Interner Bericht

Terminologie des Übersetzerbaus

Peter Knauber
Stefan Vorwieger
Reinhard Eppler

255 / 94

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-67653 Kaiserslautern

Terminologie des Übersetzerbaus

Peter Knauber
Stefan Vorwieger
Reinhard Eppler

255 / 94

Herausgeber: **AG Programmiersprachen und Compilerbau**
Leiter: **Prof. Dr. Ing. H.-W. Wippermann**

Kaiserslautern, Oktober 1994

Inhalt

1. Begriffsdefinitionen und Abgrenzungen 1

1.1 Theoretische Grundlagen zu formalen Sprachen 1

1.2 Definitionen für den Compilerbau 6

1.2.1 Scanner 6

1.2.2 Parser 8

1.3 Literaturvergleich 10

2. Produktionen regulärer Grammatiken 15

2.1 Einfache Scanner-Regeln (SR) 15

2.2 Erweiterte Scanner-Regeln (ESR) 16

2.3 Der Lookahead-Operator 19

2.4 Die Ergebnisform 20

2.5 Kommentar-Regeln 21

3. Produktionen kontext-freier Grammatiken 22

3.1 Elemente der Backus-Naur-Form (BNF) 22

3.1.1 Ursprüngliche BNF 22

3.1.2 Aktuelle BNF 22

3.2 Erweiterte Backus-Naur-Form (EBNF) 23

3.3 Zuordnung eines Lexems zu einem Tokentag 26

Anhang A: EBNF-Syntax 28

Anhang B: Notation von Grammatiken 31

Repräsentation im FrameMaker 31

Regeln zur Formatumsetzung 35

Objektorientierte Darstellung 37

Anhang C: Literatur 38

1 Begriffsdefinitionen und Abgrenzungen

In diesem Kapitel sollen die wichtigsten theoretischen Begriffe im Zusammenhang mit formalen Sprachen, die als Grundlage für den Bau von Übersetzern dienen, definiert und erläutert werden. Dabei werden für diese Begriffe Definitionen angegeben und etwaige Abweichungen von der gängigen Literatur erläutert.

Kapitel 1.1 beschäftigt sich mit den allgemeinen Definitionen für formale Sprachen, Kapitel 1.2 geht auf die Verwendung dieser Begriffe für den speziellen Bereich des Übersetzerbaus ein und definiert weitere Begriffe speziell für diesen Bereich. Kapitel 1.3 vergleicht die eingeführten Definitionen mit gängigen Verwendungen in der Literatur.

1.1 Theoretische Grundlagen zu formalen Sprachen

Um ein einheitliches Begriffsfundament zu schaffen, ist es zunächst notwendig, ein ausreichend abgeschlossenes theoretisches Fundament zu definieren, auf dem speziellere Begriffe für den Compilerbau aufsetzen können.

Wir beginnen zunächst mit der Definition der kleinsten Einheit, dem Atom, einer formalen Sprache:

Definition 1 *Vokabular, Symbol und Symbolfolge*

Eine endliche, nichtleere Menge $\mathcal{V}$ von Symbolen heißt Vokabular.
Eine beliebig lange, aber endliche Folge von Symbolen wird Symbolfolge genannt.
Eine Symbolfolge der Länge n ist eine Abbildung $\{1, \ldots, n\} \rightarrow \mathcal{V}$ und kann folgendermaßen notiert werden:

$\omega_1\omega_2\omega_3\ldots\omega_n$, mit $\omega_i \in \mathcal{V}$.

Die Menge aller Symbolfolgen über einem Vokabular $\mathcal{V}$ der Länge n ist

$\mathcal{V}^n = \{\omega \mid \omega\colon \{1, \ldots, n\} \rightarrow \mathcal{V}\}$

Eine leere Symbolfolge, d. h. eine Folge der Länge Null, wird mit ε bezeichnet. Sie ist einziges Element der Menge $\mathcal{V}^0$, d. h. $\mathcal{V}^0 = \{\varepsilon\}$.

Die Menge aller Symbolfolgen über einem Vokabular $\mathcal{V}$ ist der Kleene-Abschluß von $\mathcal{V}$, d. h.

$$\mathcal{V}^* = \bigcup_{n \geq 0} \mathcal{V}^n$$

Seien ω eine Symbolfolge der Länge n und ξ eine Symbolfolge der Länge m, d. h. $\omega = \omega_1\omega_2\ldots\omega_n$ und $\xi = \xi_1\xi_2\ldots\xi_m$, so ist $\omega\xi$ eine Symbolfolge der Länge $n+m$ mit $\omega\xi = \omega_1\omega_2\ldots\omega_n\xi_1\xi_2\ldots\xi_m$. ❑

Definition 2 *Präfix und Rest einer Symbolfolge*

Gegeben sei ein Vokabular $\mathcal{V}$und eine Symbolfolge $\omega = \xi\psi$ mit $\xi, \psi \in \mathcal{V}^*$. Es ist

ξ ein *Präfix* und ψ der *Rest* von ω, wenn $\xi \neq \varepsilon$ gilt. ❑

Man beachte hierbei, daß das Präfix eine Symbolfolge der Länge ≥ 1 ist. Dagegen ist erlaubt, daß der Rest einer Symbol gleich ε ist.

Der Übergang zu einer formale Sprache geschieht zunächst völlig abstrakt. Es wird nicht festgelegt, wie eine Sprache beschrieben werden kann.

Definition 3 *Formale Sprache über einem Vokabular*

Gegeben sei ein Vokabular $\mathcal{V}$. Eine Teilmenge von $\mathcal{V}^*$ heißt *formale Sprache* $\mathcal{L}$ über $\mathcal{V}$, d. h. $\mathcal{L} \subseteq \mathcal{V}^*$. ❑

Ebenso abstrakt wird der Satz einer Sprache definiert:

Definition 4 *Sätze einer formalen Sprache*

Gegeben sei ein Vokabular $\mathcal{V}$. Die Elemente σ einer formalen Sprache $\mathcal{L}$ über $\mathcal{V}$ heißen *Sätze*, d. h. $\sigma \in \mathcal{L}$. Sätze sind also die gültigen Symbolfolgen aus $\mathcal{V}^*$ und müssen immer bezüglich einer Sprache betrachtet werden, d. h.

$$\mathcal{L} \subseteq \mathcal{V}^* \Rightarrow \sigma \in \mathcal{L} \wedge \sigma \notin \mathcal{V}^*\text{-}\mathcal{L}.$$ ❑

An dieser Stelle soll betont werden, daß man nicht von einem bestimmten Vokabular zu einer Sprache sprechen kann, vielmehr gibt es unendlich viele Vokabulare über eine Sprache, nämlich jede echte Erweiterung des anfangs gegebenen Vokabulars $\mathcal{V}$. Es gilt:

Sind $\mathcal{V}$ und $\mathcal{W}$ Vokabulare: $\mathcal{L} \subseteq \mathcal{V}^* \Rightarrow \mathcal{L} \subseteq (\mathcal{V} \cup \mathcal{W})^*$ mit $\mathcal{V} \cap \mathcal{W} = \emptyset$.

Eine Sprache setzt sich aus einer syntaktischen und semantischen Beschreibung zusammen. Die Intention der nachfolgenden Seiten ist die nähere Betrachtung der Syntax.

Die einfachste Form einer syntaktischen Beschreibung ist eine Menge, die alle Sätze enthält. Auf diese Weise sind aber nur sehr einfache Sprachen erklärt. Die meisten Sprachen bestehen aus einer unendlichen Anzahl von Sätzen, so daß zu einer formelleren Beschreibung gegriffen werden muß. Diese besteht zunächst aus einem Vokabular, das die Basis für die Sätze einer Sprache bildet. Weiterhin benötigt man Vorschriften, wie die Struktur eines Satzes aufgebaut sein bzw. werden kann.

Der Grund für eine (formale wie auch natürliche) Sprache ist die Kommunikation (Mensch zu Mensch, Mensch zu Maschine), die sich in zwei Bereiche aufteilen läßt: Das Sprechen und das Verstehen.

Aus syntaktischer Sicht ist das Sprechen einer Sprache nichts anderes als das Erzeugen von Sätzen, deren Bedeutung irrelevant ist. Man spricht auch von einem *erzeugenden System*, wenn ein Algorithmus aufgrund einer syntaktischen Beschreibung alle Sätze einer Sprache aufzählen kann.

Das Verstehen eines Satzes einer Sprache reduziert sich auf das Analysieren der Struktur einer Symbolfolge, wenn man sich auf den syntaktischen Bereich beschränkt. Hat man einen totalen Algorithmus, der aufgrund der Beschreibung einer Sprache in endlicher Zeit entscheiden kann, ob eine gegebene Symbolfolge ein Satz dieser Sprache ist, so spricht man von einem *analysierenden* oder *akzeptierenden System*.

Es ist klar, daß die Art der Sprachbeschreibung an die Art des Umgangs mit der Sprache (erzeugen, akzeptieren) angepaßt ist, um den jeweiligen Algorithmus möglichst einfach zu halten. Man beachte, daß in beiden Fällen die beschriebene Sprache dieselbe sein muß.

Im Bereich von Programmiersprachen verwendet man im ersten Fall Grammatiken, im zweiten Fall verbirgt sich die Struktur in einer Tabelle. Ein Algorithmus zur Analyse von Programmiersprachen ist ein Automat, der auf dieser Tabelle arbeitet. Polak: "In some sense the interpreter of the parsing table defines a semantics for parsing tables and the tables themselves could be considered the formal definition of the syntax." Grundlage für den Algorithmus eines erzeugenden Systems ist eine Ableitungsrelation über ein Semi-Thue-System, die im folgenden definiert wird.

Definition 5 *Semi-Thue-System, Ableitungsrelation, Ableitungsfolge*

Ein *Semi-Thue-System* (*STS*) ist ein Paar $\mathcal{STS} = (\mathcal{V}, \mathcal{P})$, bestehend aus einem Vokabular $\mathcal{V}$ und einer endlichen Menge von Zwei-Tupeln $\mathcal{P} \subseteq \mathcal{V}^* \times \mathcal{V}^*$.

Eine *Ableitungsrelation* $\underset{\mathcal{STS}}{\Rightarrow}$ (bezüglich $\mathcal{STS}$) über $\mathcal{V}^*$ ist eine zweistellige, transitive Relation, mit

$\xi \underset{\mathcal{STS}}{\Rightarrow} \psi$, wenn es $\omega, \omega' \in \mathcal{V}^*$ und $(\upsilon_1, \upsilon_2) \in \mathcal{P}$ gibt mit

$\xi = \omega\upsilon_1\omega'$ und $\psi = \omega\upsilon_2\omega'$.

Falls der Bezug der Ableitungsrelation zum $\mathcal{STS}$ klar ist, schreiben wir $\Rightarrow$ statt $\underset{\mathcal{STS}}{\Rightarrow}$.

Abkürzend für eine mehrfach angewandte Ableitungsrelation schreiben wir

$\xi \overset{n}{\Rightarrow} \psi$, wenn es $\omega_0, \omega_1, \ldots, \omega_n \in \mathcal{V}^*$ gibt mit

$\xi = \omega_0$, $\psi = \omega_n$ und $\omega_i \Rightarrow \omega_{i+1}$ für $0 \le i < n$,

und

$\xi \overset{*}{\Rightarrow} \psi$, wenn es ein $n \ge 0$ gibt mit $\xi \overset{n}{\Rightarrow} \psi$.

Wir bezeichnen $\omega_0, \omega_1, \ldots, \omega_n$ als *Ableitungsfolge*. ❑

Wir benötigen weitere Einschränkungen, um eine Sprache generieren zu können. Hierzu führen wir den Begriff der Grammatik ein, die uns durch Angabe einer endlichen Zahl von Regeln ermöglicht, eine Sprache mit unendlich vielen Sätzen zu beschreiben.

Definition 6 *Grammatik*

Eine *Grammatik* $\mathcal{G}$ ist ein Quadrupel $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$, wobei

$\mathcal{STS} = (\mathcal{V}, \mathcal{P})$ ein STS ist. Wir bezeichnen $\mathcal{STS}$ das zur Grammatik $\mathcal{G}$ gehörige STS. Es ist daher $\underset{\mathcal{G}}{\Rightarrow}$ über $\mathcal{STS}$ als $\underset{\mathcal{STS}}{\Rightarrow}$ definiert.

Es gilt $\mathcal{T} \subset \mathcal{V}$ mit $\mathcal{T}$ nicht leer und wir setzen

$\mathcal{N} = \mathcal{V} - \mathcal{T}$.

$\mathcal{N}$ und $\mathcal{T}$ bezeichnen somit disjunkte Vokabulare, d. h.

$\mathcal{N} \cap \mathcal{T} = \varnothing$,

wobei die Elemente aus $\mathcal{N}$ Nonterminale und die Elemente aus $\mathcal{T}$ Terminale genannt werden.

$\mathcal{P}$ ist (wie bei dem STS) eine endliche Menge von Zwei-Tupeln, wobei hier die syntaktische Struktur etwas eingeschränkt wird:

$\mathcal{P} \subseteq \mathcal{V}^{*\prime} \times \mathcal{V}^*$ mit $\mathcal{V}^{*\prime} = \mathcal{V}^* \mathcal{N} \mathcal{V}^*$.

Wir schreiben $\xi \rightarrow \psi$ in $\mathcal{P}$ statt $(\xi, \psi) \in \mathcal{P}$. Elemente aus $\mathcal{P}$ werden Produktionen (Regeln) genannt. Wir sprechen von der linken Seite ξ und der rechten Seite ψ einer Regel. In einer Grammatik enthalten demnach alle linken Seiten einer Produktion mindestens ein Nonterminal.

S ist ein ausgezeichnetes Symbol, das sog. Startsymbol, aus $\mathcal{N}$, d. h. $S \in \mathcal{N}$. Für die Ableitungsfolge $\underset{\mathcal{G}}{\Rightarrow}$ bedeutet dies, daß alle Ableitungsfolgen mit S beginnen müssen. ❑

Die Tupel der Ableitungsrelation sind ebenso wie die Sätze einer formalen Sprache i. a. nicht endlich; eine Aufzählung zur Spezifizierung ist also nicht möglich. Daher bedient man sich zur Spezifizierung einer Menge von Zwei-Tupeln über $\mathcal{V}^*$, die oben als Produktionen eingeführt wurden. Im weiteren beschränken wir uns auf formale Sprachen, die durch eine endliche Menge dieser Produktionen beschrieben werden können.

Definition 7 *Definition einer Sprache durch eine Grammatik*

Gegeben sei eine Grammatik $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$. Die von $\mathcal{G}$ beschriebenen formalen Sprache $\mathcal{L}$ wird erzeugt über den transitiven Abschluß von $\mathcal{V}^*$ unter $\underset{\mathcal{G}}{\Rightarrow}$, d. h.

$$\mathcal{L}(\mathcal{G}) = \{ \sigma \mid S \overset{+}{\underset{\mathcal{G}}{\Rightarrow}} \sigma \wedge \sigma \in \mathcal{T}^* \} .$$

Eine *Grammatik* $\mathcal{G}$ ist damit ein erzeugendes System für eine formale Sprache. Wir sagen auch, eine Grammatik *definiert* eine formale Sprache. ❑

Man beachte, daß die Sätze einer formalen Sprache nur aus den Terminalen gebildet werden. Mit der Bedingung, daß die linke Seite einer Produktion immer ein Nonterminal enthält, stehen die Sätze der erzeugten Sprache immer am Ende einer Ableitungsfolge.

Die Ableitungsrelation und damit die formale Sprache hängt wie oben dargelegt von der Form (Syntax) der Produktionen ab. N. Chomsky hat 1956 eine Einteilung von Grammatiken vorgeschlagen, die heute noch Gültigkeit hat:

Definition 8 *Klassifizierung von Grammatiken und Sprachen (Chomsky-Hierarchie)*

Eine Grammatik $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ mit $\mathcal{N} = \mathcal{V} - \mathcal{T}$ heißt

- vom *Typ 0*, falls

  jede Produktion die Form

  $\xi \rightarrow \psi$ mit $\xi \in \mathcal{V}^+$, $\psi \in \mathcal{V}^*$ hat,

- vom *Typ 1* (*kontext-sensitiv*), falls

  jede Produktion die Form

  $S \rightarrow \varepsilon$ oder
  $\xi A \psi \rightarrow \xi \nu \psi$ mit $\xi, \psi \in \mathcal{V}^*$, $A \in \mathcal{N}$, $\nu \in \mathcal{V}^+$ hat,

- vom *Typ 2* (*kontext-frei*), falls

  jede Produktion die Form

  $$A \to \xi \text{ mit } A \in \mathcal{N}, \xi \in \mathcal{V}^* \text{ hat,}$$

- vom *Typ 3* (*rechts-linear, regulär*), falls

  jede Produktion die Form

  $$A \to \tau B \text{ mit } \tau \in \mathcal{T} \text{ und } A, B \in \mathcal{N} \text{ oder}$$
  $$A \to \tau \text{ mit } \tau \in \mathcal{T} \cup \{\varepsilon\} \text{ und } A \in \mathcal{N} \text{ hat.}$$

Eine formale Sprache $\mathcal{L}$ heißt vom *Typ i*, falls es eine Grammatik $\mathcal{G}$ vom *Typ i* gibt mit $\mathcal{L} = \mathcal{L}(\mathcal{G})$. ❑

Für eine Typ 0-Grammatik gelten kaum Einschränkungen für die Form der Produktionen:

- Auf der linken Seite ihrer Produktionen steht eine von Elementen aus dem Vokabular $\mathcal{V}$, diese hat mindestens die Länge eins. Nach der Definition einer Grammatik kommt in dieser Folge mindestens ein Nonterminal vor.
- Die rechte Seite ist beliebig.

Für eine Grammatik vom Typ 1 gilt:

- Auf der linken Seite jeder Produktion kann nur noch genau ein Nonterminal erscheinen. Das Nonterminal wird auf der rechten Seite durch eine nicht-leere Symbolfolge ersetzt. Auf der linken Seite können noch weitere Terminale auftreten, so daß die Anwendung dieser Regel von ihrer Umgebung, dem Kontext, abhängig ist. Alle Terminale der linken Seite treten wieder auf der rechten auf.
- Die rechte Seite darf das Startsymbol nicht enthalten.

Für eine Grammatik vom Typ 2 gilt:

- Die linke Seite jeder Regel besteht nur noch aus einem Nonterminal. Für die Anwendung dieser Regeln gibt es keinen Kontext als Bedingung. (Auf dieser Stufe ist die Menge der Nonterminale sofort bestimmbar.)
- Die rechte Seite ist beliebig.

Für eine Grammatik vom Typ 3 gilt:

- Für die linke Seite gelten dieselben Einschränkungen wie für Typ 2-Grammatiken.
- Auf der rechten Seite darf nur dann ein Nonterminal erscheinen, wenn zuvor ein Terminal notiert wurde. Ansonsten darf die rechte Seite nur aus einem Terminal oder der leeren Symbolfolge bestehen.

Bemerkung: Die Chomsky-Hierarchie ist eine echte Hierarchie, d. h. $\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0$ mit $\mathcal{L}_i$ ist formale Sprache vom Typ i (Beweis s. [Avenhaus]). Eine erzeugende Grammatik für eine formale Sprache $\mathcal{L}$ legt zwar den Typ von $\mathcal{L}$ fest, macht aber **keine** Aussage darüber, von welchen Typ $\mathcal{L}$ **nicht** ist. Beispiel: Ist eine Grammatik vom Typ 0 und 2, aber nicht vom Typ 1, so ist nach obiger Feststellung die erzeugte Sprache trotzdem auch vom Typ 1 , da $\mathcal{L}_2 \subseteq \mathcal{L}_1$. Man kann nämlich eine Grammatik von Typ 1 finden, die diesselbe Sprache erzeugt und vom Typ 1 ist. Eine Aussage, ob die Sprache vom Typ 3 ist, läßt sich nur mit einigem theoretischen Aufwand machen.

1.2 Definitionen für den Compilerbau

Die Aufgabe eines Compilerbauers ist es, ein Programm zu schaffen, das eine Folge von Zeichen (das Programm) in eine Folge von Anweisungen für einen Prozessor bzw. für eine Rechnerarchitektur zu transformieren. Eine Aufgabe des Compilers besteht also darin, das Programm auf "Korrektheit" zu überprüfen, d. h. seine Struktur zu analysieren und zu akzeptieren. Ein gültiges Programm stellt einen Satz einer formalen (Programmier-)Sprache dar. Teile eines Compilers ergeben somit ein *akzeptierendes System* für eine Sprache, die zunächst durch eine Grammatik definiert (erzeugt) wird.

Eine Aufgabe des Compilerbauers ist es also, ein erzeugendes System (die Grammatik) in ein akzeptierendes System (den akzeptierenden Teil eines Compilers) zu transformieren. Reguläre (Typ 3-) und kontext-freie (Typ 2-) Sprachen sind durch verschiedene Automaten relativ leicht als akzeptierendes System zu implementieren. Der Aufwand steigt jedoch unverhätnismäßig bei Typ 1- und Typ 0-Sprachen. Da die meisten Programmiersprachen vom Typ 2 sind, werden im folgenden nur reguläre und kontext-freie Sprachen untersucht.

Der akzeptierende Teil eines Compilers wird üblicherweise in zwei Teile unterteilt: in den Scanner und den Parser. Die etwas lockere Ausdrucksweise aus einem Großteil der vorhandenen Literatur, wie "Der Scanner bildet die erste Phase des Compilers. Seine Hauptaufgabe besteht darin, Eingabezeichen zu lesen und als Ausgabe eine Folge von Symbolen zu erzeugen, die der Parser syntaktisch analysiert." [AhoSethiUllman] oder "Nun wird ... ein Scanner eingesetzt, um einzelne Symbole aus der einzulesenden Zeichenfolge herauszulesen. Dieser Scanner verkörpert die sogenannten lexikographischen Regeln, welche bestimmen, wie Symbole aus einzelnen Zeichen aufgebaut sind." [Wirth84] soll im folgenden genauer definiert werden.

1.2.1 Scanner

Ein Scanner enthält eine Menge von akzeptierenden Systemen für eine endliche Menge von *regulären Sprachen* zusammen mit einem geeigneten (Auswahl-) Verfahren. Der Scanner wird beschrieben durch die Grammatiken, die diese regulären Sprachen definieren. Wir notieren reguläre Sprachen mit Hilfe regulärer Ausdrücke wie sie in Kapitel 2 definiert sind. Dazu betrachten wir die folgenden Definitionen:

Definition 9 *Terminal eines Scanners*

Die *Terminale eines Scanners* sind diejenigen Symbole, mit denen ein Programm notiert wird. Wir sprechen im Zusammenhang mit Scannern nicht mehr von Symbolen, sondern von Zeichen. Dies stellt den Übergang einer abstrakten zu einer konkreten Sprache dar, die dann mit konkreten Symbolen, den Zeichen, notiert wird. ❑

Um Programmiersprachen auf unterschiedlichen Rechnersystemen bzw. -architekturen portabel zu halten, werden die Terminale meist aus einer Schnittmenge aller verfügbaren Zeichensätze gewählt, dem ASCII-Zeichensatz.

Definition 10 *Nonterminal eines Scanners*

Genau alle linken Seiten der Produktionen der Grammatiken, die die Sprachen für den Scanner beschreiben, bilden die Menge der *Nonterminale des Scanners*. ❑

Definition 11 *Satz eines Scanners*

Jeder Satz aus der Menge der Sprachen, die der Scanner akzeptiert, ist ein *Satz des Scanners*. ❑

Definition 12 *Literal*

Jeder Satz einer einelementigen Sprache, die der Scanner akzeptiert, wird als *Literal* bezeichnet. Ein *Literal* beschreibt also eine Sprache, die nur aus einer nichtleeren, konstanten Symbolfolge besteht. ❑

Wir sprechen nicht vom "Literal eines Scanners", weil die Bezeichnung *Literal* oft auch aus Sicht des Parsers für dieselbe nichtleere, konstante Zeichenfolge verwendet wird.

Definition 13 *Tokentag und Lexem*

Ein *Tokentag* ist ein Symbol, das einer Sprache des Scanners eindeutig zugeordnet ist.

Als *Lexem* bezeichnet man die konkrete Symbolfolge, d. h. Zeichenfolge, einer Sprache, die vom Scanner als Satz akzeptiert wird. ❑

Definition 14 *Token*

Als *Token* bezeichnet man das n-Tupel (Tokentag, Token-Ergebnisform, weitere Attribute ...).

Die Token-Ergebnisform ist eine Zeichenfolge, die üblicherweise das Lexem enthält. Das *Token* enthält als *Tokentag* das Symbol der Sprache, zu der das Lexem akzeptiert wurde. Eventuelle weitere Attribute sind frei wählbar (z. B. Position des Lexems in der bisher betrachten Symbolfolge usw.). ❑

Ein Scanner muß eine Menge von Sprachen gleichzeitig akzeptieren. Daher kann man sagen, er vereinigt in sich eine Menge von akzeptierenden Systemen für die zugehörigen Sprachen. Der Scanner arbeitet auf einer Zeichenfolge und liefert als Ergebnis ein Token oder einen Fehler. Wird der Scanner aufgefordert, ein neues *Token* zu liefern, überprüft er, welche der Sprachen ein Präfix der Eingabe als *Satz* enthalten. Diese Sätze sind Kandidaten für die *Lexeme* des neuen Tokens. Existieren mehrere Kandidaten, muß der Scanner eine Auswahl treffen, die üblicherweise über die Länge des *Lexems* entschieden wird. Die längste Zeichenfolge, die als *Satz des Scanners* akzeptiert werden kann, entscheidet über das neue *Token*. Enthalten mehrere Sprachen dieses längste Lexem, entscheidet üblicherweise die Reihenfolge der Sprachnotation über die Auswahl, d. h. es wird das *Tokentag* der Sprache gewählt, die in der Beschreibung des Scanners zuerst angegeben ist. Existiert kein Kandidat für ein *Lexem*, meldet der Scanner einen Fehler, da kein *Satz des Scanners* vorliegt.

Ein deterministischer endlicher Automat (DEA) eignet sich für die Implementierung eines akzeptierenden Systems zu einer regulären Sprache. Die DEAen verschiedener regulärer Sprachen lassen sich unter Berücksichtigung des oben beschriebenen Auswahlverfahrens zu einem

neuen DEA zusammenfassen, so daß die Beschreibung eines Scanners (als Menge von erzeugenden Systemen) automatisch in ein akzeptierendes System umzusetzen ist.

1.2.2 Parser

Ein Parser stellt ein akzeptierendes System für eine *kontext-freie Sprache* dar, die zunächst mittels kontext-freier Grammatik definiert wird (s. Kapitel 3). Diese Sprache wird als die eigentliche Programmiersprache bezeichnet.

Der Parser fordert den Scanner auf, ihm neue Token zu liefern. Die Token, die der Scanner an den Parser liefert, sind die Symbole für die kontext-freie Sprache, die der Parser akzeptiert.

Definition 15 *Terminal eines Parsers, Terminal-Name*

Ein *Terminal eines Parsers* ist ein Symbol, das von einem Scanner als Tokentag zurückgeliefert wird.

Das Terminal wird über einen Namen, dem *Terminal-Namen*, angegeben. ❑

Definition 16 *Nonterminal eines Parsers, Nonterminal-Name*

Genau alle linken Seiten der Regeln der Grammatik, die die kontext-freie Sprache für den Parser beschreibt, bilden die Menge der *Nonterminale des Parsers*.

Das Symbol der linken Seite einer Produktion heißt *Nonterminal-Name*. ❑

Wir notieren kontext-freie Sprachen mittels Backus-Naur Form (BNF) oder erweiterter Backus-Naur Form (EBNF) wie sie in Kapitel 3 beschrieben wird..

Definition 17 *Satz eines Parsers*

Jedes Programm, welches der Parser akzeptiert, ist ein *Satz des Parsers*. ❑

Wir stellen noch weitere Ansprüche an die einer Programmiersprachdefinition zugrundeliegende Grammatik. Dazu folgen einige weitere Definitionen.

Definition 18 *Syntaxbaum, Strukturbaum, Definition wie [MaurerWilhelm]*

Sei $\mathcal{G} = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ eine kontext-freie Grammatik zu einer Sprache $\mathcal{L}$ mit $\mathcal{N} = \mathcal{V} - \mathcal{T}$. Sei $\mathcal{B}$ ein Baum, bei dem die Ausgangskanten jedes Knotens geordnet sind. $\mathcal{B}$ heißt ein *Syntax-* bzw. *Strukturbaum* für einen Satz $\sigma \in \mathcal{L}(\mathcal{G})$ gdw gilt:

- Alle internen Knoten des Baumes sind mit Symbolen aus $\mathcal{N}$ markiert, alle Blätter mit Symbolen aus $\mathcal{T} \cup \{\varepsilon\}$.
- Ist ein beliebiger innerer Knoten mit ξ markiert und sind seine Kinder in dieser Reihenfolge mit $\psi_1, \ldots, \psi_n$ markiert, wobei gilt $\xi \in \mathcal{N}$, $\psi_1, \ldots, \psi_n \in \mathcal{N} \cup \mathcal{T}$, so ist $\xi \rightarrow \psi_1 \ldots \psi_n$ eine Produktion aus $\mathcal{P}$.
  Ist ein beliebiger innerer Knoten mit ξ markiert und ist sein einziges Kind mit ε markiert, wobei gilt $\xi \in \mathcal{N}$, so ist $\xi \rightarrow \varepsilon$ eine Produktion aus $\mathcal{P}$.
- Die Blattfolge von $\mathcal{B}$ ergibt den Satz σ.

Die Wurzel ist mit dem Startsymbol S der Grammatik markiert. ❑

Jede *Ableitungsfolge* läßt sich als *Syntaxbaum* notieren, der zugehörige Baum ist wie folgt zu konstruieren: Es wird ein Wurzelknoten erzeugt und mit dem Startsymbol markiert. Dann wird, ausgehend von Startsymbol die gewünschte Ableitungsfolge erzeugt. Bei jeder Anwendung einer Produktion $\xi \rightarrow \psi_1 \ldots \psi_n$ auf ein Nonterminal ξ (in der Ableitungsfolge) werden n Subknoten geordnet an den Knoten des Baumes, der mit ξ markiert ist, angefügt und mit $\psi_1 \ldots \psi_n$ markiert. Dieser Vorgang wird wiederholt, bis alle Nonterminale in der Ableitung ersetzt wurden. Der resultierende Baum ist der zur *Ableitungsfolge* gehörige *Syntaxbaum*.

Zu jedem Satz einer Sprache gibt es mindestens eine *Ableitungsfolge*. Daher gibt es auch zu jedem Satz einer Sprache mindestens einen *Syntaxbaum*.

Definition 19 *Eindeutigkeit eines Satzes, einer Grammatik*

Ein Satz einer Sprache heißt *eindeutig*, wenn er genau einen Syntaxbaum besitzt.

Eine Grammatik heißt *eindeutig*, wenn es keinen mehrdeutigen Satz in der durch sie erzeugten Sprache gibt. ❑

Es gibt meistens mehrere Möglichkeiten, zu einem Satz einer Sprache einen Syntaxbaum zu aufzubauen. Existieren z. B. die folgenden Produktionen in einer Grammatik:

$$\begin{aligned} \xi &\rightarrow \xi_1\xi_2 \,. \\ \xi_1 &\rightarrow \psi_1 \,. \\ \xi_2 &\rightarrow \psi_2 \,., \text{ wobei } \xi, \xi_1, \xi_2 \in \mathcal{N} \text{ und } \psi_1, \psi_2 \in \mathcal{T} \end{aligned}$$

Wurde die Produktion $\xi \rightarrow \psi_1\psi_2$ in einer Ableitungsfolge zu einem Satz verwendet, so ist es ohne Bedeutung, ob danach zuerst die Produktion ψ_1 oder ψ_2 angewendet wird. In beiden Fällen resultiert der gleiche *Syntaxbaum*.

Es muß sichergestellt sein, daß alle Sätze einer Programmiersprache eindeutig sind, d. h. eine Sprache muß durch eine eindeutige Grammatik erzeugt werden.

Definition 20 *Linksableitung, Rechtsableitung*

Sei $\omega_0, \omega_1, \ldots, \omega_n \in \mathcal{V}^*$ eine Ableitungsfolge für einen Satz $\sigma \in \mathcal{T}^*$ aus dem Startsymbol S mit $S = \omega_0$, $\sigma = \omega_n$. $\omega_0, \omega_1, \ldots, \omega_n$ heißt eine *Linksableitung* von σ, wenn beim Schritt von ω_i nach ω_{i+1} jeweils eine Produktion auf das am weitesten links stehende Nonterminal angewendet wurde.

Entsprechend heißt $\omega_0, \omega_1, \ldots, \omega_n$ heißt eine *Rechtsableitung* von σ, wenn beim Schritt von ω_i nach ω_{i+1} jeweils eine Produktion auf das am weitesten rechts stehende Nonterminal angewendet wurde. ❑

Es läßt sich zeigen, daß sich zu jeder Linksableitung eines Satzes eine korrespondierende Rechtsableitung finden läßt, die den gleichen Strukturbaum erzeugt. Für jeden Satz einer Sprache, die durch eine eindeutige Grammatik erzeugt wird, gibt es genau eine Links- und eine Rechtsableitung.

Definition 21 *Metasymbol, Metasprache*

Metasymbole sind die zur Beschreibung einer Sprache $\mathcal{L}$ notwendigen Symbole, die nicht Terminal- oder Nonterminalsymbole von $\mathcal{L}$ sind. Die beschreibende Sprache nennt man

Metasprache $\mathcal{L}_{\mathcal{M}}$. *Metasymbole* einer Sprache $\mathcal{L}$ sind ausschließlich Terminalsymbole der zugehörigen *Metasprache* $\mathcal{L}_{\mathcal{M}}$.

Das bisher verwendete Zeichen $\rightarrow$ zur Notation von Produktionen ist ein Metasymbol.

Durch einen relativ begrenzten Zeichenvorrat ist es häufig nicht zu vermeiden, daß für Meta- und Terminalsymbole diesselbe Repräsentation gewählt werden muß. Falls eine Unterscheidung aus dem Kontext nicht möglich ist, wird das Metasymbol durch doppelte Unterstreichung gekennzeichnet (z. B. $\underline{\underline{\rightarrow}}$).

1.3 Literaturvergleich

Im folgenden soll die unterschiedliche Begriffswelt der zugrundeliegenden Literatur erläutert, entwirrt und von der eigenen abgesetzt werden. Dabei werden zur besseren Lesbarkeit und Zuordnung die selbstdefinierten Begriffe *kursiv*, die Fremdbegriffe **fett** gedruckt.

Ein kurzer Überblick über die verglichene Literatur sowie die Abweichungen zu den eigenen Begriffen befinden sich in Tabelle 1.

[AhoSethiUllman] wird zwar zur Standardliteratur für den Bereich Compilerbau gezählt, kann aber als Nachschlagewerk nur in eingeschränktem Maß verwendet werden. In ihrem ständig konkreter werdenden Erzählstil werden beiläufig Definitionen eingewoben, die zunächst sehr schwammig und auch teilweise inkonsistent geraten. Diese Begriffe werden dann erst in späteren Kapiteln in abweichender Bedeutung definiert und verwendet.

So wird zunächst strikt zwischen **Zeichen** und **Symbol** unterschieden, ein **Symbol** hat hier die Bedeutung eines *Token*. Später wird ein **Alphabet** (*Vokabular*) als eine **Zeichenklasse** bestehend aus einer endlichen Menge von **Symbolen** eingeführt, so daß ein **String** als eine endliche Folge von **Symbolen** definiert werden kann (*Symbolfolge*). **Satz** und **Wort** wird als Synonym für **String** benutzt, nicht aber in seiner Bedeutung unterschieden.

Das *Präfix* ist definiert als diejenige Zeichenfolge eines Strings, die übrig bleibt, wenn am rechten Ende des Strings null oder mehrere Zeichen entfernt werden. Der Rest eines gegebenen Strings bleibt undefiniert, obwohl dieser Begriff für weitere Erläuterungen nützlich wäre.

Sprachen werden zunächst als eine beliebige Menge von **Strings** über einem **Alphabet** eingeführt, später ist die Sprache die Menge aller herleitbaren **Wörter** einer *Grammatik*, wobei nur **kontext-freie Grammatiken** erwähnt werden. Die Herleitbarkeit wird in späteren Kapiteln über die **Ableitungsrelation**, die exakt definiert wird, genauer beschrieben. Nicht erwähnt wird das *Semi-Thue-System* als Grundlage von Grammatiken.

An dieser Stelle wird das Wort ersetzt durch die **Satzform**, die dadurch definiert ist, daß sie aus einer **Grammatik** hergeleitet werden kann. Eine **Satzform**, die keine Nichtterminale enthält, wird **Satz** genannt; dies entspricht unserer Definition eines *Satzes*. Der Begriff der **Satzform** wird später nicht mehr verwendet.

Reguläre **Grammatiken** werden mittels **regulärer Ausdrücke**, die eine Notation zur Spezifikation von Mustern sind, erklärt. Ein Muster stellt somit eine *reguläre Sprache* dar.

Viele unserer Definition sind in mehr oder weniger veränderter Form aus [Avenhaus] entnommen. Um bestimmte Assoziationen bei den Grundbegriffen zu verhindern, haben wir auf

✔: übereinstimmende Definition
⊥: Definition fehlt

Tabelle 1: Literaturvergleich der verwendeten Begriffe aus den theoretischen Grundlagen

verw. Begriffe / Autoren	Vokabular	Symbol	Symbolfolge	Präfix u. Rest einer Symbolf.	formale Sprache	Satz einer Sprache	Semi-Thue-System	Ableitungs-relation	Ableitungs-folge	Grammatik	Klassifizierung von Gramm.
[AhoSethi-Ullman]	Alphabet Zeichen-klasse	✔	String, Satz, Wort	Präfix: ✔ Rest: ⊥	✔, Menge v. Strings über Alphabet	Satz / Wort einer Grammatik	⊥	✔	✔	nur kontext-frei	⊥
[Avenhaus]	Alphabet	Buchstabe	Wort	⊥	✔	(⊥)	✔	✔	✔	Startsymbol = Axiom, ✔	✔
[Broy]	Alphabet ist spezielles ~	Zeichen	Wort	⊥	def. über Alphabet	⊥	⊥	⊥	⊥	Syntax	⊥
[GoosWaite]	✔, Alphabet	✔	✔, String	✔, head & tail	(✔), formal systems	✔, sentence	✔, general rewriting system	✔,derivative relation	✔, derivation	✔	✔
[Gries]	✔, Alphabet	Word, Symbol	String	✔	✔	✔, sentence	⊥	(✔)	(✔)	(✔)	✔
[MaurerWilhelm]	Alphabet	Zeichen	Wort, Satzform	Präfix: ✔ Rest: ⊥	(✔)	Wort	Automaten für reguläre Ausdrücke und kontext-freie Grammatiken			nur regulär und kontext-frei	⊥
[Polak]	⊥	Zeichen, Token	Zeichen-kette, Wort	⊥	✔	⊥	⊥	✔	✔	Marken, (✔)	⊥
[Wipper-mann]	(✔), $\mathcal{N} \cup \mathcal{T}$	⊥	⊥	⊥	✔, def. über synt. Menge	Satz einer Grammatik	⊥	(✔)	✔, Ersetzungsfolge, Ableitung	nur kontex-frei	⊥
[Wirth84]	✔	(Grund-) Symbol	✔	⊥	def. über 4-Tupel	✔	⊥	(✔)	✔, Herleitung	Syntax, Substitutionsregel = Produktion	(✔)

Begriffe wie **Alphabet** (für *Vokabular*), **Buchstabe** (für *Symbol*) und **Wort** (für *Symbolfolge*) verzichtet. [Avenhaus] unterscheidet nicht zwischen Wort und Satz einer Sprache.

In [Broy] findet man ein breites Spektrum an theoretischen Grundlagen für den Informatiker, jedoch reduziert sich die Information über den Bereich Grundlagen für den Compilerbau auf zwei Seiten:

Ein *Symbol* ist ein **Zeichen**; eine **Zeichenkette** (auch ein **Wort** einer bestimmten Länge) entspricht der bekannten *Symbolfolge*. Auf die Definition eines gültigen *Wortes/Satzes einer Sprache* wird verzichtet. Eine Menge von Zeichen wird erst als **Alphabet** (*Vokabular*) bezeichnet, wenn auf ihr eine lineare Ordnung definiert ist. Die Notwendigkeit dieser Ordnung bleibt jedoch im Weiteren verborgen.

Als *formale Sprache* bezeichnet [Broy] einerseits eine Teilmenge aus der Menge der **Zeichenfolgen** (wie gewohnt), andererseits ist sie die Menge der syntaktisch korrekten **Programme**, die durch die **Syntax** beschrieben wird. Die **Syntax** entspricht also unserer *Grammatik*. Auf die Grundlagen von Grammatiken (wie *Semi-Thue-System, Ableitung*, ...) wird nicht eingegangen.

Ähnlich wie bei [Avenhaus] haben wir uns stark an [GoosWaite] angelehnt, da diese eine verständliche Abfolge von Definitionen liefern. Geringe Abweichungen bestehen nur in der Übersetzung, beziehen sich aber nicht auf den Inhalt.

Von allen hier untersuchten Autoren verwendet [Gries] einen Definitionsschatz, der am meisten verwirrt. Ähnlich wie in [AhoSethiUllman] versucht er zunächst informell, den Leser an das Thema heranzuführen. Die hier verwendeten Begriffe werden teilweise später mit anderem Inhalt formell definiert, was leicht konfus wirkt. Genau wie bei [AhoSethiUllman] ist auch hier ein roter Faden nur mit Mühe zu erkennen.

Elemente eines *Alphabets* nennt er **Symbole** oder **Worte**, *Symbolfolgen* sind **Strings**. Bezüglich einer Grammatik spricht er von einem Vokabular: Ähnlichkeiten oder Unterschiede zu einem Alphabet werden nicht genannt!

Als Ursprung einer Sprache führt er die Definition einer Grammatik ein, die nur von einem Startsymbol abhängt. **Regeln** zur Beschreibung von *Symbolfolgen* werden nicht angegeben. (Eine **Grammatik** ist zunächst kein 4-Tupel, vielmehr extrahiert er aus einem gegebenen Satz von Regeln alle **Nonterminale**, die übrigen *Symbole* sind die **Terminale**. In einem zusätzlichen Kapitel wird später die Grammatik formell korrekt definiert und auch nach dem Schema von [Chomsky] klassifiziert.) Mit Hilfe dieser (vom Himmel gefallenen) Regeln können **Ableitungen** gebildet werden. Die dazu notwendige *Ableitungsrelation* hat zwar ein Zeichen ($\Rightarrow$), bleibt aber unbenannt. Ein **String** am Ende einer **Ableitungsfolge** wird **Wort**, das zuvor ein Synonym für **Symbol** war, genannt. **Worte** heißen auch **satzartige Form**; wenn diese nur noch aus Terminalen bestehen, sind es **Sätze**. So sind **Sätze** die eigentliche Elemente einer Sprache. Die Notwendigkeit für die Verwendung des Begriffs **satzartige Form** bleibt unklar.

Die theoretischen Grundlagen für den Übersetzerbau fallen in [MaurerWilhelm] relativ kurz aus. Obwohl das gesamte Werk sehr formal aufgebaut ist, entsteht der Eindruck, die Grundlagen seien zwar unvermeidbar, aber doch Ballast, der möglichst gering gehalten werden sollte.

Ein **Alphabet** besteht aus **Zeichen** (unsere *Symbole*), deren Folge **Wort** genannt wird. **Worte** über einem **Alphabet** werden auch **Symbole** genannt. (Hier wird in der Betrachtungsweise - wie bei vielen Autoren - die Sicht eines Scanners und Parsers gemischt.) Es werden sog. **Sym-**

bolklassen eingeführt, die *Symbole* gleicher syntaktischer Struktur enthalten. Diese Klassen beziehen sich jedoch nur auf *Symbole* des Scanners; sie fassen eine Menge von *Symbolen* zusammen, die für den Parser äquivalent sind.

Eine Betrachtung (*Klassifikation*) *von Grammatiken* beschränkt sich auf die für die syntaktische Analyse notwendige kontext-freie Grammatik. Bei der Beschreibung von regulären Sprachen wird nur auf reguläre Ausdrücke eingegangen.

[Polak] fordert die formale Beschreibung von Programmiersprachen, um einen zugehörigen Compiler verifizieren zu können. Dennoch fallen seine Begriffsdefinitionen sehr kurz aus. Die **Syntax** einer Sprache beschreibt, wie Programme dieser Sprache als Zeichenketten repräsentiert werden. Diese Syntax unterteilt er in **Micro Syntax** oder **Token Syntax** und die **Satzstruktur** von Programmen. Die **Micro Syntax** beschreibt den syntaktischen Aufbau der kleinsten syntaktischen Einheiten, die hier Token genannt werden. Es wird der Begriff der **Tokenklasse** eingeführt und dazu verwendet, diese kleinsten syntaktischen Einheiten zu gruppieren, so daß sie jeweils mittels einer regulären Sprache beschrieben werden können. Als Definition einer regulären Sprache wird die Eingabe für einen Scanner-Generator genannt, genauere Angaben fehlen ebenso wie der Zusammenhang zwischen den Token und ihrer Verwendung als Terminalsymbole in der (übergeordneten) Satzstruktur.

[Polak] erweitert den Begriff der *Grammatik* um **Marken**, mit denen die einzelnen Produktionen eindeutig gekennzeichnet werden. Diese **Marken** werden für die spätere Verwendung in *Strukturbäumen* eingeführt. Eine **markierte Grammatik** ist ein 5-Tupel bestehend aus Mengen von **Nonterminalsymbolen**, **Terminalsymbolen**, **Marken**, **Produktionen** und einem **Startsymbol** aus der Menge der Nonterminalsymbole. Ein Vokabular wird nicht definiert, dementsprechend auch nicht die Nonterminalsymbole. Es fehlt ebenfalls die Erklärung von erzeugenden Systemen und somit der Zweck des Startsymbols.

[Wippermann] stellt in seinem Vorlesungsskript den Stoff leicht lesbar dar, jedoch verliert er durch den Verzicht auf formale Definitionen an Genauigkeit.

Symbol und damit zusammenhängende Begriffe werden weggelassen. Es wird nur auf die **kontext-freie Grammatik** (hier als 4-Tupel) eingegangen, in deren Bezug der Begriff **Vokabular** auftritt. Ebenfalls in diesem Zusammenhang werden **Terminale T** und **Nonterminale** als *Symbole* erwähnt. Die Folge von Terminalen wird nur mit **T*** bezeichnet, ist aber ansonsten namenlos. Die genaue Bedeutung des Symbols **T*** wird wohl als bekannt vorausgesetzt, läßt sich aber notfalls aus dem Kontext schließen.

Die rechte Seite einer Produktion wird **kontext-freier Ausdruck (CFX)** genannt. Damit können Elemente der Sprache erzeugt werden (Hinweis auf erzeugendes System), was durch Beispiele ergänzt wird. Dieser Vorgang wird dann im Zusammenhang mit der Syntaxanalyse **Ableitung** genannt. Die *Ableitungsrelation* $\Rightarrow$ hat keinen Namen. Die Elemente, die durch die Produktionen abgeleitet werden können und nur aus Terminalen bestehen, sind **Sätze einer Grammatik**. Der Begriff **satzartige Form** tritt hier ebenfalls unmotiviert auf. Als **Sprache** wird wie in [Broy] die Menge der syntaktisch korrekten Programme aufgeführt.

[Wirth84] gibt in Relation zum Umfang des gesamten Werkes viel theoretische Grundlagen preis. Er überhäuft den Leser nicht mit unnötigen Begriffen, alles wirkt sehr einfach und klar.

Er definiert zunächst eine Sprache als eine Menge von Symbolfolgen, die über eine **Syntax** (Synonym für *Grammatik*) wohldefiniert sind. *Symbolfolgen* können auch **Wort** genannt werden. Die Elemente einer **Sprache** heißen **Sätze**. Genauer definiert wird eine Sprache über ein

4-Tupel, das stark an eine *Grammatik* erinnert. Da er die Definition der *Grammatik* wegläßt, entfällt auch hier eine *Klassifikation* sowie die Erwähnung eines *Semi-Thue-Systems*. Eine *Ableitung* (hier **Herleitung**) geschieht in der bekannten Weise durch Anwendung von Produktionen. Dies wird durch Beispiele erläutert. Als Sprachen werden nur **kontext-sensitive** und **-freie** erwähnt. Die lexikalische Analyse und damit die Gefahr einer Vermischung von Sichtweisen (Scanner $\leftrightarrow$ Parser) kommt nicht vor.

2 Produktionen regulärer Grammatiken

In diesem Kapitel wird eine Metasprache für die Notation regulärer Grammatiken angegeben. Zunächst werden in Kapitel 2.1 einfache Scanner-Regeln beschrieben, die dann zum einfacheren Gebrauch für die Beschreibung des Scanners einer Programmiersprache in Kapitel 2.2 erweitert werden. Kapitel 2.3 beschreibt die Notation für vorausschauendes Scannen mittels Lookahead-Operator. In Kapitel 2.4 wird beschrieben, wie zusätzliche Attribute in ein Token eingesetzt werden. Kapitel 2.5 beschreibt die Notation von Kommentar-Regeln für Programmiersprachen.

In Anhang A findet sich die genaue Syntaxdefinition der erweiterten Scanner-Regeln, wie sie hier eingeführt wird. Sie wird durch eine vereinfachte Form der EBNF (s. Kapitel 3) beschrieben.

Allen hier verwendeten Darstellungsformen liegt der ASCII-Zeichensatz zugrunde.

2.1 Einfache Scanner-Regeln (SR)

Ein Scanner akzeptiert eine Menge von Sprachen. Jede dieser Sprachen wird durch eine Grammatik beschrieben, die wenige, meist nur eine, Produktionen enthält. Die Gesamtheit dieser Produktionen nennen wir Scanner-Regeln (SR). Zur Beschreibung dieser Produktionen werden neben den Terminal- und Nonterminalsymbolen verschiedene Metasymbole (s. Definition 21) benötigt.

SRn beschreiben reguläre Sprachen. Die Produktionen regulärer Grammatiken haben entsprechend der Chomsky-Hierarchie (s. Definition 8) den folgenden Aufbau:

$A \rightarrow \tau B$ mit $\tau \in \mathcal{T}$ und $A, B \in \mathcal{N}$ oder

$A \rightarrow \tau$ mit $\tau \in \mathcal{T} \cup \{\varepsilon\}$ und $A \in \mathcal{N}$.

Eine SR wird durch einen Bezeichner (den Nonterminal-Namen des Scanners) benannt. Eins der Metasymbole "=", "→" oder "::=" trennt die linke Seite einer Regel von der rechten. Die rechte Seite besteht aus einem Terminal gefolgt von einem Nonterminalnamen oder einem einzelnen Terminal oder ε.

Es gelten folgende Vereinbarungen zur Notation:

- Als Trenner der linken und rechten Seite einer Produktion können die Metasymbole "::=", "=" oder "→" benutzt werden.
- Scannernonterminale werden durch ihren Namen notiert. Dieser muß mit einem Buchstaben beginnen und darf dann eine beliebige Folge aus Buchstaben, Ziffern, "-" und "_" enthalten.
- Zeichen werden in (einfache oder doppelte) Anführungszeichen "..." oder '...' eingeschlossen. Innerhalb der einfachen Anführungszeichen ist jedes druckbare Zeichen außer dem Zeilenendezeichen und einem einfachen Anführungszeichen erlaubt. Innerhalb der doppelten Anführungszeichen ist jedes druckbare Zeichen außer dem Zeilenendezeichen und einem doppelten Anführungszeichen erlaubt.

  Nicht druckbare Zeichen können durch ihren ASCII-Wert (in dezimaler Darstellung) notiert werden.

- ε beschreibt die leere Zeichenfolge.
- Jede Produktion wird durch einen Punkt "." abgeschlossen.

Die rechte Seite von SR nennen wir *(einfache) reguläre Ausdrücke*. Elemente der regulären Ausdrücke sind also ε, Zeichen (Terminale), Nonterminalnamen.

Im folgenden Kapitel werden wir diese Notation erweitern.

2.2 Erweiterte Scanner-Regeln (ESR)

Wir erweitern nun die SR zu *erweiterten Scanner-Regeln (ESR)*, indem wir die regulären Ausdrücke um die Metasymbole "|", "*", "+", "?", "{", "}", "..", "(", ")" zu *erweiterten regulären Ausdrücken* ergänzen. Die erweiterten regulären Ausdrücke erweitern nicht die Ausdruckskraft der einfachen regulären Ausdrücke. Es werden nur verkürzende Schreibweisen für häufig gebrauchte Teilausdrücke eingeführt. Die Elemente der erweiterten regulären Ausdrücke sind im einzelnen:

1. Zeichenmenge

   Eine Zeichenmenge kann Zeichen, Zeichenintervalle (s. Punkt 2) und Bezeichner enthalten. Bezeichner dürfen wiederum für Zeichen, Zeichenintervalle oder Zeichenmengen stehen. In diesem Zusammenhang ist zu beachten, daß Zeichenintervalle (s. Punkt 2) und Alternativen (s. Punkt 8) als andere Notationsform für Zeichenmengen gelten. Eine Zeichenmenge wird mit Hilfe der geschweiften (Mengen-) Klammern ("{" und "}") notiert. Damit können die SR

   ```
   A       =    "a" .
   A       =    "b" .
   A       =    "c" .
   ```

   wie folgt abgekürzt werden:

   ```
   A       =    { "a" "b" "c" } .
   ```

   Eventuell geschachtelte Mengen werden auf eine Ebene gebracht und vereinigt.

   Beispiel: {"#" "A".."Z" digit}

   Der vordefinierte Bezeichner **ANY** repräsentiert eine Zeichenmenge, die alle Zeichen außer dem Zeilenendezeichen enthält.

2. Zeichenintervall

   Ein Zeichenintervall stellt eine abkürzende Schreibweise dar für eine Menge, die alle Zeichen zwischen linkem und rechtem Grenz-Zeichen und die beiden Grenz-Zeichen selbst beinhaltet. Dabei muß das linke Grenz-Zeichen vor dem rechten liegen. Es liegt die Ordnung des ASCII-Zeichensatzes zugrunde. Ein Zeichenintervall wird durch Angabe der beiden Grenz-Zeichen, getrennt durch zwei Punkte "..", notiert.

   Beispiel:

   Die Menge der Ziffern kann statt der SR-Notation

```
digit       =   "0" .
digit       =   "1" .
...
digit       =   "9" .
```

in ESR wie folgt notiert werden:

```
digit       =   "0" .. "9" .
```

Ein Zeichenintervall stellt eine andere Schreibweise für Zeichenmengen (s. Punkt 1) dar.

3. Obligatorische Wiederholung: 1 bis N

Bei der Beschreibung eines Scanners werden häufig unbeschränkte Zeichenfolgen benötigt, die mindestens die Länge 1 haben. Z. B. kann ein Bezeichner aus mindestens einem, jedoch unbeschränkt vielen Buchstaben bestehen. Die obligatorische Wiederholung muß in SR mit Hilfe von Rekursion beschrieben werden:

```
A           =   "a" A .
A           =   "a" .
```

Die Regel A definiert die Sätze "a", "aa", "aaa", etc. Da die Wiederholung durch Rekursion nicht immer auf den ersten Blick ersichtlich ist, steht in ESR ein Konstruktor für die Notation der obligatorischen Wiederholung zur Verfügung. Dies ist das Pluszeichen "+". Das Pluszeichen bezieht sich auf den direkt vorangestellten regulären Ausdruck. Somit können die obigen Regeln wie folgt notiert werden:

```
A           =   "a" + .
```

4. Optionale Wiederholung: 0 bis N

Bei der Beschreibung eines Scanners werden häufig Zeichenfolgen von unbeschränkter Länge benötigt, z. B. kann eine reelle Zahl 0 bis unbeschränkt viele Ziffern vor dem Dezimalpunkt haben. Die optionale Wiederholung muß in SR mit Hilfe von Rekursion beschrieben werden:

```
A           =   "a" A .
A           =   ε .
```

Die Regel A definiert die Sätze ε, "a", "aa", "aaa", etc. Da die Wiederholung durch Rekursion nicht immer auf den ersten Blick ersichtlich ist, steht in ESR ein Konstruktor für die Notation der optionalen Wiederholung zur Verfügung. Dies ist der Kleene-Stern "*". Der Kleene-Stern bezieht sich auf den direkt vorangestellten regulären Ausdruck. Somit können die obigen Regeln wie folgt notiert werden:

```
A           =   "a" * .
```

5. Option

In regulären Sprachen kommt es oft vor, daß ein Teil einer Zeichenfolge optional stehen kann, d. h. eine Zeichenfolge ist sowohl mit als auch ohne diesen Teilsatz ein Satz der Sprache. Eine abkürzende Schreibweise dafür stellt der Postfix-Optionsoperator "?" zur Verfügung. Damit entsprechen die SR

```
A           =   "a" .
A           =   ε .
```

den ESR

```
A       =   "a"? .
```

Der Optionsoperator bezieht sich auf den direkt vorangestellten regulären Ausdruck.

6. Sequenz, Zeichenkette

Wir erweitern die Schreibweise der SR um die n-fache Konkatenation, für die jedoch kein neues Metasymbol eingeführt wird. Damit wird es möglich, beliebig lange Folgen regulärer Ausdrücke zu notieren, ohne Hilfsregeln einzuführen. Z. B. muß die Sequenz-Darstellung des Satzes "1a2b" in SR wie folgt notiert werden:

```
A       =   "1" A1 .
A1      =   "a" A2 .
A2      =   "2" A3 .
A3      =   "b" .
```

In ESR können wir diese Regeln wie folgt zusammenfassen:

```
A       =   "1" "a" "2" "b" .
```

Sequenzen von aufeinander folgenden Zeichen können verkürzt als Zeichenketten beschrieben werden. Eine Zeichenkette schließt eine Folge druckbarer Zeichen in einfache oder doppelte Anführungszeichen ein. Das jeweilige Anführungszeichen darf nicht in der Folge enthalten sein. Die obige Regel A kann dann wie folgt notiert werden:

```
A       =   "1a2b" .
```

7. Mengendifferenz

Der Mengendifferenzoperator "-" bildet die Differenz zweier Zeichenmengen. Das Ergebnis ist wieder eine Zeichenmenge.

Beispiel:

Die Menge aller Zeichen außer dem Zeilenendezeichen und dem Punkt kann in ESR wie folgt notiert werden:

```
set     =   ANY - "." .
```

Da Zeichenintervalle (s. Punkt 2) lediglich eine andere Schreibweise für Zeichenmengen darstellen, können diese ebenfalls in einer Mengendifferenz verwendet werden.

8. Alternative

In der Beschreibung eines Scanners können mehrere Regeln angegeben werden, um das gleiche Nonterminal abzuleiten. Wir fassen diese Regeln zu einer neuen Regel zusammen und trennen die einzelnen Alternativen durch senkrechte Striche "|". Damit entsprechen die SR

```
A       =   "a" .
A       =   "b" .
```

der ESR

```
A       =   "a" | "b" .
```

Die Alternative von Zeichen stellt eine andere Schreibweise für Zeichenmengen (s. Punkt 1) dar und kann daher auch in einer Mengendifferenz (s. Punkt 7) verwendet werden.

Die Alternative hat die schwächste Bindung aller Operatoren der erweiterten regulären Ausdrücke.

Für die zusätzlichen Metasymbole der ESR muß eine Bindungspriorität angegeben werden. Die Bindungspriorität fällt entsprechend der Reihenfolge der obigen Auflistung. Die geringste Priorität hat also die Alternative. Die geschweiften Klammern für die Mengenbildung dienen der Gruppierung der in ihnen eingeschlossenen Ausdrücke, die genauso als Einheit betrachtet werden wie Zeichenintervalle.

Unter Umständen ist es notwendig, die gegebene Priorität zu durchbrechen. Durch die runden Klammern ("(" und ")") ist es möglich, reguläre Ausdrücke anders zu gruppieren:

9. Gruppierung

   Die Gruppierung wird benötigt, wenn ein Operator ("|", "*", "+", "?") auf eine Folge von Ausdrücken angewendet werden soll, die nicht schon implizit (d. h. durch "{", "}", "-" oder "..") gruppiert sind. In ESR hat z. B. die Alternative eine schwächere Bindung als eine Folge von Terminalen, so daß man zwei Regeln benötigt, um die Sätze "abd" oder "acd" einer Sprache zu beschreiben:

```
A       =   "a" B "d" .
B       =   "b" | "c" .
```

   Um beliebig andere Bindungen erzwingen zu können, kann man die runden Klammern ("(" und ")") zur Gruppierung verwenden. Damit läßt sich die obige Sprache in nur einer Regel formulieren:

```
A       =   "a" ( "b" | "c" ) "d" .
```

   Die Gruppierung hat die höchste Priorität.

Im Unterschied zu SR dürfen in ESR nicht mehrere Produktionen mit gleichem Namen angegeben werden. Ebenfalls ist die Verwendung des Symbols ε in ESR nicht erlaubt, da alle Produktionen die ε enthalten stattdessen auch den Optionsoperator "?" verwenden können.

Bezeichner dürfen in SR nicht rekursiv oder korekursiv verwendet werden!

2.3 Der Lookahead-Operator

In einigen Sprachen ist es nötig, einige Zeichen vorauszuschauen, um bestimmen zu können, welcher Sprache des Scanners das aktuelle Lexem zuzuordnen ist. So sind z. B. in FORTRAN [ANSI78] Schlüsselwörter keine reservierten Worte, d. h. es können Variablen gleichen Namens existieren. Daher muß der Scanner aus dem Kontext entscheiden, ob der aktuelle Bezeichner ein Schlüsselwort oder eine Variable bezeichnet. Er muß dafür einige Zeichen vorausschauen. Das Schlüsselwort **DO** z. B. wird daran erkannt, daß nachfolgend zwei Buchstaben- und Ziffernfolgen stehen, die durch ein Gleichheitszeichen "=" getrennt werden, worauf ein Komma "," folgt.

Für solche Fälle gibt es den sogenannten Lookahead-Operator, der als schräger Strich "/" notiert wird. Besteht eine Scanner-Regel aus zwei regulären Ausdrücken, die durch den Lookahead-

Operator getrennt werden, so wird eine Zeichenfolge nur dann als Satz der zugehörigen Sprache akzeptiert, wenn sie auch ohne den Operator akzeptiert worden wäre. Entscheidet der Scanner, daß das nächste *Token* das Kennzeichen dieser Sprache tragen soll, so wird als *Lexem* nur die Zeichenfolge akzeptiert, die durch den Ausdruck bis zum Lookahead-Operator beschrieben ist. Die Zeichenfolge, die eine Ableitung des zweiten Ausdrucks ist, bleibt dem *Rest* der Eingabe zugehörig und steht für die nächste Eingabe zur Verfügung.

Beispiel:

```
Do-Keyword      =    "DO" / (letter | digit)* "=" (letter | digit)* "," .
letter          =    "a" .. "z" | "A" .. "Z" .
digit           =    "0" .. "9" .
```

Diese Form das FORTRAN-Schlüsselwort **DO**. Die Buchstaben- und Ziffernfolgen dürfen aber noch nicht als gelesen gelten.

Die vom Lookahead akzeptierte Zeichenfolge trägt zur Länge des Lexems bei. So ist es z. B. möglich, einem Token bei der Auswahl Vorrang vor einem anderen gleichlangen Token zu geben, wenn eine bestimmte Zeichenfolge nachfolgt.

2.4 Die Ergebnisform

Unter Umständen soll die Zeichenfolge, die einen Satz eines Scanners darstellt, modifiziert werden, bevor sie als *Token-Ergebnisform* in das nächste *Token* einfließt. Z. B. kann es nötig sein, vor der Übersetzung in eine Zwischen- oder Zielsprache Modifikationen an den Zeichen eines Bezeichners vorzunehmen oder Fluchtsymbole in Strings zu ersetzen.

Zu diesem Zweck gibt es eine optionale Ergebnisform für Scanner-Regeln. Diese wird durch einen geschlängelten Pfeil "~>" von den vorangehenden regulären Ausdrücken getrennt und enthält eine Folge von Literalen und dem Dollarzeichen "$". Die Konkatenation dieser Zeichenfolgen wird - anstatt des *Lexems* - als *Token-Ergebnisform* ausgewiesen; dabei steht das Dollarzeichen für das tatsächlich gelesene *Lexem*. Wird keine Ergebnisform angegeben, so wird das *Lexem* als *Token-Ergebnisform* ausgewiesen und in das nächste *Token* eingesetzt oder als Resultat an eine andere Scanner-Regel weitergegeben, die diese auf der rechten Seite verwendet. Soll der Scanner keine *Token-Ergebnisform* zurückgeben, muß die leere Zeichenkette "" als Ergebnisform angegeben werden.

Beispiel:

Die Scanner-Regeln

```
SC: Letter      =    {"A".."Z" "a".."z"} .
SC: ExclMark    =    "!" ~> "" .
SC: SingleQuote =    "'" ~> '\"' .
SC: String      =    {Letter Digit ExclMark SingleQuote "."}* ~> "|" $ "|" .
```

führen bei der Eingabe

```
"I said 'Hello Man!'."
```

zu der Ausgabe

```
"|I said \"Hello Man\".|"
```

2.5 Kommentar-Regeln

Kommentare haben in Programmiersprachen eine besondere Funktion. Sie dienen zur Dokumentation, haben also keine Bedeutung für das Programm. Dennoch muß ihre Notation beschrieben werden, da sie zusammen mit dem eigentlichen Programm geschrieben und später bei der Übersetzung analysiert werden. Da Kommentare nur insofern bedeutungstragend sind, daß sie ignoriert werden müssen ohne einen Fehler zu erzeugen, können sie bereits in der Phase der lexikalischen Analyse, d. h. vom Scanner, "aussortiert" werden. Der Scanner wird also, wenn er einen Kommentar findet, kein Token an den Parser liefern, sondern den Kommentar bis zu seinem Ende überlesen und dem Parser das erste Token nach dem Kommentar liefern.

Da Kommentare vom Scanner erkannt und analysiert werden, liegt es nahe, sie analog zu Scanner-Regeln mittels regulärer Ausdrücke zu beschreiben. Scanner-Regeln zur Beschreibung eines Kommentars haben daher keine Ergebnisform.

In manchen Programmiersprachen, z. B. in Modula-2 [Wirth 80], können Kommentare geschachtelt werden, d. h. ein Kommentar kann innerhalb eines anderen stehen. Es ist daher notwendig, Bezeichner von Kommentar-Regeln (im Unterschied zu Scanner-Regeln) rekursiv auf der rechten Seite einer Kommentar-Regel zuzulassen. Dies ist eine echte Erweiterung zu der Definition regulärer Sprachen entsprechend der Definition von Chomsky (s. Definition 8).

3 Produktionen kontext-freier Grammatiken

In diesem Kapitel wird eine Metasprache für die Notation kontext-freier Grammatiken angegeben. In Kapitel 3.1 werden zunächst die Elemente der Backus-Naur-Form vorgestellt, die dann in Kapitel 3.2 um zusätzliche Metasymbole zur Extended Backus-Naur-Form (EBNF) erweitert werden. Diese zusätzlichen Metasymbole ermöglichen eine kürzere und besser lesbare Notation der Grammatik von Programmiersprachen. Kapitel 3.3 stellt dann ein zusätzliches Konzept vor, das nützlich für die Implementierung von Compilern für Sprachen mit bestimmten Eigenschaften ist.

In Anhang A findet sich die genaue Syntaxdefinition der EBNF wie sie hier verwendet wird. Sie wird durch eine vereinfachte Form der EBNF selbst beschrieben.

3.1 Elemente der Backus-Naur-Form (BNF)

3.1.1 Ursprüngliche BNF

Die Produktionen kontext-freier Grammatiken werden üblicherweise in Backus-Naur-Form (BNF) oder einer Erweiterung davon (Extended Backus-Naur-Form, EBNF) notiert. Die BNF wurde erstmalig bei der Definition von ALGOL 60 verwendet und von P. Naur [Naur] definiert.

Die BNF enthält die folgenden Metasymbole: "::=", "<", ">".

Regeln der BNF werden durch einen Bezeichner (den Nonterminal-Namen) eingeleitet. Die Zeichenfolge "::=" trennt die linke Seite der Produktion von der rechten. Die rechte Seite besteht aus einer Sequenz von Parserterminalen und -nonterminalen. Terminal- und Nonterminal-Namen werden zur Unterscheidung zu Literalen in spitze Klammern ("<", ">") gesetzt.

3.1.2 Aktuelle BNF

Im folgenden weichen wir von der üblichen Notation ab und führen dafür die folgenden zusätzlichen Metasymbole ein: "=", "→", "...", '...', ".", **`Fettdruck`**. Dafür entfallen die spitzen Klammern als Metasymbole. Es gelten folgende Vereinbarungen zur Notation:

- Als Trenner der linken und rechten Seite einer Produktion können außer der Zeichenfolge "::=" auch das Gleichheitszeichen "=" oder der Ableitungspfeil "→" benutzt werden.
- Parsernonterminale werden durch ihren Namen notiert.
- Literale werden in (einfache oder doppelte) Anführungszeichen "..." oder '...' eingeschlossen oder **`fett`** (s. Anhang 2) gesetzt.
- Andere Parserterminale werden durch ihren Namen notiert.
- Jede Produktion wird durch einen Punkt "." abgeschlossen, d. h. eine Regel kann über das Ende einer Zeile hinausreichen.

Die folgenden Konstrukte sind gültige Teile der rechten Seite einer BNF-Produktion:

1. Nonterminal-Namen (des Parsers)

   Ein Nonterminal-Name wird ohne besondere Kennzeichen notiert. Er muß mit einem Buchstaben beginnen und darf dann eine beliebige Folge aus Buchstaben, Ziffern, "-" und "_" enthalten.

2. Terminale
   - Terminal-Namen (des Parsers)

     Ein Terminal-Name wird ohne besondere Kennzeichen notiert. Er muß mit einem Buchstaben beginnen und darf dann eine beliebige Folge aus Buchstaben, Ziffern, "-" und "_" enthalten.

   - Literale

     Ein Literal kann wahlweise in einfache oder doppelte Anführungszeichen gesetzt werden oder im FrameMaker durch den Zeichentyp "Syn: Keyword" gekennzeichnet werden (s. Anhang 2). Diesem Format kann dann ein beliebiges Erscheinungsbild - üblicherweise fett oder unterstrichen - zugeordnet werden.

     Innerhalb der einfachen Anführungszeichen ist jedes druckbare Zeichen außer dem Zeilenendezeichen und einem einfachen Anführungszeichen erlaubt. Innerhalb der doppelten Anführungszeichen ist jedes druckbare Zeichen außer dem Zeilenendezeichen und einem doppelten Anführungszeichen erlaubt.

3. Epsilon

   ε beschreibt das leere Wort.

Beispiel:

Die ursprüngliche BNF-Regel

<program> ::= PROGRAM <id> ; <declarations> <compound-statement> END .

kann also wie folgt notiert werden:

```
program        "→"  PROGRAM id ";"
                      declarations
                      compound-statement
                    END "." .
```

3.2 Erweiterte Backus-Naur-Form (EBNF)

Oft wird die BNF durch zusätzliche Konstrukte erweitert, um bei einer Sprachdefinition an Lesbarkeit zu gewinnen. Die zusätzlichen Konstrukte werden unter Zuhilfenahme folgender Metasymbole notiert: "|", "[", "]", "{", "}", "(", ")", "*", "+".

Die erweiterte BNF gewinnt nicht an Ausdruckskraft; die neuen Konstrukte dienen nur der kürzeren Notation oder der besseren Lesbarkeit. Es sind im einzelnen:

1. Alternative

   In der BNF können mehrere Produktionen angegeben werden, um das gleiche Nonterminal abzuleiten. In der EBNF werden diese Regeln zu einer neuen Regel zusammengefaßt und

die verschiedenen rechten Seiten durch senkrechte Striche "|" getrennt. Damit entsprechen die BNF-Regeln

```
A       ::=   B C
A       ::=   "d"
A       ::=   ε
```

der EBNF-Regel

```
A       =     B C | "d" | ε .
```

Die Alternative hat die schwächste Bindung von allen EBNF-Konstrukten.

2. Gruppierung

In der BNF benötigt man zwei Regeln, um eine Alternative zu notieren. So benötigt man die folgenden zwei Regeln, um die Sätze "a b d" oder "a c d" einer Sprache zu beschreiben

```
A       ::=   "a" "b" "d"
A       ::=   "a" "c" "d"
```

In der EBNF hat die Alternative eine schwächere Bindung als eine Folge von Symbolen, deshalb benötigt man für das obige Problem immer noch zwei Regeln:

```
A       =     "a" B "d" .
B       =     "b" | "c" .
```

Um beliebig andere Bindungen erzwingen zu können, kann man die runden Klammern ("(" und ")") zur Gruppierung verwenden. Damit läßt sich die obige Sprache in nur einer Regel lesbar formulieren:

```
A       =     "a" ( "b" | "c" ) "d" .
```

3. Option

In einer Programmiersprache kommt es oft vor, daß ein Teil eines Satzes optional stehen kann, d. h. eine Symbolfolge ist sowohl mit als auch ohne diesen Teilsatz ein Satz der Sprache. Eine abkürzende Schreibweise dafür stellen die eckigen Optionsklammern ("[" und "]") zur Verfügung. Damit entsprechen die BNF-Regeln

```
A       ::=   B C
A       ::=   ε
```

der EBNF-Regel

```
A       =     [ B C ] .
```

Die Optionsklammern bilden automatisch eine Gruppe aus den eingeklammerten Symbolen.

4. Optionale Wiederholung: 0 bis N

In Programmiersprachen ist es häufig notwendig, Symbolfolgen von unbeschränkter Länge zu notieren, z. B. Listen von Bezeichnern. Diese müssen in der BNF mit Hilfe von Rekursion beschrieben werden:

```
<A>     ::=   a <A>
<A>     ::=   ε
```

```
<B>          ::=   a b <B>
<B>          ::=   ε
```

Die Regel A definiert die Sätze ε, a, a a, a a a, etc., die Regel B definiert die Sätze ε, a b, a b a b, a b a b a b, etc. Da die Wiederholung durch Rekursion nicht immer auf den ersten Blick ersichtlich ist, stehen in der EBNF verschiedene Konstruktoren für die Notation von 0- bis n-maliger Wiederholung zur Verfügung. Es sind dies der Kleene-Stern "*" sowie die geschweiften Klammern ("{" und "}"). Der Kleene-Stern bezieht sich auf das direkt vorangestellte Symbol (das kann auch eine Gruppe von Symbolen sein, s. Punkt 2), die geschweiften Klammern fassen üblicherweise mehrere Symbole zusammen, die als Gruppe wiederholt werden können. Somit können die obigen Regeln wie folgt notiert werden:

```
A            =     "a" * .
B            =     { "a" "b" } .
```

alternativ:

```
B            =     ( "a" "b" )* .
```

Die geschweiften Klammern bilden automatisch eine Gruppe aus den eingeklammerten Konstrukten.

5. Obligatorische Wiederholung: 1 bis N

Die ein- bis n-malige Wiederholung von Symbolen wird in der BNF wiederum durch Rekursion ausgedrückt. So definiert die Regel A die Sätze a, a a, a a a, etc., die Regel B die Sätze a b, a b a b, a b a b a b, etc.:

```
<A>          ::=   a <A>
<A>          ::=   a
<B>          ::=   a b <B>
<B>          ::=   a b
```

Wiederum stehen in der EBNF verschiedene Konstruktoren für die Notation von 1- bis n-maliger Wiederholung zur Verfügung. Es sind dies das Pluszeichen "+" sowie die geschweiften Klammern mit angehängtem Plus ("{" und "}+"). Das Pluszeichen bezieht sich auf das direkt vorangestellte Symbol (das kann auch eine Gruppe von Symbolen sein, s. Punkt 2), die geschweiften Klammern fassen üblicherweise mehrere Symbole zusammen, die wiederholt werden können. Somit können die obigen Regeln wie folgt notiert werden:

```
A            =     "a" +.
B            =     { "a" "b" }+ .
```

alternativ:

```
B            =     ( "a" "b" )+ .
```

Auch hier bilden die geschweiften Klammern automatisch eine Gruppe aus den eingeklammerten Konstrukten.

6. Wiederholung mit Trennelement

Oft werden in Programmiersprachen Gruppen von Symbolen mit einem speziellen Zeichen getrennt, z. B. kann in einer Folge von Anweisungen das Semikolon als Trennzeichen verwendet werden. Allgemein kann man dies in der BNF etwa so formulieren:

```
<A>          ::=   "a"
<A>          ::=   "a" "b" <A>
```

Hier ist nicht leicht ersichtlich, daß eine Liste von a's getrennt durch b's als gültiger Satz definiert wird. Günstiger ist wiederum die Darstellung mittels geschweifter Klammern ("{" und "}") als Iterationskonstrukt und der Angabe eines Listentrennelements innerhalb dieser Klammern, das durch zwei (senkrechte oder schräge) Striche ("||" oder "//") abgetrennt wird. Die obige Syntaxregel läßt sich dann wie folgt in EBNF definieren:

```
A            =     { "a" // "b" } .
```

Die geschweiften Klammern und das Trennzeichen bilden automatisch zwei Gruppen aus den eingeklammerten Konstrukten: die erste Gruppe bilden die Symbole zwischen der öffnenden Klammer und dem Trennzeichen, die zweite Gruppe bilden die Symbole zwischen dem Trennzeichen und der schließenden Klammer.

Selbstverständlich können nicht nur Symbole in die verschiedenen EBNF-Konstrukte geschachtelt werden, sondern auch die verschiedenen EBNF-Konstrukte ineinander.

Im Unterschied zur BNF ist es nicht erlaubt, mehrere Regeln gleichen Namens anzugeben. Diese müssen in einer Regel als Alternativen formuliert werden.

Für die zusätzlichen Metasymbole der EBNF muß eine Bindungspriorität angegeben werden. Die verschiedenen Klammerarten (rund, eckig und geschweift) dienen der Gruppierung der in ihnen eingeschlossenen Konstrukte. Der Kleene-Stern und das Pluszeichen haben geringere Priorität und beziehen sich immer auf das unmittelbar davor stehende Konstrukt. Die geringste Priorität hat die Alternative; sie bindet schwächer als die Sequenz, die ohne Operator notiert wird. Diese schwache Bindung der Alternative entspricht der ursprünglichen Notation in der BNF, bei der Alternativen auf mehrere Regeln verteilt werden mußten.

3.3 Zuordnung eines Lexems zu einem Tokentag

Es ist unter Umständen notwendig, in einer Sprache neue Terminale der Sprache selbst zu definieren. Beispiele dafür sind die Neudefinition von Operatoren, die in Infix-Darstellung notiert werden sollen, oder die Typnamen-Definition wie in ANSI C [KernighanRitchie] vorgesehen. Es handelt sich dann nicht mehr um eine kontext-freie sondern um eine kontext-sensitive Sprache, jedoch ist die Abhängigkeit vom Kontext stark eingeschränkt. Meist reicht es aus (wie in den Beispielen oben), sich auf eine andere Produktion zu beziehen, in der ein neues *Lexem* vereinbart und diesem ein bestimmtes *Tokentag* zugeordnet wird.

Unsere Definition der EBNF bietet eine Notation für diesen speziellen Fall. Dazu werden zwei neue Metasymbole eingeführt: ";" und "->". Das Semikolon ";" trennt die eigentliche EBNF-Regel von der Zuordnung Lexem zu Tokentag. Der Pfeil "->" gibt an, welches Lexem welchem neuen Tokentag zugeordnet werden soll. Lexem wie Token-Tag werden durch Namen bezeichnet. Der Terminal-Name, der das Lexem bezeichnet, muß in der aktuellen Regel eindeutig definiert sein!

Zum Beispiel kann eine Operatorneudefinition wie folgt notiert werden. Gegeben sei die folgende Scanner-Regel, die den (Parser-) Terminal-Namen "AddOp" definiert:

```
SC: AddOp        =     "+" | "-" .
```

Die folgende EBNF-Regel gibt an, wie ein neuer Operator definiert wird:

```
OperatorDefinition =  OP-DEF OpString ParamList IS RoutineBody END ;
                      OpString -> AddOp.
OpString           =  '"' (ANY - {'"'})* '"'.
```

Kommt beim Parsen eines konkreten Programmms die Regel *OperatorDefinition* zur Anwendung, so liefert der Parser das gewohnte Token. Zusätzlich wird das Lexem, welches *OpString* zugeordnet wurde, als Erweiterung der Regel *AddOp* eingetragen.

Tritt im folgenden Programmtext das gefundene Lexem für *OpString* auf, so wird dies behandelt, als sei es Teil der Regel *AddOp,* d. h. die Regel *AddOp* kommt zur Anwendung und als Ergebnis liefert der Parser das Tokentag von *AddOp* und das erwähnte Lexem.

Es bleibt Aufgabe der semantischen Analyse, die korrekte Verwendung als Operator zu überprüfen. Jedoch ist auf diese Art und Weise die Verwendung als Infix-Operator möglich.

Anhang A: EBNF-Syntax

In diesem Anhang ist die Syntax der EBNF durch eine vereinfachten Form der EBNF selbst beschrieben. Die Syntax gibt die genaue Definition aus Kapitel 3 wieder und entspricht der Definition der Arbeiten von G. Kühn [Kühn] und M. Teichmann [Teichmann]. Die unten angegebenen (Parser-) Regeln entsprechen den folgenden Vereinbarungen:

- Die zuerst angegebene Regel definiert das Startsymbol der Grammatik.
- Eine Regel beginnt mit einem Namen, gefolgt von einem Gleichheitszeichen "=" und einem EBNF-Konstrukt, und endet mit einem Punkt ".".
- Zeichen bzw. Zeichenfolgen werden in einfachen oder doppelten Anführungszeichen notiert.
- Namen bestehen aus einer nichtleeren Folge von Buchstaben. Namen von Nonterminalen werden gemischt aus Groß- und Kleinschrift notiert; sie beginnen mit einem Großbuchstaben. Namen von Terminalen werden ausschließlich mit Großbuchstaben notiert.
- Runde Klammern ("(" und ")") gruppieren die enthaltenen Konstrukte.
- Optionale Konstrukte werden in eckige Klammern ("[" und "]") eingeschlossen.
- Optionale Wiederholungen (0-N) werden durch Anfügen von "*" gekennzeichnet.
- Obligatorische Wiederholungen (1-N) werden durch Anfügen von "+" gekennzeichnet.
- Alternative Konstrukte werden durch "|" getrennt.

```
Grammar          =   { ScannerRule | CommentRule | AsciiComment }
                     ParserRule
                     { ParserRule | ScannerRule | CommentRule
                     | AsciiComment } .

ParserRule       =   "P:" ["r:"] ["->>"]
                     Nonterminal ("=" | "::=" | "⟶") ParserExpression
                     [";" Terminalname "->" (Terminalname | Nonterminal)] "." .
ParserExpression =   Term ("|" Term)* .
Term             =   Factor+ | "ε" | "eps:" .
Factor           =   Unit ["*" | "+"] .
Unit             =   Compound | Option | Repetition | LITERAL
                 |   Terminalname | Nonterminal .
Compound         =   "(" ParserExpression ")".
Option           =   "[" ParserExpression "]".
Repetition       =   OptionalRepetition
                 |   RequiredRepetition
                 |   SeparatedRepetition .
OptionalRepetition = "(" ParserExpression ")*"
                 |   "{" ParserExpression "}*"
                 |   "{" ParserExpression "}" .
```

```
RequiredRepetition =    "(" ParserExpression ")+"
                    |   "{" ParserExpression "}+" .
SeparatedRepetition=    "{" ParserExpression ("||" | "//") ParserExpression "}" .
Terminalname       =    IDENT .
Nonterminal        =    IDENT .

ScannerRule        =    "S:" IDENT ("=" | "::=") ScannerExpression
                        [ "/" ScannerExpression ]
                        [ "~>" ResultForm ] "." .
ScannerExpression  =    Sequence { "|" Sequence } .
Sequence           =    { BasicExpression Postfix* }+ .
BasicExpression    =    CharacterLiteral
                    |   LITERAL
                    |   (Set | IDENT) [ "-" (Set | IDENT) ]
                    |   "(" ScannerExpression ")" .
CharacterLiteral   =    Character [".." Character ] .
Set                =    "{" { CharacterLiteral | LITERAL | IDENT } "}" .
Postfix            =    "?" | "*" | "+" .
ResultForm         =    CharOrLiteral [ "$" [ CharOrLiteral ] ]
                    |   "$" [ CharOrLiteral ] .
Character          =    ( SQCHARACTER | DQCHARACTER )
                    |   NUMBER .
CharOrLiteral      =    Character | LITERAL .

CommentRule        =    "C:" Ident ("=" | "::=") ScannerExpression
                        [ "/" ScannerExpression ] "." .

AsciiComment       =    ASCII_COMMENT .
```

Eine EBNF-Grammatik muß mindestens eine Parserregel enthalten. Die mit "->>" markierte Regel ist die Startregel. Gibt es keine so markierte Regel, wird die zuerst angegebene Regel als Startregel angesehen. Gibt es mehrere mit "->>" markierte Regeln, ist das ein Fehler. Definitionen von rechtsassoziativen Operatoren werden durch ein zusätzliches "r:" gekennzeichnet.

Jeder Name muß durch genau eine Regel definiert sein.

Im folgenden sind die Terminale des Parsers angegeben. Sie sind als Scanner-Regeln notiert und entsprechen den folgenden Vereinbarungen:

- Vordefinierte Namen sind LETTER (Menge der Groß- und Kleinbuchstaben), DIGIT (Menge der Ziffern 0 bis 9) und ANY (Menge aller Zeichen mit Ausnahme des Zeilenendezeichens).
- Mengen von Zeichen werden in geschweifte Klammern { } eingeschlossen.

- Die Mengendifferenz wird durch das Minuszeichen "-" ausgedrückt.

```
IDENT          =   LETTER (LETTER | DIGIT | "-" | "_")* .
LITERAL        =   '"' (ANY - {"})+ '"' | "'"(ANY - {'})+ "'" .
SQCHARACTER    =   \' -{\n \'} \' .
DQCHARACTER    =   \" -{\n \"} \" .
NUMBER         =   DIGIT+ .
ASCII_COMMENT=     "--" ANY*
```

Anhang B: Notation von Grammatiken

Wie bereits oben erläutert werden die Grammatiken für Programmiersprachen mittels BNF und EBNF notiert. Scanner- und Parsergeneratoren akzeptieren diese Beschreibung in modifizierter Form bedingt durch benötigte Zusatzinformation (semantische Aktionen). Jedoch muß die Eingabe für diese Generatoren mit Hilfe von ASCII-Zeichen notiert werden. Die Lesbarkeit dieser Notation ist fraglich.

Weiter möchte man Zusatzinformation für den Leser unterbringen (z. B. Kennzeichnung der Regeln für den Parser- und Scannergenerator) bzw. besser lesbar darstellen oder eine Grammatik durch erklärenden Text, beispielsweise in einem Sprachreport, unterbrechen. Wünschenswert ist es z. B, Schlüsselwörter in der Grammatikbeschreibung im Report besonders zu kennzeichnen (durch Fettschrift oder Unterstreichung), während Generatoren an dieser Stelle Anführungsstriche verlangen. Das erfordert oftmals eine redundante Repräsentation mit der Folge, daß mehrere Dokumente gewartet und konsistent gehalten werden müssen. Der berechtigte Wunsch nach einer einheitlichen Darstellungsform entsteht.

Die geschilderte Situation enthält mehrere Problematiken. Zum einen muß ein Editor gefunden werden, in dem spezielle Formatierungen wie Fettdruck und Unterstreichung und Sonderzeichen, die außerhalb des ASCII-Zeichenvorrats liegen, möglich sind. Zum anderen müssen Bereiche als wichtig oder unwichtig markiert werden können. Hierzu eigenen sich Textverarbeitungssysteme mit symbolischer Formatverwaltung für Zeichen und Absätze. Da die Generatoren nicht um- bzw. neugeschrieben werden sollen, muß ein Übersetzer eingesetzt werden, der diese Formatierungen in ASCII-Format umwandelt und die wichtigen Bereiche aus einem Dokument filtern kann.

Anhang B.1: Repräsentation im FrameMaker

In der Arbeitsgruppe Programmiersprachen und Compiler wird als Textverarbeitungsprogramm FrameMaker eingesetzt, welches die erwünschten Eigenschaften in vollem Umfang unterstützt. Dieses Programm arbeitet absatz- und zeichenorientiert, d. h. es ist möglich, Absatz- und Zeichentypen zu benennen und diesen Typen bestimmte Formatierungseigenschaften zuzuordnen. Dabei sind die Eigenschaften eines Absatzes unabhängig von seinem Namen wählbar, während alle Absätze eines Namens das gleiche äußere Erscheinungsbild zeigen. Das gleiche gilt für Zeichentypen.

Die für einen Übersetzergenerator wichtigen Bereiche können entsprechend markiert werden, so daß es möglich ist, ausschließlich anhand der Namen von Absatz- und Zeichentypen zu entscheiden, ob eine Syntaxregel, eine Scanner-Regel, ein Kommentar oder sonstiger Text vorliegt. Diese wichtigen Bereiche werden automatisch aus einem Dokument extrahiert und in ein geeignetes ASCII-Format transformiert.

Basierend auf dem ASCII-Austauschformat des FrameMakers (MIF) wurde ein Übersetzer entwickelt [Vorwieger], der die geforderten Formattransformationen und Filterungen gesteuert durch Regeldateien durchführt. Diese Regeln geben die Übersetzungen von FrameMaker-Formaten in die gewünschten ASCII-Formate an, wobei alle nicht aufgeführten Absatztypen ignoriert werden. Die Regeln für Zeichentypen beziehen sich nur auf angegebene Absatztypen.

Speziell für die Darstellung von Grammatiken für Scanner- und Parsergeneratoren sind für die Darstellung im FrameMaker verschiedene Namen von Absatz- und Zeichentypen vorgegeben. Diese können aus einem leeren Dokument, einem sog. Template, übernommen werden. Allen vordefinierten Namen ist die Zeichenkette "Syn: " vorangestellt.

Zu den im folgenden beschriebenen Absatzformaten existiert eine Regeldatei mit den entsprechenden Transformationsregeln.

Anhang B.1.1: Absatzformate

1. Syn: ParserTitle

Dieses Absatzformat ist als einleitender Absatz für die Parser-Regeln gedacht. Er erzeugt eine grafische Überschrift ("Parser") und wird ansonsten interpretiert wie der Absatz *Syn: Parser* (s. Punkt 2). Daher macht seine Anwendung nur Sinn, wenn alle Parser-Regeln in einem Block zusammengefaßt sind oder jedesmal daraufhingewiesen werden soll, daß es sich bei den darauffolgenden Regeln um Angaben für den Parser handelt. Ansonsten verwendet man nur die Absatzformate *Syn: Parser* und *Syn: Rassoc Parser*. Ein Beispiel für einen solchen Absatz ist:

PARSER

```
program          =  { fct_def | class_def | method_def }.
```

2. Syn: Parser

Diesen Namen bekommen alle linksassoziativen Syntaxregeln (bis auf die Überschriftsabsäze, *Syn: ParserTitle*). Beispiel für einen solchen Absatz:

```
fct_def          =    ( FCT ID bound_vars IS clause_sequ END-FCT )
                 |    ( FCT ID bound_vars IS let END-FCT ).
```

3. Syn: Rassoc Parser

Wird in der Grammatik ein rechtsassoziativer Operator beschrieben, so soll dieser von den anderen (linksassoziativen) Operatoren unterschieden werden. Das kann durch einen besonderen Schrifttyp oder durch ein vorangestelltes Zeichen (s. unten) dokumentiert werden. Beispiel für einen solchen Absatz:

```
ℜ: pot_ex        =    factor { potop factor }.
potop            =    "^".
```

4. Syn: OperatorTitle

Eine alternative Schreibweise für die Notation von Operatoren, ihre Assoziativität und Bindungsstärke kann eine Operatortabelle darstellen. Das Absatzformat *Syn: OperatorTitle* leitet eine Operatortabelle ein. Für die sinnvolle Anwendung gilt dasselbe wie für Punkt 1. Operator-Regeln werden ansonsten mit den Absatz *Syn: Operator* oder *Syn: Rassoc Operator* bezeichnet. Beispiel:

OPERATOR

```
Op1 = "*" | "/".
```

5. Syn: Operator

   Diesen Namen bekommen die linksassoziativen Einträge der Operatortabelle. Dieses Format zeichnet sich nicht durch eine besondere Form aus.

6. Syn: Rassoc Operator

   Diesen Namen erhalten die rechtsassoziativen Einträge der Operatortabelle. Dieser Absatz zeichnet sich nicht durch ein besonderes Format aus.

7. Syn: ScannerTitle

   Dieser Absatz kann die Definition der Scanner-Regeln einleiten. Für die sinnvolle Anwendung gilt dasselbe wie für Punkt 1. Scanner-Regeln sind ansonsten mit *Syn: Scanner* oder *Syn: Comment-Def* zu kennzeichnen.

   Beispiel für den einleitenden Absatz für Scanner-Regeln:

SCANNER

```
SC: alpha       =    letter | "@" | "%" | "&".
```

8. Syn: Scanner

   Diesen Namen bekommen Scanner-Regeln, die keine Kommentare spezifizieren. Ein der Regel vorangestelltes "SC: " ist Teil des Formates für Scanner-Regeln. Beispiel:

```
SC: STRG        =    '"' ANY - { '"' }* '"'.
```

9. Syn: Comment-Def

   Mit diesem Namen werden Ausdrücke bezeichnet, die die Notation von Kommentaren beschreiben. Die Beschreibung eines Kommentars, der mit "--" eingeleitet wird und bis zum Zeilenende geht, wird beispielsweise folgendermaßen beschrieben:

```
Comment = "--" ANY*.
```

10. Syn: ASCII-Comment

    Mit diesem Namen werden spezielle Absätze eingeleitet, die Benutzer-Kommentare innerhalb der Grammatik enthalten. Bei einer Konvertierung der FrameMaker-Repräsentation in eine äquivalente ASCII-Repräsentation (s. Anhang B.2) wird der Inhalt dieser Absätze übernommen und wiederum besonders als Benutzer-Kommentar gekennzeichnet.

Anhang B.1.2: Die Zeichenformate

Zeichenformate werden bei der Transformation nur berücksichtigt, wenn sie auf Zeichen angewendet werden, die Teil eines Absatzes sind, für den ebenfalls eine Transformationsregel angegeben ist.

Im Gegensatz zu den Absatztransformationen, werden Zeichenketten mit Zeichenformaten, für die keine Transformationsregeln existieren, unverändert (d. h. ohne Transformation) übernommen.

1. Syn: StartNT

   Das mit diesem Format gekennzeichnete Nonterminal bezeichnet das Start-Nonterminal der Grammatik, falls nicht die erste Syntaxregel verwendet werden soll. Dieses Format hat kein besonderes äußeres Erscheinungsbild.

2. Syn: Keyword

   Schlüsselworte der Grammatik können in doppelte Anführungszeichen gesetzt werden oder mit diesem Zeichenformat verbunden werden. Im Beispiel ist **NOT** mit dem Zeichenformat *Syn: Keyword* markiert und wird in "NOT" umgesetzt:

   ```
   monop        =    NOT | "+" | "-".
   ```

3. Syn: Comment

   Dieses Zeichenformat dient der Kennzeichnung von Kommentaren in Regelabsätzen. Da Kommentare nicht zu den eigentlichen Regeln gehören, d. h. auch nicht als Teil der Regel interpretiert werden dürfen, müssen sie ignoriert werden. Jede als Kommentar markierte Zeichenkette wird nicht als ein Teil der Regel angesehen. Das Zeichenformat sollte sich vom Rest der Regel absetzen, um die Zeichenkette als Kommentar erkennen zu können. Der Kommentar könnten im Dokument z. B. kursiv erscheinen:

   ```
   constr       =    "<" sum_ex_list [ "|" expr ] ">"
                |    ID "." "(" attr_val_sequ ")".      "[" ... "]" für Maps
   ```

   Bemerkung: Durch die Markierung mit einem Zeichenformat ist einen weitere Markierung durch ein spezielles kommentar-einleitendes Symbol nicht mehr notwendig.

4. Syn: Symbol Set

   Manchmal werden Zeichen aus einem anderen als dem Standard-Zeichsatz zur Notation von EBNF-Regeln benötigt, z. B. das Epsilon "ε" für die leere Produktion. Diese stammen üblicherweise aus dem Symbol-Zeichensatz und werden durch diesen Zeichentyp gekennzeichnet. Beispiel:

   ```
   Anweisungsfolge  =    Anweisung Anweisungsfolge | ε .
   ```

Anhang B.2: Regeln zur Formatumsetzung

Die im folgenden aufgelisten Regeln zur Formatwandlung von Zeichen und Absätzen sind folgendermaßen zu interpretieren:

Jede Regel besteht aus zwei Zeilen. Die erste Zeile spezifiziert zunächst den Regeltyp:

- *P:* Diese Regel definiert die Formatumwandlung für Absätze ("**p**aragraph format").
- *C:* Diese Regel definiert die Formatumwandlung für Zeichen ("**c**haracter format").
- *SC*: Diese Regel definiert die Formatumwandlung für spezielle Zeichen innerhalb eines Zeichenformates ("**s**pecial characters within a **c**haracter format").

Der zweite Teil der ersten Zeile gibt den Namen des Formats an. Die eigentliche Angabe für die Formatumwandlung steht in der zweiten Zeile.

Für *P*:- und *C*:-Regeln besteht diese aus einer Folge von Zeichenketten, deren Konkatenation das Ergebnis der Formatwandlung ist. Die Zeichenkette, die mit dem Format markiert ist, wird mit <string> angegeben. Jede weitere Zeichenkette wird in einfache oder doppelte Hochkommata geschrieben.

Der Regeltyp *SC:* unterscheidet eine linke und eine rechte Seite, getrennt duch einen Ableitungspfeil "->". Die rechte Seite beschreibt die Zeichenkette in Hochkomma, durch die das Zeichen der linken Seite ersetzt werden soll. Die Angabe der linken Seite erfolgt in der ASCII-Repräsentation des FrameMaker (MIF).

Näheres s. [Vorwieger].

Anhang B.2.1: Die Absatzformatregeln

P: Syn: ParserTitle
"P: " <string>

P: Syn: Parser
"P: " <string>

P: Syn: Rassoc Parser
"P:r: " <string>

P: Syn: OperatorTitle
"O: " <string>

P: Syn: Operator
"O: " <string>

P: Syn: Rassoc Operator
"O:r: " <string>

P: Syn: ScannerTitle
"S: " <string>

P: Syn: Scanner
"S: " <string>

P: Syn: Comment-Def
"C: " <string>

P: Syn: ASCII-Comment
"-- " <string>

Anhang B.2.2: Die Zeichenformatregeln

C: Syn: StartNT
"->>" <string>

C: Syn: Keyword
'"' <string> '"'

C: Syn: Comment
""

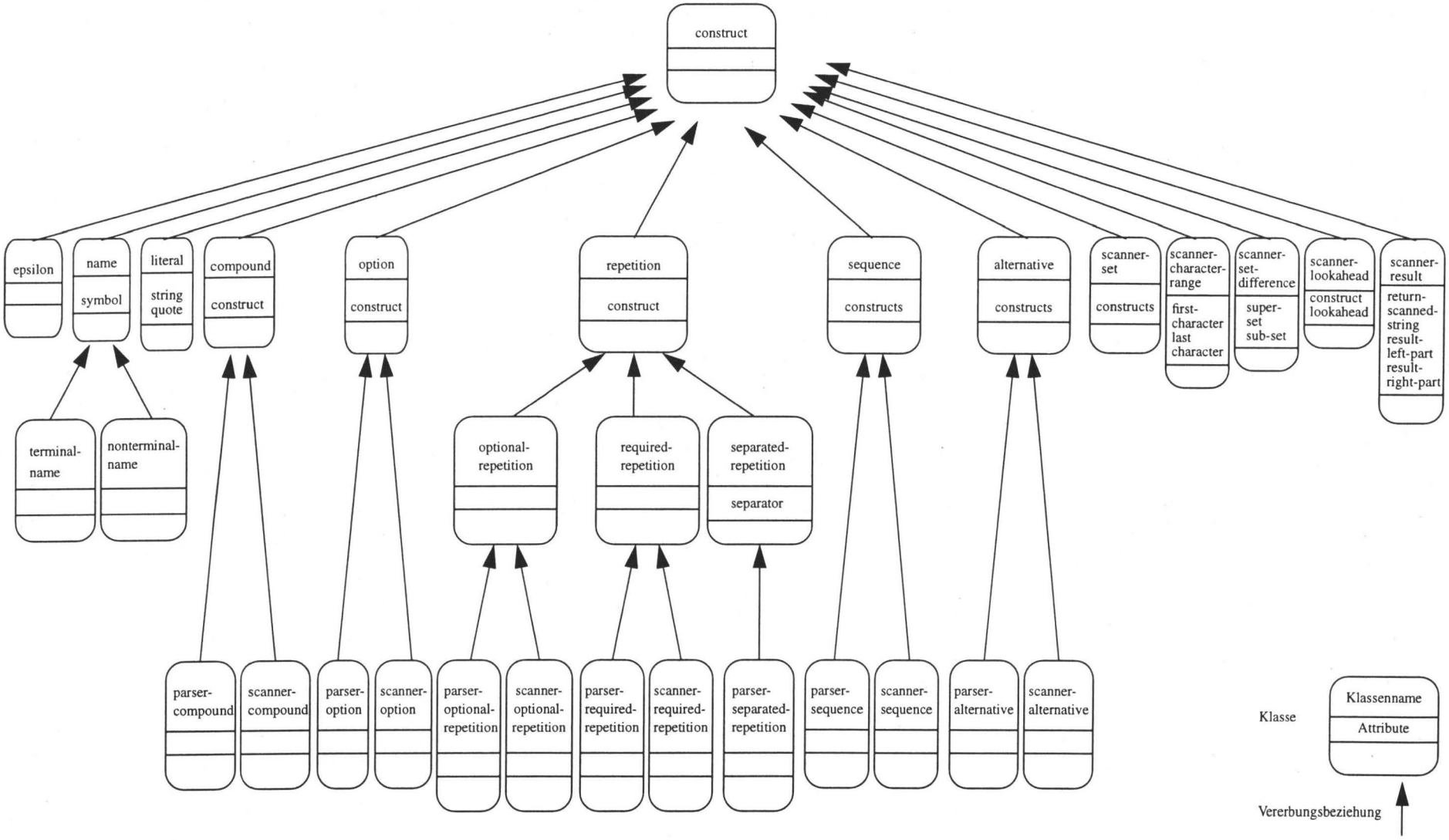
C: Syn: Symbol Set
'"' <string> '"'

SC: Syn: Symbol Set
e -> "eps:"

SC: Syn: Symbol Set
\xae -> "::="

Die letzte Regel beschreibt die Ersetzung des Zeichens → durch die Zeichenfolge "::=".

Anhang B.3: Objektorientierte Darstellung



Anhang C: Literatur

[AhoSethiUllman] A.V. Aho, R. Sethi, J.D. Ullman, *Compilerbau*. Addison-Wesley, 1988

[ANSI78] American National Standards Institute, Inc.: *Programming Language FORTRAN*, ANSI X3.9-1978, New York, 1978

[Avenhaus] Jürgen Avenhaus, *Einführung in die theoretische Informatik. Berechenbarkeit - Maschinen - Sprachen*. Script, Universität Kaiserslautern, FB Informatik, 1985

[Broy] Manfred Broy, *Informatik. Eine grundlegende Einführung*. Springer-Verlag Berlin [u.a.], 1992

[Chomsky] N. Chomsky, *Three Models for the Description of Language*. IRE Transactions on Information Theory IT-2, 113-124, 1956

[GoosWaite] G. Goos, W. M. Waite, *Compiler Construction*. Springer-Verlag New York Berlin Heidelberg Tokio, 1985

[Gries] David Gries, *Compiler Construction for digital Computers*. John Wiley & Sons, Inc New York [u.a.], 1971

[KernighanRitchie] B. W. Kernighan, D. M. Ritchie, *Programmieren in C; Zweite Ausgabe ANSI-C*, Carl Hanser Verlag München Wien, 1990

[KMA82] A. J. Kfoury, Robert N. Moll, Michael A. Arbib, *A programming approach to computability*. Springer-Verlag New York, 1982.

[Kühn] G. Kühn, *Entwurf und Implementierung eines Scannergenerators*. Projektarbeit Universität Kaiserslautern, 1994

[MaurerWilhelm] D. Maurer, R. Wilhelm, *Übersetzerbau*. Springer-Verlag Berlin Heidelberg New York, 1992

[Naur] P. Naur, *Revised Report on the Algorithmic Language ALGOL 60*. Communications of the ACM 6(1), 1-17, 1963

[Polak] W. Polak,
Compiler Specification and Verification. Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg New York, 1981

[Teichmann] M. Teichmann,
Entwurf und Implementierung eines Tools zur Analyse von Grammatiken. Projektarbeit Universität Kaiserslautern, 1994

[Vorwieger] S. Vorwieger,
FKT - ein Filter- und Formatkonvertierungs-Tool, Interner Bericht in Vorb., Universität Kaiserslautern, FB Informatik, 1994

[Wippermann] Hans-Wilm Wippermann,
Struktur und Implementierung imperativer Programmiersprachen. Script, Universität Kaiserslautern, FB Informatik, 1991

[Wirth 80] N. Wirth,
The Programming Language Modula-2. Report Nr. 36 ETH Zürich, Institut für Informatik, 1980

[Wirth84] Niklaus Wirth,
Compilerbau: eine Einführung. - 3., überarb. u. erw. Aufl. Teubner-Verlag Stuttgart, 1984