von Namen, eine Adresse von Stringkonstanten usw. Die Werte dieser Attribute liefert meist der Scanner, zumindest, wenn er um einige Funktionen zur Buchhaltung vermehrt ist. In der Praxis spielen also die abgeleiteten Attribute bei (manchen) Terminalsymbolen eine große Rolle. Später bei der Vorstellung der Generierungsverfahren stören sie etwas in der Darstellung, ohne aber Implementierungsschwierigkeiten zu machen. Wir werden sie dann weglassen.

In einer Attributgrammatik in Normalform existiert also für jedes definierende Attributvorkommen in einer Produktion genau eine semantische Regel, und deren Argumente sind ausschließlich angewandte Attributvorkommen. Betrachten wir ein Exemplar der Produktion in einem Syntaxbaum. Dann müssen die Exemplare zu angewandten Attributvorkommen ihre Werte also von außerhalb (des Exemplars) der Produktion erhalten, ererbte Attribute auf der linken Seite der Produktion aus dem oberen Baumfragment und abgeleitete Attribute auf der rechten Seite aus den darunterliegenden Teilbäumen bzw. durch „externe Regeln", siehe Abbildung 9.6.

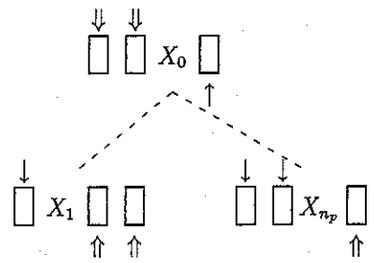

Abb. 9.6: Eine (attributierte) Produktion. Ererbte Attributvorkommen sind als Kästchen links von syntaktischen Symbolen, abgeleitete als Kästchen rechts von Symbolen dargestellt. Einfache Pfeile deuten semantische Regeln der Produktion an, doppelte den Informationsfluß von außerhalb in die Regeln hinein.

9.2.1 Die Semantik einer Attributgrammatik

Was ist die Semantik einer Attributgrammatik? Unter bestimmten Bedingungen legt sie für jeden Syntaxbaum t der zugrundeliegenden kontextfreien Grammatik eine Zuordnung von Attributwerten zu Knoten von t auf folgende Weise fest:

Sei n ein Knoten von t. Wir betrachten n als ein Wort über $\mathbb{N}$ und definieren die Konkatenation $n.j$ (kürzer auch nj) wie üblich, außer daß wir $n.0 = n$ festlegen. $symb(n) \in V_N \cup V_T$ sei das Symbol, welches n markiert. Ist $symb(n) \in V_N$, so sei $prod(n)$ die an n angewendete Produktion. Für jedes Attribut $a \in Attr(symb(n))$ liegt an n ein **Attributexemplar** a_n vor. Diesem Exemplar soll ein Wert aus seinem Wertebereich D_a zugeordnet werden. Die Werte der verschiedenen Attributexemplare zu dem Exemplar von $prod(n)$ müssen

in folgender Beziehung stehen: Sei $val(a_m)$ der Wert vom Exemplar von a beim Knoten m. Ist $a_i = f_{p,a,i}(b^1_{j_1}, \ldots, b^k_{j_k})$ eine semantische Regel von $prod(n) = p$, so induziert sie die folgende Relation zwischen Werten von Attributexemplaren:

$$val(a_{ni}) = f_{p,a,i}(val(b^1_{nj_1}), \ldots, val(b^k_{nj_k}))$$

Attribute und Attributvorkommen „existieren" also ab dem Augenblick, in dem die Attributgrammatik aufgeschrieben wird, also auf jeden Fall dann, wenn ein Attributauswerter generiert werden soll. Attributexemplare existieren erst zur Übersetzungszeit, wenn ein Syntaxbaum aufgebaut ist, d.h. zu der Zeit, wenn die Attributexemplare ausgewertet werden. Sei $a \in Inh(X)$ ein Attribut eines Nichtterminals X, und $p : X \longrightarrow XY$ sei eine Produktion. Dann hat a zwei Vorkommen in a_0 und a_1 in p. Zu p gehöre die semantische Regel $a_1 = f(a_0)$. Abbildung 9.7 (a) zeigt diese Beziehung zwischen den Attributvorkommen von p, (b) zeigt die daraus induzierte Beziehung zwischen den zugehörigen Attributexemplaren einer Anwendung von p in einem Syntaxbaum.

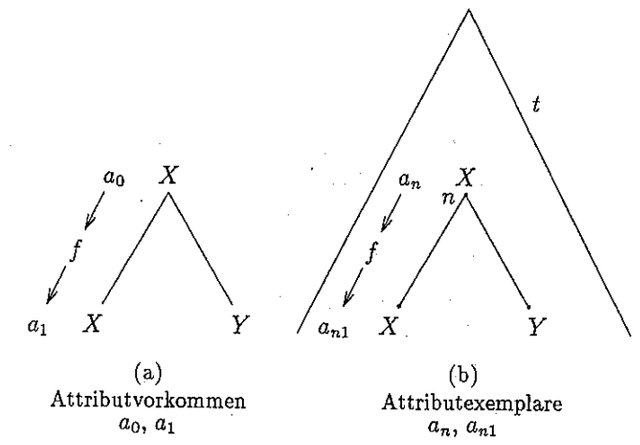

Abb. 9.7: (a) Attributvorkommen, semantische Regeln zur Generierungszeit, (b) Attributexemplare, induzierte semantische Relation zur Übersetzungszeit.

Für die Menge aller Attributexemplare in t, $V(t) = \{a_m\}_{m \in t,\, a \in Attr(symb(m))}$ ergibt sich somit ein Gleichungssystem in den Unbekannten a_m. Ist dieses Gleichungssystem rekursiv (zyklisch), so kann es mehrere oder auch keine Lösung haben. Ist es nicht rekursiv, so hat es genau eine Lösung, d.h. genau einen Wert pro Attributexemplar. Man nennt eine Attributgrammatik **wohlgeformt**, wenn keines dieser Gleichungssysteme rekursiv ist. In diesem Fall ist die Semantik der Attributgrammatik also die eindeutige Zuordnung von Attributwerten zu den Knoten der Syntaxbäume der zugrundeliegenden kontextfreien Grammatik.

9.2.2 Eine Notation für Attributgrammatiken

Im folgenden werden einige Beispielgrammatiken in der folgenden Syntax gegeben:

AttrGramm	⟶ *GrammName: Nonts; Attrs; Rules; Functions*
GrammName	⟶ **attribute grammar** *Name*
Nonts	⟶ **nonterminals** *NonEmptyNamList*
Attrs	⟶ **attributes** *AttrDeclList*
AttrDeclList	⟶ *AttrDeclList; AttrDecl* | ε
AttrDecl	⟶ *Direction Name* **with** *NonEmptyNamList DomSpec*
Direction	⟶ **inh** | **syn**
DomSpec	⟶ **domain** *Domain*
Rules	⟶ **rules** *RuleList*
RuleList	⟶ *RuleList Rule* | *Rule*
Rule	⟶ *CfRule SemRuleList*
CfRule	⟶ *Name ' → ' NamList*
SemRuleList	⟶ *SemRuleList SemRule* | ε
SemRule	⟶ *Attr ' = ' Name (AttrList)* | *Attr ' = ' Attr* | *Attr ' = ' Const*
AttrList	⟶ *AttrList Attr* | *Attr*
Attr	⟶ *Name.Name* | *Name Index.Name*
Functions	⟶ **functions** *FunctDefList*
FunctDefList	⟶ *FunctDefList FunctDef* | ε
NamList	⟶ *NonEmptyNamList* | ε
NonEmptyNamList	⟶ *NonEmptyNamList Name* | *Name*

Natürlich müssen für die Sprache, in der Attributgrammatiken aufgeschrieben werden, auch Kontextbedingungen gelten; die wichtigsten werden angegeben:

- Es dürfen auf der linken Seite von CfRules, (kontextfreien) Produktionsregeln, nur Namen stehen, die hinter **nonterminals** aufgelistet sind.
- Als Attributbezeichnung sind $X.Y$ und $X_i.Y$ zugelassen, wobei X ein Nichtterminal aus der vorangehenden kontextfreien Produktion und Y ein Attributname ist, der laut **attributes**-Spezifikation mit X assoziiert ist. $X_i.Y$ bezeichnet das Attributvorkommen von Y beim i-ten Vorkommen von X auf der rechten Seite, falls $i > 0$, und beim Vorkommen von X auf der linken Seite, wenn $i = 0$.

- Jedes definierend in einer Produktion auftretende Attribut muß in genau einer semantischen Regel der Produktion links auftreten, und nur definierende Vorkommen dürfen auf der linken Seite auftreten. Auf der rechten Seite von semantischen Regeln dürfen nur angewandte Vorkommen von Attributen aus der Produktion stehen (Normalform).
- Für jedes in einer semantischen Regel auftretende Funktionssymbol muß genau eine Definition im functions-Teil existieren.

9.3 Einige Attributgrammatiken

Die folgenden Attributgrammatiken beschreiben einige Varianten des Typberechnungsproblems für arithmetische Ausdrücke. Sie dienen später als Beispiele für die Zugehörigkeit zu bestimmten Klassen von Attributgrammatiken und zur Demonstration der Implementierungsprobleme und -techniken.

Zur Definition der funktionalen Beziehung zwischen Attributvorkommen benutzen wir eine funktionale Programmiersprache mit folgenden Eigenschaften:

- die verschiedenen Fälle für die Funktion werden durch die zugelassenen Kombinationen von Argumentwerten und Variablen angegeben;
- für einen Fall unwesentliche Argumente werden durch eine anonyme Variable '_' bezeichnet;
- fällt ein aktuelles Argumenttupel unter mehrere Fälle, so wird der „speziellste“ Fall ausgewählt; dabei sind Konstante spezieller als Variable, und ein Tupel t_1 ist spezieller als ein Tupel t_2, wenn t_1 aus t_2 durch eine Substitution von Konstanten für Variablen hervorgeht. (Der Fall unvergleichlicher, aber sich in obiger Art überlappender Fälle tritt hier nicht auf)
- mehrfach in einem Tupel auftretende Variable müssen auf die gleichen Werte treffen, um den entsprechenden Fall auszuwählen.

Die erste Attributgrammatik, $\mathbf{AG_1}$, dargestellt in Beispiel 9.3.1, beschreibt die Typberechnung für Ausdrücke mit den Operatoren $+, -, *, /$ und $\div$ und Variablen und Konstanten vom Typ *integer* oder *real*.

Beispiel 9.3.1 (Attributgrammatik $\mathbf{AG_1}$)

attribute grammar AG_1:
nonterminals *E, T, F, P, Aop, Mop;*
attributes **syn** *typ* **with** *E, T, F, P, c, v* **domain** {*int, real*};
syn *op* **with** *Aop, Mop* **domain** {'+','−','*','/','÷'};

rules

1 : $E \rightarrow E\ Aop\ T$
$E_0.typ = f_1(E_1.typ,\ T.typ)$

2 : $E \rightarrow T$
$E.typ = T.typ$

3 : $T \rightarrow T\ Mop\ F$
$T_0.typ = f_3(\ Mop.op,\ T_1.typ,\ F.typ\ )$

4 : $T \rightarrow F$
$T.typ = F.typ$

5 : $F \rightarrow P$
$F.typ = P.typ$

6 : $P \rightarrow c$
$P.typ = c.typ$

7 : $P \rightarrow v$
$P.typ = v.typ$

8 : $P \rightarrow (E)$
$P.typ = E.typ$

9 : $Aop \rightarrow +$
$Aop.op = '+'$

10 : $Aop \rightarrow -$
$Aop.op = '-'$

11 : $Mop \rightarrow *$
$Mop.op = '*'$

12 : $Mop \rightarrow /$
$Mop.op = '/'$

13 : $Mop \rightarrow \div$
$Mop.op = '\div'$

functions:

$f_1\ int\ int$	=	int	$f_3\ '*'\ int\ int$	=	int
$f_1\ \_\ \_$	=	$real$	$f_3\ '\div'\ int\ int$	=	int
			$f_3\ '*'\ \_\ \_$	=	$real$
			$f_3\ '/'\ \_\ \_$	=	$real$

□

Wir betrachten das Problem isoliert von der sonstigen semantischen Analyse, d.h. wir setzen voraus, daß für die terminalen Operanden, Konstanten und Variablen, der Typ bereits berechnet wurde. Wie das mithilfe von Attributgrammatiken geschieht, werden wir später sehen.

An diesem Beispiel kann man eine statistisch sehr gut belegte Beobachtung machen; ein Großteil der semantischen Regeln sind Gleichungen (identische Übergaben, Kopieraktionen) zwischen zwei Attributvorkommen. Es ist typisch für Attributgrammatiken, daß an einigen „Stellen" der Grammatik Attribute berechnet, ihre Werte dann lange Strecken transportiert und anschließend an anderen Stellen der Grammatik benutzt werden. Deshalb treffen wir jetzt eine neue Konvention, die die Aufschreibung von Attributgrammatiken von den meisten identischen Übergaben befreit:

> Bei Fehlen einer semantischen Regel für ein ererbtes Attributvorkommen auf der rechten Seite (abgeleitetes Vorkommen auf der linken Seite) wird eine identische Übergabe von einem gleichnamigen ererbten Attributvorkommen auf der linken Seite (abgeleiteten Attributvorkommen auf der rechten Seite) angenommen.

Natürlich muß im Fall der identischen Übergabe an ein abgeleitetes Attribut der linken Seite genau ein gleichnamiges abgeleitetes Attribut auf der rechten Seite auftreten. Die folgenden Beispiele benutzen bereits diese Konvention.

Jetzt folgen zwei Varianten der Typberechnung für arithmetische Ausdrücke, die eine bessere Lokalisierung von Typfehlern erlauben. Denn berechnet sich in

einem Ausdruck $e_1 \div e_2$ gemäß $\mathbf{AG_1}$ z.B. der Typ von e_2 zu *real*, so wird eine Fehlermeldung wegen der Nichtdefiniertheit der Funktion f_3 für diese Kombination von Argumenten gegeben. Es ist aber nicht klar, wo die Ursache für diesen Fehler liegt, etwa in einem reellen terminalen Operanden oder in einer Division in e_2. Die nächste Attributgrammatik, $\mathbf{AG_2}$, in Beispiel 9.3.2, beschreibt, wie in rechten Operanden von $'\div'$ die Ursache für diesen Typfehler lokalisiert werden kann. Sie reicht die Information, daß als Ergebnistyp *integer* verlangt ist, in einem Attribut *obltyp* in Unterausdrücke herunter.

Beispiel 9.3.2 (Attributgrammatik $\mathbf{AG_2}$)

attribute grammar AG_2:
nonterminals $S, E, T, F, P, Aop, Mop;$
attributes **syn** *typ* **with** E, T, F, P, c, v **domain** $\{int, real\};$
syn *op* **with** Aop, Mop **domain** $\{'+','-','*','/','\div'\};$
inh *obltyp* **with** E, T, F, P **domain** $\{int, unspec\};$

rules

0 : $S \rightarrow E$
$E.obltyp := unspec$

1 : $E \rightarrow E\ Aop\ T$
$E_0.typ = f_1(E_1.typ,\ T.typ)$

2 : $E \rightarrow T$

3 : $T \rightarrow T\ Mop\ F$
$T_0.typ = f_3^1(Mop.op,\ T_0.obltyp,\ T_1.typ,\ F.typ)$
$F.obltyp = f_3^2(Mop.op,\ T_0.obltyp)$

4 : $T \rightarrow F$

5 : $F \rightarrow P$

6 : $P \rightarrow c$
$P.typ = f_6(P.obltyp,\ c.typ)$

7 : $P \rightarrow v$
$P.typ = f_6(P.obltyp,\ v.typ)$

8 : $P \rightarrow (E)$

9 : $Aop \rightarrow +$
$Aop.op = '+'$

10 : $Aop \rightarrow -$
$Aop.op = '-'$

11 : $Mop \rightarrow *$
$Mop.op = '*'$

12 : $Mop \rightarrow /$
$Mop.op = '/'$

13 : $Mop \rightarrow \div$
$Mop.op = '\div'$

functions:

$f_3^1\ '*'\ int\ int\ int$	=	int	$f_3^2\ '\div'\ \_$	=	int
$f_3^1\ '/'\ unspec\ \_\ \_$	=	$real$	$f_3^2\ \_\ int$	=	int
$f_3^1\ '\div'\ \_\ int\ int$	=	int	$f_3^2\ \_\ \_$	=	$unspec$
$f_3^1\ '*'\ \_\ \_\ \_$	=	$real$			
$f_6\ int\ int$	=	int	$f_6\ int\ int$	=	int
$f_6\ \_\ \_$	=	$real$	$f_6\ unspec\ t$	=	t

□

Die Attributgrammatik $\mathbf{AG_2}$ hat eine etwas unnatürliche Unsymmetrie an sich (der Eingeweihte merkt, weshalb sie so konstruiert wurde). Nur rechte Operanden von „$\div$" bekommen den von ihnen verlangten Typ mitgeteilt. Diese Unsymmetrie wird jetzt beseitigt, indem die Produktion 3 durch die neue Produktion 3′ wie in Beispiel 9.3.3 ersetzt wird. Die sich ergebende Grammatik werde mit $\mathbf{AG_3}$ bezeichnet.

Beispiel 9.3.3 (Attributgrammatik AG_3)
Sie ergibt sich aus $\mathbf{AG}_2$ durch Ersetzung von Produktion 3 durch 3'.

attribute grammar AG_3:
nonterminals *S, E, T, F, P, Aop, Mop;*
attributes **syn** *typ* **with** *E, T, F, P, c, v* **domain** $\{int, real\}$;
syn *op* **with** *Aop, Mop* **domain** $\{'+','-','*','/','\div'\}$;
inh *obltyp* **with** *E, T, F, P* **domain** $\{int, unspec\}$;

rules

0 : $S \rightarrow E$
$E.obltyp := unspec$

1 : $E \rightarrow E\ Aop\ T$
$E_0.typ = f_1(E_1.typ,\ T.typ)$

3' : $T \rightarrow T\ Mop\ F$
$T_0.typ = f_3^1(Mop.op,\ T_0.obltyp,\ T_1.typ,\ F.typ)$
$T_1.obltyp = f_3^2(Mop.op,\ T_0.obltyp)$
$F.obltyp = f_3^2(Mop.op,\ T_0.obltyp)$

5 : $F \rightarrow P$

7 : $P \rightarrow v$
$P.typ = f_6(P.obltyp,\ v.typ)$

9 : $Aop \rightarrow +$
$Aop.op = '+'$

11 : $Mop \rightarrow *$
$Mop.op = '*'$

2 : $E \rightarrow T$

4 : $T \rightarrow F$

6 : $P \rightarrow c$
$P.typ = f_6(P.obltyp,\ c.typ)$

8 : $P \rightarrow (E)$

10 : $Aop \rightarrow -$
$Aop.op = '-'$

12 : $Mop \rightarrow /$
$Mop.op = '/'$

13 : $Mop \rightarrow \div$
$Mop.op = '\div'$

functions:

$f_3^1\ '*'\ int\ int\ int$	=	*int*	$f_3^2\ '\div'\ \_$	=	*int*
$f_3^1\ '/'\ unspec\ \_\ \_$	=	*real*	$f_3^2\ \_\ int$	=	*int*
$f_3^1\ '\div'\ \_\ int\ int$	=	*int*	$f_3^2\ \_\ \_$	=	*unspec*
$f_3^1\ '*'\ \_\ \_\ \_$	=	*real*			
$f_1\ int\ int$	=	*int*	$f_6\ int\ int$	=	*int*
$f_1\ \_\ \_$	=	*real*	$f_6\ unspec\ \mathrm{t}$	=	t

□

Die nächste Grammatik erhalten wir, indem wir als zusätzlichen Operator die Potenz zulassen. In der Einleitung wurde allerdings gezeigt, daß dann die Typ-Eigenschaft keine statische Eigenschaft mehr ist, weil der Typ von $e_1 \uparrow e_2$ für integer-Ausdrücke e_1 und e_2 von der Größe von e_2 abhängt. Deshalb machen wir für $\mathbf{AG}_4$ eine Einschränkung; integer-Exponenten zu einer integer-Basis müssen nichtnegative Werte haben. Als zusätzliche Attribute enthält $\mathbf{AG}_4$ ein *value*-Attribut bei Konstanten und ein *oblsize*- Attribut für die E, T, F und P. In letzterem wird festgehalten, welche Größe der Wert des Ausdrucks (vom Typ *int*) haben muß. Der Wert des Attributs kann dann dazu benutzt werden, Laufzeittests mit der entsprechenden Bedingung zu erzeugen.

Beispiel 9.3.4 (Attributgrammatik AG_4)

attribute grammar AG_4:
nonterminals *S, E, T, F, P, Aop, Mop;*
attributes **syn** *typ* **with** *E, T, F, P, c, v* **domain** $\{int, real\}$;
syn *op* **with** *Aop, Mop* **domain** $\{'+','-','*','/','\div'\}$;
syn *value* **with** *c* **domain** $int \cup real$;
inh *obltyp* **with** *E, T, F, P* **domain** $\{int, unspec\}$;
inh *oblsize* **with** *E, T, F, P* **domain** $\{unspec, nonneg\}$

rules

0 : $S \rightarrow E$
$E.obltyp := unspec$
$E.oblsize := unspec$

1 : $E \rightarrow E\ Aop\ T$
$E_0.typ = f_1(E_1.typ,\ T.typ)$

2 : $E \rightarrow T$

3' : $T \rightarrow T\ Mop\ F$
$T_0.typ = f_3^1(Mop.op,\ T_0.obltyp,\ T_1.typ,\ F.typ)$
$T_1.obltyp = f_3^2(Mop.op,\ T_0.obltyp)$
$F.obltyp = f_3^2(Mop.op,\ T_0.obltyp)$

4 : $T \rightarrow F$

5' : $F \rightarrow P \uparrow F$
$F_1.oblsize = f_{5'}^2(P.typ)$
$P.oblsize = f_{5'}^3(F_1.typ)$
$F_0.typ = f_{5'}^1(F_0.obltyp,\ P.typ,\ F_1.typ)$

6' : $P \rightarrow c$
$P.typ = f_{6'}(P.obltyp,\ P.oblsize,\ c.typ,\ c.value)$

7' : $P \rightarrow v$
$P.typ = f_{7'}(P.obltyp,\ v.typ)$

8 : $P \rightarrow (E)$

9 : $Aop \rightarrow +$
$Aop.op = '+'$

10 : $Aop \rightarrow -$
$Aop.op = '-'$

11 : $Mop \rightarrow *$
$Mop.op = '*'$

12 : $Mop \rightarrow /$
$Mop.op = '/'$

13 : $Mop \rightarrow \div$
$Mop.op = '\div'$

functions:

$f_3^1\ '*'\ int\ int\ int$	=	*int*	$f_3^2\ '\div'\ \_$	=	*int*
$f_3^1\ '/'\ unspec\ \_\ \_$	=	*real*	$f_3^2\ \_\ int$	=	*int*
$f_3^1\ '\div'\ \_\ int\ int$	=	*int*	$f_3^2\ \_\ \_$	=	*unspec*
$f_3^1\ '*'\ \_\ \_\ \_$	=	*real*			
$f_1\ int\ int$	=	*int*			
$f_1\ \_\ \_$	=	*real*			
$f_{5'}^1\ int\ int\ int$	=	*int*	$f_{5'}^2\ int$	=	*nonneg*
$f_{5'}^1\ \_\ \_\ \_$	=	*real*	$f_{5'}^2\ real$	=	*unspec*
$f_{6'}^3\ int$	=	*unspec*	$f_{6'}\ int\ nonneg\ int\ nonneg$	=	*int*
$f_{6'}^3\ real$	=	*nonneg*	$f_{6'}\ int\ unspec\ int\ \_$	=	*int*
			$f_{6'}\ unspec\ \_\ \mathrm{t}\ \_$	=	t
$f_{7'}\ int\ int$	=	*int*			
$f_{7'}\ unspec\ \mathrm{t}$	=	t			

□

Ein weiteres Beispiel ist die folgende Attributgrammatik **Bin_to_Dec**, die die Umrechnung von Binär- in Dezimalzahlen beschreibt.

Beispiel 9.3.5 (Attributgrammatik Bin_to_Dec)

attribute grammar Bin_to_Dec:
nonterminals {*N, BIN, BIT* };
attributes **syn** *l* **with** *BIN* **domain int**;
syn *v* **with** *N, BIN, BIT* **domain real**;
inh *r* **with** *BIN, BIT* **domain int**;
rules

1 : $N \rightarrow BIN.\ BIN$
$N.v = BIN_1.v + BIN_2.v$
$BIN_1.r = 0$
$BIN_2.r = -\ BIN_2.l$

2 : $BIN \rightarrow BIN\ BIT$
$BIN_0.v = BIN_1.v + BIT.v$
$BIN_0.l = BIN_1.l + 1$
$BIN_1.r = BIN_0.r + 1$
$BIT.r = BIN_0.r$

3 : $BIN \rightarrow \varepsilon$
$BIN.v = 0$
$BIN.l = 0$

4 : $BIT \rightarrow 1$
$BIT.v = 2^{BIT.r}$

5 : $BIT \rightarrow 0$
$BIT.v = 0$

□

Wir betrachten nun, wie die Grammatik Binärzahlen in Dezimalzahlen umrechnet. Die Werte werden in dem Attribut v berechnet. Die Regeln für v sind einsichtig. Nur die Regel für der Produktion 4 erfordert, daß der Rang r der Position von *BIT* bekannt ist. Der Rang entspricht der Position von *BIT* relativ zum binären Punkt. Positionen links vom Punkt haben den Rang $0, 1, 2, \ldots$; diejenigen rechts vom Punkt haben den Rang $-1, -2, -3, \ldots$, jeweils vom Punkt aus gezählt.

Wie werden die Stellen vor dem Punkt behandelt? Das *BIT*, das am weitesten rechts steht, hat den Rang 0. Wenn dieser Anfangswert gegeben ist, kann man leicht das Attribut r in der rekursiven Produktion 2 ausrechnen. Der *BIT*-String rechts vom Punkt ist etwas schwieriger zu behandeln. Der Rang des am weitesten rechts stehenden *BIT*s entspricht der Länge des Strings. Deswegen wird mit dem abgeleiteten Attribut l die Länge des *BIT*-Strings bestimmt und mit diesem Wert dann das r-Attribut auf der rechten Seite initialisiert.

Beispiel 9.3.6 Die Attributgrammatik **BoolExp** beschreibt die Codeerzeugung für die sogenannte Kurzschlußauswertung Boolescher Ausdrücke. Der erzeugte Code für einen Booleschen Ausdruck hat die folgenden Eigenschaften:

- es werden nur Lade-Befehle und bedingte Sprünge generiert;
- für die Booleschen Operatoren **and, or** und **not** werden keine Befehle erzeugt;
- die Teilausdrücke des Ausdrucks werden von links nach rechts ausgewertet;

- von jedem (Teil-)Ausdruck werden nur die kleinsten Teilausdrücke ausgewertet, die den Wert des ganzen (Teil-)Ausdrucks eindeutig bestimmen.

Für den Booleschen Ausdruck (*a* **and** *b*) **or not** *c* mit Booleschen Variablen a, b und c wird die folgende Befehlsfolge erzeugt:

```
        LOAD a
        JUMPF L1          jump-on-false
        LOAD b
        JUMPT L2          jump-on-true
L1:     LOAD c
        JUMPT L3
L2:     Code für den Nachfolger im Wahrfall
L3:     Code für den Nachfolger im Falschfall
```

Die Attributgrammatik **BoolExp** beschreibt die Codeerzeugung, dabei insbesondere die Erzeugung von Sprungzielen (Marken) für Teilausdrücke und den Transport solcher Marken an die primitiven Teilausdrücke, von denen aus man diese Marken anspringen kann. Jeder Teilausdruck wird mit zwei Marken versorgt, der Marke des Nachfolgers im „Falschfall" und der des Nachfolgers im „Wahrfall".

Außerdem wird im abgeleiteten Attribut *jcond* berechnet, wie die Korrelation des Wertes des ganzen Ausdrucks mit dem Wert seines am weitesten rechts stehenden Identifiers ist. Hat *jcond* bei einem Ausdruck e den Wert *true*, so heißt das:

Wenn bei der Ausführung der Übersetzung von e der letzte Identifier geladen wird, so ist dessen Wert gleich dem Wert von e.

Hat *jcond* den Wert *false*, so sind die Werte des letzten Identifiers von e und von e die Negation voneinander. Entsprechend gilt für die Codeerzeugung:

Wenn für den letzten Identifier eines Ausdrucks e ein **LOAD** erzeugt wurde, so bestimmt der *jcond*-Wert von e, ob der Sprung zum Wahrnachfolger ein **JUMPT** (im Falle *jcond* = *true*) oder ein **JUMPF** (im Falle *jcond* = *false*) sein muß.

Um einen Kontext für Boolesche Ausdrücke zu haben, nehmen wir noch eine Produktion für die zweiseitige bedingte Anweisung hinzu. Der Nachfolger im Wahrfall ist der then-Teil, der im Falschfall der else-Teil. Wie üblich, erzeugen wir am Ende der Bedingung einen bedingten Sprung zum else-Teil. Dieser testet die Bedingung *E* auf *false*. Deswegen wird in der gencjump-Funktion als erster Parameter **not** *jcond* benutzt.

attribute grammar BoolExp:
nonterminals *IFSTAT, STATS, E, T, F;*

attributes **inh** *tsucc, fsucc* **with** *E,T,F* **domain string**;
syn *jcond* **with** *E,T,F* **domain bool**;
syn *code* **with** *IFSTAT, E,T,F* **domain string**;

rules
IFSTAT → **if** *E* **then** *STATS* **else** *STATS* **fi**
E.tsucc = *t*
E.fsucc = *e*
IFSTAT.code = *E.code* ++ *gencjump*(**not** *E.jcond*, *e*) ++
t: ++ *STATS*$_1$*.code* ++ *genujump*(*f*) ++ *e:* ++ *STATS*$_2$*.code* ++ *f:*
E → *T*
E → *E* **or** *T*
E$_1$*.fsucc* = *t*
E$_0$*.jcond* = *T.jcond*
E$_0$*.code* = *E*$_1$*.code* ++ *gencjump*(*E*$_1$*.jcond*, *E*$_0$*.tsucc*) ++ *t:* ++ *T.co de*
T → *F*
T → *T* **and** *F*
T$_1$*.tsucc* = *f*
T$_0$*.jcond* = *F.jcond*
T$_0$*.code* = *T*$_1$*.code* ++ *gencjump*(**not** *T*$_1$*.jcond*, *T*$_0$*.fsucc*) ++ *f:* ++ *F.code*
F → (*E*)
F → **not** *F*
F$_1$*.tsucc* = *F*$_0$*.fsucc*
F$_1$*.fsucc* = *F*$_0$*.tsucc*
F$_0$*.jcond* = **not** *F*$_1$*.jcond*
F → **id**
F.jcond = *true*
F.code = **LOAD id**.*identifier*

Verwendete Hilfsfunktionen:

genujump(*l*) = **JUMP** *l*
gencjump(*jc*, *l*) = **if** *jc* = *true*
then JUMPT *l*
else JUMPF *l*
fi

□

9.4 Die Generierung von Attributauswertern

In diesem Abschnitt befassen wir uns mit der Auswertung von Attributen, genauer Attributexemplaren, in Syntaxbäumen. Die Attributgrammatik definiert für jeden Syntaxbaum der zugrundeliegenden kontextfreien Grammatik ein Gleichungssystem. Die Unbekannten darin sind die Attributexemplare zu den Knoten des Syntaxbaums. Nehmen wir an, das Gleichungssystem sei nicht rekursiv. Dann könnte man es durch ein Eliminationsverfahren lösen und damit den Attributexemplaren Werte zuordnen. In jedem Eliminationsschritt müßte man jeweils

das nächste Attributexemplar suchen, welches nur von bereits berechneten Exemplaren abhängt, und seinen Wert berechnen. Dieses wäre ein vollkommen dynamisches Vorgehen, da es beim Eliminationsverfahren keinerlei Informationen darüber ausnutzt, woraus dieses Gleichungssystem entstanden ist.

Statische Attributauswertungsverfahren nutzen Wissen über die Attributgrammatik aus. Jede Attributgrammatik beschreibt Abhängigkeiten zwischen den Attributvorkommen der Produktionen. Dabei hängt ein Attributvorkommen a_i von einem Vorkommen b_j ab, wenn b_j ein Argument für die semantische Regel von a_i ist. Diese Abhängigkeiten bestimmen die Abhängigkeiten zwischen Attributexemplaren in den oben genannten Gleichungssystemen. Wenn ein entsprechender Generator die Abhängigkeiten in der Attributgrammatik analysiert, so kann er gegebenenfalls einen Attributauswerter erzeugen, der nicht jeweils nach dem nächsten auszuwertenden Attributexemplar suchen muß, sondern die Attributexemplare gemäß einer statisch bestimmten „Besuchsreihenfolge" auswertet. Betrachten wir noch einmal Abbildung 9.6. Sei diese Produktion, p, irgendwo im Inneren eines Baumes t angewendet. Die Attributauswertung erfordert dann ein Zusammenspiel von Berechnungen, die lokal zu dem Vorkommen der Produktion sind, und solchen in der Umgebung. Eine lokale Berechnung eines (Exemplars eines) definierenden Vorkommens bei X_0 stellt der nächsten Produktion, q, oberhalb einen neuen Wert zur Verfügung. Eine Berechnung eines Attributexemplars am gleichen Knoten, die lokal zu q stattfindet, macht p einen neuen Wert verfügbar, der eventuell wieder neue lokale Berechnungen in p ermöglicht. Ähnliches passiert zwischen diesem Vorkommen von p und den darunter angewandten Produktionen. Möchte man dieses Zusammenspiel statisch planen, so muß man u.a. globale Abhängigkeiten analysieren. Diese Abhängigkeiten beschreiben für ein Nichtterminal X, von welchen Attributen es die Werte vom Kontext geliefert bekommen kann, wenn es ihm die Werte gewisser Attribute liefert. Der Kontext kann dabei ein Teilbaum für X oder ein oberes Baumfragment für X sein. Ausgangspunkt für die Berechnung der globalen Abhängigkeitsrelationen sind natürlich die produktionslokalen Abhängigkeiten.

Zur Behandlung dieser Generierungsverfahren müssen wir eine Reihe von Begriffen, Algorithmen und Eigenschaften kennenlernen.

9.4.1 Attributabhängigkeiten

Die semantischen Regeln zur Produktion p einer Attributgrammatik induzieren eine Relation auf der Menge $V(p)$ der Attributvorkommen der Produktion.

Definition 9.4.1 (produktionslokale Abhängigkeit)
Die **produktionslokale Abhängigkeitsrelation** $Dp(p) \subseteq V(p) \times V(p)$ zur Produktion p ist folgendermaßen definiert:

$$b_j \; Dp(p) \; a_i \qquad \text{genau dann, wenn} \qquad a_i = f_{p,a,i}(\ldots, b_j, \ldots)$$

für eine semantische Regel zu p. Das Vorkommen des Attributs b bei X_j steht also in der Relation zum Vorkommen von a bei X_i, oder a_i **hängt von** b_j **ab**,