß ist, ist die Anzahl der Tokenklassen verhältnismäßig begrenzt. Typisch sind ca. 50 verschiedene Token.

Die Unterteilung der Tokenklassen ist nicht eindeutig festgelegt und hängt von der Implementierung der Weiterverarbeitung ab. So besteht z.B. die Möglichkeit, alle Operatoren zusammen zu fassen. Man könnte aber auch jedem Operator eine eigene Tokenklasse zuweisen. Abbildung 2.6 zeigt einen Ausschnitt der in der Sprache C möglichen Token.

Es reicht allerdings nicht aus, die reine Klasseninformation der Token an den Parser weiter zu reichen, da hierbei die Information über den *Wert* des Tokens verloren gehen würde. Deshalb werden die Token mit einem Attribut versehen, welches den Wert des Tokens enthält. Optional lassen sich auch noch weitere Attribute hinzufügen, die z.B. für Debuggingausgaben von Nutzen sind, wie die Angabe der Zeilennummer. Der Typ des Wert-Attributes ist dabei abhängig von der Tokenklasse, wie Abbildung 2.7 illustriert. Für die

Tokenklasse	Attribut1	Wert1	Attribut2	Wert2
Schlüsselwort	String Wort	“while“	int Zeile	12
Operator	String Typ	“PLUS“	int Zeile	5
Bezeichner	String Name	“Variable123“	int Zeile	6
Ganze Zahl	int Wert	-17	int Zeile	6
Fließkommazahl	Float Wert	1.456	int Zeile	8

Abbildung 2.7: Tokenattribute

Implementierung von Token ist bedingt durch diese Struktur ein objektorientierter Ansatz am besten geeignet.

2.2.5 Diskrete und Nicht-Diskrete endliche Automaten

Im vorangegangenen Abschnitt wurde der Formalismus der Regulären Ausdrücke entwickelt, mit dessen Hilfe sich die lexikalische Struktur einer Sprache beschreiben lässt. Jetzt soll noch die Arbeitsweise des Scanners in Verbindung mit den Regulären Ausdrücken geklärt werden.

NFAs

Das Pattern Matching lässt sich durch den Entwurf eines *endlichen Zustandsautomaten* (FA, “finite automaton“) implementieren. Ein endlicher Zustandsautomat besteht aus einer Menge von endlich vielen benannten Zuständen sowie gerichteten Übergängen zwischen Paaren von Zuständen. Ein Übergang findet unter einer bestimmten Bedingung statt, im Falle des Scanners dem Lesen eines Zeichens.

Beim sogenannten *Nicht-Diskreten endlichen Automaten* (NFA) ist ein Übergang auch mit einem “leeren“ Zeichen erlaubt. Solche Übergänge ohne Bedingung werden ϵ*-Übergänge* genannt. Außerdem ist beim NFA die Definition mehrerer möglicher Übergänge zu verschiedenen Folgezuständen für das selbe Zeichen erlaubt. Der Automat besitzt weiterhin einen Startzustand sowie einen oder mehrere Endzustände. Das Erreichen eines Endzustandes entspricht einem Match eines Regulären Ausdrucks. Man spricht auch vom *Akzeptieren* einer Zeichenkette.

Das Beispiel in Abbildung 2.8 zeigt einen einfachen NFA für den Regulären Ausdruck a | ab. Die Zustände werden bei dieser Notation durch nummerierte Kreise dargestellt, die Übergänge durch Pfeile. Neben den Übergängen sind ihre Bedingungen angegeben. Der Startzustand ist durch einen offenen Pfeil ausgezeichnet, die Endzustände durch eine doppelte Umrandung.

Diese Art der Darstellung ist zwar sehr übersichtlich, eignet sich aber nicht besonders zur Implementierung eines Automaten. Deshalb werden Zustände und Übergänge für die

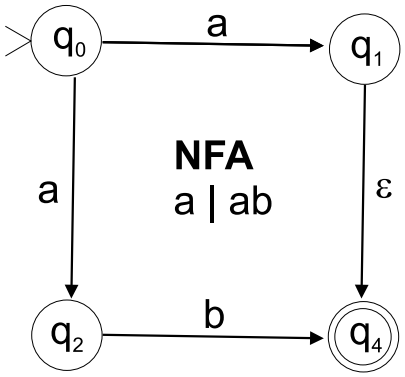

Abbildung 2.8: Einfacher NFA

Eingabezeichen:	a	b	ϵ
q_0	q_1,q_2	$\perp$	$\perp$
q_1	$\perp$	$\perp$	q_3
q_2	$\perp$	q_3	$\perp$
q_3 (Ende)	$\perp$	$\perp$	$\perp$

Abbildung 2.9: Übergangsmatrix

Implementierung besser in tabellarischer Form dargestellt. In einer solchen *Übergangsmatrix* werden die möglichen Zustände über den möglichen einzulesenden Zeichen aufgetragen. In den entsprechenden Feldern werden dann die durch das Zeichen erreichbaren Zielzustände eingetragen (Abbildung 2.9).

Der Algorithmus zur Interpretation eines NFA läuft nun wie folgt:

Eingabe: Zeichenstrom

Ausgabe: “akzeptiert“ oder “nicht akzeptiert“

Algorithmus: Sei S der Startzustand, z das aktuelle Eingabezeichen, M die Menge aller mit z erreichbarer Zustände, t das durch Erreichen eines Endzustands definierte Token, p die Eingabeposition (Anzahl der gelesenen Zeichen).

1. M:=∅, t:=$\perp$
2. Solange Eingabezeichen ≠ EOF:
   a) M wird um alle mit ϵ erreichbaren Zustände erweitert
   b) Falls M Endzustände enthält, wird t auf das durch den ersten Endzustand definierte Token gesetzt und p gemerkt. Damit lässt sich die Auflösung von Mehrdeutigkeiten sowie der “longest Match“ realisieren.
   c) neues z wird gelesen
   d) M’ wird gebildet als Menge aller Zustände, die von einem Zustand aus M mit z erreichbar sind

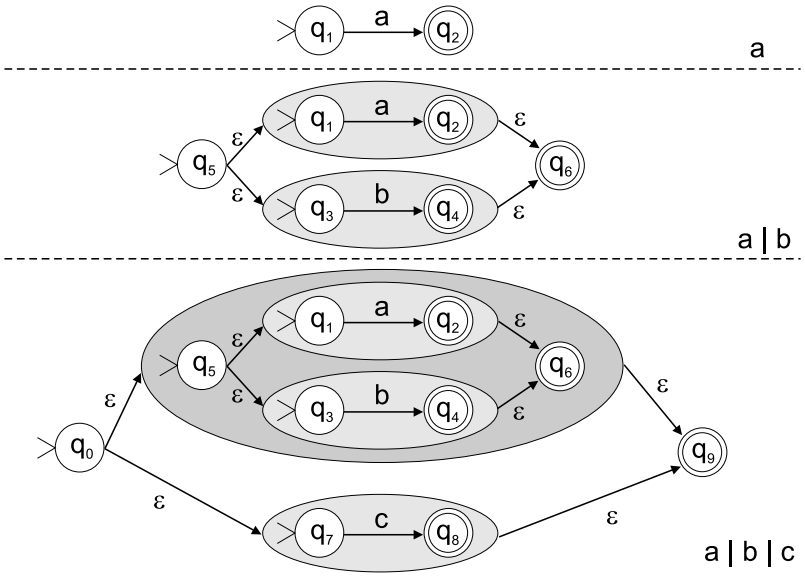

Abbildung 2.10: Konstruktion eines NFAs

e) M:=M' (M enthält nun die neue Menge erreichbarer Zustände)

f) Falls M=∅:
Ist t=⊥ wurde die Zeichenkette nicht akzeptiert (Ende mit Fehler).
Sonst wird t ausgegeben, die Eingabe auf das p-te Zeichen zurückgesetzt und M und t neu initialisiert (M:={S}, t:=⊥).

3. Falls M Endzustände enthält wird das durch den ersten Endzustand definierte Token t ausgegeben. (Zeichenkette akzeptiert)
Falls nicht wurde die Zeichenkette nicht akzeptiert (Ende mit Fehler).

Der Automat wird konstruiert, in dem ein Regulärer Ausdruck in seine Grundelemente zerlegt wird. Für jedes Grundelement wird nun ein Automat gebildet und aus diesen Automaten das Gesamtkonstrukt zusammengefügt. Als Grundelemente genügen bereits *Sequenz, Alternative* und *Iteration* (Abbildung 2.10).

Der NFA stellt eine einfache Möglichkeit zur Konstruktion eines Scanners zur Verfügung. Für diese Einfachheit muss man allerdings eine geringe Geschwindigkeit und einen hohen Speicherverbrauch in Kauf nehmen. Schließlich muss durch die Nicht-Eindeutigkeit der Übergänge für jeden Arbeitsschritt des Scanners (Einlesen eines Zeichens) eine Menge von Zuständen und eine Menge von Übergängen pro Zustand betrachtet werden. Der Speicherverbrauch sowie die Laufzeit des Scanners steigen dadurch quadratisch mit der Anzahl der Zustände des NFAs.

DFAs

Eine deutlich effizientere Verarbeitung erreicht man mit einem *Deterministischen Endlichen Automaten* (Deterministic Finite Automaton, DFA). Bei diesem gibt es von einem

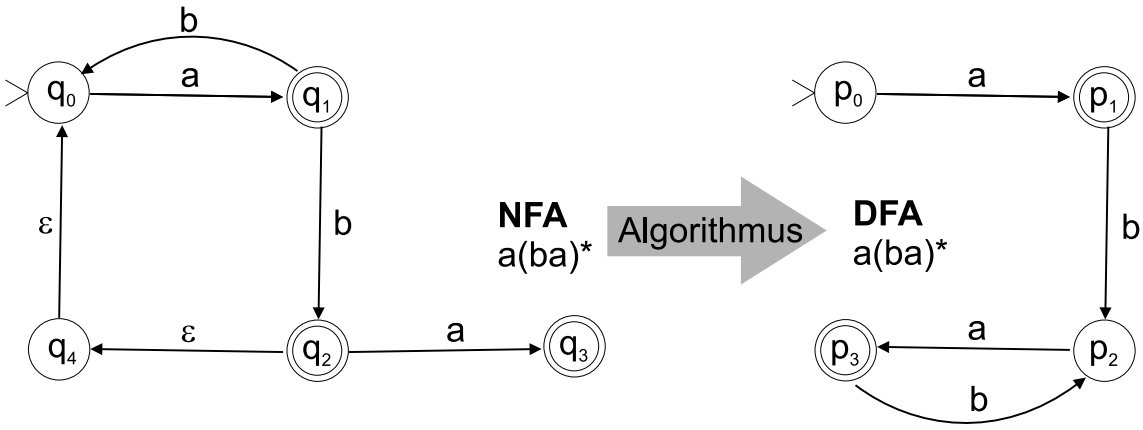

Abbildung 2.11: NFA → DFA

Zustand aus entweder *keinen* oder *genau einen* Übergang pro eingelesenem Zeichen. ϵ-Übergänge sind dabei nicht mehr erlaubt. Durch diese Vorgehensweise wird die Betrachtung von Übergangsmengen pro Zustand eingespart.

NFAs bieten trotz ihrer Ineffizienz den Vorteil einer sehr einfachen Konstruktion. Eine elegante Methode für die Konstruktion eines DFAs wäre deshalb, ihn aus einem NFA abzuleiten. Dies ist mittels eines einfachen Algorithmus auch möglich. Die Grundidee dabei ist, den NFA zu durchlaufen und Zustände, die durch die gleichen Eingabezeichenfolgen erreicht werden können, zusammen zu fassen. Weiterhin werden mit ϵ-Übergängen erreichbare Zustände in der sogenannten *ϵ-Closure* zusammen gefasst. Das Beispiel in Abbildung 2.11 illustriert die Konstruktion eines DFAs aus einem NFA.

Da nun in jedem Zustand des Automaten die Übergänge für jedes Zeichen eindeutig sind, vereinfacht sich der Algorithmus wie folgt:

Eingabe: Zeichenstrom

Ausgabe: “akzeptiert“ oder “nicht akzeptiert“

Algorithmus: Sei t das durch Erreichen eines Endzustands definierte Token, p die Eingabeposition (Anzahl der gelesenen Zeichen).

1. Automat wird in Startzustand gesetzt
2. Zeichen z wird eingelesen, Übergang in durch das Zeichen festgelegten Zielzustand
   Falls Zielzustand undefiniert:
   - Falls t=$\perp$: Ende mit Fehler (Zeichenkette nicht akzeptiert).
   - Sonst: Ausgabe von t, Neustart des Algorithmus mit p+1 bis z=EOF. (Zeichenkette akzeptiert)

   Falls Zielzustand Endzustand, Token t und Position p merken für “longest Match“

3. Wiederholung von Schritt 2

Dieser Algorithmus ist deutlich weniger aufwändig als der des NFAs. Um ein optimales Ergebnis zu erreichen, lässt sich nun noch die Anzahl der Zustände auf ein Minimum reduzieren. In Abbildung 2.11 z.B. könnte man auf die Zustände p_0 und p_1 verzichten, ohne das Ergebnis des Automaten zu verändern. Um den Rahmen dieser Arbeit nicht zu sprengen, soll auf das Optimierungsverfahren allerdings nicht näher eingegangen werden.

2.3 Die Grammatik

2.3.1 Grammatiken und formale Sprachen

Eine formale Sprache besteht aus einer Menge von gültigen Zeichenketten, den *Lexemen*, welche die Menge aller in der Sprache erlaubten Wörter festlegt. Auch ein einzelnes Zeichen kann bereits ein Wort bilden, z.B. ein Operator wie “>“. Verschiedene Lexeme ähnlicher Bedeutung fasst man zu Klassen zusammen. Diese Klassen werden *Token* genannt. In genannten Beispiel wären das alle Vergleichsoperatoren, also "==", "<", "<=", "!=", etc. Um den “Satzbau“ der Sprache festzulegen, wird ein Regelwerk definiert, das wie folgt arbeitet:

- Die Menge der in der Lexeme wird erweitert um eine Menge von *Zwischensymbolen*. Man spricht von *Terminalen* (diese entsprechen Token) und *Nichtterminalen* (Zwischensymbole). Manche Grammatiken erlauben auch das ϵ*-Symbol* (“leeres“ Symbol), d.h. die Ableitung eines Nichtterminals “ins Nichts“.
- Es werden eine Reihe von Regeln festgelegt, mit deren Hilfe sich die Nichtterminale *ableiten* lassen, d.h. aus ihnen werden neue Terminale oder Nichtterminale gebildet. Terminale lassen sich, wie der Name bereits nahelegt, nicht ableiten.
- Zusätzlich wird ein *Startsymbol* eingeführt. Dieses ist ein spezielles Nichtterminal, auch die “Wurzel“ der Grammatik genannt. Von diesem Symbol aus lassen sich durch die wiederholte Anwendung verschiedener Ableitungsregeln alle in der formalen Sprache erlaubten Anordnungen von Terminalen bilden.
- Diese Menge aus Terminalen, Nichtterminalen und Ableitungsregeln bezeichnet man als *Grammatik*.

Als kleines Beispiel soll Abbildung 2.12 den Sachverhalt verdeutlichen. Die übliche Notation unterscheidet Terminale und Nichtterminale mit Groß- und Kleinschreibweise bzw. Groß- und Kleinbuchstaben.

Start	→	Summe
Summe	→	Zahl + Zahl |
		Zahl – Zahl
Zahl	→	INTEGER |
		FLOAT

Abbildung 2.12: Eine einfache Grammatik

Kontextfrei			**Kontextsensitiv**		
S	→	A	S	→	A
A	→	aua |	mAu	→	muu |
		aha			miau

Abbildung 2.13: Kontextfreie und kontextsensitive Grammatik

2.3.2 Ableitungen und Reduktionen

Mit dieser einfachen Grammatik lassen sich alle Ausdrücke beschreiben, in denen zwei Zahlen, die Ganze Zahlen oder Gleitkommazahlen sein können, miteinander summiert oder von einander subtrahiert werden. Wie man sieht, bestehen die Ableitungsregeln aus einer linken Seite, die mindestens ein Nichtterminal enthalten muss, da sonst keine weitere Ableitung möglich wäre, sowie einer rechten Seite, die sich beliebig aus Terminalen und Nichtterminalen zusammen setzt. Das Ableiten geschieht dadurch, dass Nichtterminalsymbole auf der linken Seite durch die entsprechende rechte Seite ersetzt werden können.

Bestehen alle Regeln auf der linken Seite aus genau einem Nichtterminalsymbol, spricht man von einer *kontextfreien* Grammatik. Dies bedeutet, dass das Nichtterminalsymbol bedingungslos durch die rechte Seite der Regel ersetzt werden kann. Bei einer *kontextsensitiven* Grammatik ist dies nur möglich, wenn das Nichtterminalsymbol im richtigen Kontext steht, z.B. zwischen zwei bestimmten Terminalen wie in Abbildung 2.13. Kontextfreie Grammatiken sind im Vergleich zu den unter Umständen extrem komplexen kontextsensitiven Grammatiken sehr gut geeignet, um Programmiersprachen zu beschreiben.

Der Ableitungsvorgang lässt sich auch umgekehrt betreiben. Man spricht in diesem Fall nicht von Ableitungen, sondern von *Reduktionen* (Abbildung 2.14). Bei einer Reduktion werden die rechten Seiten der Regeln durch die Linken ersetzt. Beim parsen nach diesem Verfahren wird der Vorgang fortgeführt, bis das Startsymbol erreicht ist, was einen erfolgreichen Parsevorgang bedeutet. Lässt sich das Startsymbol nicht durch Reduktionen erreichen, wurde ein nach dieser Grammatik nicht erlaubter Satz eingelesen, es liegt ein *Syntaxfehler* vor.

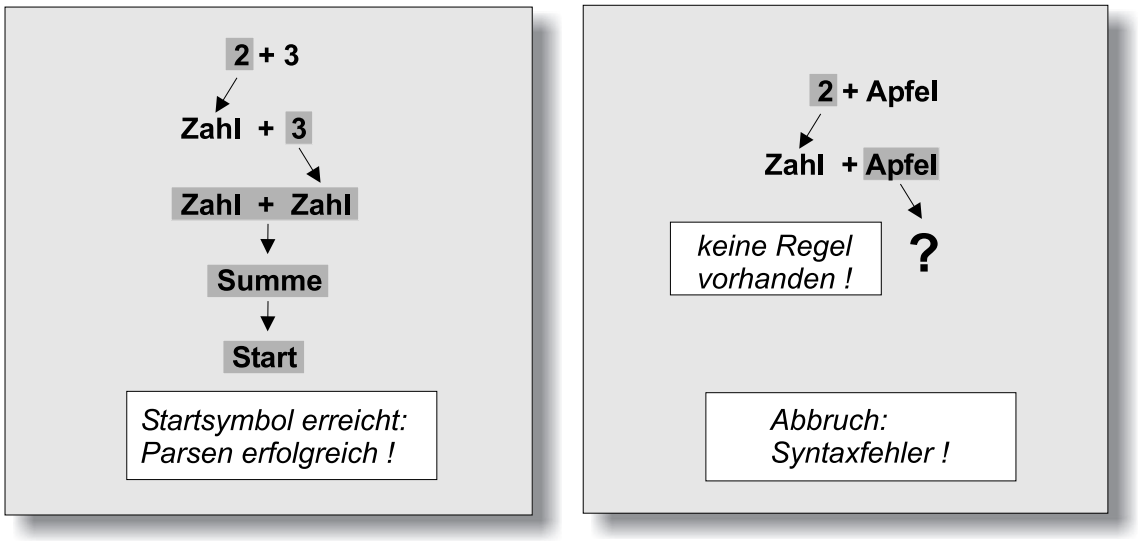

Abbildung 2.14: Reduktion der Symbole

Wurzel	→	Ausdruck
Ausdruck	→	Ausdruck + Ausdruck |
		Ausdruck – Ausdruck |
		INTEGER

Abbildung 2.15: Rekursive Grammatik

2.3.3 Rekursive Grammatiken

In den Regeln einer Grammatik lässt sich selbstverständlich auch Rekursion verwenden. Dadurch ist es möglich, auch komplexere Sprachen elegant zu beschreiben, wie das Beispiel in Abbildung 2.15 zeigt.

Diese Grammatik, die beliebig lange Summen und Produkte beschreibt, ist dank des Einsatzes von Rekursion sehr kurz. Allerdings bringt die Rekursion auch Probleme mit sich. So ist zum Beispiel bei einem Eingabetext der Art:

$$2 + 3 * 4$$

nicht klar, ob der Parser zuerst 3*4 oder 2+3 reduziert. Für eine korrekte Auswertung des Ausdrucks ist dies allerdings entscheidend. Eine Grammatik, die mit diesem Problem behaftet ist, wird *mehrdeutig* genannt (Abbildung 2.16).

Eine Lösung dieses Problems bietet die Anwendung von Assiotiativitätsregeln, wie auch schon aus der Mathematik bekannt. Dabei werden den in der Sprache erlaubten Operatoren Prioritäten zugewiesen, so dass Operatoren mit höherer Priorität eine stärkere Bindung besitzen als solche geringerer Priorität. Dabei kann es auch vorkommen, dass Operatoren gleiche Priorität besitzen. In diesem Fall erfolgt die Reduktion einheitlich links- oder

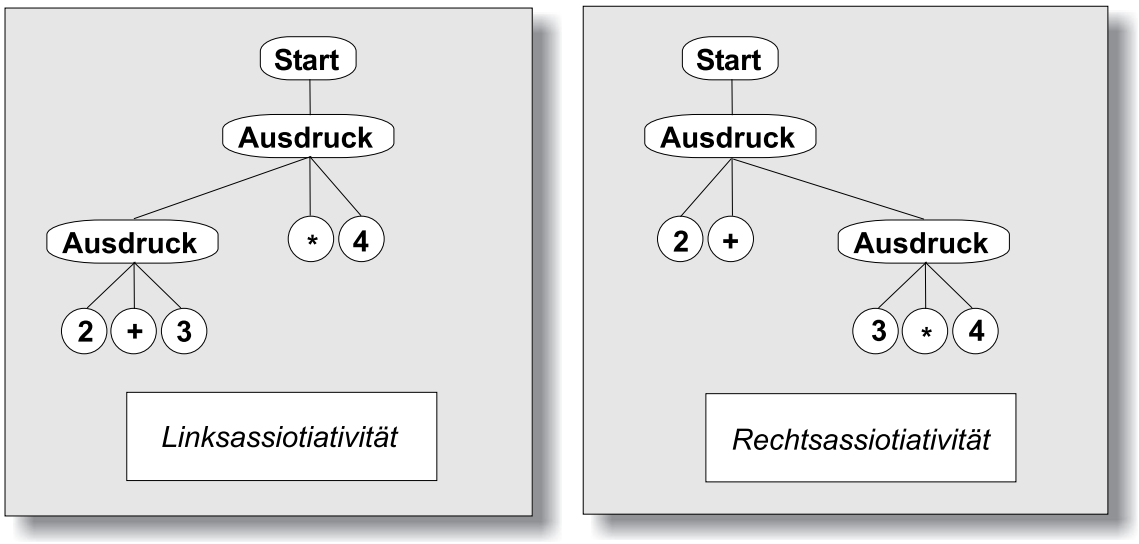

Abbildung 2.16: Mehrdeutigkeit

rechtsassiotiativ.

2.4 Parser und syntaktische Analyse

2.4.1 Parsebaum und Syntaxbaum

Ein Parser analysiert einen Eingabetext auf seine syntaktische Struktur auf der Grundlage einer vorgegebenen Grammatik. Bei einem erfolgreichen Durchlauf wird vom Parser die syntaktische Struktur des Textes in einem Baum gespeichert. Dieser *Parsebaum* repräsentiert die vollständige Eingabe und alle Ableitungsschritte. Die Blätter des Baumes stellen dabei den Eingabetext (Terminale) dar, in den Knoten sind die Zwischensymbole (Nichtterminale) enthalten.

Für die weitere Verarbeitung wird der Parsebaum zu einem *Syntaxbaum* vereinfacht. Dieser enthält nur noch die für die syntaktische Struktur relevanten Anteile. Einzelne Ableitungsschritte und Gruppierungsterminale (z.B. alle Klammertypen) werden entfernt. Typen von syntaktischen Gebilden, z.B. eine Zuweisung oder eine Multiplikation, werden im übergeordneten Knoten verschlüsselt und zugehörige Operatoren (in diesem Fall ‘=‘ bzw ‘∗‘) werden ebenfalls entfernt.

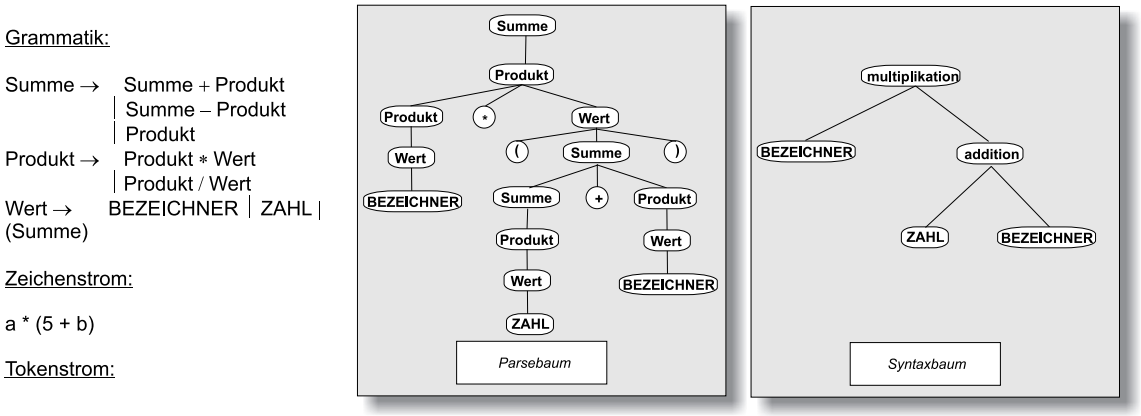

Abbildung 2.17: Parsebaum und Syntaxbaum

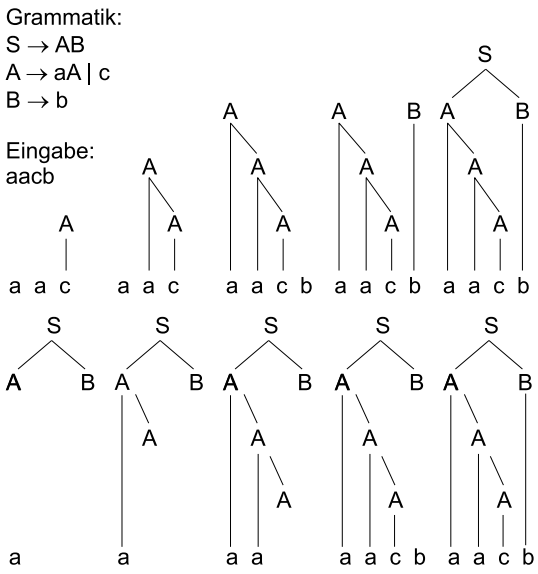

Abbildung 2.18: oben: Rechtsableitung (Bottom-Up), unten: Linksableitung (Top-Down)

2.4.2 Links- und Rechtsableitung

LL-Parser

Für die Konstruktion von Parsern unterscheidet man hauptsächlich zwischen zwei Verfahren: Den *LL* Parsern und den *LR*-Parsern. Am einfachsten lassen sich LL-Parser konstruieren. Das erste L steht dabei für die Verarbeitung des vom Scanner angelieferten Token-Stroms von *l*inks nach rechts, das zweite L für die Linksableitung der Eingabe. Linksableitung bedeutet, das immer zuerst das am weitesten links stehende Nichtterminal einer rechten Seite einer Regel abgeleitet wird. Dabei wird der Parsebaum von der Wurzel zu den Blättern hin aufgebaut, man spricht deshalb auch von einem *Top-Down*-Verfahren (in der Informatik wachsen Bäume von oben nach unten).

Bild 2.18 demonstriert die Methode der Linksableitung. Dabei werden an den Wurzelknoten solange Unterknoten angefügt, bis alle Nichtterminale abgeleitet sind (Erfolg) oder die Ableitung eines Nichtterminals nicht mehr möglich ist (Fehler).

Beim Hinzufügen der Unterknoten kann es allerdings verschiedene Möglichkeiten geben. Deshalb liest der Parser ein oder mehrere Token voraus, um eine eindeutige Entscheidung treffen zu können. Dieses Verfahren wird *Lookahead* genannt. Einen LL-Parser, der n Zeichen vorausliest, bezeichnet man als LL(n) Parser. In der Praxis kommen fast nur LL(1)-Parser zum Einsatz.

LL-Parser lassen sich sehr einfach implementieren. Allerdings setzen sie voraus, dass ihre Grammatik frei von *Linksrekursionen* ist, d.h., auf der rechten Seite einer Ableitungsregel darf nicht ganz links das gleiche Nichtterminalsymbol stehen wie auf der linken Seite der Regel. Ist dies doch der Fall, kann es beim parsen zur Endlosrekursion kommen.

LR-Parser

Im Gegensatz zu den LL-Parsern wird bei LR-Parsern immer das am weitesten rechts stehende Nichtterminal abgeleitet. Der Parserbaum wird hier von den Blättern zur Wurzel hin aufgebaut, deshalb wird hier auch von einem *Bottom-Up*-Verfahren gesprochen. Im Compilerbau werden eher LR-Parser als LL-Parser eingesetzt. Das hat verschiedene Gründe: Zum einen unterliegt der LR-Parser nicht dem Nachteil der Endlosrekursionen durch Linksrekursion. Für die meisten Programmiersprachen sind deshalb auch eher LR-Grammatiken als LL-Grammatiken verfügbar. Ein weiterer Grund ist, das sich syntaktische Fehler mit dem Bottom-Up-Verfahren schneller erkennen lassen als mit dem Top-Down-Verfahren.

2.4.3 Lookahead, First- und Followmengen

Mit Hilfe des Lookahead-Mechanismus kann der Parser entscheiden, welche Ableitungsregel als nächstes ausgewählt wird. Um das Lookahead mit den für die Entscheidung benötigten Informationen zu versorgen, stehen zwei Funktionen zur Verfügung: *FIRST* und *FOLLOW*.

Die Funktion FIRST(γ), angewendet auf ein Nichtterminal , bildet die Menge aller Elemente, die als erstes Terminal aus γ abgeleitet werden können. Ist t ein Terminal, N ein Nichtterminal sowie α eine Anordnung von Terminalen und Nichtterminalen, so bedeutet $t \in$ FIRST (N), das es für beliebiges α eine Ableitung $N \rightarrow t\alpha$ gibt.

Die von First zurückgegebene Menge reicht allerdings nicht immer für eine Entscheidung aus, vor allem dann, wenn die Grammatik ϵ-Ableitungen zulässt. Lautet die Grammatik z.B.

$$\begin{array}{lcl} S & \rightarrow & XY \\ X & \rightarrow & a \mid \epsilon \\ Y & \rightarrow & c \end{array}$$

so ist nicht klar, wie "X" abgeleitet werden muss, wenn das nächste Token "c" ist. In solchen Fällen kommt die Funktion FOLLOW zum Einsatz. FOLLOW(γ) bildet die Menge aller Terminale, die in einem Ableitungsschritt auf γ folgen können. Dabei wird die von FIRST erzeugte Menge um die mit ϵ-Ableitungen erreichbaren Terminale erweitert.

2.4.4 Implementierungs-Strategien

Für die Implementierung eines Parsers stehen zwei unterschiedliche Strategien zur Verfügung. Die für den Programmierer zunächst einmal naheliegend erscheinende Methode ist, die grammatikalische Struktur in einzelnen Funktionen abzubilden, die sich gegenseitig aufrufen wie in Abbildung 2.19. Diese Methode wird *Rekursiver Abstieg* genannt, einen so implementierten Parser nennt man auch *Recursive-Decent-Parser*. Der Vorteil dieser Methode ist die Verständlichkeit des Programmcodes, der sich für kleinere Aufgaben, z.B. dem parsen von einfacheren mathematischen Ausdrücken, auch schnell entwickeln lässt.

Diese Verfahren hat allerdings auch Nachteile: Zum einen ist der Einsatz von unter Umständen tief verschachtelten Rekursionen sehr speicherintensiv. Zum anderen muss der Programmcode des Parsers speziell für eine einzige Grammatik entworfen werden. Ändert man die Grammatik, zieht das zwangsweise Änderungen am Programmcode nach sich. Packt man dagegen die Grammatik-Informationen in Datenstrukturen, die von Parser nur ausgewertet werden, muss bei Änderung der Grammatik nur die Datenstruktur geändert werden, was sich auch automatieren lässt.

Dieses Strategie wird beim *Tabellengesteuerten Parser* eingesetzt. Wie auch schon bei der Übergangsmatrix eines Endlichen Automaten eines Scanners werden die Informationen über aktuelle Zustände des Parsers, vom Parser durchzuführende Aktionen in Abhängigkeit eines eingelesenen Tokens sowie den resultierenden Zielzuständen in einer *Parsetabelle* abgelegt. Der Programmcode des Parsers ist dabei unabhängig von der Grammatik. Die zur Grammatik passende Parsetabelle ist zwar von Hand nur mühsam zu implementieren, lässt sich aber auch automatisch erzeugen.

2.4.5 Parsetabelle und Stack

Mit Hilfe der First- und Followmengen lässt sich zu jeder Kombination bereits eingelesener Token vorhersagen, welche Token noch folgen müssen, damit die Eingabe bezüglich der Grammatik korrekt ist. Durch das Lookahead ist weiterhin das nächste einzulesende Token bekannt. Mit Hilfe dieser Informationen lässt sich nun eine Matrix anfertigen wie in Abbildung 2.20.

```
float Summe() {
        Wert = Produkt()
        if Operator == ‘+‘
                Wert += Summe()
        else
                Wert -= Summe()
return Wert
}

float Produkt() {
Wert = ID()
        if Operator == ‘*‘
                Wert *= Produkt()
        else
                Wert /= Produkt()
return Wert
}

float ID() {
Wert = get_value_from_string()
return Wert
}
```

Abbildung 2.19: Auswerten eines Ausdrucks mit rekursivem Abstieg

Grammatik: S → E
E → (E + E)
E → id

Shift-Tabelle:

Zustand	**id**	**+**	**(**	**)**	**EOF**
Z0	shift goto Z3	⊥	shift goto Z2	⊥	⊥
Z1	⊥	⊥	⊥	⊥	accept
Z2	shift goto Z3	⊥	shift goto Z2	⊥	⊥
Z3	⊥ ⊥	reduce E→id	⊥	reduce E E→id	reduce E→id
Z4	⊥	shift, goto Z7	⊥	⊥	⊥
Z7	shift, goto Z3	⊥	shift goto Z2	⊥	⊥
Z8	⊥	⊥	⊥	shift, goto Z10	⊥
Z10	⊥	reduce E→(E+E)	⊥	reduce E→(E+E)	reduce E→(E+E)

Reduce-Tabelle:

Zustand	**E**
Z0	goto Z1
Z2	goto Z4
Z7	goto Z8

Abbildung 2.20: Parsetabelle

Wie der Scanner bildet auch der Parser einen endlichen Automaten. Die Parsetabelle besteht aber anders als die Übergangsmatrix eines Scanners aus zwei Teilen. Im ersten Teil wird festgelegt, ob der Parser eine Regel reduziert (*reduce*) oder ein weiteres Zeichen einliest (*shift*). Wird ein weiteres Zeichen eingelesen, ändert sich auch der Zustand entsprechend. Im zweiten Teil steht der Zielzustand, der erreicht wird, wenn eine Regel reduziert wird. Da dieser Zustand nicht vom aktuellen, sondern vom vorhergehenden Zustand des Automaten abhängig ist, wird dafür eine zweite Tabelle angelegt.

Der Ablauf ist in Abbildung 2.21 dargestellt. Der Parser liest die Token (Lookahead-Token) aus dem Eingabestrom und legt den daraus resultierenden Zustand auf einem Stack ab (TOS). Aus der Parsetabelle lässt sich nun für jede Kombination aus Lookahead und TOS eine Anweisung entnehmen. Lautet diese nach einer Regel zu reduzieren, werden die zugehörigen Zustände vom Stack entfernt. Ist keine Anweisung definiert, so liegt ein Syntaxfehler vor. Ist der Stack leer, ohne dass ein Fehler aufgetreten ist, wurde die Eingabe

Stack	Eingabestring	Aktion
Z0	(E + E) EOF	shift, goto Z2
Z0, Z2	E + E) EOF	shift, goto Z3
Z0, Z2, Z3	+ E) EOF	red. E→id, goto Z4
Z0, Z2, Z4	+ E) EOF	shift, goto Z7
Z0, Z2, Z4, Z7	E) EOF	shift, goto Z3
Z0, Z2, Z4, Z7, Z3	) EOF	red. E→id, goto Z8
Z0, Z2, Z4, Z7, Z8	) EOF	shift, goto Z10
Z0, Z2, Z4, Z7, Z8, Z10	EOF	red. E→(E+E), goto Z1
Z0, Z1	EOF	accept

Abbildung 2.21: Ablauf des Parsers

akzeptiert.

2.4.6 Symboltabellen

Da sich Programmiersprachen im Allgemeinen durch kontextfreie Grammatiken beschreiben lassen, aber deshalb trotzdem nicht kontextfrei sind, wird dem Parser oft noch die Aufgabe der semantischen Analyse aufgeladen. So ist es zum Beispiel in den meisten Programmiersprachen nicht möglich, eine Variable ohne vorherige Deklaration zu benutzen. Wird dies doch getan, so liegt zwar kein syntaktischer Fehler vor, der laut Grammatik nicht erlaubt wäre, aber ein Semantischer.

Fehler dieser Art lassen sich schon während des Parserdurchlaufs entdecken. Dazu muss der Parser (oder ggf. sogar schon der Scanner) eine *Symboltabelle* (Abbildung 2.22) anlegen, in die die Variablen und Informationen über Gültigkeitsbeireich, Typ, etc. eingetragen werden. Anhand dieser Tabelle lässt sich dann z.B. überprüfen, ob eine Variable vor ihrer Verwendung deklariert wurde. Solche Bedingungen, die bereits während der syntaktischen Analyse abgeprüft werden können, werden als Eigenschaften der statischen Semantik bezeichnet.

Ob die semantische Analyse schon im Parser beginnt, bleibt dabei dem Programmierer überlassen. Im Interesse eines schnellen Auffindens von Fehlern ist dies allerdings empfehlenswert.

Object	Type	Value	Location
i	int	5	Code-Block 15
arg	double	11.5	global
i	String	"Hallo"	Code-Block 17
c_ref	int	undef	Code-Block 17
...	...	...	...

Abbildung 2.22: Symboltabelle, “Location“ gibt den Deklarationsort und damit die Gültigkeit der Variable an

3 Scanner- und Parsergeneratoren

3.1 Der Scannergenerator JLex

Ein Scanner lässt sich durchaus von Hand implementieren. Durch die vielfältigen Optimierungsmöglichkeiten beim Entwurf von Hand ist ein solcher Parser im allgemeinen einem generierten überlegen, was die Performance betrifft. Allerdings ist dieser Ansatz mühsam, wenn komplizierte Konstrukte wie zum Beispiel Gleitkommazahlen aus dem Eingabestrom zu identifizieren sind. Änderungen und Erweiterungen sind meist nur umständlich durchzuführen, damit steigt auch die Anfälligkeit für Fehler.

In einigen Hochsprachen lassen sich für die lexikalische Analyse auch fertige Scannerklassen einsetzen. Für Java steht hierzu die Klasse “java.io.StreamTokenizer“ zur Verfügung. Damit lassen sich unter anderem Identifier, Zahlen, Quoted Strings, verschiedene Kommentare etc. erkennen. Das Verhalten der Scannerobjekte kann durch Modifikation der Methoden eingestellt werden. Vorteile dabei ist ein schneller Entwurf eines Parser aus schon getesteten und damit fehlerfreien Elementen. Nachteilig wirkt sich die Tatsache aus, dass die zu erkennenden Token weitestgehend vordefiniert und damit nicht allgemein verwendbar sind, so übersteigt zum Beispiel die Unterscheidung von Gleitkomma- und Integer-Zahlen schon die Möglichkeiten der Klasse.

Ein eleganterer Ansatz, der fast genauso schnell zum Ziel führt, ist deshalb der Einsatz eines Scannergenerators wie *JLex*. JLex ist eine Neumimplementierung des Scannergenerators flex, der hauptsächlich für die Programmiersprache C entworfen wurde. JLex ist für den Bau von Scannern in Java-Umgebungen konzipiert und vollständig in Java geschrieben.

3.1.1 Konfiguration

JLex generiert aus einer Tabelle aus egrep-artigen Mustern, die einzelne Token beschreiben, Code für verschiedene Funktionen zur Texterkennung und Weiterverarbeitung. Die Tabelle befindet sich zusammen mit weiteren Angabe zur Konfiguration des Scanners in einer Steuerdatei.

Die Steuerdatei von JLex besteht aus drei Teilen: der *user-code-section*, den *JLex directives* und der *regular expression rules*. Die einzelnen Sektionen sind durch eine Zeile, die mit “% %“ beginnt, voneinander getrennt

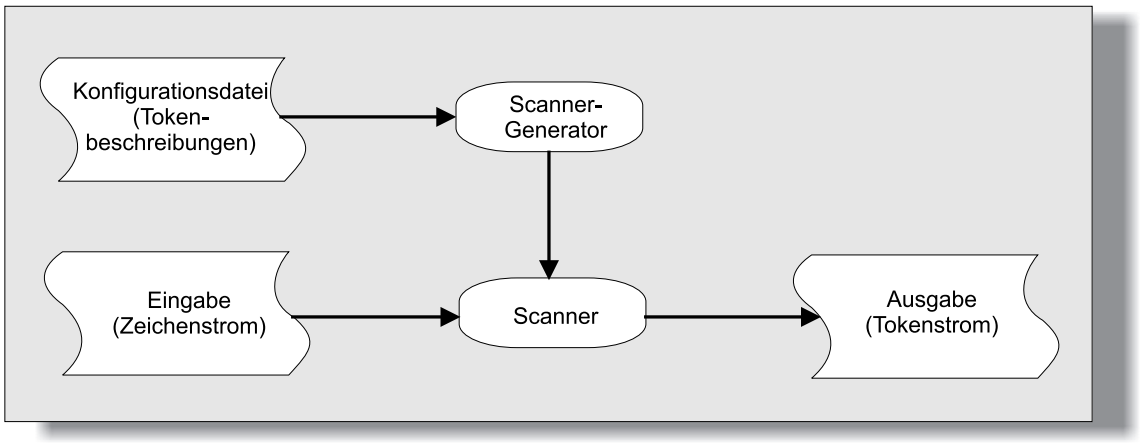

Abbildung 3.1: Funktionsweise eines Scannergenerators

```
letter      [a-zA-Z]
digit       [0-9]
whitespace  [ \t\n]
...
```

Abbildung 3.2: Makros in den JLex-Direktiven

Alle Zeilen in der user-code-section werden direkt an den Anfang des erzeugten Codes kopiert. Damit lassen sich *package*- und *include*-Anweisungen angeben sowie Hilfsklassen anlegen, die allerdings nicht *public* deklariert sein dürfen und somit nur vom erzeugten Scanner selbst angesprochen werden können. Die hier angegebenen Code-Fragmente befinden sich noch ausserhalb des Klassenkörpers des eigentlichen Scanners.

In den JLex-direktiven wird das Verhalten der erzeugten Scanner-Klasse gesteuert. Dies geschieht durch Anweisungen, die mit “%“ beginnen. Soll die Scanner-Klasse zum Beispiel um eigenen Code erweitert werden, so muss dieser Zwischen die Anweisungen “%{“ und “%}“ geschrieben werden. Auch Name und Zugriffsrechte sowie weitere Modifikatoren und Eigenschaften lassen sich hier mit Anweisungen wie %public, %class und %implements definieren. Mit %eofval lässt sich ein Anweisungblock definieren, der bei erreichen des Eingabestrom-Endes angesprungen wird und einen EOF-Wert liefert. Weiterhin lassen sich Makros für Reguläre Ausdrücke definieren, wodurch sich eine bessere Übersichtlichkeit erreichen lässt.

Der “regular expression rules“-Abschnitt besteht aus Regeln, nach denen die Zeichen der Eingabe in die einzelnen Tokens zerlegt werden sollen. Eine Regel besteht einer optionalen Zustandsliste, einem regulären Ausdruck und einem Anweisungs-Block. Matcht ein Regulärer Ausdruck, wird der entsprechende Anweisungsblock ausgeführt.

Treffen mehrere Reguläre Ausdrücke zu, werden Mehrdeutigkeiten aufgelöst, indem die

```
%%
%{
Integer value;
%}
%type int
%eofval{
return Taschenrechner.EOF;
%eofval}
ws = [\t\r\n\b\015] +
%%

 "+"          { return Taschenrechner.PLUS; }
 "*"          { return Taschenrechner.MULT; }
 [0-9][0-9]*  { this.value = new Integer(yytext());
              return Taschenrechner.ID;}
 {ws}         {System.out.print(yytext());}
```

Abbildung 3.3: JLex-Konfigurationsdatei

Regel angewandt wird, die auf die längste Zeichenkette passt ("longest match"). Bei gleicher Länge gewinnt die zuerst deklarierte Regel. Durch das Definieren und Zuweisen von Zuständen zu bestimmten Regeln kann man erreichen, das bestimmte Regeln in bestimmten Zuständen des Scanners zusätzlich angewandt werden.

3.1.2 Erzeugter Code

JLex erzeugt als Scanner, wenn der Name nicht durch die Konfiguration geändert wurde, die Klasse "Yylex". Yylex hat zwei public Konstruktoren, einer akzeptiert als Eingabestrom einen java.io.Reader, während der andere einen java.io.InputStream benötigt. Für die Weiterverarbeitung der erkannten Token werden drei Methoden generiert:

yylex: Die Methode yylex gibt das nächste einen Alias-Wert für das nächste erkannte Token zurück, im Folgenden als Token-Nummer bezeichnet. Die Token-Nummern sind Integer, die vom Benutzer oder von einem Parsergenerator vorher definiert werden müssen.

yytext: Mit der yytext-Methode lässt sich die als letztes Token erkannte Zeichenkette erfragen. Liefert yylex beispielsweise die Token-Nummer für einen String, so wird von yytext der Inhalt des Strings geliefert.

yylen: yylen liefert als Rückgabewert die Anzahl der Zeichen, die zum Erkennungsmuster des von yylex erkannten Tokens gelesen wurden.

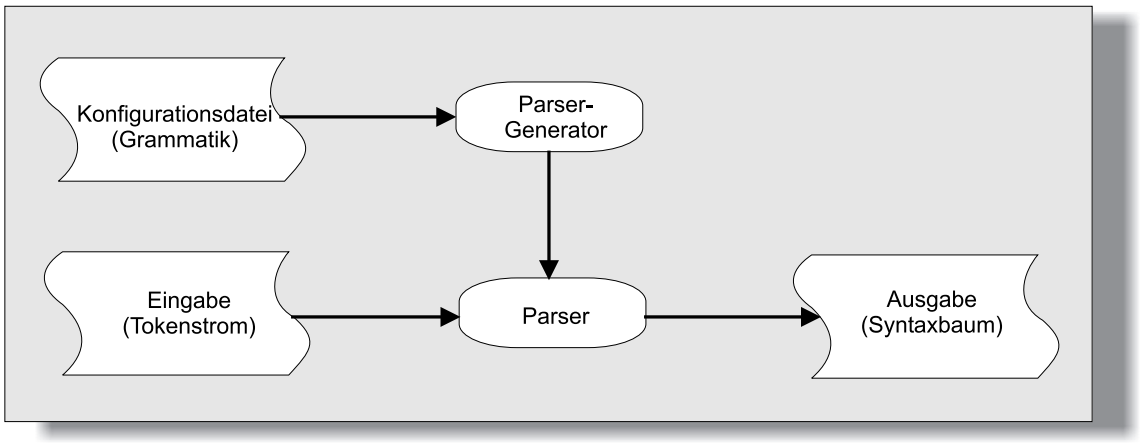

Abbildung 3.4: Funktionsweise eines Parsergenerators

3.2 Der Parsergenerator jay

Ähnlich wie schon beim Scannergenerator ist die Grundidee beim Parsergenerator die Trennung von Programmcode und Regelwerk. Mit diesem Ansatz lässt sich der Parser schnell und flexibel an Änderungen der Grammatik anpassen, ohne den Programmcode zu ändern und zeitraubenden Tests unterziehen zu müssen.

Auch die Konfiguration des Parsergenerators läuft ähnlich ab wie bei einem Scannergenerator. Der Parsergenerator liest eine Konfigurationsdatei ein, in dem das Regelwerk der Grammatik sowie weitere Konfigurationsangaben enthalten sind. Aus diesen Angaben wird dann der Programmcode für einen Parser generiert, der Eingabetexte bezüglich der gegebenen Grammatik analysieren kann.

Ein bekannter Vertreter der Parsergeneratoren ist der von Dr. Bernd Kühl und Prof. Dr. Axel-Tobias Schreiner an der Universität Osnabrück entwickelte “jay“. Jay ist eine Weiterentwicklung des Parsergenerators “yacc“, der sich in der UNIX-Welt bereits zu einem Standard-Werkzeug entwickelt hat. Yacc ist in C implementiert worden und vor allem für den C-Compilerbau konzipiert. Mit jay wurde yacc an die Anforderungen der Java-Welt angepasst.

3.2.1 Konfiguration

Die Konfiguration von jay erfolgt im Unterschied zu der von JLex über zwei Dateien: Der eigentlichen Konfigurationsdatei sowie dem sogenannten *skeleton*, in dem der Programmcode des generierten Parsers definiert wird.

Die Konfigurationsdatei von jay ist ähnlich aufgebaut wie die von JLex. Auch sie besteht aus drei Teilen: dem *prolog*, dem *local*-Teil und dem *epilog*. Wie schon bei JLex sind die drei Teile durch ein ‘%%‘voneinander getrennt.

```
%{
package arith_simple;

import java.io.*;
import java.util.*;

public class Arith {
%}

%token    Double Number 99
%type     Double expr

%left     '+'  '-'
%left     '*'  '/'
%right    UNARY

%start    prog

%% // public Object yyparse(yyInput yyLex) throws IOException, yyException

expr :    expr '+'expr { $$ = new Double($1.doubleValue()+$3.doubleValue()); }
          | expr  '-' expr  { $$ = new Double($1.doubleValue()-$3.doubleValue()); }
          | expr  '*' expr  { $$ = new Double($1.doubleValue()* $3.doubleValue()); }
          | expr  '/' expr  { $$ = new Double($1.doubleValue()/$3.doubleValue()); }
          | '+' expr  %prec UNARY { $$ = $<>2; } // can suppress class
          | '-' expr  %prec UNARY { $$ = new Double(-$2.doubleValue()); }
          | '(' expr  ')'  { $$ = $2; }
          | Number  // $$ = yyDefault($1);

prog :    /* null */
          | prog expr  '\n'  { System.out.println("\t"+$2); }
          | prog  '\n'
%%
```

Abbildung 3.5: jay-Konfigurationsdatei

Im prolog-Teil lässt sich der Kopf der zu erzeugenden Parser-Klasse angeben. Dieser wird direkt an den Anfang des erzeugten Codes kopiert. Zusätzlich sind weitere Konfigurationsangaben möglich, wie zum Beispiel die Angabe von Prioritäten zum Erstellen von Assotiativitätsregeln.

Im local-Teil steht die eigentliche Grammatik in BNF sowie die Aktionen, die beim Erkennen einer Regel ausgeführt werden sollen. Weiterhin lässt sich in diesem Teil noch Code angeben, der an den Anfang der erzeugten Interpreterfunktion kopiert wird, z.B. um lokale Variablen anzulegen.

Der epilog-Teil wird einfach direkt ans Ende der erzeugten Quellcode-Datei kopiert und beendet damit die Parser-Klasse.

Die Steuerdatei skeleton enthält das Grundgerüst des erzeugten Parsercodes. Sie besteht zum einen aus Code-Fragmenten, die direkt in die zu erzeugende Quellcode-Datei kopiert werden, und weiterhin aus Funktionsaufrufen. Die Funktionsaufrufe wiederum sorgen dafür, dass der prolog-Teil kopiert wird, die Parsertabelle eingefügt wird usw. Der erzeugte Parser wird also anhand des skeletons sukzessive zusammengesetzt. Sollen also prinzipielle Änderungungen am Parser vorgenommen werden, genügt es möglicherweise, die skeleton-Datei anzupassen, ohne in den jay-Quellcode eingreifen zu müssen.

3.2.2 Erzeugter Code

Der von jay erzeugte Code besteht im wesentlichen aus Token-Konstanten, einem Scanner-Interface und der eigentlichen Parser-Methode *yyparse*. Da die Token vom Scanner als Integer-Werte übergeben werden müssen, generiert jay eine statische Konstante für jedes in der Grammatik angegebene Token. Die Konstanten sind public und können somit vom Scanner verwendet werden.

Das in Abbildung 3.6 abgebildete Scanner-Interface stellt drei Methoden bereit, die vom Scanner implementiert werden müssen:

advance () versucht das nächste Token einzulesen und gibt bei Erfolg den Bool-Wert true zurück. Ist kein weiteres Token mehr verfügbar wird false zurückgegeben.

token () gibt die Integer-Konstante zurück, die das gerade eingelesene Token repräsentiert.

value () gibt den Wert des gerade eingelesenen Tokens zurück, also eine Zeichenkette im Falle eines Strings oder einen Float-Wert im Falle einer Gleitkommazahl. Der zurückgegebene Wert ist vom Typ “Object“ und muss deshalb für eine Weiterverarbeitung noch entsprechend umgewandelt werden.

Die Parsermethode *yyparse* schließlich parst den Eingabestrom und gibt im allgemeinen einen Syntaxbaum zurück. Der Aufbau des Syntaxbaumes wird durch die in der Grammatik angegebenen Aktionen vom Benutzer gesteuert.

```
public Interface yyInput {
        boolean advance () throws java.io.IOException;
        int token ();
        Object value ();
}
```

Abbildung 3.6: Scanner-Interface von jay

3.2.3 Konflikte

Bei Umsetzung der Grammatik in eine Steuerdatei für jay ist darauf zu achten, dass es beim parsen nicht zu Konflikten kommt. Man unterscheidet dabei zwischen zwei Arten von Konflikten: dem “Shift-Reduce-Konflikt“ und dem “Reduce-Reduce-Konflikt“.

Shift-Reduce-Konflikt

Von einem Shift-Reduce-Konflikt spricht man, wenn nicht klar ist, eine Reduktion vorgenommen werden (*Reduce*) oder erst ein weiteres Token eingelesen werden soll (*Shift*). Dieser Konflikt lässt sich nicht immer vermeiden. Ein klassisches Beispiel hierfür ist das sogenannte “Dangling-Else-Problem“: Bei einem Ausdruck der Form

```
statement: IF condition THEN statement
        | IF condition THEN statement ELSE statement
```

ist nicht klar, ob “statement“ ohne “ELSE“ reduziert werden oder “ELSE“ geshiftet werden soll. Dies ist problematisch, wenn mehrere “IF“-Anweisungen verschachtelt sind, wie im folgenden Fall:

```
if (a < b)
then if (c < d)
     then ...
else ...
```

Das “ELSE“ kann dabei keinem “IF“ eindeutig zugeordnet werden. Jay meldet einen solchen Konflikt und shiftet dabei stets bis zum längstmöglichen Ausdruck. Im obigen Fall würde dadurch das “ELSE“ dem innersten “IF“ zugeordnet werden.

Reduce-Reduce-Konflikt

Beim Reduce-Reduce-Konflikt ist nicht eindeutig, zu welchem Nichtterminal reduziert werden soll. Auch dieser Konflikt lässt nicht immer auf grammatikalischer Ebene lösen. Ein Beispiel dafür ist der Gleichheitsoperator, der in manchen Programmiersprachen gleichermaßen als Vergleichs- und Zuweisungsoperator dient:

```
statement: Variable '='condition
        | Variable '='expression
condition: expression
        | expression '<' expression
```

In diesem Fall stehen zwei Alternativen für die Reduktion von "statement" zur Verfügung. Der Konflikt wird von jay ebenfalls gemeldet. Dabei wird die erste Alternative zur Reduktion benutzt. Dieses Verhalten kann in der weiteren Verarbeitung allerdings zu logischen Fehlern führen.

3.3 Zusammenspiel von JLex und jay

Wie bereits erwähnt ist dient das Interface yyInput als Schnittstelle dem von jay erzeugten Parser und einem Scanner. Möchte man das Scanner-Generator-Tool JLex einsetzen, muss zusätzlich zum erzeugten Scanner das Interface implementiert werden, da der von JLex erzeugte Scanner zunächst einmal nur die Methode "yylex ()" bereitstellt, welche einen Integer-Wert zurückliefert. Dieser Integer-Wert repräsentiert die erkannten Token und muss vorher vom Benutzer definiert werden. In unserem Fall, d.h. bei der Verwendung von jay als Parsergenerator, werden die Token-Nummern für die Token von jay als Integer-Konstanten erzeugt. Das Zusammenspiel zwischen JLex und jay ist in Abbildung 3.7 skizziert.

Die von JLex erzeugte Methode "yylex ()" gibt per Default ein Objekt der Klasse "Yytoken" zurück. Dieses Objekt ist nicht Teil der erzeugten Scannerklasse und muss deshalb vom Anwender implementiert werden. Durch die Direktive "%integer" bzw. "%type" lässt sich der Rückgabewert auf Integer bzw. beliebig setzen. Yylex gibt dann den in den Aktionen durch die *return*-Anweisung deklarierten Wert zurück, also z.B. die Token-Nummer.

Um einen funktionsfähigen Parser inklusive Scanner generieren zu können, der sich zum Beispiel für den Einsatz in einem Compiler eignet, muss also zunächst einmal das Tool jay mit entsprechender Grammatik und definierten Aktionen aufgerufen werden. Dieses erzeugt Token-Nummern für jedes deklarierte Token. Da das Interface yyInput allerdings drei Werte erwartet, nämlich die Angabe, ob ein weiteres Token vorliegt, die Token-Nummer des Tokens und den eigentlichen Token-Wert, muss der Scanner dementsprechend erweitert werden.

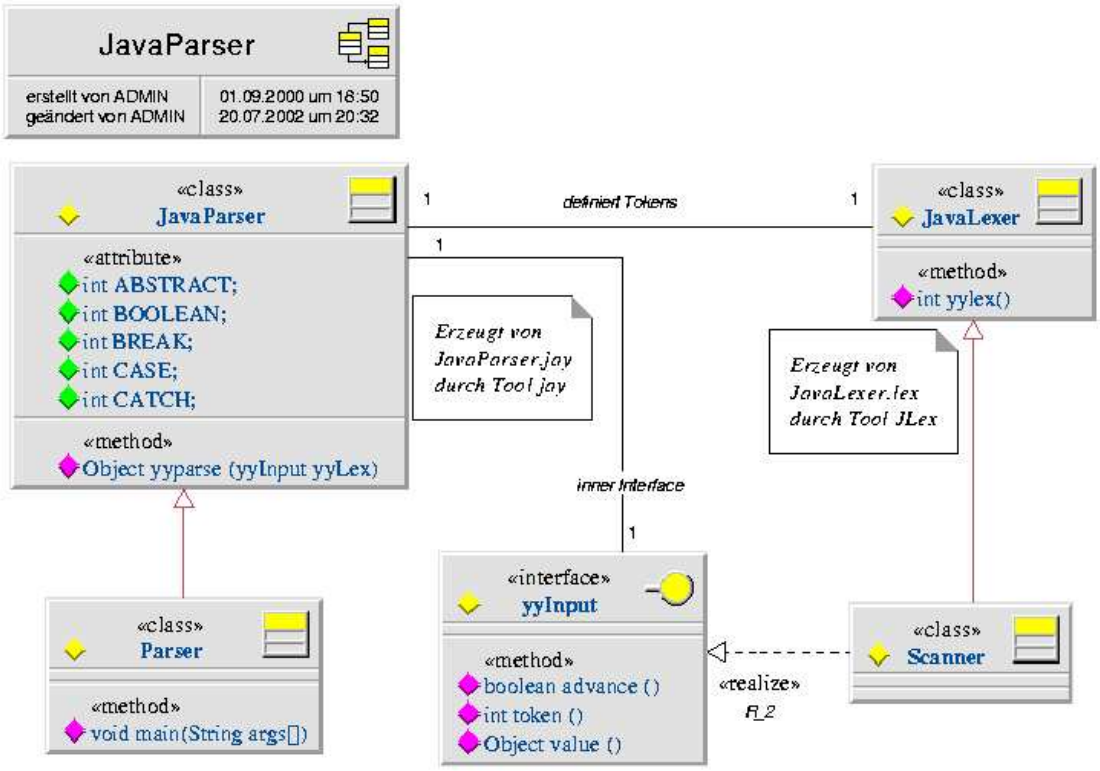

Abbildung 3.7: Zusammenspiel zwischen JLex und Java am Beispiel eines Java-Compilers

```
%{
class Taschenrechner {
%}

%token PLUS
%token MULT
%token <Integer> ID
%token EOF

//PRIORITAETEN EINFUEGEN
//TYPEN VON TOKENS UND NICHTTERMINALEN EINFUEGEN
%left PLUS
%left MULT
%start S

%type <Integer> E
%%
S : E          { System.out.println($1);}
E : E PLUS E { $$ = new Integer ($1.intValue() + $3.intValue());}
  | E MULT E { $$ = new Integer ($1.intValue() * $3.intValue());}
  | ID         { yyDefault($1);}
%%
}
```

Abbildung 3.8: jay-Konfiguration für den Minimal-Taschenrechner

Ein bewährtes Vorgehen ist dabei das Einführen eines Token-Attributes in die Scanner-Klasse. Dieses Token-Attribut ist ein Objekt welches den Wert des Tokens als “Object“ sowie die Token-Nummer als Integer enthält. Als erste Aktion beim erkennen eines Tokens wird ein neues Tokenattribut erzeugt. Dabei wird die Token-Nummer entsprechend gesetzt sowie das Wert-Objekt mit Hilfe der Methode “yytext ()“ erzeugt. Dieses neu erzeugte Token wird dann im Token-Attribut gespeichert. Die Methoden “token ()“ und “value ()“ des Interfaces können so durch Auslesen der des Token-Attributs implementiert werden.

Um die Methode “advance ()“ ebenfalls zu implementieren, genügt es ans Ende jedes Aktionsblocks ein “return true“ zu stellen. “advance ()“ wird so direkt durch “yylex ()“ implementiert. Bei diesem Verfahren muss der Rückgabewert von “yylex ()“ auf Boolean gesetzt werden. Um auch das Ende der Eingabe erkennen zu können, muss weiterhin der EOF-Wert über die Direktive “%eofvalue {}“ auf “false“ gesetzt werden.

In Abbildung 3.8 bzw. 3.9 sind entsprechende Konfiguration am Beispiel eines Minimal-Taschenrechner zu sehen.

```
%%
%{
    Integer value;
%}
%eofval{
    return Taschenrechner.EOF;
%eofval}
ws = [ \t\r\n\b\015]+
%%

"+" { return Taschenrechner.PLUS; }
"*" { return Taschenrechner.MULT; }
[0-9][0-9]*      { this.value = new Integer (yytext());
                   return Taschenrechner.ID; }
{ws} {System.out.print(yytext());}
```

Abbildung 3.9: JLex-Konfiguration für den Minimal-Taschenrechner

4 Das neue Design

Wie anfänglich bereits beschrieben soll das neue Design vor allem zwei Punkte erfüllen:

1. Die Übergabe der Token vom Scanner an den Parser soll objektorientiert erfolgen
2. Der Benutzer soll das Scanner-Interface nicht länger selbst implementieren müssen.

Um diese zwei Forderungen zu erfüllen, wurden sowohl jay als auch JLex modifiziert. Durch die Überarbeitung von jay werden nun keine Integer-Konstanten mehr als Token generiert, sondern Token-Klassen, die Wert und Integerkonstante als Attribut besitzen.

Durch die objektorientierte Übergabe der Token an den Parser konnte das Interface auf die Methode “advance ()“ reduziert werden, die jetzt ein Token-Objekt oder *null* zurückgibt. Token-Nummer und Token-Wert können dem Token-Objekt entnommen werden. Die Methoden “token ()“ und “value ()“ wurden dadurch überflüssig.

Damit der erzeugte Parser wie bisher arbeiten kann, musste die Interpretervorlage “skeleton“ entsprechend angepasst werden, um mit den neuen Tokenobjekten umgehen zu können.

Jlex wurde um zwei Direktiven in der Steuerdatei ergänzt, welche die Erzeugung einer Scannerklasse erwirken, durch die das von jay geforderte Scanner-Interface implementiert wird. Die vorgenommenen Modifikationen sind im Folgenden im Detail beschrieben.

4.1 Modifikationen an jay

4.1.1 Modifikationen am Quellcode

Der Quellcode von jay setzt sich aus 12 Teilen zusammen:

main.c Das Hauptprogramm, welches den Ablauf steuert

defs.h Eine Headerdatei, in der hauptsächlich Konstanten enthalten sind, aber auch Definitionen benötigter Strukturen

reader.c Stellt Funktionen zum Einlesen der Konfigurationsdaten bereit

symtab.c Enthält Strukturen und Funktionen zur Erstellung und Manipulation der Symboltabelle

lr0.c Enthält Funktionen für das LR0-Parsen

lalr.c Enthält Funktionen zur Lookahead-Implementierung

mkpar.c Enthält Funktionen zur Generierung des Parsers

verbose.c Funktionen zur Erstellung einer von Menschen lesbaren Ausgabe (Ablaufverfolgung). Die Ausgabe erolgt in eine Datei

output.c Funktionen zur Ausgabe des erzeugten Parsers in eine Datei

closure.c Enthält Funktionen zur Hüllengenerierung für den Endlichen Automaten

warshall.c Implementierung des "Warshall-Algorithmus" zur Optimierung des Endlichen Automaten

error.c Funktionen zur Fehlerbehandlung

Für die Erzeugung von Token-Klassen genügte es, die Datei "output.c" zu modifizieren. Output.c ist verantwortlich für die Ausgabe des Parsercodes und wird von der Interpreter-Vorlage "skeleton" gesteuert. Die Steuerung geschieht über das parsen bestimmter Schlüsselwörter in skeleton. Wird ein solches Schlüsselwort gefunden, bewirkt dies den Aufruf einer entsprechenden Funktion in output.c. Im skeleton können folgende Schlüsselworte für die Steuerung verwendet werden. Dabei ist die Reihenfolge einzuhalten, da die einzelnen Ausgaben stets an die Ausgabedatei angehängt werden und sonst unter Umständen sinnloser Code entsteht:

tokens Aufruf von output_defines(strtok(NULL, "\r\n")). Die dem Schlüsselwort "tokens" folgenden Zeichen werden eingelesen und als Modifikatoren bzw. Typ der Token-Nummern-Konstanten verwendet. Dann werden die Konstanten ausgegeben. Ursprünglich waren die Konstanten vom Typ "public static final int".

prolog Aufruf von output_stored_text(prolog_file, prolog_file_name). Der Programmcode aus dem *prolog*-Teil der Konfigurationsdatei, der bereits in eine temporäre Datei abgespeichert wurde, wird in die Ausgabedatei ausgegeben.

debug Aufruf von output_debug (). Erweitert die Ausgabe um Debugging-Informationen.

local Aufruf von output_stored_text(local_file, local_file_name). Der Programmcode, der im *local*-Teil der Konfigurationsdatei angegeben und ebenfalls in eine temporäre Datei gespeichert wurde, wird in die Ausgabedatei kopiert.

actions Aufruf von output_semantic_actions (). Die Aktionen, die vom Benutzer im *local*-Teil beim Auftreten einer grammatikalischen Regel festgelegt wurden, werden durch diesen Aufruf in Form eines "Switch-Case"-Blocks in die Ausgabedatei ausgegeben.

```
class yyTokenclass {
   public int tokennr;
   public Object value;

   yyTokenclass () {
      this.tokennr=-1;
   }
   yyTokenclass (Object o) {
      this.value = o;
   }
}
```

```
final class ID extends yyTokenclass {
   ID(Object o) {
      super(o);
      this.tokennr = 259;
}
```

Abbildung 4.1: Tokensuperklasse yyTokenclass, Tokenklasse

tables Aufruf von output_rule_data (), output_yydefred (), output_actions (). Bewirkt die Ausgabe von Tabellen, die für den Endlichen Automaten des Parsers benötigt werden.

epilog Aufruf von output_trailing_text (). Der Programmcode, der im *epilog*-Teil der Konfigurationsdatei angegeben wurde, wird in die Ausgabedatei kopiert.

Um statt Integer-Konstanten Token-Objekte zu erzeugen, wurde die Funktion “output_defines (char *prefix)“ entsprechend geändert. Die Funktion besteht im wesentlichen aus einer Schleife, welche alle deklarierten Token abarbeitet. Die Anzahl der Token wurde bereits im Programmteil “reader.c“ in die Variable “ntokens“ gespeichert, ebenso wie die Namen und Nummern der Token, die in den Arrays “symbol_name[]“ bzw. “symbol_value[]“ gespeichert wurden. Für jedes Token in “symbol_name[]“ wird nun der Präfix “prefix“, also in der Ursprungsversion “public static final int“ ausgegeben, gefolgt vom Namen des Tokens, ‘=‘ und seiner Nummer.

An Stelle dieser Ausgabe wurde nun die Ausgabe einer Tokenklasse mit dem Namen des Tokens gesetzt. Diese Klasse erbt die Attribute “int tokennr“ und “Object value“ von einer Superklasse namens “yyTokenclass“, wie in Abbildung 4.1 zu sehen. Der Konstruktor wird so ausgegeben, dass “tokennr“ beim Anlegen des Objekts auf die entsprechende Token-Nummer gesetzt wird. Die Superklasse “yyTokenclass“ ist in “skeleton“ definiert.

4.1.2 Modifikationen an “skeleton“

Die Datei enthält “skeleton“ einerseits den Code für den Interpreter und dient weiterhin zur Steuerung der Ausgabe. Sie ist in folgender Syntax notiert:

- Zeilen, die mit einem Punkt (‘.‘) beginnen, werden direkt in die Ausgabedatei kopiert.
- Zeilen, die mit einem ‘t‘ beginnen, werden in die Ausgabedatei kopiert, wenn das Debugging eingeschaltet wurde.

- Zeilen, die mit einer Raute (‘#‘) beginnen, sind Kommentare.

Beginnt eine Zeile ohne eines der genannten Zeichen, so wird der Inhalt der Zeile als Aufruf einer der oben genannten Funktionen von “output.c“ interpretiert.

Die Erzeugung der Token-Klassen wurde wie bereits erläutert durch die Änderungen an “output.c“ erreicht. Der Code für Superklasse “yyTokenclass“ wurde einfach am Ende der skeleton-Datei eingefügt, so dass die Klassendeklaration am Ende der erzeugten Ausgabedatei auftaucht. “yyTokenclass“ enthält zwei Konstruktoren:

yyTokenclass (Object o) Erzeugt ein Tokenobjekt und setzt das “value“-Attribut auf das übergebene Objekt o. Dieser Konstruktor wird nur vom Konstruktor einer Token-Klasse aufgerufen.

yyTokenclass () Dieser Konstruktor erzeugt ein “leeres“ Tokenobjekt mit negativer Tokennummer. Dieses Objekt wird vom Interpreter benötigt, der ursprünglich bei negativer Token-Nummer das nächste Token eingelesen hatte.

Der Code für den Interpreter ist ursprünglich für den Umgang mit Integer-Werten entwickelt worden. Der erzeugte Endliche Automat arbeitet im neuen Design intern weiterhin mit Integer-Werten. Allerdings musste der Interpreter-Code angepasst werden, um die Verwendung der Tokenobjekte zu ermöglichen.

Zunächst einmal wurden die Methoden “int token ()“ und “Object value ()“ des Interfaces auskommentiert. Sie sind im neuen Design nicht mehr notwendig.

Im Original-skeleton wurde vom Interpreter eine Token-Nummer in die Integer-Variable yyToken eingelesen. Beim Einlesen dieser Variable wurde geprüft, ob der Aufruf von “advance ()“ *true* zurückliefert, ob also ein weiteres Token eingelesen werden konnte. War dies der Fall, wurde yyToken der Rückgabewert der Funktion “token ()“ zugewiesen.

Der Typ von “yyToken“ wurde im neuen Design auf “yyTokenclass“ geändert. Dadurch ist es nicht mehr notwendig, zu prüfen, ob ein weiteres Token vorliegt. Es genügt, die Methode “advance ()“ aufzurufen, die jetzt beim vorliegen eines Tokens das entsprechende Token-Objekt bzw. beim nicht vorhanden sein den Wert *null* zurückgibt. Der Rückgabewert wird yyToken zugewiesen. Alle Aufrufe von “token ()“ und “value ()“ mussten durch Zuweisung von “yyToken.tokennr“ bzw. “yyToken.value“ ersetzt werden. Abbildung 4.2 zeigt einen Ausschnitt aus der neuen skeleton-Datei.

Da der FA des Parsers auch für das Erreichen des Dateiendes ein Token mit der Nummer 0 erwartet, wurde ein weiteres Token “EOF“ angelegt, welches erzeugt wird, wenn “advance ()“ den Wert *null* zurückliefert. Alternativ besteht die Möglichkeit, als EOF-Rückgabewert im Scanner ein EOF-Objekt zu benutzen.

```
/* modified by diho, 14.04.2004
   if "empty token", get next token */
if (yyToken.tokennr <0) {
/* modified by diho, 10.04.2004
call yyLex.advance in any case, check if yyToken is EOF */
   yyToken = yyLex.advance(); /*? yyLex.token() : 0;*/
/* modified by diho, 27.04.2004
if "null", create EOF token */
if (yyToken == null) {yyToken = new EOF();}
                    .
                    .
                    .
/* modified by diho, 14.04.2004
   replaced yyLex.value() by yyToken.value*/
yyVal = yyToken.value;
/* modified by diho, 14.04.2004
   orig:  yyToken = -1 */
yyToken = new yyTokenclass();
```

Abbildung 4.2: Ausschnitt aus der skeleton-Datei

4.2 Modifikationen an JLex

Um dem Benutzer die Implementierung des Scanner-Interfaces von jay abzunehmen, sollte das Interface für den Benutzer transparent implementiert werden. Dieses Ziel wurde dadurch erreicht, das JLex jetzt in der Lage ist, einen Scanner zu erzeugen, der das Interface automatisch implementiert. Dies wird durch zwei hinzugefügte Direktiven erreicht:

%jayscanner: Der Text nach dieser Direktive wird als Name des zu generierenden Scanners interpretiert. Ausserdem wird bei Verwendung der Direktive ein Flag gesetzt. Das Flag wird bei der Ausgabe des Scannercodes abgefragt. Ist das Flag nicht gesetzt, verhält sich der erzeugte Scanner wie bei der Originalversion von JLex.

%jayparser: Der Text nach dieser Direktive wird als Name des Parsers interpretiert, dessen Interface implementiert werden soll. Diese Direktive hat keinen Effekt, wenn sie nicht zusammen mit der "%jayscanner"-Direktive verwendet wird.

Zunächst einmal wurde der Default-Rückgabetyp der Scanner-Funktion "yylex ()" von "yyToken" auf "yyTokenclass" geändert. Dies ist eine reine Namensänderung, die der besseren Verständlichkeit dienen soll. Da der Typ "yyToken" vom Benutzer implementiert werden muss, hätte man das jay-skeleton auch diesen erzeugen lassen können. Der Name des Typs ist im Quellcode von JLex wie alle Namen in einem Char-Array gespeichert.

Zusätzlich wurden einige weitere Variablen eingeführt:

```
class TRScanner extends Yylex implements TRParser.yyInput {
        public TRScanner (java.io.Reader reader) {
                super (reader);
        }
        //public int token() is not longer necessary

        public yyTokenclass advance() throws java.io.IOException {
                yyTokenclass ret = yylex();
                return ret;
        }

        //public Object value () is not longer necessary

}
```

Abbildung 4.3: von JLex erzeugter Scanner

private boolean jayparser Das Flag, das die Verwendung der "%jayscanner"-Direktive anzeigt. Default ist *false*.

char m_scanner_name [] Der Name des zu generierenden Scanners. Default ist "Scanner".

char m_parser_name [] Der Name des Parsers, dessen Interface implementiert werden soll. Default ist "Parser".

Die Erzeugung der Ausgabedatei übernimmt bei JLex eine Methode namens "generate ()". Ähnlich wie auch schon bei der Erzeugung des Parsers ruft "generate ()" nacheinander einige Methoden auf, die wiederum bestimmte Codefragmente erzeugen, welche von den Einträgen in der Konfigurationsdatei abhängen. Hier wurde eine weitere Methode "implement_interface ()" hinzugefügt, die aufgerufen wird, wenn das Flag "jayparser" gesetzt ist.

"Implement_interface ()" gibt den Code für die eine Klasse aus, deren Namen durch "char m_scanner_name []" gegeben ist und das Interface yyInput eines Parsers mit dem durch "char m_parser_name []" gegebenen Namen implementiert. Der Konstruktor erwartet ein Objekt vom Typ java.io.Reader. Als Methode ist einzig "advance ()" implementiert. Die Methode ruft "yylex ()" auf und gibt das erhaltene Objekt vom Typ "yyTokenclass" zurück. Der erzeugte Scanner ist in Abbildung 4.3 zu sehen.

4.3 Zusammenspiel von JLex und jay

Betrachtet man nur die Erstellung der Konfigurationsdateien, so hat sich beim Zusammenspiel zwischen JLex und jay im Vergleich zu den Originalversionen nicht viel geändert.

```
%%
%jayscanner TRScanner
%jayparser TRParser
%eofval{
    return null;
%eofval}
ws = [ \t\r\n\b\015]+
%%

"+" { return new PLUS(yytext()); }
"*" { return new MULT(yytext()); }
[0-9][0-9]*      { return new ID(new Integer (yytext())); }
{ws} {System.out.print(yytext());}
```

Abbildung 4.4: neue JLex-Konfiguration für den Minimal-Taschenrechner

Lediglich bei der JLex-Konfigurationsdatei müssen nun über die Direktiven “%jayscanner“ und “%jayparser“ die Namen des Scanners und des Parsers angegeben werden.

Bei der Definition der Aktionen entfällt die Rückgabe des *true*-Wertes, da die Interface-Methode “advance ()“ jetzt direkt das Token-Objekt zurückgibt. Als EOF-Wert muss dazu allerdings über die Direktive “%eofvalue“ im *prolog*-Teil der Wert *null* eingestellt werden, da ein *null*-Objekt vom Parser als Ende der Eingabe interpretiert wird.

Das Übergeben von Token an der Parser geschieht nun nicht mehr über Integer-Konstanten und setzen einer Internen Variable (siehe Abbildung 3.9) sondern durch direkte Rückgabe eines Token-Objekts wie in Abbildung 4.4. Die Konfiguration für jay ist dabei unverändert wie bereits in Abbildung 3.8

Die Klassen für die Token-Objekte werden von jay im Vorfeld erzeugt. Um die Komplexität gering zu halten, wurden die Klassen ohne Package-Deklaration erzeugt. Sie müssen deshalb im gleichen Verzeichnis wie die Quelldatei des Scanners liegen.

Da der Scanner im neuen Design von JLex erzeugt wird aber jay für die Erzeugung der Token-Klassen zuerst ausgeführt werden muss, lässt er sich nicht innerhalb der jay-Konfigurationsdatei verwenden. Um Scanner und Parser benutzen zu können, muss also der Benutzer eine Hauptklasse von Hand anlegen, die Scanner und Parser miteinander verbinden. Dieses Verfahren ist durch das “Henne und Ei“-Problem bei der Erzeugung von Parser und Scanner leider unumgänglich, ist aber schnell und einfach zu bewerkstelligen. Ein Beispiel für eine Hauptklasse ist in Abbildung 4.5 zu sehen.

```
public class TRmain {
    public static void main (String [] args) {
        TRScanner scanner = new TRScanner (new java.io.InputStreamReader (System.in));
        TRParser parser = new TRParser() ;
        try {
            parser.yyparse(scanner);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Abbildung 4.5: Hauptklasse für den Minimal-Taschenrechner

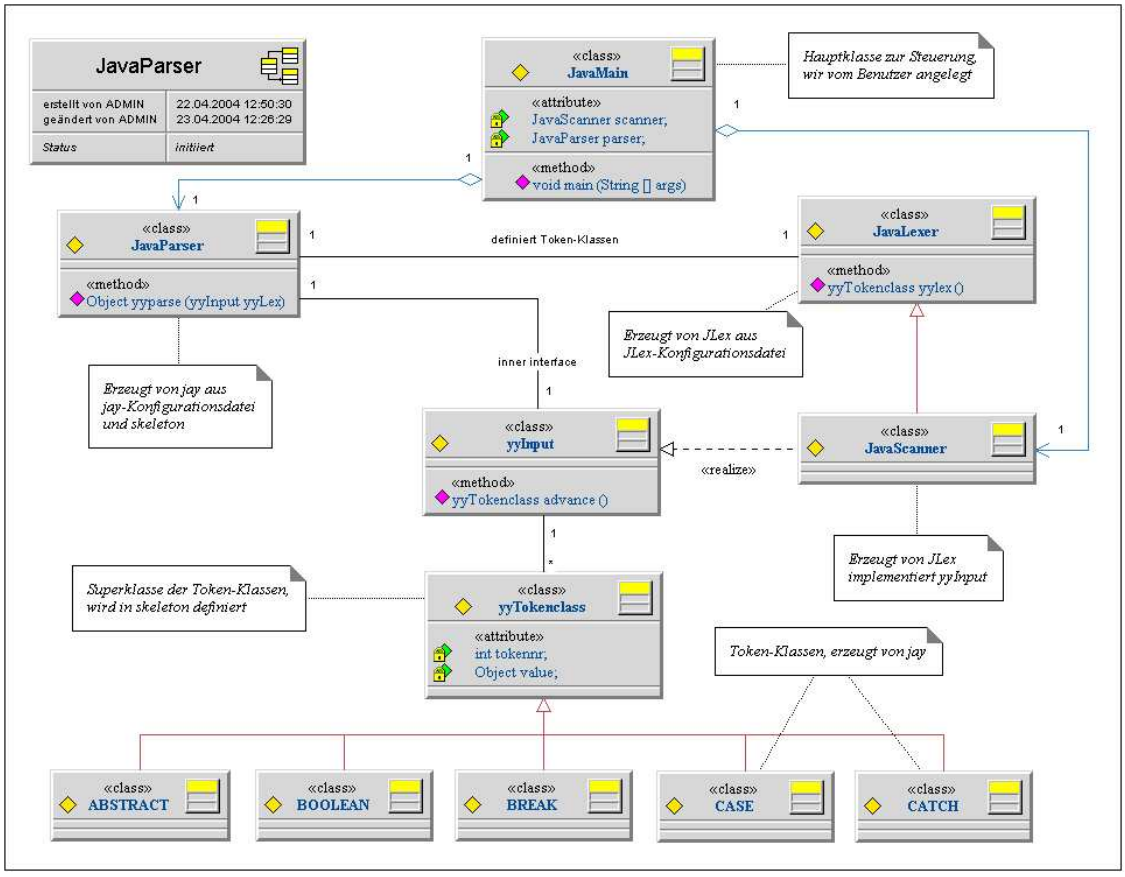

Abbildung 4.6: Zusammenspiel von JLex und jay im neuen Design

5 Resumee

Das neue Design, wie es in Abbildung 4.6 dargestellt ist, bietet die Möglichkeit einer objektorientierten Token-Übergabe eines Scanners an einen mit jay erzeugten Parser. Bei der Verwendung des Scanner-Generators JLex entfällt darüber hinaus das Implementieren des in jay verwendeten Interfaces.

Die Aufgabenstellung wurde somit vollständig erfüllt.

Die im letzten Kapitel beschriebene Vorgehensweise zur Erzeugung eines Parsers funktioniert bisher jedoch nur, wenn sich die Quelldateien von Scanner und Parser im gleichen Verzeichnis befinden.

Sollen Packages verwendet werden, ist es notwendig, die Erzeugung der Token-Klassen sowie ihrer Superklasse im skeleton entsprechend abzuändern. Die Einbindung eines Package-Konzepts, welches über Direktiven im jay-Steuerfiles konfiguriert werden kann, steht noch aus.

Weiterhin wurden die Attribute der Token-Klassen durchgängig als *public* deklariert. Sie ermöglichen somit direkten Zugriff von beliebigen anderen Klassen aus. Diese Eigenschaft vereinfacht zwar den Umgang mit den erzeugten Klassen und ist im Compilerbau eher unbedeutend. Für den Einsatz in Programmen, bei der Zugriffsrechte eine größere Rolle spielen, ist sie aber möglicherweise unerwünscht.

Für eine Weiterentwicklung des Designs wäre eine Neukonzeption der Kapselung eine interessante Aufgabe.

A Literatur

Skript zur Vorlesung “Compilerbau“, Prof. Dr. R Schiedermeier, FB07 Informatik/Mathematik, FH München

Skript zur Vorlesung “Informatik III“, Prof. Dr. Martin Plümicke, FB Technische Informatik, BA Horb

Skript zu Vorlesung “Compilerbau mit Java“, Prof. Dr. Axel Tobias Schreiner, FB Informatik/Mathematik, Universität Osnabrück

“jay - Ein yacc für Java“, Prof. Dr. Axel T. Schreiner und Dr. Bernd Kühl, Fachbereich Mathematik/Informatik, Universität Osnabrück

Folien zur Vorlesung “Compilerbau und formale Sprachen“, Prof. Dr. Uwe Schmidt, FB Informatik, FH Wedel

“JLex: A lexical analyzer generator for Java“, Elliot Berk, Department of Computer Science, Princeton University

Skript zur Vorlesung “Compilerbau“, U. Grude, C. Knabe, A. Solymosi, FB VI Informatik, TFH Berlin

B Inhalt der CD

Die CD enthält den Quellcode der Studienarbeit sowie die angegebene Beispielkonfiguration:

- **jay:** Der modifizierte Quellcode des Parsergenerators jay
- **JLex:** Der modifizierte Quellcode des Scannergenerators JLex
- **finaltest:** Das Taschenrechnerbeispiel. Generieren des Beispiels mit dem Shell-Skript *generate.sh*.
- **compilertest:** Test mit dem im 4. Semester angefertigten Java-Compiler. Generieren der Dateien mit "make", für Tests siehe README-Datei.
  Anm.: Die Methoden *Semantikcheck* und *Codegenerierung* wurden deaktiviert.
- **Studienarbeit:** Quellen und Postscript-File der Studienarbeit.