3

Wort

Am Anfang war das Wort.

In diesem Kapitel wiederholen wir vom Stoff von *Theoretische Informatik* gerade soviel von Chomsky-Grammatiken, wie für unser Thema wichtig ist, und betrachten dann kontextfreie Regeln mit Variablen. Diese sogenannten Zweistufen-Grammatiken erlauben es, auch kontext-sensitive Sprachen zu beschreiben und Parsierer dafür zu generieren. Wir werden zwei Arten von Zweistufen-Grammatiken kennenlernen: van-Wijngaarden-Grammatiken und Attribut-Grammatiken.

3.1 Chomsky-Grammatiken

Formale Sprachen werden als Worte über einem endlichen *Vokabular* V definiert, dessen Elemente *Symbole* genannt werden. V^* bezeichnet die (endlichen) *Wörter* über V. Wörter werden einfach durch Hintereinanderschreiben von Symbolen gebildet ("*Juxtaposition*"). Wenn a, b und c Symbole aus V sind, so ist also $abcb$ ein Wort aus V^*; die Verkettungsoperation auf Symbolen und Wörtern bleibt also implizit. Das *leere Wort* wird mit $\Box$ bezeichnet. Die *Länge* eines Wortes w gibt die Anzahl der in ihm enthaltenen Symbole an und wird als $|w|$ geschrieben; also ist $|abcb| = 4$ und $|\Box| = 0$.

In Chomsky-Grammatiken wird dem als *terminal* angesehenen Vokabular V ein *nichtterminales Vokabular* N mit $N \cap V = \{\}$ hinzugefügt.[1]

Regeln sind Paare $\langle \alpha, \beta \rangle$ von Wörtern aus $(N \cup V)^*$, wobei die linke Seite α mindestens ein Nichterminal enthält. Eine Grammatik ist ein Tupel $\Gamma = \langle V, N, R, Z \rangle$ wobei R eine endliche *Regelmenge* und $Z \in N$ das *Startsymbol* ist. Direkte Ableitungen sind definiert mit

[1] Nichtterminale werden in der Literatur manchmal *syntaktische Variablen* genannt. Wir vermeiden diese Terminologie, weil wir Nichtterminale später mit (Kontext-) Variablen parameterisieren wollen.

$$\frac{\gamma, \delta \in (N \cup V)^*, \langle \alpha, \beta \rangle \in R}{\gamma\alpha\delta \Rightarrow_R \gamma\beta\delta}$$

D. h., Ableitungen bilden den Abschluss der Regelmenge bezüglich der Verkettung mit Präfixen γ und Suffixen δ. Wir benutzen die übliche Notation für die Abschlüsse der Ableitungen $\Rightarrow_R$ und definieren die *Sprache* der Grammatik Γ als

$$L_\Gamma = \{w \in V^* \mid Z \Rightarrow_R^* w\}$$

Beim Umgang mit Chomsky-Grammatiken verwenden wir folgende Konventionen:

- Großbuchstaben A, B, C, D und Z stehen für Nichtterminale, insbesondere bezeichnet Z das Startsymbol einer Grammatik. (In konkreten Beispielen verwenden wir manchmal Bezeichner wie "*name*" oder "$\langle$*name*$\rangle$".)
- Kleinbuchstaben a, b, c, d, e stehen für Terminale. (In konkreten Beispielen verwenden wir manchmal unterstrichene Zeichenketten wie `id` oder `[`.)
- Griechische Buchstaben α, β, γ und δ stehen für Wörter aus nichtterminalen und terminalen Symbolen.
- Kleinbuchstaben u, v, w, x, y und z stehen für terminale Wörter.
- Regeln $\langle \alpha, \beta \rangle$ schreiben wir als $\alpha \to \beta$. (In konkreten Beispielen benutzen wir manchmal die Notation der Backus-Naur-Form. Dann steht "$\alpha ::= \beta_1 \mid \cdots \mid \beta_n$" für n Regeln $\alpha \to \beta_i$.)

Wir verwenden folgende Operationen auf Sprachen $L, L' \subseteq V^*$:

- $L \cup L'$ ist die Vereinigung von Sprachen.
- $L \cap L'$ ist der Durchschnitt von Sprachen.
- $\bar{L} = \{w \in V^* \mid w \notin L\}$ das Komplement einer Sprache.
- $LL' = \{ww' \mid w \in L, w' \in L'\}$ bezeichnet die Verkettung von Sprachen.
- Für $n \geqslant 0$ bezeichnet $L^n = LL^{n-1}$ die n-fache Wiederholung einer Sprache; $L^1 = L$ und L^0 ist die *leere Sprache* $\{\}$, nicht zu verwechseln mit der einelementigen Sprache $L_\square = \{\square\}$, die nur das leere Wort enthält.[2]
- $L^+ = \bigcup_{n>0} L^n$ und $L^* = L^+ \cup \{\}$ bezeichnen den transitiven bzw. transitiv-reflexiven Abschluss der Sprache L.

Darüber hinaus benutzen wir Abkürzungen wie a^n für die Sprache $\{a^n \mid n > 0\}$, $a^n b^n$ für die Sprache $\{a^n b^n \mid n > 0\}$, $a^n b^n c^n$ für die Sprache $\{a^n b^n c^n \mid n > 0\}$.

3.1.1 Die Chomsky-Hierarchie

Die Regeln von Wortgrammatiken können verschieden stark eingeschränkt werden. Dies führt zu folgender *Chomsky-Hierarchie* von Grammatiken mit den Typen 0 bis 3. (Rechts stehen die Namen der Automaten, die Sprachen des entsprechenden Typs erkennen können.)

[2] L_0 ist neutral bzgl. Vereinigung, und $L_\square$ bzgl. Verkettung von Sprachen. Es gilt $LL_0 = L_0$, aber $L \cup L_\square = L$ gilt nur, wenn $\square \in L$.

Typ	Regeln	Sprachen	Erkennung
0	$\alpha \to \beta$	rekursiv aufzählbar	Turing-Maschine
1	$\lvert\alpha\rvert \leqslant \lvert\beta\rvert$	kontextsensitiv	linear beschränkter A.
2	$\alpha = A$	kontextfrei	Keller-A.
3	$\beta = a$ oder $\beta = bB$	regulär	endlicher A.

Eine Grammatik ist vom Typ $i \in \{0, 1, 2, 3\}$, wenn alle ihre Regeln vom Typ i sind. Eine Sprache ist vom Typ i, wenn sie von einer Grammatik des Typs i erzeugt werden kann. Die Klassen von Sprachen des Typs i enthalten die Sprachen jedes höheren Typs echt; das gilt auch für die Regeln verschiedenen Typs, abgesehen von dem Sonderfall, dass die kontextfreie Regel $A \to \square$ nicht kontext-sensitiv ist; man kann jedoch zeigen, dass für die Generierung kontextfreier Sprachen höchstens eine leere Startregel $Z \to \square$ benötigt wird, so dass alle anderen Regeln die Größenbeschränkung der kontextsensitiven Regeln erfüllen.

Einige Fragestellungen aus der Theorie der formalen Sprachen sind auch für uns relevant.

Äquivalenz. Zwei Grammatiken heißen *äquivalent* wenn sie die gleiche Sprache generieren. Manchmal möchte man eine Grammatik *transformieren*, d.h., in eine äquivalente Grammatik umformen. Tritt in einer Grammatik ein Nichtterminal N nicht auf einer rechten Regelseite auf, kann man alle Regeln, wo N in der linken Seite auftaucht, entfernen. Die so entstandene Grammatik definiert immer noch dieselbe Sprache.

Normalformen. Eingeschränkte Formen von Regeln sind unter Umständen einfacher zu studieren als der allgemeine Fall. Wenn die Regeln aller Grammatiken einer Grammatikklasse sich in eine bestimmte äquivalente Form bringen lassen, nennt man dies eine *Normalform*. In Chomsky-Form haben beispielsweise alle Regeln entweder die Form $A \to a$ oder $A \to BC$. Dies ist eine Normalformen für kontextfreie Regeln.

Entscheidbarkeitsprobleme. Für (Klassen von) Grammatiken möchte man wissen, ob eine bestimmte Eigenschaft P allgemein, also für alle Grammatiken der Klasse *entscheidbar* ist, d.h., sich mit einem Algorithmus bestimmen lassen. Z. B. könnte die Eigenschaft P sein: "*Ist die Sprache der Grammatik leer?*"

Abgeschlossenheit. Wir haben einige Operationen auf Sprachen kennengelernt (Vereinigung, Verkettung). Oft ist man interessiert zu wissen, ob eine Klasse von Sprachen unter so einer Operation abgeschlossen ist, d.h., ob das Operationsergebnis immer in derselben Sprachklasse ist. So könnte man sich fragen, für welchen Typ i von Chomsky-Sprache L auch ihr Komplement $\bar{L}$ vom Typ i ist.

3.1.2 Kontextfreie Grammatiken

Kontextfreie Chomsky-Grammatiken werden weithin benutzt, z.B. um die Syntax von Programmiersprachen zu definieren. Hier sollen nur einige Eigenschaften kontextfreier Grammatiken und Sprachen rekapituliert werden, die im Folgenden noch nützlich sein werden.

Kanonische Ableitungen. Man kann die Ableitungsrelation so einschränken, dass jeweils das am meisten links bzw. rechts in einem Wort stehende Nichterminal ersetzt wird.

$$\frac{w \in V^*, \beta \in (N \cup V)^*, A \to \alpha \in R}{wA\beta \overset{l}{\Rightarrow}_R w\alpha\beta} \qquad \frac{\beta \in (N \cup V)^*, w \in V^*, A \to \alpha \in R}{\beta Aw \overset{r}{\Rightarrow}_R \beta\alpha w}$$

Links- und Rechts-Ableitungen reichen aus, um alle Worte einer Sprache abzuleiten.

Theorem 3.1.

$$Z \Rightarrow_R^* w \text{ genau dann wenn } Z \overset{l}{\Rightarrow}_R^* w \text{ genau dann wenn } Z \overset{r}{\Rightarrow}_R^* w$$

Damit sind Links- und Rechts-Ableitungen Strategien im Sinne von Definition 2.4, die normalisierend sind.

Ableitungsbäume. Ein Baum B ist *Ableitungsbaum* einer kontextfreien Grammatik $\Gamma = \langle V, N, R, Z\rangle$ wenn gilt:

1. Jeder Knoten v in B hat eine Markierung $\ell(v) \in V \cup \{\square\}$.
2. Die Wurzel von B ist mit Z markiert.
3. Jeder innere Knoten ist mit einem Nichtterminal markiert.
4. Wenn ein Knoten v die geordneten Nachfolger $v_1, \ldots, v_k$ hat (in Abbildungen von links nach rechts gelesen), dann ist die Markierung $\ell(v) \to \ell(v_1)\cdots\ell(v_k)$ eine Regel von R.
5. Wenn ein Knoten mit $\square$ markiert ist, ist er ein Blatt und einziger Nachfolger seines Vorgängers v, und $\ell(v) \to \square$ ist eine Regel in R.

Der *Rand front*(B) eines Ableitungsbaums besteht aus den Markierungen seiner Blätter in Präfix-Ordnung, ist also ein Wort aus V^*.

$\mathcal{B}_\Gamma$ soll im Folgenden die Menge aller Ableitungsbäume zu Γ bezeichnen.

Definition 3.1. Eine kontextfreie Grammatik heißt *mehrdeutig*, wenn ihre Sprache Wörter enthält, die mehrere Linksableitungen, Rechtsableitungen oder Ableitungsbäume haben.

Beispiel 3.1 (Eine mehrdeutige kontextfreie Grammatik). Die Regeln

$$R = \{Z \to E\,\underline{.},\ E \to \underline{\texttt{id}},\ E \to \underline{(}E\underline{)},\ E \to E\underline{+}E,\ E \to E\underline{*}E\}$$

definieren eine kontextfreie Grammatik

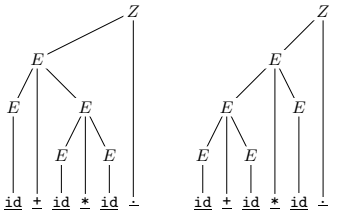

Abb. 3.1. Zwei Ableitungsbäume für id+id*id.

$$\Gamma = \langle V = \{\underline{.}, \underline{\texttt{id}}, \underline{(}, \underline{)}, \underline{+}, \underline{*}\}, N = \{Z, E, \}, R, Z\rangle$$

Das Wort id+id*id. hat zwei Linksableitungen:

$$\begin{aligned}
E\underline{.} \Leftarrow Z &\Rightarrow E\underline{.}\\
E\underline{+}E\underline{.} \Leftarrow &\Rightarrow E\underline{*}E\underline{.}\\
\underline{\texttt{id}}\underline{+}E\underline{.} \Leftarrow &\Rightarrow E\underline{+}E\underline{*}E\underline{.}\\
\underline{\texttt{id}}\underline{+}E\underline{*}E\underline{.} \Leftarrow &\Rightarrow \underline{\texttt{id}}\underline{+}E\underline{*}E\underline{.}\\
\underline{\texttt{id+id*}}E\underline{.} \Leftarrow &\Rightarrow \underline{\texttt{id+id*}}E\underline{.}\\
\underline{\texttt{id+id*id.}} \Leftarrow &\Rightarrow \underline{\texttt{id+id*id.}}
\end{aligned}$$

Also ist die Grammatik mehrdeutig. Die Ableitungsbäume dazu sind in Abbildung 3.1 zu sehen.

Das Beispiel zeigt, dass mehrdeutige Grammatiken – zumindest für Programmiersprachen – ungeeignet sind. Von den hier gezeigten Ableitungen entspricht nur die erste den gängige Vorrang-Regeln der Arithmetik ("Punktrechnung vor Strichrechnung").

Theorem 3.2. *Mehrdeutigkeit von kontextfreien Grammatiken ist nicht entscheidbar.*

Theorem 3.3 (Disjunktheit kontextfreier Sprachen). *Für kontextfreie Grammatiken Γ und Γ' ist die Frage*

$$L_\Gamma \cap L_{\Gamma'} = \{\}\,?$$

nicht entscheidbar.

3.1.3 Parsieren

Das Wortproblem ist für kontextfreie Grammatiken entscheidbar. Dies gilt auch schon für kontext-sensitive Grammatiken. Für kontextfreie Grammatiken gibt es aber darüber hinaus auch effektive Parsierer. Ein *Parsierer* entscheidet

nicht nur das Wortproblem für eine Grammatik Γ, sondern liefert, wenn ein Wort w in der Sprache L_Γ ist, die Menge $\{B \in \mathcal{B}_\Gamma \mid \mathit{front}(B) = w\}$ aller Ableitungsbäume für w.

Allgemeine Parsier-Verfahren für kontextfreien Grammatiken haben den Zeitaufwand $\mathcal{O}(n^3)$ (bzw. $\mathcal{O}(n^2)$ wenn die Grammatik eindeutig ist). Das Parsieren von Teilklassen der eindeutigen kontextfreien Grammatiken, wie $LR(k)$ und $LL(k)$, hat linearen Zeitaufwand.

Deshalb wird die Syntax von Programmiersprachen oft mit kontextfreien Grammatiken definiert, wobei verschiedene Schreibweisen wie die *erweiterte Backus-Naur-Form* (kurz *EBNF*) benutzt werden. Aus so einer Grammatik lässt sich – mindestens wenn sie eindeutig ist – leicht ein Parsierer generieren, der die Syntax der Sprache erkennt.

3.1.4 Kontextsensitivität

Programmiersprachen sind nicht kontextfrei. Deshalb beschreiben kontextfreie Grammatiken immer eine Obermenge aller "wirklich" syntaktisch wohlgeformten Programme. Wir erläutern dies anhand von zwei Erweiterungen der Grammatik aus Beispiel 3.1.

Beispiel 3.2 (Kontextbedingungen in Programmiersprachen). Fügen wir den Regeln der Grammatik in Beispiel 3.1 drei Regeln hinzu:

$$E \to \texttt{let}\ \texttt{id} = E\ \texttt{in}\ E \qquad E \to E \leq E \qquad E \to \texttt{number}$$

Wir betrachten wir das Programm

$$\texttt{let}\ \texttt{x} = \texttt{1} < \texttt{2}\ \texttt{in}\ \texttt{x} + \texttt{y}$$

Dieses Programm entspricht der kontextfreien Grammatik, verletzt aber zwei gängige Kontextbedingungen von Programmiersprachen:

Identifizierung: Jeder in einem Programm benutzte Bezeichner muss eine passende Vereinbarung besitzen. Der Bezeichner y wird in dem Ausdruck aber nicht vereinbart.

Typisierung: Der Typ von Operanden muss zum Typ der Operation passen. Im Programm wird x beim ersten Auftreten ein Wahrheitswert gegeben, aber beim zweiten Auftreten als Operand einer arithmetischen Operation benutzt.

Kontextbedingungen wie diese können mit kontextfreien Grammatiken prinzipiell nicht beschrieben werden. Sie könnten zwar mit kontextsensitiven Chomsky-Grammatiken beschrieben werden, aber solche Beschreibungen wären sehr umständlich. Und sie würden auch praktisch nicht viel weiter helfen. Denn es gibt keine effektiven Parsierverfahren für kontextsensitive Chomsky-Grammatiken.

Gesucht wird also eine Erweiterung von kontextfreien Grammatiken, mit der sich praktisch auftretende Kontextbedingungen gut beschreiben lassen, und für die es auch ein effektives Parsierverfahren gibt. Darum geht es im nächsten Abschnitt.

3.2 Zweistufen-Grammatiken

Um kontextsensitive Sprachen zu definieren, werden kontextfreie Grammatiken erweitert, indem ihre Nichtterminale parameterisiert werden. Kontextfreie Regeln werden damit zu *Regelschemata*, aus denen durch Instanziierung der Parameter unendliche Mengen von kontextfreien Regeln erzeugt werden, mit denen dann die Wörter der Sprache abgeleitet werden.

Zweistufengrammatiken unterscheiden sich in der Art, wie die Parameter der Nichtterminale beschrieben werden:

- *Van-Wijngaarden-Grammtiken* benutzen kontextfreie Meta-Grammatiken, um die zulässigen Instanziierungen der Parameter zu definieren.
- *Attribut-Grammatiken* benutzten eine semantische Basis (Datentypen und Funktionen darauf), mit denen die Parameter ausgewertet werden können.

In den nächsten Unterabschnitten werden wir die Grammatiken kurz vorstellen. Insbesondere werden wir auch Einschränkungen dieser Formalismen betrachten, die das Parsieren vereinfachen.

3.2.1 Van-Wijngaarden-Grammatiken

Van-Wijngaarden-Grammatiken verwenden kontextfreie Regelschemata, denen Nichtterminale Wörter sind, die von einer zweiten Stufe von Meta-Grammatiken erzeugt werden.

Definition 3.2 (Van-Wijngaarden-Grammatik). Eine *van-Wijngaarden-Grammatik* (kurz *W-Grammatik*) ist ein Tupel $G = \langle V, M, R_\mathrm{m}, X, H, T, R_\mathrm{h}, \langle z \rangle \rangle$, mit folgenden Komponenten:

- V ist eine endliche Menge von *Meta-Terminalen*.
- M ist eine endliche Menge von *Meta-Nichtterminalen* mit $M \cap V = \{\}$ und $(M \cup V) \cap \{\langle, \rangle, \to\} = \{\}$.
- $R_\mathrm{m} \subseteq (M \times (M \cup V)^*)$ ist eine endliche Menge von kontextfreien *Meta-Regeln*, so dass für jedes Meta-Nichtterminal $A \in M$ die kontextfreie Grammatik $\langle V, M, R_\mathrm{m}, A \rangle$ die kontextfreie *Meta-Sprache* L_A definiert.
- $X = (X_A)_{A \in M}$ ist eine Familie von *Variablen* für Meta-Nichtterminale.
- H ist eine endliche Menge von *Hyper-Nichtterminalen* der Form $\langle \alpha \rangle$ mit $\alpha \in (X \cup V)^*$.
- T ist eine endliche Menge von *Terminalen*.
- $R_\mathrm{h} \subseteq (H \times (H \cup T)^*)$ ist eine endliche Menge von kontextfreien *Hyperregeln*
- $\langle z \rangle \in H$ mit $z \in T^*$ ist das *Startsymbol*.

Eine Familie $\sigma = (\sigma_A \colon X_A \to L_A)_{A \in M}$ von Funktionen wird *Substitution* genannt. Die Fortsetzung von σ auf ein Hyper-Nichtterminal von der allgemeinen Form $h = \langle u_0 x_1 \cdots x_n u_n \rangle$ (mit $x_i \in X$, $u_0, u_i \in V^*$ für $\leqslant i \leqslant n$) ist dann gegeben als

$$h^\sigma = \langle u_0\, \sigma(x_1) \cdots \sigma(x_n)\, u_n \rangle$$

Dann kann σ weiter fortgesetzt werden auf eine Hyperregel mit der allgemeinen Form $r = h_0 \to w_0\, h_1 \cdots h_k\, w_k$ (mit $h_i \in H$ und $w_i \in T^*$ für $0 \leqslant i \leqslant k$):

$$r^\sigma = h_0^\sigma \to w_0\, h_1^\sigma \cdots h_k^\sigma\, w_n$$

Damit kann nun die *Sprache* von G definiert werden als diejenige, die von der kontextfreien Grammatik $G_\mathrm{s} = \langle T, N, R_\mathrm{s}, \langle z \rangle\rangle$ erzeugt wird, wobei die *Nichtterminale* N und *Grundregeln* R_g so definiert sind:

$$N = \{h^\sigma \mid h \in H, \sigma \text{ Substitution}\}$$
$$R_\mathrm{g} = \{r^\sigma \mid r \in R_\mathrm{h}, \sigma \text{ Substitution}\}$$

In Beispielen verwenden wir kursiv geschriebene Bezeichner für Meta-Symbole, wobei Meta-Nichtterminale und auch Variablen groß geschrieben sind. Terminale werden unterstrichen. Wir trennen linke und rechte Regelseiten wie in erweiterter Backus-Naur-Form, durch "::=", und fassen Regeln mit gleichen linken Seiten mit "|" zusammen. Dies tun wir sowohl auf der Meta- als auch auf der Hyperstufe.

Beispiel 3.3 (Eine Zweistufen-Grammatik für Ausdrücke). Die in den Beispielen 3.1 und 3.2 kontextfrei definierte Sprache wird hier um einige Operatoren erweitert und mit allen Kontextbedingungen definiert.

Die Meta-Regeln der Zweistufengrammatik beschreiben die Struktur von Typen und Vereinbarungssequenzen. Die Struktur von Bezeichnern und Zahlen wird nur angedeutet. Die Variablenmengen für die Meta-Nichtterminale werden rechts angegeben.

$T ::= \mathit{bool} \mid \mathit{int}$	$X_T = \{T, L, R\}$
$E ::= \square \mid I\ \mathit{of}\ T\ E$	$X_E = \{E, E_1, E_2\}$
$I ::= \mathit{id}\ a \mid \mathit{id}\ x \mid \ldots$	$X_I = \{X, Y\}$
$N ::= \mathit{num}\ \mathit{one} \mid \mathit{num}\ \mathit{two} \mid \ldots$	$X_N = \{N\}$

In den Regelschemata werden die Regeln aus Beispiel 3.1 mit Typausdrücken parameterisiert, die gemäß den Meta-Regeln gebildet sind.

⟨*program*⟩ ::= ⟨*T expression in* □⟩

⟨*T expression in E*⟩
::= ⟨*X identifier*⟩ ⟨*where X is T in E*⟩
| `let` ⟨*X identifier*⟩ ⟨*unless X is in E*⟩ `=` ⟨*L expression in E*⟩ `in` ⟨*T expression in E X of L*⟩
| `(` ⟨*T expression in E*⟩ `)`
| ⟨*L expression in E*⟩ ⟨*L x R to T operator*⟩ ⟨*R expression in E*⟩

⟨*int expression in E*⟩ ::= ⟨*N number*⟩

⟨*int x int to bool operator*⟩ ::= `>` | `>` | `<=` | `>=` | `=` | `/=`

⟨*int x int to int operator*⟩ ::= `+` | `-` | `*` | `div` | `mod`

⟨*unless X is in* □⟩ ::= □

⟨*unless X is in Y of T E*⟩ ::= ⟨*unless X equals Y*⟩ ⟨*unless X is in E*⟩

⟨*where X is T in* E_1 *X of T* E_2⟩ ::= □

Wir nehmen an, das Nichtterminal "⟨*unless X equals Y*⟩" sei so vordefiniert, dass es □ ableitet genau dann wenn X und Y ungleich sind. Die Regeln für das Nichtterminal "⟨*unless X is in E*⟩" überprüfen, ob ein Bezeichner X in der Umgebung E bereits vereinbart wurde – dies wäre eine doppelte Vereinbarung. Die Regel für das Nichtterminal "⟨*where X is T in E*⟩" überprüft, ob ein Bezeichner X in der Umgebung E vereinbart wurde. Alle diese Nichtterminale sind insofern speziell, als sie keine Wörter ableiten, außer □, wenn eine Bedingung für ihre Parameter erfüllt ist. Sie werden *Prädikate* genannt, und ähneln tatsächlich auch den Prädikaten der Logik.

Wir demonstrieren, wie Ableitungen mit W-Grammatiken Kontextbedingungen respektieren.

⟨*program*⟩
⇒ ⟨*int expression in* □⟩
⇒ `let` ⟨*id x identifier*⟩ ⟨*unless x is in* □⟩ `=` ⟨*bool expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x` ⟨*unless x is in* □⟩ `=` ⟨*bool expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x =` ⟨*bool expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x =` ⟨*int expression in* □⟩ ⟨*int x int to bool operator*⟩ ⟨*int expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x =` ⟨*num one number*⟩ ⟨*int x int to bool operator*⟩ ⟨*int expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x = 1` ⟨*int x int to bool operator*⟩ ⟨*int expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 <` ⟨*int expression in* □⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 <` ⟨*num two number*⟩ `in` ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 < 2 in` ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 < 2 in` ⟨*bool expression in id x of bool*⟩ ⟨*bool x int to int operator*⟩ ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 < 2 in` ⟨*id x identifier*⟩ ⟨*where id x is bool in id x of bool*⟩ ⟨*bool x int to int operator*⟩ ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 < 2 in x` ⟨*where id x is bool in id x of bool*⟩ ⟨*bool x int to int operator*⟩ ⟨*int expression in id x of bool*⟩
⇒ `let x = 1 < 2 in x` ⟨*bool x int to int operator*⟩ ⟨*int expression in id x of bool*⟩

In der letzten Zeile kann die Ableitung nicht weiter gehen, weil es keinen "⟨*bool x int to int operator*⟩" gibt. Weil das "bool" durch die Vereinbarung von "`x`" fixiert ist, geht es hier nicht weiter, denn eine Kontextbedingung ist verletzt. Dies nennt man eine *Sackgasse* ("*blind alley*").

Ersetzt man `<` durch `+`, beginnt der erste Teil der Ableitung so:

program
$\Rightarrow^*$ `let x = 1 + 2 in x`
⟨*int x int to int operator*⟩ ⟨*int expression in id x of int*⟩

(Im "Environment" der *expression* wird dann "*x of int*" eingetragen.) Jetzt findet man einen passenden Operator, und die Ableitung kann fortgesetzt werden:

$\Rightarrow$ `let x = 1 + 2 in x +` ⟨*int expression in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x +` ⟨*id y identifier*⟩ ⟨*where id y is int in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x + y` ⟨*where id y is int in id x of int*⟩

Hier ist wieder eine Sackgasse, weil "`y`" nicht vereinbart wurde. Nur wenn man "`y`" durch "`x`" ersetzt, kann die Ableitung zu Ende geführt werden und sieht dann so aus:

$\Rightarrow^*$ `let x = 1 + 2 in x +` ⟨*id x identifier*⟩ ⟨*where id x is int in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x + x` ⟨*where id y is int in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x + x`

Dieses Programm erfüllt die Kontextbedingungen, die definiert werden sollten. Man kann sich leicht überzeugen, dass alle Ableitungen die Kontextbedingungen einhalten.

In der Ableitung haben wir Grundregeln verwendet, also alle Variablen in den Hyperregeln voll substituiert. Das ist nicht besonders klug, weil wir so die möglichen Ableitungen einschränken, bevor es nötig ist. Im ersten Schritt legen wir uns beispielsweise darauf fest, einen Ausdruck vom Typ "int" abzuleiten, und im zweiten darauf, dass der lokale Hilfsausdruck den Typ "bool" haben und an den Bezeichner "`x`" binden soll. Es ist geschickter, die Variablen erst dann zu instanziieren, wenn sie durch ein abgeleitetes Terminal festgelegt sind. Beispielsweise sollte X erst dann mit *id x* instanziiert werden, wenn der Bezeichner `x` abgeleitet werden soll, und der Typ T sollte erst dann mit "bool" instanziiert werden, wenn der Operator "$<$" festlegt, dass der Hilfsausdruck diesen Typ hat.

Definition 3.3. Über einer Metastufe $\langle U, M, R_\mathrm{m}\rangle$ wird eine *unvollständige Substitution* definiert als eine Familie von Funktionen

$$\sigma = (\sigma_m \colon X_m \to \mathcal{F}_m)_{m\in M}$$

wobei die *Satzformen* der Meta-Grammatiken definiert sind als

$$\mathcal{F}_A = \{\alpha \in (X \cup V)^* \mid A \Rightarrow^*_{R_M} \mathit{typ}(\alpha)\}$$

Hierbei ordnet die Funktion $\mathit{typ}\colon X \to M$ jeder Variablen $x \in X_A$ ihren Typ A zu und wird auf naheliegende Weise zu einer Funktion auf Wörtern $(X\cup V)^*$ erweitert, die wir auch mit *typ* bezeichnen.

Wir benutzen die Regeln, die mit unvollständigen Substitutionen aus den Hyperregeln abgeleitet werden können, für folgende Ableitung des (korrigierten) Programms aus Beispiel 3.3:[3]

⟨*program*⟩
$\Rightarrow$ ⟨*T expression in* □⟩
$\Rightarrow$ `let` ⟨*X identifier*⟩ ⟨*unless X is in* □⟩ `=` ⟨*L expression in* □⟩
`in` ⟨*T expression in X of L*⟩
$\Rightarrow$ `let x` ⟨*unless id x is in* □⟩ `=` ⟨*L expression in* □⟩
`in` ⟨*T expression in id x of L*⟩
$\Rightarrow$ `let x =` ⟨*L expression in* □⟩ `in` ⟨*T expression in id x of L*⟩
$\Rightarrow$ `let x =` ⟨*L expression in* □⟩ ⟨*L x R to L operator*⟩
⟨*R expression in* □⟩ `in` ⟨*T expression in id x of L*⟩
$\Rightarrow$ `let x =` ⟨*num one number*⟩ ⟨*int x R to L operator*⟩
⟨*R expression in* □⟩ `in` ⟨*T expression in id x of L*⟩
$\Rightarrow$ `let x = 1` ⟨*int x R to L operator*⟩ ⟨*R expression in* □⟩
`in` ⟨*T expression in id x of L*⟩
$\Rightarrow$ `let x = 1 +` ⟨*int expression in* □⟩ `in` ⟨*T expression in id x of int*⟩
$\Rightarrow$ `let x = 1 <` ⟨*num two number*⟩ `in` ⟨*T expression in id x of int*⟩
$\Rightarrow$ `let x = 1 < 2 in` ⟨*T expression in id x of int*⟩
$\Rightarrow$ `let x = 1 < 2 in` ⟨*L expression in id x of int*⟩
⟨*L x R to T operator*⟩ ⟨*R expression in id x of int*⟩
$\Rightarrow$ `let x = 1 < 2 in` ⟨*X identifier*⟩ ⟨*where X is L in id x of int*⟩
⟨*L x R to T operator*⟩ ⟨*R expression in id x of int*⟩
$\Rightarrow$ `let x = 1 < 2 in x` ⟨*where id x is L in id x of int*⟩
⟨*L x R to T operator*⟩ ⟨*T expression in id x of int*⟩
$\Rightarrow$ `let x = 1 < 2 in x`
⟨*int x R to T operator*⟩ ⟨*T expression in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x +` ⟨*int expression in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x +` ⟨*id x identifier*⟩ ⟨*where id x is int in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x + x` ⟨*where id x is int in id x of int*⟩
$\Rightarrow$ `let x = 1 + 2 in x + x`

Bei dieser Art der Ableitung müssen wir Nichtterminale in der Ableitung an linke Regelseiten angleichen, wie zum Beispiel

⟨*where X is L in id x of int*⟩
an ⟨*where X is T in* E_1 *X of T* E_2⟩

Dies ist allgemein bekannt als das *Unifizierungsproblem für kontextfreie Sprachen*, und für W-Grammatiken als das *Referenz-Problem*. In diesem Beispiel können die Wörter *unifiziert* werden, denn die Substitution

$$T \mapsto L, E_1 \mapsto \square, E_2 \mapsto \square,$$

vereinheitlicht beide Nichtterminale zu ⟨*where X is T in X of T*⟩. Leider gilt im Allgemeinen aber:

[3] Diese Regeln sind Hyperregeln, also keine reinen Grundregeln!

Theorem 3.4. *Das Referenz-Problem für W-Grammatiken ist nicht entscheidbar.*

Beweis. Seien $\langle\alpha\rangle$ und $\langle\beta\rangle$ zwei Hypern-Nichtterminale (oder allgemein: Wörter einer kontextfreien Sprache). Wir konstruieren die kontextfreien Grammatiken

$$G = \langle V, M, R_\mathrm{m} \cup \{Z \to \alpha\}, Z\rangle \text{ und } H = \langle V, M, R_\mathrm{m} \cup \{Z \to \beta\}, Z\rangle$$

Dann lassen sich $\langle\alpha\rangle$ und $\langle\beta\rangle$ genau dann unifizieren, wenn $L_G \cap L_H \neq \{\}$. Disjunktheit von kontextfreien Grammatiken ist aber nach Satz 3.3 im Allgemeinen nicht entscheidbar.

Deshalb funktioniert diese Art der Ableitung für uneingeschränkte W-Grammatiken leider nicht.

Da die Disjunktheit für *reguläre* Sprachen entschieden werden kann, könnte man sich auf reguläre Meta-Sprachen beschränken. Man kann zeigen, dass dies die Mächtigkeit von W-Grammatiken nicht einschränkt. Die Meta-Sprachen in Beispiel 3.3 sind regulär – die angegebenen Regeln nicht, aber sie ließen sichj in reguläre Regeln transformieren. (Allgemein reicht sogar eine reguläre Grammatik über einem einzigen Nichtterminal.)

Die Unifikation von regulären Sprachen ist aber im Allgemeinen sehr aufwändig. Deshalb machen wir eine andere, weiter gehende Einschränkung

Definition 3.4 (Transparenz). Eine W-Grammatik wie in Definition 3.2 heißt *transparent*, wenn gilt:

1. Alle Meta-Grammatiken $\langle U, M, R_\mathrm{m}, A\rangle$ (für $A \in M$) sind eindeutig.
2. Die Metastufe hat ein *Wurzel-Nichtterminal* $Z \in M$, so dass für alle Hyper-Nichtterminale $\langle\alpha\rangle \in H$ gilt: $typ(\alpha) \in \mathcal{F}_Z$.

Theorem 3.5. *Für transparente W-Grammatiken ist das Referenz-Problem entscheidbar.*

Beweis. Nach Definition haben alle Hyper-Nichtterminale und die aus ihnen ableitbaren Grund-Nichtterminale eindeutige Ableitungsbäume. Deshalb können wir diese Wörter über der Metasufe immer eindeutig durch ihre Ableitungsbäume darstellen.

Das Referenz-Problem reduziert sich damit auf das Unifizieren von Ableitungsbäumen, was der Unifikation in Prädikatenlogik erster Stufe entspricht. Dafür gibt es lineare Algorithmen.

Beispiel 3.4 (Eine transparente W-Grammatik). Die Meta-Grammatiken in Beispiel 3.3 sind schon eindeutig und erfüllen damit die erste Bedingung von Definition 3.4.

Für den zweiten Punkt erweitern wir die Meta-Grammatik um folgende Regeln für das Wurzel-Nichtterminal Z:

$$\begin{aligned} Z ::= & \textit{ program} \\ \mid & \textit{ T expression in E} \\ \mid & \textit{ T x T to T operator} \\ \mid & \textit{ I identifier} \\ \mid & \textit{ N number} \\ \mid & \textit{ unless I is in E} \\ \mid & \textit{ where I is T in E} \end{aligned}$$

Alle Hyper-Nichtterminale bis auf $\langle \textit{where X is T in } E_1 \textit{ X of T } E_2\rangle$ (in der letzten Hyperregel) können aus der Meta-Grammatik für Z abgeleitet werden. In diesem Fall sind zwar alle Substitutionen von E_1 *X of T* E_2 aus E ableitbar, aber nicht der Ausdruck selbst. Aus E können nur Ausdrücke der Form *X of T* E_2 abgeleitet werden. Eine kleine Änderung der Definition des Prädikats schafft Abhilfe:

$$\begin{aligned} \langle \textit{where X is T in } \square\rangle &::= \square \\ \langle \textit{where X is T in Y of T' } E\rangle &::= \langle \textit{unless X equals Y}\rangle\ \langle \textit{where X is T in } E\rangle \end{aligned}$$

Dann erfüllt die Grammatik auch die zweite Bedingung von Definition 3.4, denn alle Nichtterminale sind nun aus von Z ableitbar.

Für transparente W-Grammatiken können Parsierer erzeugt werden, indem zunächst ihre unterliegende kontextfreie Grammatik bestimmt wird und die Prädikate isoliert werden:

Definition 3.5 (Unterliegende kontextfreie Grammatik). Sei G eine transparente W-Grammatik wie in Definitionen 3.2 und 3.4. Wegen der Transparenz von G gibt es für jedes Hyper-Nichtterminal $\langle\beta\rangle \in H$ eine eindeutige Ableitung

$$Z \Rightarrow_{R_\mathrm{m}} \alpha \Rightarrow^*_{R_\mathrm{m}} typ(\beta)$$

vom Wurzel-Nichtterminal Z. Das *unterliegende Nichtterminal* $\overline{\langle\beta\rangle}$ erhält man dann, indem man in $\langle\alpha\rangle$ jedes Meta-Nichtterminal durch "␣" ersetzt. Dann sind die unterliegenden Nichtterminale gegeben mit $U = \{\bar{h} \mid h \in H\}$ und die unterliegenden Regeln als

$$R_\mathrm{u} = \{\bar{h} \to w_0\, \bar{h}_1 \cdots \bar{h}_k\, w_k \mid h \to w_0\, h_1 \cdots h_k\, w_k \in R_\mathrm{h}\}$$

Die *Prädikate* von G sind diejenigen Nichtterminale $P \in U$, für die gilt: Wenn $P \Rightarrow^*_{R_\mathrm{u}} w$ mit $w \in V^*$, dann ist $w = \{\square\}$.

Dann enthält man die unterliegende kontextfreie Grammatik als $\bar{G} = \langle V, U, R'_\mathrm{u}, \langle z\rangle\rangle$ wobei in R'_u für jedes Prädikat $P \in U$ alle unterliegenden Regeln für P durch eine einzige Regel von der Form $P \to \square$ ersetzt wurden.

Die unterliegende Grammatik kann effektiv bestimmt werden. Es ist nämlich entscheidbar, ob die Sprache einer kontextfreien Grammatik nur das leere Wort enthält.

Lemma 3.1. *Für alle transparenten W-Grammatiken ist die generierten Sprache in der Sprache ihrer unterliegenden Grammatik enthalten.*

Mit diesem Lemma bekommt man einen Ansatz, ein Parsierverfahren für transparente W-Grammatiken zu entwickeln. Dazu geht man so vor:

1. Mit einem Parsierer für die unterliegende Grammatik werden Ableitungsbäume eines Wortes konstruiert.
2. Für jeden Knoten eines Ableitungsbaumes werden die Baumunifikationsprobleme gelöst, die sich aus den Hyperregeln der unterliegenden Regeln ergeben.
3. Alle Prädikatknoten werden nach den Hyperregeln abgeleitet.
4. Wenn alle Unifikationsprobleme gelöst werden können – und nur dann – ist das Wort in der Sprache der W-Grammatik.

Dieses Verfahren ist praktikabel, besonders wenn die unterliegende kontextfreie Grammatik eindeutig und z.B. mit dem $LR(k)$-Verfhren parsierbar ist. Dann wird höchstens ein unterliegender Ableitungsbaum konstruiert, und dann rekursiv eine Menge von Baum-Unifikationsproblemen und Prädikat-Ableitungen für diesen Baum.

Die unterliegende Grammatik von Beispiel 3.3 entspricht allerdings der aus Beispiel 3.1, und die ist mehrdeutig, wie wir aus dem letzten Abschnitt wissen. Auch dies kann aber behoben werden, wie man in allen Lehrbüchern zum Übersetzerbau nachlesen kann.

Theorem 3.6 (Mächtigkeit von W-Grammatiken). *Jede rekursiv aufzählbare Sprache lässt sich mit einer W-Grammatik oder transparenten W-Grammatik beschreiben.*

Das ist mehr als gewollt, denn das Wortproblem ist für rekursiv aufzählbare Sprachen nicht entscheidbar. Für transparente W-Grammatiken mag dies verwundern. Es ist aber leicht zu zeigen, dass die Ableitungen der Prädikate nicht immer terminieren müssen, so dass trotz der Parsierbarkeit der unterliegenden kontextfreien Grammatik die Frage, ob alle Prädikate gelten, nicht immer beantwortet werden kann.

Das Terminieren der Prädikatableitungen kann man für konkrete Beispiele dadurch zeigen, dass man nachweist, dass Prädikatableitungen zu immer "kleineren" Prädikat-Nichtterminalen führen. Dann müssen Ableitungen immer irgendwann das leere Wort ableiten, oder in einer Sackgasse stecken bleiben.

Die Prädikate $\langle \mathit{unless} \ldots \rangle$ und $\langle \mathit{where} \ldots \rangle$ in Beispiel 3.4 haben diese Eigenschaft: In rekursiven Aufrufen ist das Prädikat-Nichtterminal auf der rechten Seite immer echt kürzer als auf der linken Seite. Deshalb wird das Parsieren für dieses Beispiel immer terminieren.

Im Allgemeinen ist es aber so, dass Termination nicht entscheidbar ist. Das werden wir noch genauer sehen, wenn wir Termersetzung betrachten (im nächsten Kapitel).

3.2.2 Attributgrammatiken

Attribut-Grammatiken können als eine andere Art von Zweistufengrammatiken aufgefasst werden, in denen die Metastufe nicht durch kontextfreie Regeln gebildet wird, sondern durch Wertemengen von Datentypen mit entsprechenden Operationen.

Beispiel 3.5 (Eine Attribut-Grammatik für Ausdrücke). Die *semantische Basis* wird als Menge von Datentypen definiert, hier in Haskell.

$$
\begin{array}{rcl}
\textbf{data}\ \mathit{Type} & ::= & \mathrm{B} \mid \mathrm{I} \\
\textbf{type}\ \mathit{Env} & = & [(X, \mathit{Type})] \\
\textbf{type}\ X & = & \mathit{String} \\
\textbf{type}\ \mathit{Num} & = & \ldots \\
\mathrm{defType} & :: & \mathit{Env} \to X \to \mathit{Type} \\
\mathrm{defType}\ ((y,t) : e)\ x & \mid & x = y \qquad \Rightarrow t \\
 & \mid & \textbf{otherwise} \Rightarrow \mathrm{defType}\ x\ e \\
\mathrm{elem} & :: & X \to \mathit{Env} \to \mathit{Bool} \\
\mathrm{elem}\ x\ [\,] & = & \mathit{False} \\
\mathrm{elem}\ x\ (h : t) & = & x = h \vee \mathrm{elem}\ x\ t \\
\mathrm{addDecl} & :: & X \to \mathit{Type} \to \mathit{Env} \to \mathit{Env} \\
\mathrm{defType}\ x\ t\ e & = & (x,t) : e
\end{array}
$$

In den attributierten Regeln werden die Regeln aus Beispiel 3.1 mit Variablen parameterisiert. In den folgenden Regeln werden die Variablen X vom Typ X, sowie die Variablen T, T', L, R vom Typ *Type* und E, E' vom Typ *Env* benutzt.

Den Parametern werden die *Richtungen* "aufwärts" (*synthesized*, "↑") oder "abwärts" (*inherited*, "↓") zugeordnet, die angeben, ob ein Attributwert von unten nach oben oder von oben nach unten "fließen" soll.

$$
\begin{array}{rcl}
\mathrm{program} & ::= & \mathrm{expression}{\uparrow}T{\downarrow}E\ \textbf{where}\ E \leftarrow [] \\
\mathrm{expression}{\uparrow}T{\downarrow}E & ::= & \mathrm{id}{\uparrow}X\ \textbf{where}\ T \leftarrow \mathrm{defType}\ X\ E\ \textbf{check}\ \mathrm{elem}\ X\ E \\
 & \mid & \underline{\texttt{let}}\ \mathrm{id}{\uparrow}X\ \underline{=}\ \mathrm{expression}{\uparrow}L{\downarrow}E\ \underline{\texttt{in}}\ \mathrm{expression}{\uparrow}T{\downarrow}E' \\
 & & \textbf{where}\ E' \leftarrow \mathrm{addDecl}\ X\ L\ E\ \textbf{check}\ \neg(\mathrm{elem}\ X\ E) \\
 & \mid & \underline{(}\ \mathrm{expression}{\uparrow}T{\downarrow}E\ \underline{)} \\
 & \mid & \mathrm{expression}{\uparrow}L{\downarrow}E\ \mathrm{operator}{\downarrow}L{\downarrow}R{\uparrow}T\ \mathrm{expression}{\uparrow}R{\downarrow}E \\
 & \mid & \mathrm{number}{\uparrow}N\ \textbf{where}\ T \leftarrow \mathrm{I}
\end{array}
$$

Für Attributgrammatiken werden typischerweise *Auswerter* generiert, die in zwei Phasen arbeiten:

1. Zunächst wird ein Ableitungsbaum für die unterliegende Grammatik erstellt. Da man meistens voraussetzt, dass die unterliegende Grammatik eindeutig ist, hat jedes Programm höchstens einen Ableitungsbaum.
2. An diesen Baum werden Attributwerte angehängt, die gemäß den Regeln und den Richtungen der Attribute berechnet werden.

Abbildung 3.2 zeigt die Auswertung des Ausdrucks, den wir auch mit der W-Grammatik abgeleitet haben.

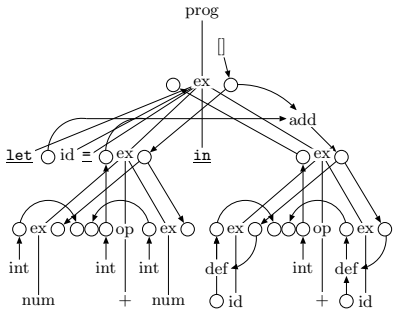

Abb. 3.2. Auswertung eines Ausdrucks

Damit die Auswertung der Attributwerte in den angegebenen Richtungen funktioniert, darf der Datenfluss nicht zyklisch sein. Dies ist in dem Beispiel gegeben. Die Überprüfungen der Attributwerte mit **check** wurde im Baum nicht berücksichtigt.

Literaturhinweise

Die Zusammenfassung von Formalen (Chomsky-) Sprachen folgt [HU79]. Parsieren für kontextfreie Grammatiken und reguläre Grammateiken und Ausdrücke ist ein wesentlicher Bestandteil des Übersetzerbaus [ALSU86]. (L-Systeme definieren *parallele* Ersetzungen auf Wörtern und werden z B. benutzt, um Wachtumsvorgänge in Pflanzen zu beschreiebn. Sie werden in [KRS97] beschrieben, hier aber noch nicht berücksichtigt.)

Das Buch [CU77] beschreibt Zweistufengrammatiken nach van Wijngaarden, die für die Definition der Programmiersprache ALGOL-68 [vWMP+76] entwickelt wurden. Diese Definition zeigt auch, dass sie *zu* elegant sind, weil die Beschreibungen sehr schwer verständlich werden können.

Attributgrammatiken wurden von Donald E. Knuth für die "Semantik kontextfreier Sprachen" entwickelt [Knu68, CM79]; auf dieser Grundlage wurden Dutzende von übersetzererzeugenden Systemen entwickelt. (Ganz ähnlich sind die Affixgrammatiken von C.H.A. Koster [Kos71].)

Kurze Einführungen in W-Grammatiken und Attributgrammatiken gibt der Übersichtsartikel [MS97, Abschnitte 4 und 5].

Die erweiterten Affixgrammatiken [Wat74] und erweiterten Attributgrammatiken [WM83] von David Watt und Ole Madsen kombinieren beide Formalismen.

Wir haben Regeln für Wörter nur als Grammatiken benutzt. Chomsky-Regeln (unbeschränkte) können auch als Berechnungsmodell benutzt werden. Das bietet aber kaum Vorteile gegenüber Termersetzung und ist deshalb nicht populär. im Übrigen definieren die *invisble productions* von W-Grammatiken Prädikate und somit Berechnungen. Beschränkt man sich auch noch auf transparente Regeln, entspricht dies ganz genau den Hornklauseln von PROLOG. Dazu später mehr, in Kapitel 4.