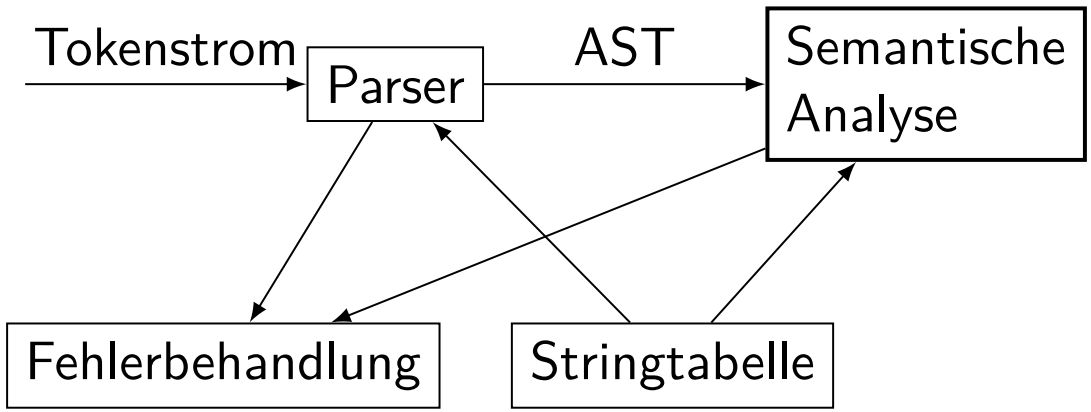
Kapitel 4

Attributierte Grammatiken

Kapitel 4: Attributierte Grammatiken

Schnittstelle Parser - semantische Analyse



Warum Attributgrammatiken?

- Die semantische Analyse hat nur den Strukturbaum, um Informationen zu gewinnen.
  - Programmiersprachen sind kompositionell definiert
  - Gesucht effiziente Berechnungsmethode
  - Art der Berechnungen ist abhängig vom jeweiligen Knotentyp
- Attributgrammatiken (AGs) sind ein systematischer Ansatz für solche Aufgaben
  - AGs erlauben eine von konkreten Berechnungsreihenfolgen unabhängige Spezifikation
  - Die Bearbeitung von XML-Bäumen ist eigentlich eine direkte Anwendung von AGs, was die meisten Leute aber nicht wissen

Attributgrammatiken (AG)

Verfahren zur Spezifikation der Eigenschaften von Bäumen

- Baum beschrieben durch kontextfreie Grammatik

wird eingesetzt zur Spezifikation der semantischen Analyse auch bei vielen anderen Aufgaben im software engineering zu gebrauchen

Denkschema:

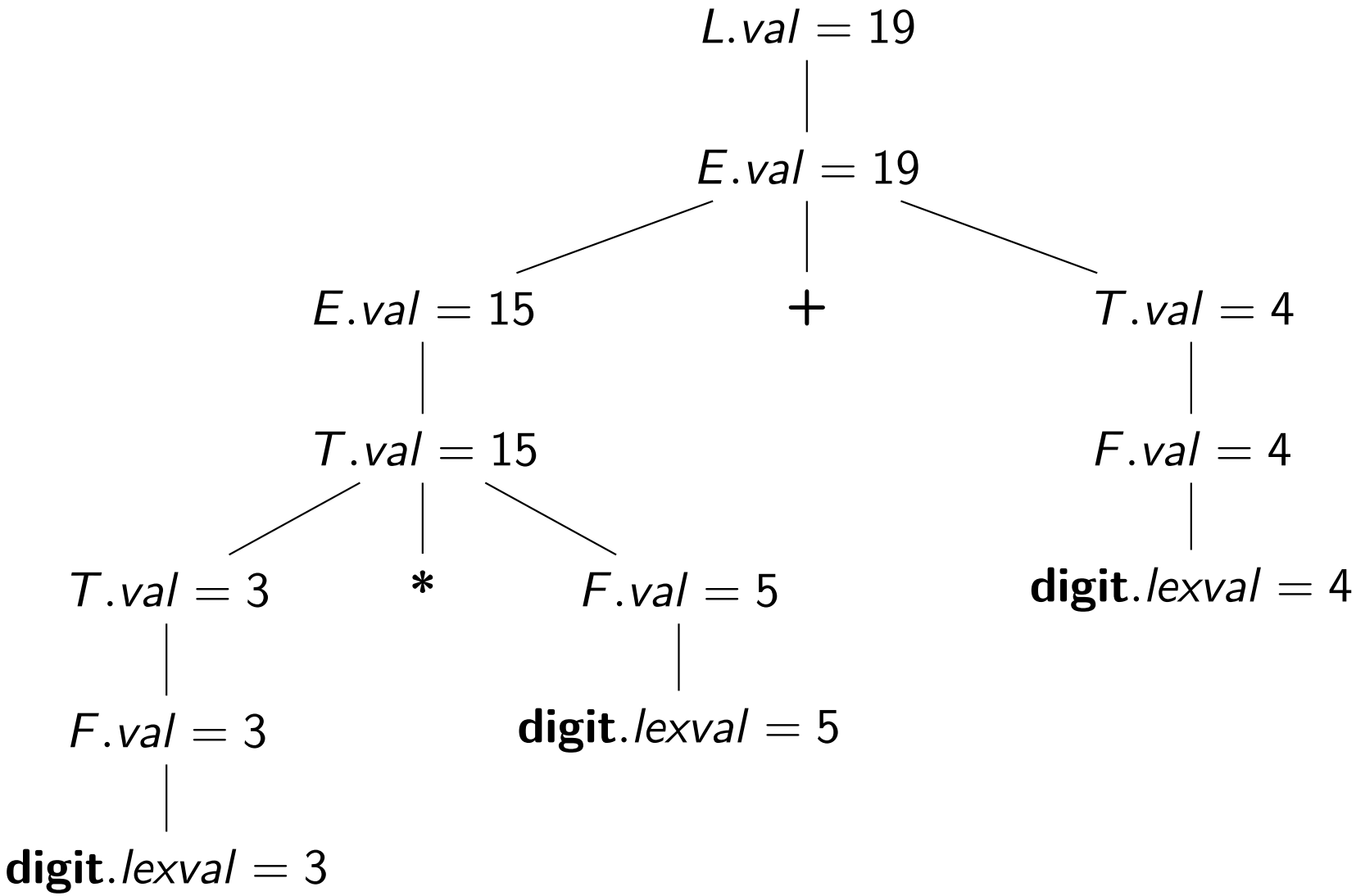
- Jeder Knoten im Baum besitzt ein oder mehrere Attribute
- Attribute können vorbesetzt sein, z.B. Positionsangaben, Bezeichner: Verweis in Stringtabelle
- Attribute verschiedener Knoten zur gleichen Produktion der kfG können voneinander abhängen
- Abhängige Attribute im Kontext einer Produktion aus anderen berechnen
- Jedes Attribut wird genau einmal berechnet, sonst gibt es (im allgemeinen) Konsistenzprobleme

Kapitel 4: Attributierte Grammatiken

Beispiel: Taschenrechner mit Attributierter Grammatik

	Produktion	**Semantische Regeln**
1)	$L \to E$	$L.val = E.val$
2)	$E \to E_1$ **+** T	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1$ ***** F	$T.val = T_1.val * F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to$ **(**E**)**	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**$.lexval$

Attributierter Parsebaum für **3 * 5 + 4**



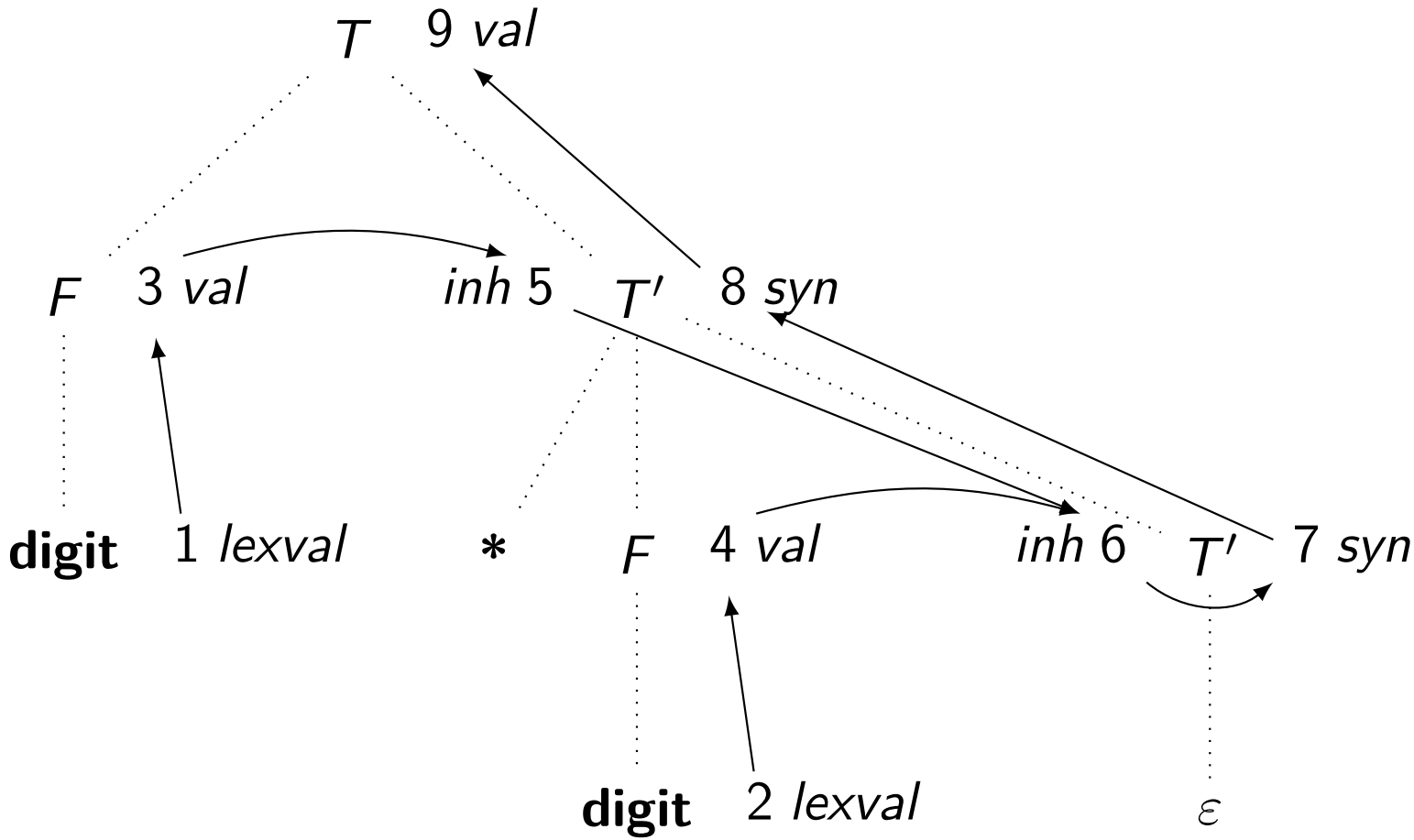
Taschenrechner Implementierung

$$
\begin{array}{ll}
L \to E & \{\ \text{print}(E.val);\ \} \\
E \to E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E \to T & \{\ E.val = T.val;\ \} \\
T \to T_1 * F & \{\ T.val = T_1.val * F.val;\ \} \\
T \to F & \{\ T.val = F.val;\ \} \\
F \to (E) & \{\ F.val = E.val;\ \} \\
F \to \textbf{digit} & \{\ F.val = \textbf{digit}.lexval;\ \}
\end{array}
$$

Grammatik mit Attributierung

	Produktion	**Semantische Regeln**
1)	$T \rightarrow FT'$	$T'.inh = F.val$
		$T.val = T'.syn$
2)	$T' \rightarrow$ ***** FT'_1	$T'_1.inh = T'.inh * F.val$
		$T'.syn = T'_1.syn$
3)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Abhängigkeitsgraph für attributierten Parsebaum

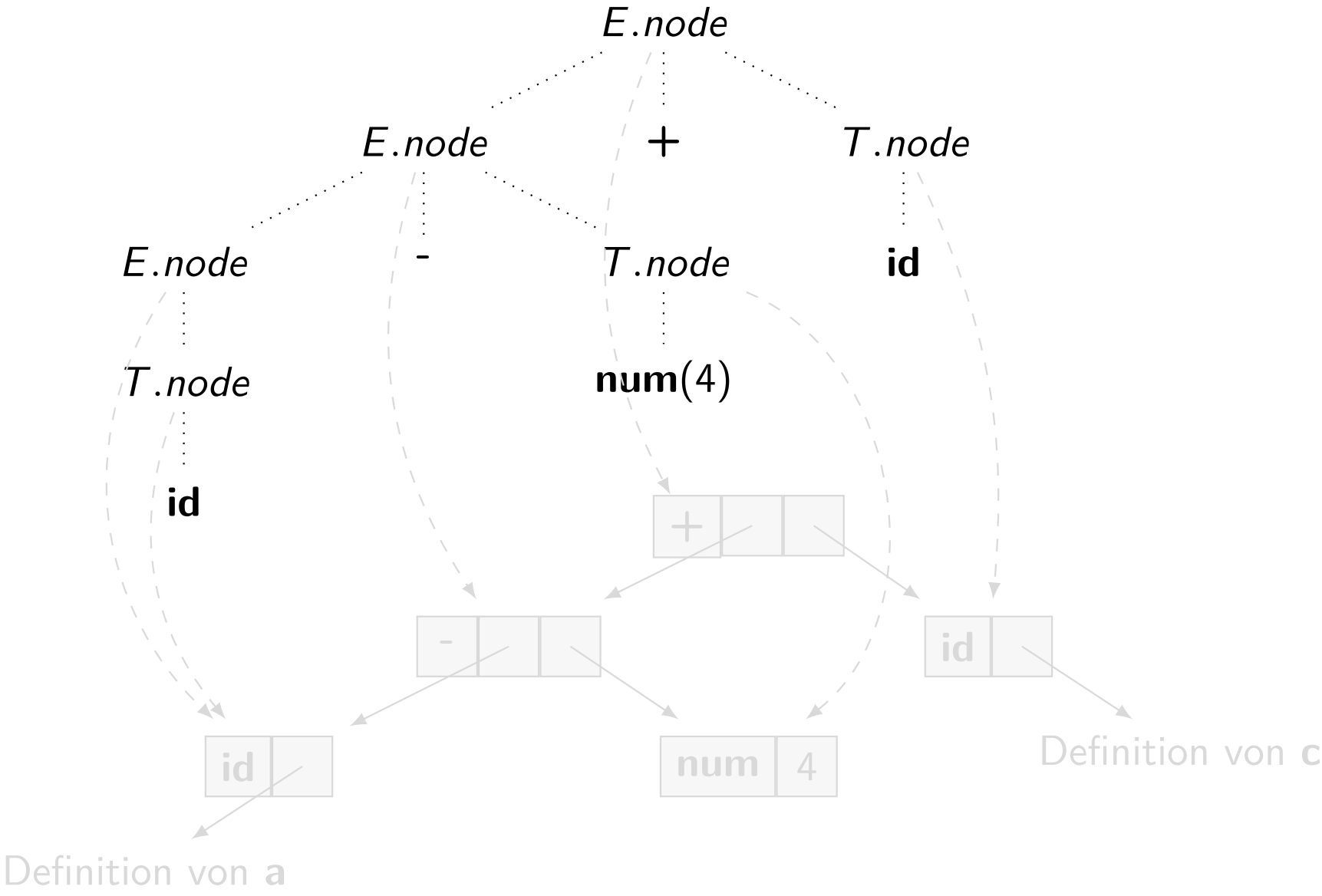


Parse-Baum für **digit * digit**.

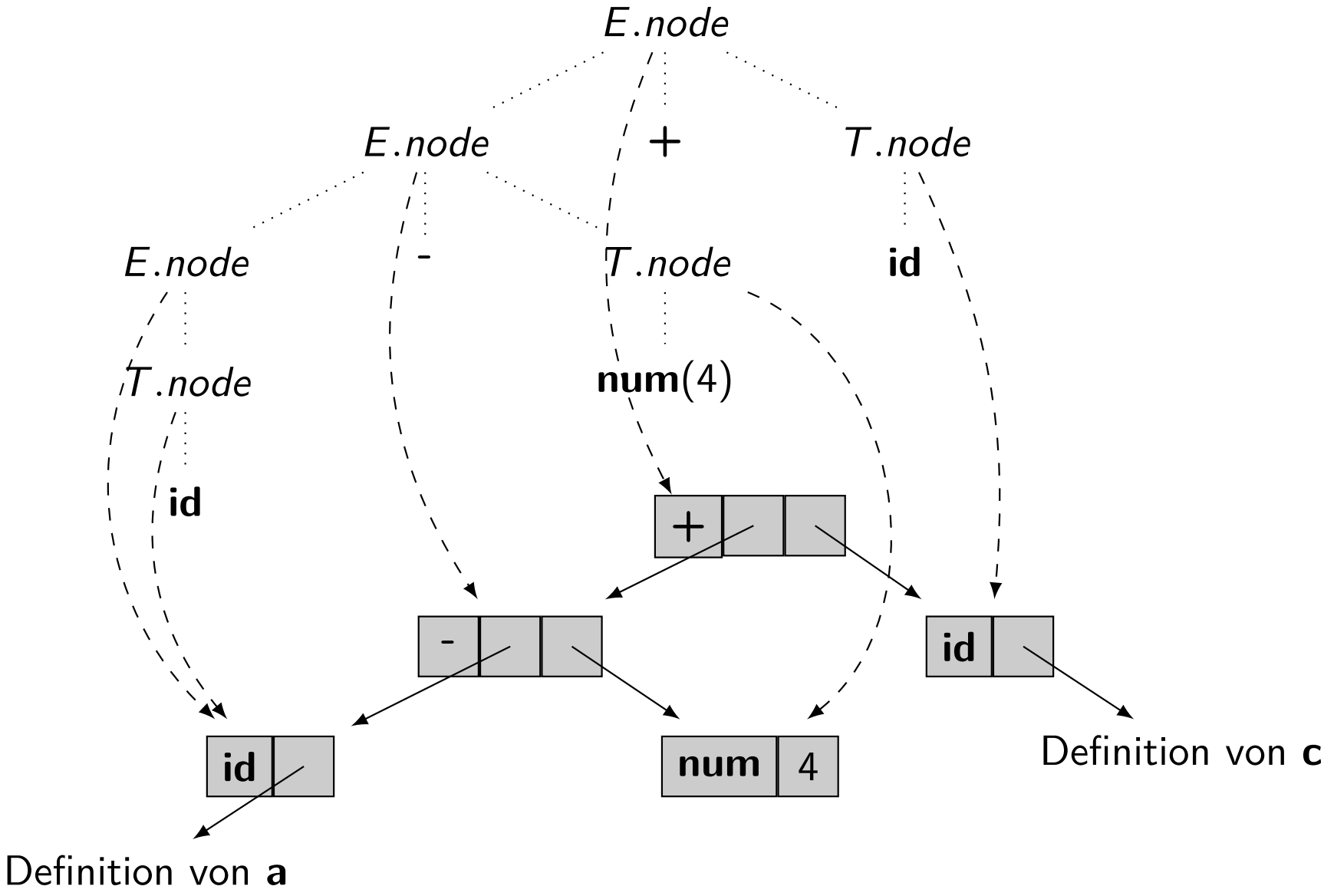
AST-Aufbau mit AGs

	Produktion	**Semantische Regeln**
1)	$E \to E_1 + T$	$E.node = \texttt{new Node}(+, E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \texttt{new Node}(-, E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \mathbf{id}$	$T.node = \texttt{new Leaf}(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \to \mathbf{num}$	$T.node = \texttt{new Leaf}(\mathbf{num}, \mathbf{num}.val)$

Beispiel AST-Aufbau



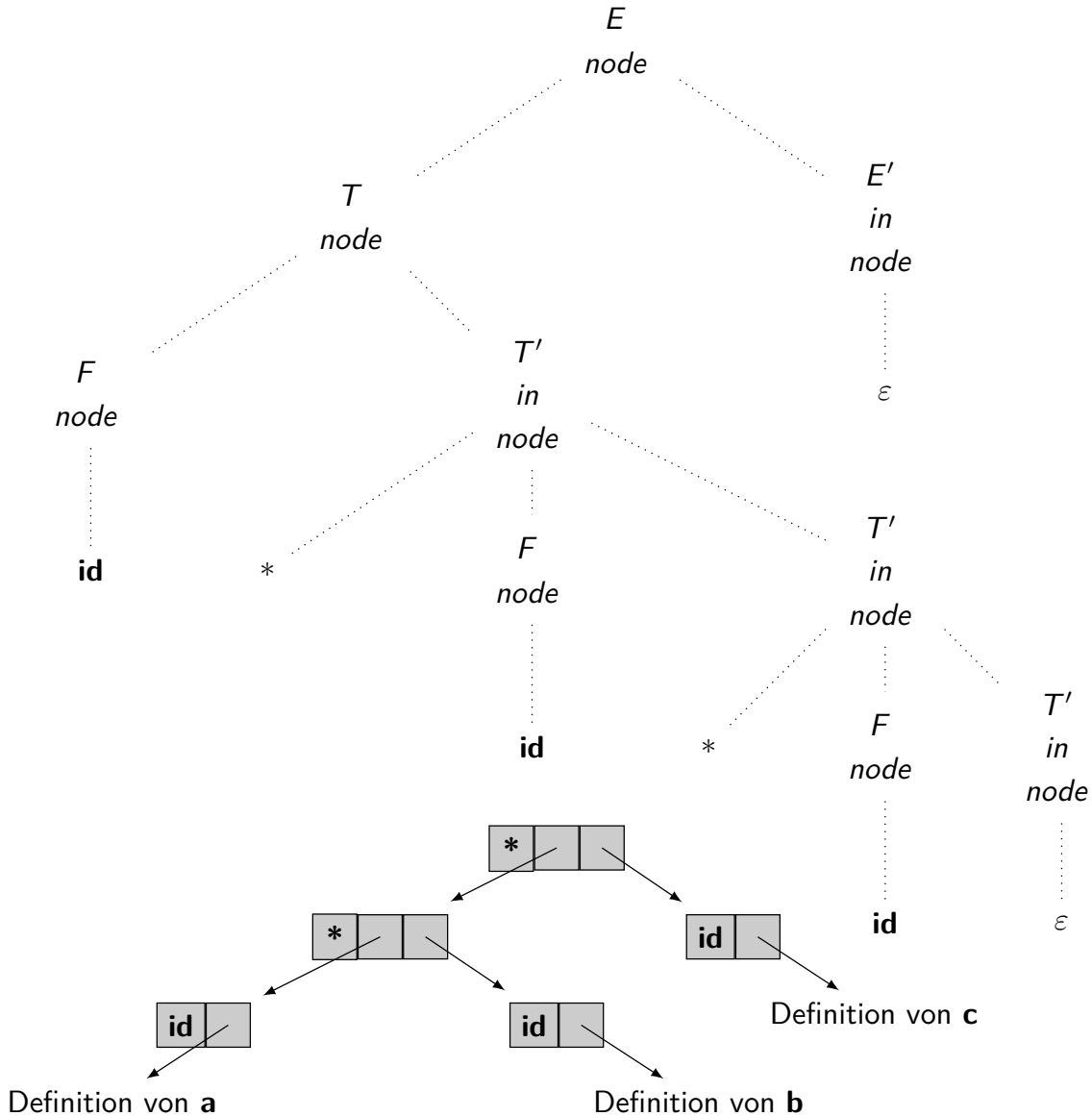
Beispiel AST-Aufbau



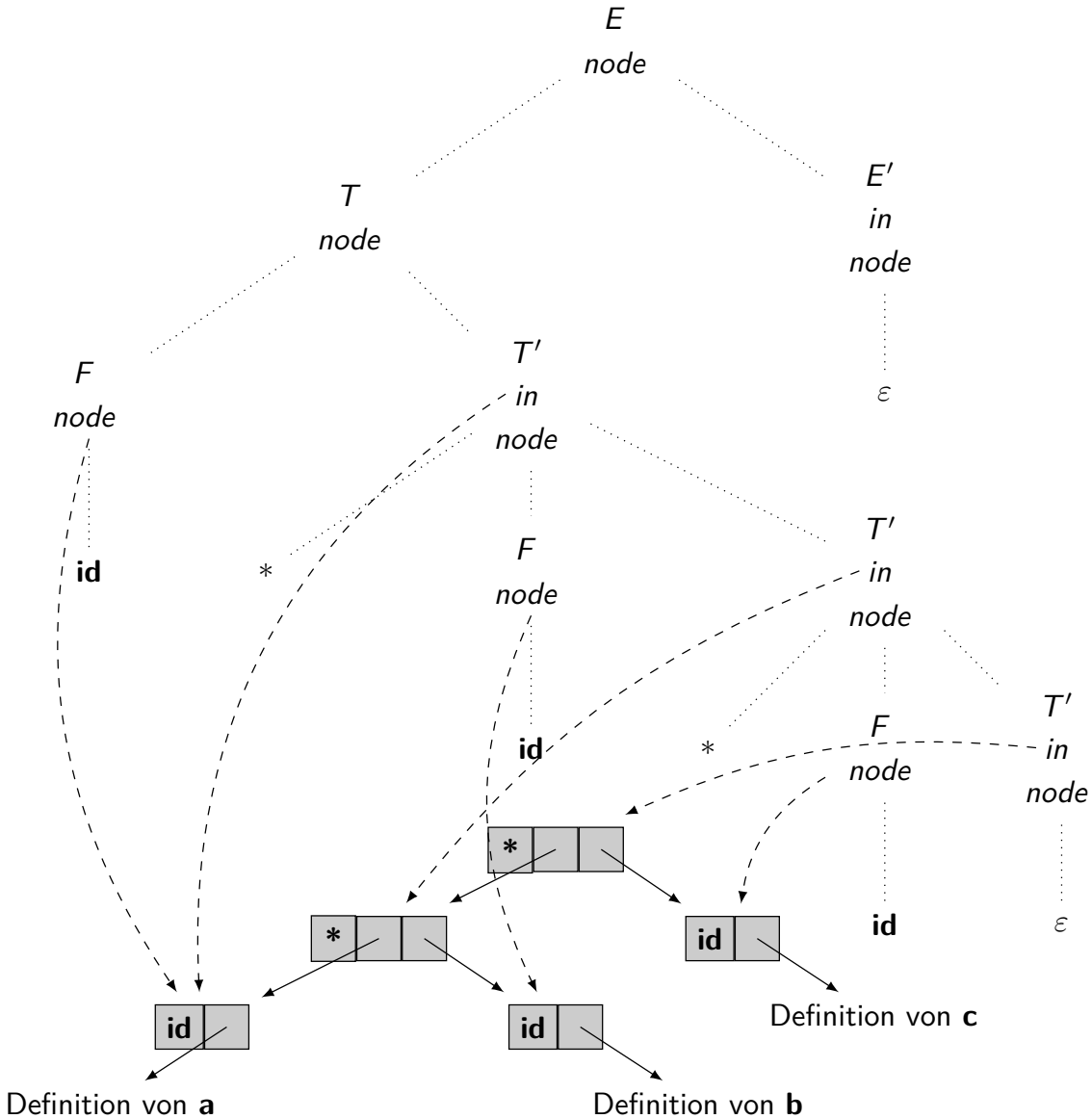
AST-Aufbau für LL(1)-Beispielgrammatik

	Produktion	**Semantische Regeln**
1)	$E \to T\ E'$	$E.node = E'.node$ $E'.in = T.node$
2)	$E' \to \varepsilon$	$E'.node = E'.in$
3)	$E'_1 \to +\ T\ E'_2$	$E'_2.in = \texttt{new Node}(+, E'_1.in, T.node)$ $E'_1.node = E'_2.node$
4)	$T \to F\ T'$	$T.node = T'.node$ $T'.in = F.node$
5)	$T' \to \varepsilon$	$T'.node = T'.in$
6)	$T'_1 \to *\ F\ T'_2$	$T'_2.in = \texttt{new Node}(*, T'_1.in, F.node)$ $T'_1.node = T'_2.node$
7)	$F \to \textbf{id}$	$F.node = \texttt{new Leaf}(\textbf{id}, \textbf{id}.entry)$
8)	$F \to (\ E\ )$	$F.node = E.node$

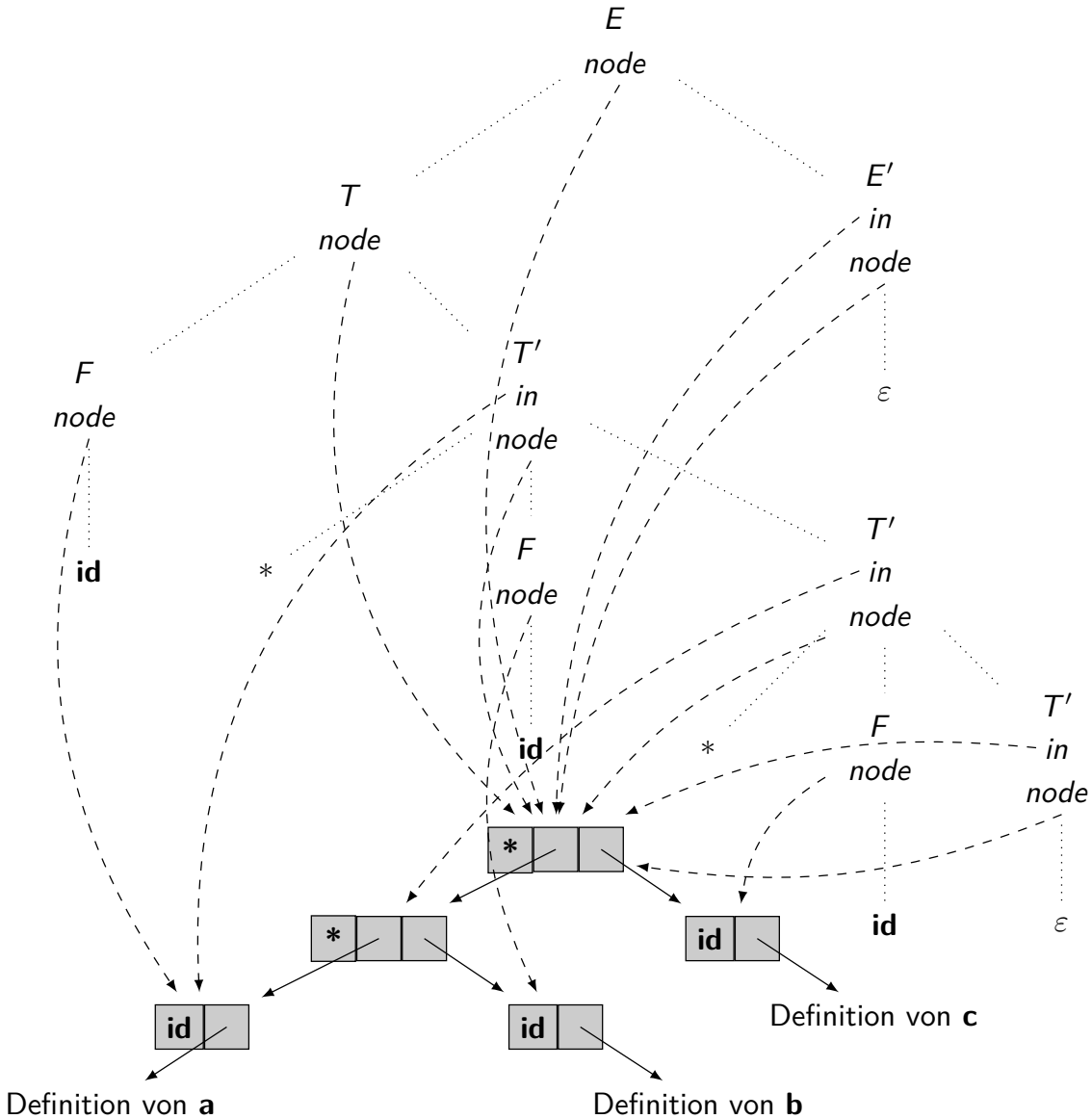
AST für $a * b * c$


E
node
T
node
E′
in
node
F
node
T′
in
node
ε
id
*
F
node
T′
in
node
id
*
F
node
T′
in
node
id
ε
*
*
id
id
id
Definition von a
Definition von b
Definition von c

AST für $a * b * c$


E
node
T
node
E′
in
node
F
node
T′
in
node
ε
id
*
F
node
T′
in
node
id
*
F
node
T′
in
node
id
ε
*
*
id
id
id
Definition von a
Definition von b
Definition von c

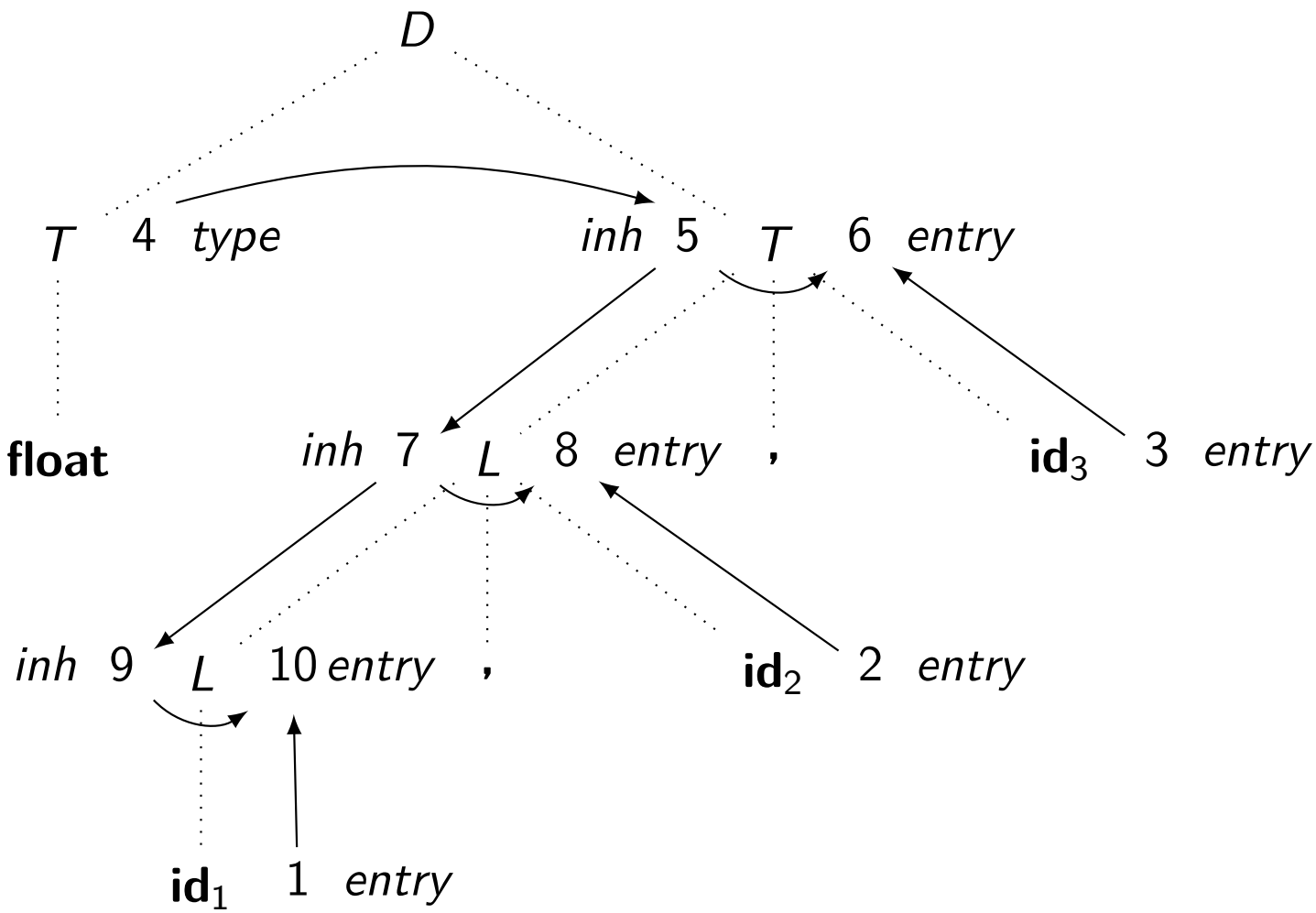
AST für $a * b * c$


E
node
T
node
E′
in
node
F
node
T′
in
node
ε
id
*
F
node
T′
in
node
id
*
F
node
T′
in
node
id
id
ε
*
*
id
id
id
Definition von a
Definition von b
Definition von c

Grammatik mit Attributierung für Typdeklarationen

	Produktion	**Semantische Regeln**
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ `Integer`
3)	$T \rightarrow$ **float**	$T.type =$ `Float`
4)	$L \rightarrow L_1$ **, id**	$L_1.inh = L.inh$
		id.*entry*.`addType`($L.inh$)
5)	$L \rightarrow$ **id**	**id**.*entry*.`addType`($L.inh$)

Abhängigkeitsgraph für attributierten Parsebaum



Parse-Baum für **float** $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$.

Attributierte Grammatik für Arraytypen

Produktion	**Semantische Regeln**
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t =$ `Integer`
$B \rightarrow$ **float**	$B.t =$ `Float`
$C \rightarrow$ **[num]** C_1	$C.t =$ `new Array`$(\textbf{num}.val, C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \varepsilon$	$C.t = C.b$

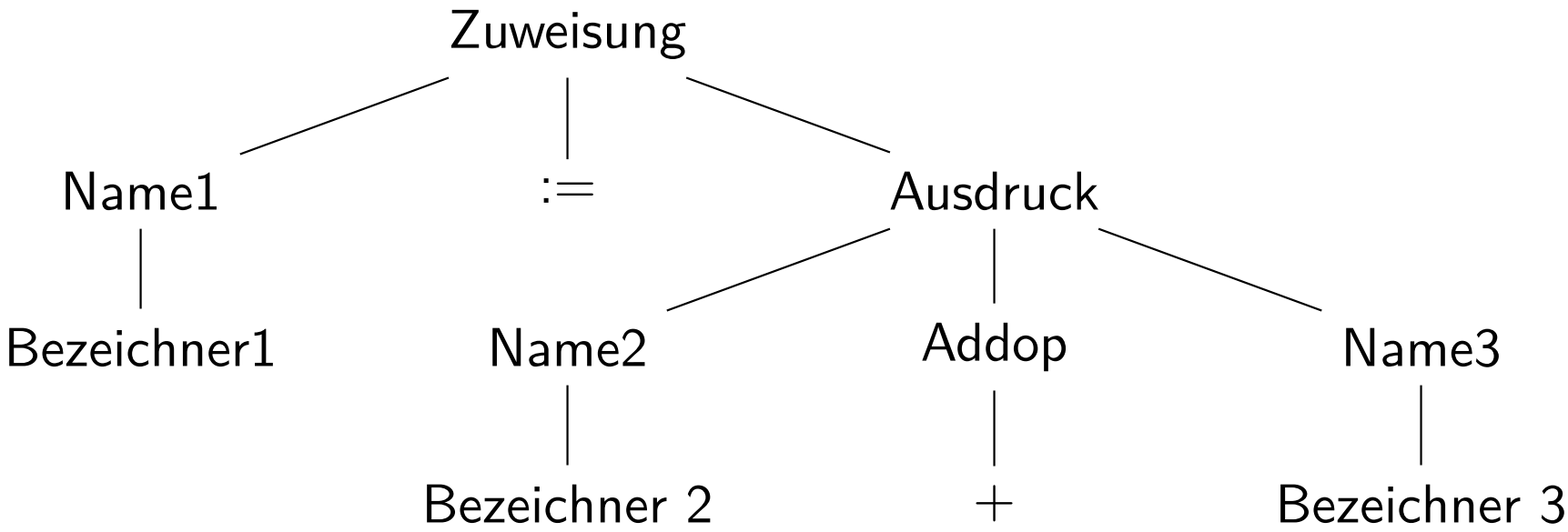
Beispiel Schriftsatz

Abbildung 5.24: Konstruieren größerer Kästen aus kleineren

Attributierte Grammatik für den Schriftsatz

	Produktion	**Semantische Regeln**
1)	$S \to B$	$B.ps = 10$
2)	$B \to B_1\ B_2$	$B_1.ps = B.ps$
		$B_2.ps = B.ps$
		$B.ht = \max(B_1.ht, B_2.ht)$
		$B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \to B_1$ **sub** B_2	$B_1.ps = B.ps$
		$B_2.ps = 0.7 * B.ps$
		$B.ht = \max(B_1.ht, B_2.ht - 0.25 * B.ps)$
		$B.dp = \max(B_1.dp, B_2.dp + 0.25 * B.ps)$
4)	$B \to$ **(** B_1 **)**	$B_1.ps = B.ps$
		$B.ht = B_1.ht$
		$B.dp = B_1.dp$
5)	$B \to$ **text**	$B.ht = \texttt{getHt}(B.ps, \textbf{text}.lexval)$
		$B.dp = \texttt{getDp}(B.ps, \textbf{text}.lexval)$

Beispiel Typanpassung - konkrete Syntax



Typanpassung - Beispiel AG

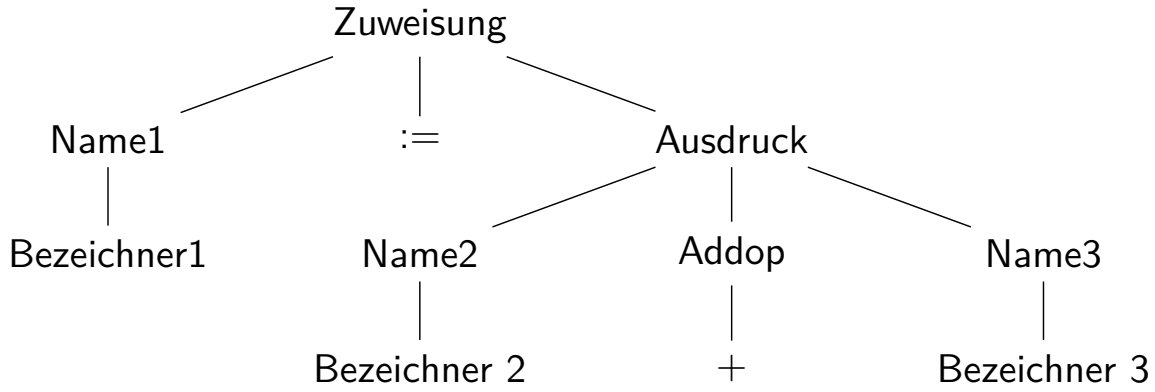
rule Zuweisung → Name ':=' Ausdruck .
attribution
Name.umg := Zuweisung.umg;
Ausdruck.umg := Zuweisung.umg;
Name.nach := Name.vor;
Ausdruck.nach := **if** Name.vor = int **then** int **else** float **end**;

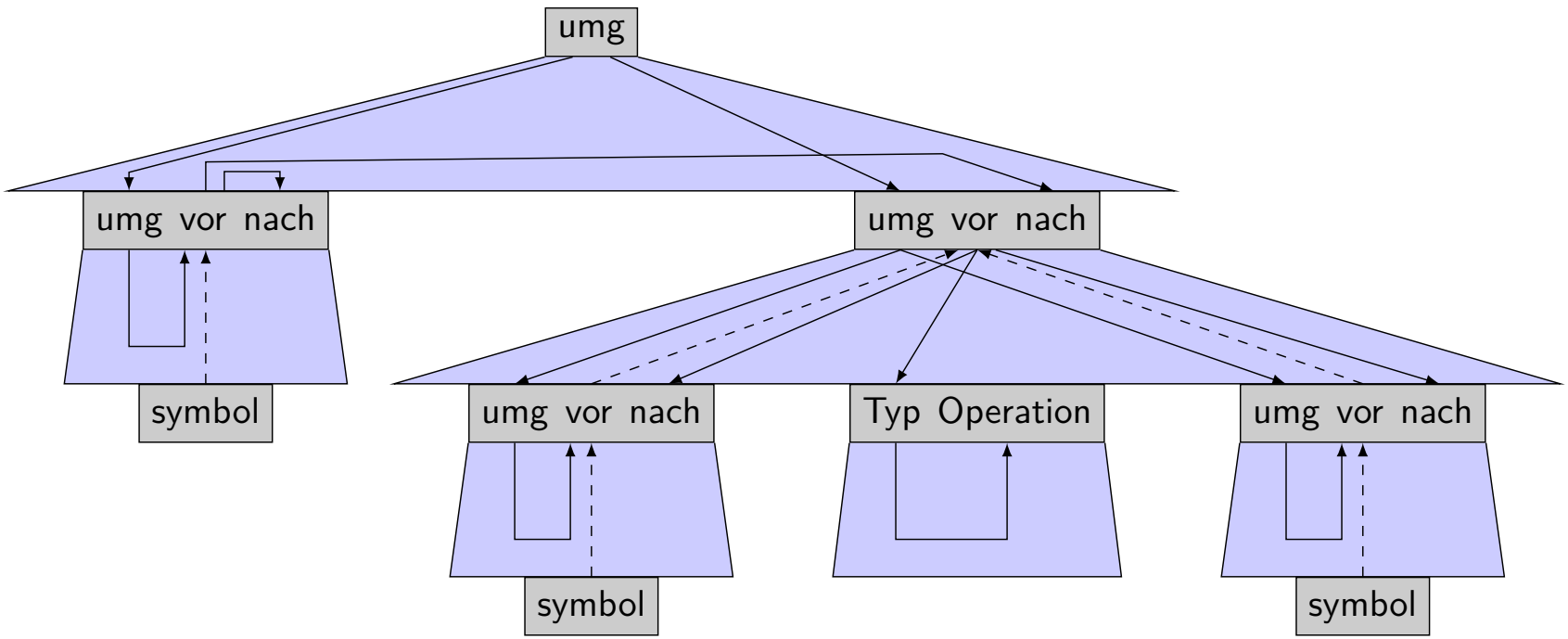
rule Ausdruck → Name addop Name .
attribution
Name[1].umg := Ausdruck.umg;
Name[2].umg := Ausdruck.umg;
Ausdruck.vor := **if** anpassbar(Name[1].vor, int) ∧ anpassbar(Name[2].vor, int)
  then int **else** float **end**;
addop.Typ := Ausdruck.vor;
Name[1].nach := Ausdruck.vor;
Name[2].nach := Ausdruck.vor;
condition anpassbar(Ausdruck.vor, Ausdruck.nach);

rule addop → '+' .
attribution addop.operation := **if** addop.Typ = int **then** int_add **else** float_add **end**;

rule Name → Bezeichner .
attribution Name.vor := definiert(Bezeichner.Symbol, Name.umg);
condition anpassbar(Name.vor, Name.nach);

Typanpassung - Auswertung an konkretem Syntaxbaum





Kapitel 4: Attributierte Grammatiken

Attributgrammatiken (D. E. Knuth 1968)

$AG = (G, A, R, B)$ mit

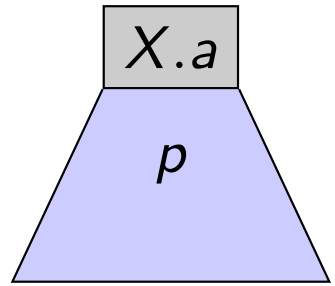
- $G = (T, N, P, Z)$ ist eine reduzierte, kontextfreie Grammatik,
- $A = \bigcup_{X \in T \cup N} A(X)$ endliche Menge von Attributen,
- $R = \bigcup_{p \in P} R(p)$ endliche Menge von Attributierungsregeln,
- $B = \bigcup_{p \in P} B(p)$ endliche Menge von Bedingungen
- $A(X) \cap A(Y) \neq \emptyset \Rightarrow X = Y$.
- Ergebnis und Argumente der Attributierungsregeln

$$X_i.a := f(\ldots, X_j.b, \ldots) \in R(p)$$

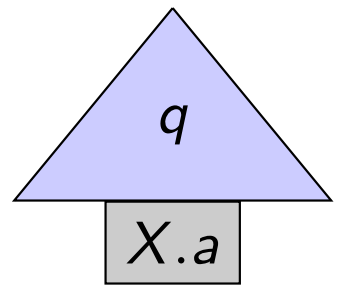
  sind Attribute, die zu einem Nichtterminal der Produktion $p : X_0 \to X_1 \ldots X_n$ gehören $(0 \leq i \wedge j \leq n)$
- $\mathrm{AF}(p) := \{X.a \mid X.a := f(\ldots) \in R(p)\}$ heißt Menge der in p definierten Attribute
- Jedes Attribut $X.a \in A(X)$ eines Knotens X im Strukturbaum eines Satzes der Sprache $L(G)$ ist mit maximal einer Regel aus R berechenbar.

Synthetisierte und ererbte Attribute

$\text{syn}(X) = \{X.a \mid \exists p : X \to X_1 \ldots X_n \in P \wedge X.a \in \text{AF}(p)\}$
Attribute in $\text{syn}(X)$ heißen synthetisiert oder abgeleitet



$\text{inh}(X) = \{X.a \mid \exists q : Y \to \mu X \omega \in P \wedge X.a \in \text{AF}(q)\}$
Attribute in $\text{inh}(X)$ heißen ererbt (inherited)



Die Mengen $\text{syn}(X)$ und $\text{inh}(X)$ sind disjunkt für alle X im Vokabular von G: Für alle $a \in A(X)$ gibt es nur eine Berechnungsregel $X.a := f(\ldots) \in R$.

Vollständigkeit einer AG

Eine AG heißt vollständig, wenn

- $\forall(p : X \to x \in P)\colon\ \mathrm{syn}(X) \subseteq \mathrm{AF}(p)$
- $\forall(q : Y \to \mu X\omega \in P)\colon\ \mathrm{inh}(X) \subseteq \mathrm{AF}(q)$
- $\mathrm{syn}(X) \cup \mathrm{inh}(X) = A(X)$
- $\mathrm{inh}(Z) = \emptyset$, wenn Z das Axiom der kfG ist

zu deutsch:

- für alle Attribute, die nicht vorbesetzt sind, gibt es eine Attributierungsregel in der „richtigen“ Produktion (linksseitig bei abgeleiteten, rechtsseitig bei ererbten Attributen)
- die Baumwurzel besitzt keine ererbten Attribute

Korrekte Attributierung

Ein Strukturbaum heißt korrekt attributiert, wenn

- die Attributierungsregeln eingehalten sind
- alle Bedingungen den Wert wahr haben

Die Bedingung $B(p)$ einer Produktion $p : X_0 \to X_1 ... X_n$ kann man als Attribut $X_0.b$ der linken Seite auffassen. Faßt man dieses Attribut mit den entsprechenden Attributen $X_i.b$ aller X_i zusammen, also

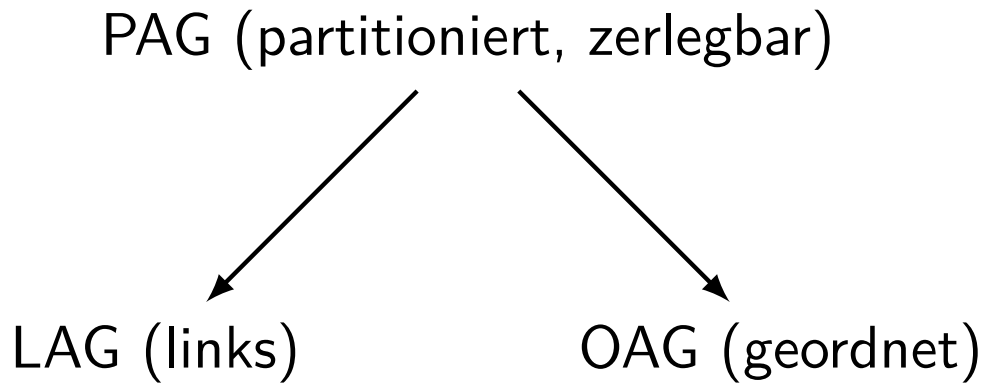
$$X_0.b := B(p) \wedge X_1.b \wedge \ldots \wedge X_n.b$$

so ist ein Strukturbaum genau dann korrekt attributiert, wenn das Attribut $z.b$ der Baumwurzel (des Axioms) wahr ist.

Aus diesem Grund sprechen wir im folgenden nur noch von Attributen und erfassen damit auch die Bedingungen.
Übung: Die Bedingungsattribute sind abgeleitete Attribute.

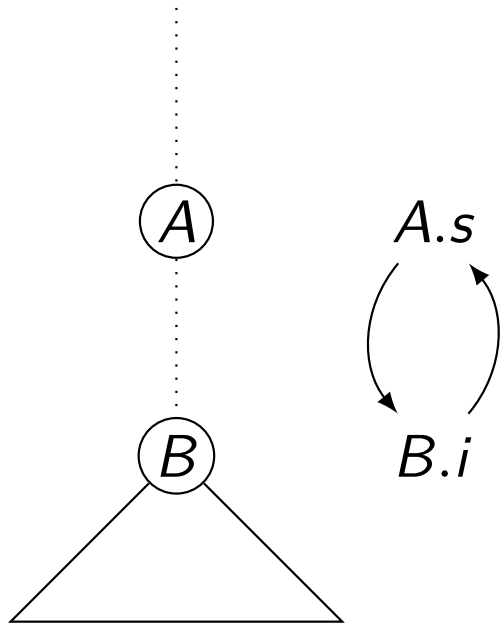
Kapitel 4: Attributierte Grammatiken

AG Hierarchie



Zyklische Abhängigkeit

Produktion	**Semantische Regeln**
$A \to B$	$A.s = B.i$ $B.i = A.s$



Direkte Attributabhängigkeiten

Die Menge der direkten Attributabhängigkeiten einer Produktion $p : X_0 \to X_1 \ldots X_n \in P$ ist gegeben durch

$$\mathrm{DDP}(p) = \{(X_i.a, X_j.b) \mid X_j.b := f(\ldots, X_i.a, \ldots) \in R(p)\}$$

Eine AG heißt lokal azyklisch, wenn $\mathrm{DDP}(p)$ für alle $p \in P$ azyklisch ist.

Beispiel DDP

	Produktion	**Semantische Regeln**
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type = \texttt{Integer}$
3)	$T \rightarrow$ **float**	$T.type = \texttt{Float}$
4)	$L \rightarrow L_1$ **,** **id**	$L_1.inh = L.inh$
		id$.entry.\texttt{addType}(L.inh)$
5)	$L \rightarrow$ **id**	**id**$.entry.\texttt{addType}(L.inh)$

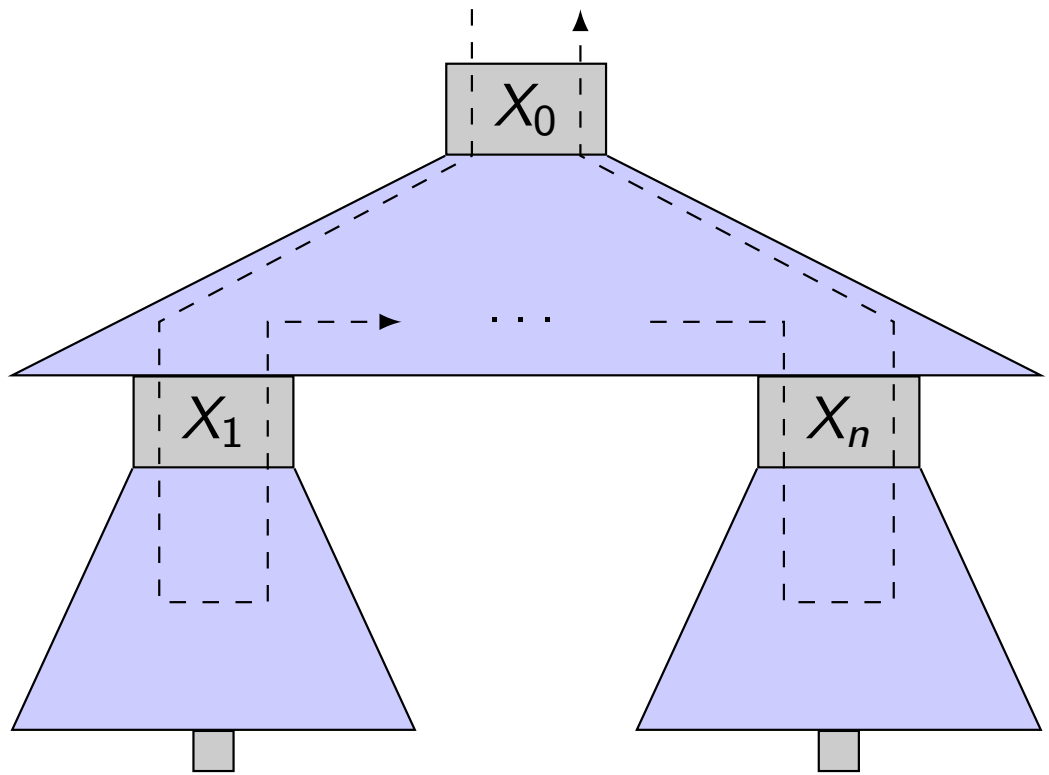
$$\begin{aligned}
\text{DDP}(D \rightarrow T\ L) &= \{(T.type, L.inh)\} \\
\text{DDP}(T \rightarrow \textbf{int}) &= \{(\texttt{Integer}, T.type)\} \\
\text{DDP}(T \rightarrow \textbf{float}) &= \{(\texttt{Float}, T.type)\} \\
\text{DDP}(L \rightarrow L_1\ \textbf{,}\ \textbf{id}) &= \{(L.inh, L_1.inh), (L.inh, \textbf{id}.entry)\} \\
\text{DDP}(L \rightarrow \textbf{id}) &= \{(L.inh, \textbf{id}.entry)\}
\end{aligned}$$

LAG(1) - Attributgrammatik

Eine AG ist links abwärts berechenbar, eine LAG(1), wenn für jede Produktion $p : X_0 \to X_1 \ldots X_n \in P$ die Attribute in der Reihenfolge

$$\text{inh}(X_0), \text{inh}(X_1), \text{syn}(X_1), \text{inh}(X_2), \ldots, \text{syn}(X_n), \text{syn}(X_0)$$

berechnet werden können.



DDP bei LAG(1)

Für jede Produktion $p : X_0 \to X_1 \ldots X_n \in P$ einer LAG(1) gilt

$$
\begin{aligned}
DDP(p) \subseteq Cl^+(&\{(X_0.a, X_1.b) \mid X_0.a \in \mathrm{inh}(X_0) \wedge X_1.b \in A(X_1)\} \\
&\cup \{(X_1.b, X_1.c) \mid X_1.b \in \mathrm{inh}(X_1) \wedge X_1.c \in \mathrm{syn}(X_1)\} \\
&\cup \{(X_1.c, X_2.d) \mid X_1.c \in \mathrm{syn}(X_1) \wedge X_2.d \in \mathrm{inh}(X_2)\} \\
&\cup \ldots \\
&\cup \{(X_n.e, X_n.f) \mid X_n.e \in \mathrm{inh}(X_n) \wedge X_n.f \in \mathrm{syn}(X_n)\} \\
&\cup \{(X_n.f, X_0.a) \mid X_n.f \in A(X_n) \wedge X_0.a \in \mathrm{syn}(X_0)\})
\end{aligned}
$$

Bemerkung: Cl^+ bezeichnet die transitive Hülle

Bemerkungen zu $\mathrm{LAG}(1)$

- Eine $\mathrm{LAG}(1)$ ist während des LL-Parsens auswertbar.
- Eine $\mathrm{LAG}(1)$ entspricht dem intuitiven Begriff eines Übersetzerlaufs.
- In der Praxis Variationen möglich, aber Grundprinzip links-abwärts bleibt erhalten.
- $\mathrm{LAG}(1)$ von Hand programmierbar.

$\mathrm{LAG}(k)$ - Attributgrammatik

Eine AG ist $\mathrm{LAG}(k)$, wenn für jede Produktion $p : X_0 \to X_1 \ldots X_n \in P$ die Attribute so in Gruppen

$$G_1, \ldots, G_k$$

zerlegt werden können, dass die Gruppen nacheinander und die Attribute jeder Gruppe nach dem $\mathrm{LAG}(1)$-Schema berechnet können.

- Entspricht dem intuitiven Begriff mehrerer Übersetzerpässe
- Erforderlich, da $\mathrm{LAG}(1)$ fast nie ausreicht

Beispiele (1/2)

Ein Einpass-Übersetzer, der an die Syntaxanalyse mit rekursivem Abstieg die semantische Analyse und Codeerzeugung unmittelbar anschließt (kein expliziter Strukturbaum), setzt eine $\mathrm{LAG}(1)$-Attributierung voraus

- Beispiel: Züricher Pascal-, Modula- und Oberon-Übersetzer
- Notwendig:
  - Vereinbarung vor erster Verwendung eines Bezeichners
  - Vorvereinbarung von Sprungmarken und verschränkt rekursiven Prozeduren, usw.

Beispiele (2/2)

Fast alle Sprachen benötigen $k = 2$, 3 oder 4. Höheres k immer nur lokal für einzelne Sprachelemente nötig

- Daher ist eine OAG meist kostengünstiger: sie spart Baumdurchläufe.
- Sie ist auch systematischer zu entwerfen: Denken in $\mathrm{LAG}(k)$ führt zu schlechtem Entwurf:
  - Zuerst wird k zu klein angenommen (Pascal benötigt tatsächlich $k = 4$!)
  - Nach Korrektur Neuentwurf nötig, um Attribute vernünftig auf die Gruppen zu verteilen.

Implementierung von LAG(1) für LL-Parser

Rekursiver Abstieg

- ererbte Attribute werden zu In-Parametern
- synthetisierte Attribute werden zu Out-Parametern
- Beispiel siehe Folien Codeerzeugung mit AGs

Tabellengesteuert

- Attribute sind an den Nichtterminalen des Stacks annotiert
- Auswertung der Attribute bei Verarbeitung der Produktionen $X \to X_1 \dots X_n$
- Beispiel siehe Folien Codeerzeugung mit AGs

Implementierung LAG(1) beim LR-Parsen

Im Allgemeinen nicht möglich, da erst beim Reduzieren feststeht, ob eine bestimmte Produktion vorleigt. Keine Attributauswertungen „in der Mitte“ der Produktion.

Lösung: Anpassung der Grammatik – Für jede nötige Auswertung innerhalb einer Produktion P:

- Erzeuge eindeutiges Nichtterminal M und Produktion M_e: $M \to \varepsilon$
- Verschiebe Auswertungscode ans Ende der neuen Produktion M_e.

Eigentlich Verstoss gegen AG-Regeln: M_e liest/schreibt Attribute von P. Beim LR-Parser aber Zugriff über den Parser-Stack möglich. Beispiel siehe Folien Codeerzeugung mit AGs.

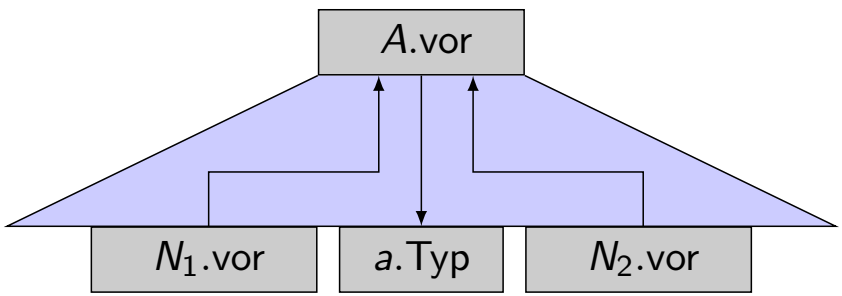
Normalisierte direkte Abhängigkeiten

Für alle $p : X_0 \to X_1 \ldots X_n \in P$ definiere die normalisierten direkten Abhängigkeiten [1]

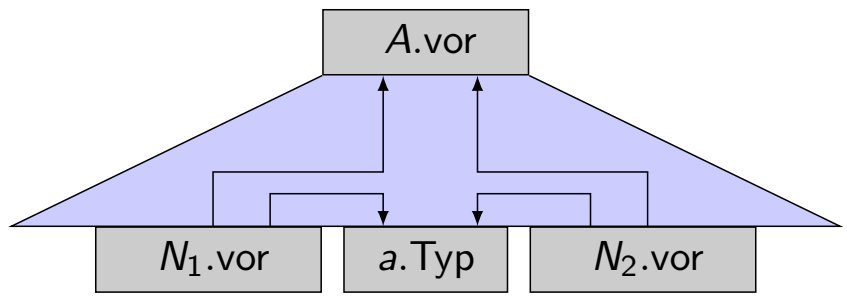
$$\mathrm{NDDP}(p) = \mathrm{DDP}(p)^+ \setminus \{(X_i.a, X_j.b) \mid X_i.a, X_j.b \in \mathrm{AF}(p)\}$$

Produktion *Ausdruck* $\to$ *Name addop Name* aus dem Beispiel:

DDP



NDDP



[1] DDP^+ bezeichnet die transitive Hülle der direkten Abhängigkeiten

Induzierte Abhängigkeiten

Berechnung von IDP und IDS:

1. Für alle $p \in P$ setze $\mathrm{IDP}(p) := \mathrm{NDDP}(p)$
2. Für alle $X \in N \cup T$ setze
   $\mathrm{IDS}(X) := \{(X.a, X.b) \mid \exists q \in P : (X.a, X.b) \in \mathrm{IDP}(q)^+\}$
3. Für alle $p : X_0 \to X_1 \ldots X_n \in P$ setze
   $\mathrm{IDP}(p) := \mathrm{IDP}(p) \cup \mathrm{IDS}(X_0) \cup \ldots \cup \mathrm{IDS}(X_n)$
4. Wiederhole 2 und 3 bis alle IDP und IDS unverändert bleiben.

IDP und IDS heißen induzierte Abhängigkeiten über Produktionen bzw. Symbole.

IDP und IDS intuitiv

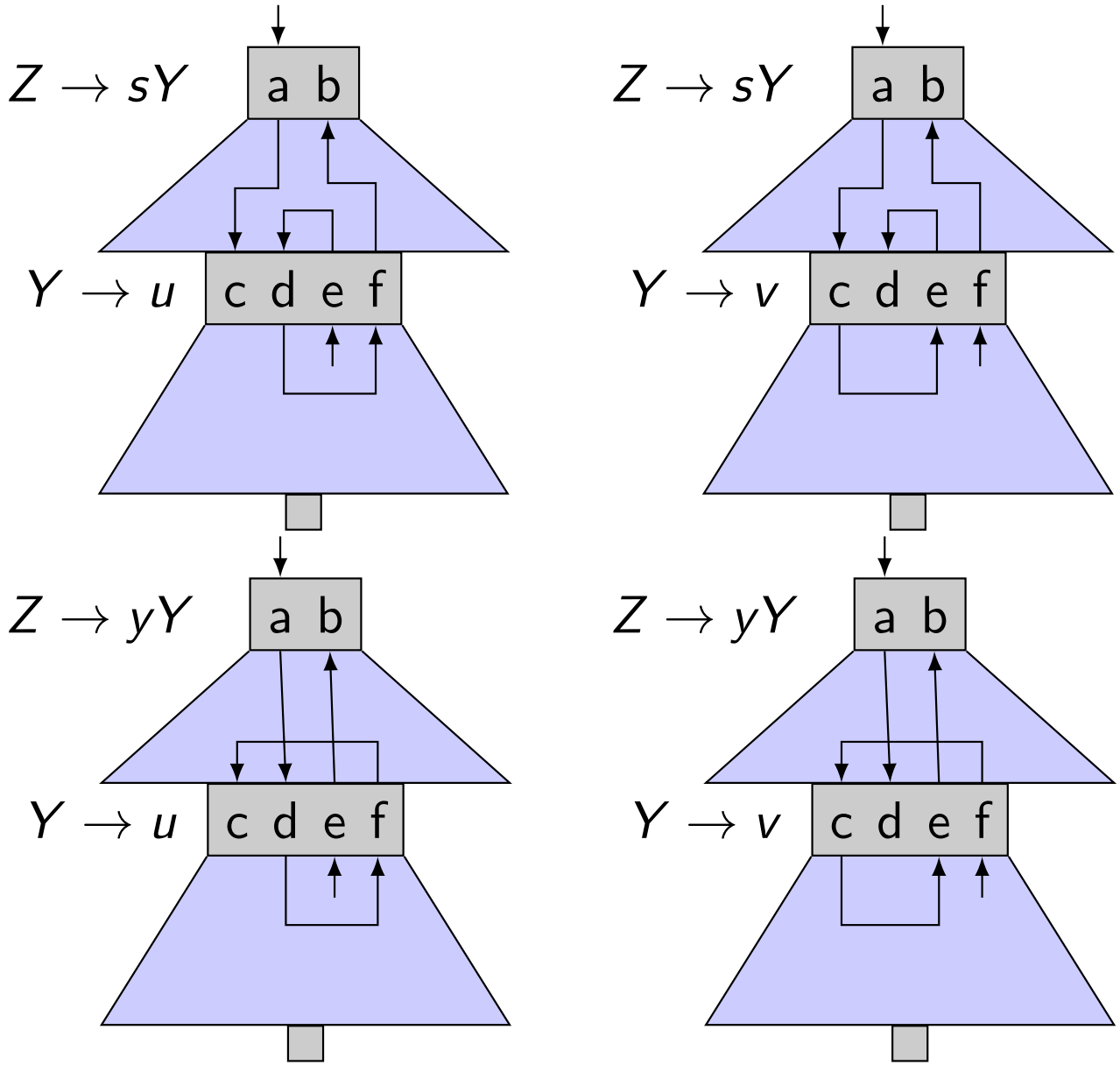
- IDS(X) enthält alle induzierten Abhängigkeiten zwischen Attributen des gleichen Nichtterminals X unabhängig von den Ober- und Unterproduktionen, die über X verbunden sind.
- IDP(p) enthält alle induzierten Abhängigkeiten zwischen Attributen der Symbole in p unabhängig davon, in welchem Kontext p im Baum erscheint.
- IDS und IDP sind pessimistische Approximationen. Die tatsächlich in einem Strukturbaum möglichen Abhängigkeiten werden überschätzt.

Aber: IDS und IDP sind statisch, unabhängig von den Strukturbäumen berechenbar!

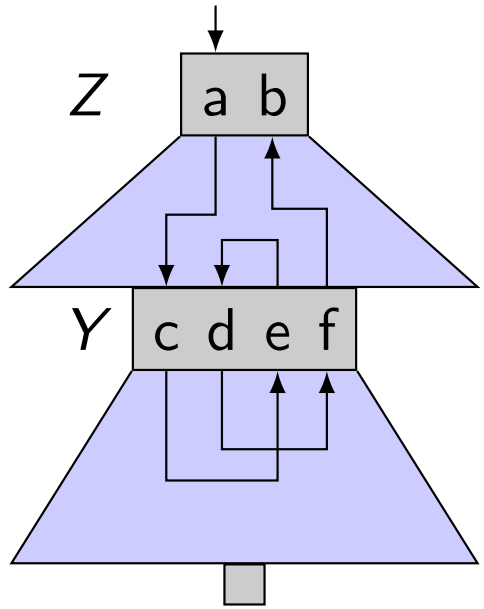
Anschaulich: Alle Abhängigkeiten aller möglichen Ableitungsbäume werden zur Gewinnung von IDS und IDP übereinandergelegt.

AG mit 4 Ableitungsbäumen

$S \rightarrow Z$
$Z \rightarrow sY \mid yY$
$Y \rightarrow u \mid v$



IDP und IDS



$\mathrm{IDS}(Z) = \{a \to b\}$ obwohl diese Abhängigkeit in keinem der vier konkreten Ableitungsbäume vorhanden ist.

Partitionierung der Attribute

Eine Partitionierung $A_1(X), \ldots, A_m(X)$ der Attribute $A(X)$, $X \in N \cup T$, heißt zulässig, wenn für alle X gilt

- $A_i(X) \subseteq \mathrm{syn}(X)$ für $i = m, m-2, \ldots$
- $A_i(X) \subseteq \mathrm{inh}(X)$ für $i = m-1, m-3, \ldots$

Beachte: $m = m(X)$ ist von dem Nichtterminal X abhängig, $A_m(X) \subseteq \mathrm{syn}(X)$!

Partitionierbare (zerlegbare) AG: PAG

Eine AG heißt partitionierbar (zerlegbar), wenn sie lokal azyklisch ist und für alle X eine zulässige Partitionierung existiert, so dass die Attributmengen immer in der Reihenfolge $A_1(X), \ldots, A_m(X)$ ausgewertet werden können (unabhängig von den Produktionen, in denen X vorkommt!).

Bei PAGs ist die Berechnungsreihenfolge unabhängig vom Strukturbaum: Attributauswerter statisch konstruierbar.

Aber: Prüfung der Eigenschaft PAG ist NP-vollständig!

Abhängigkeiten über Produktionen

Für alle X sei eine zulässige Zerlegung $A_1(X), \ldots, A_m(X)$ gegeben.

Für alle $p : X_0 \to X_1 \ldots X_n \in P$ ist

$$\begin{aligned} \mathrm{DP}(p) = \mathrm{IDP}(p) \cup \\ \{(X_i.a, X_i.b) \mid & X_i.a \in A_j(X_i) \wedge X_i.b \in A_k(X_i) \wedge \\ & 0 \leq i \leq n \wedge j < k\} \end{aligned}$$

die Abhängigkeitsrelation über der Produktion p.

Satz: AG ist zerlegbar, gdw. $\mathrm{DP}(p)$ azyklisch für alle $p \in P$.

Geordnete Attributgrammatiken: OAG

Eine AG heißt geordnet, wenn faule Auswertung eine zulässige Partitionierung liefert: Sei $T_{-1}(X) = T_0(X) = \emptyset$ und für $k > 0$

$$T_{2k-1}(X) = \{a \in \mathrm{syn}(X) \mid (a,b) \in \mathrm{IDS}(X) \Rightarrow b \in T_j(X), j \leq 2k-1\}$$
$$T_{2k}(X) = \{a \in \mathrm{inh}(X) \mid (a,b) \in \mathrm{IDS}(X) \Rightarrow b \in T_j(X), j \leq 2k\}$$

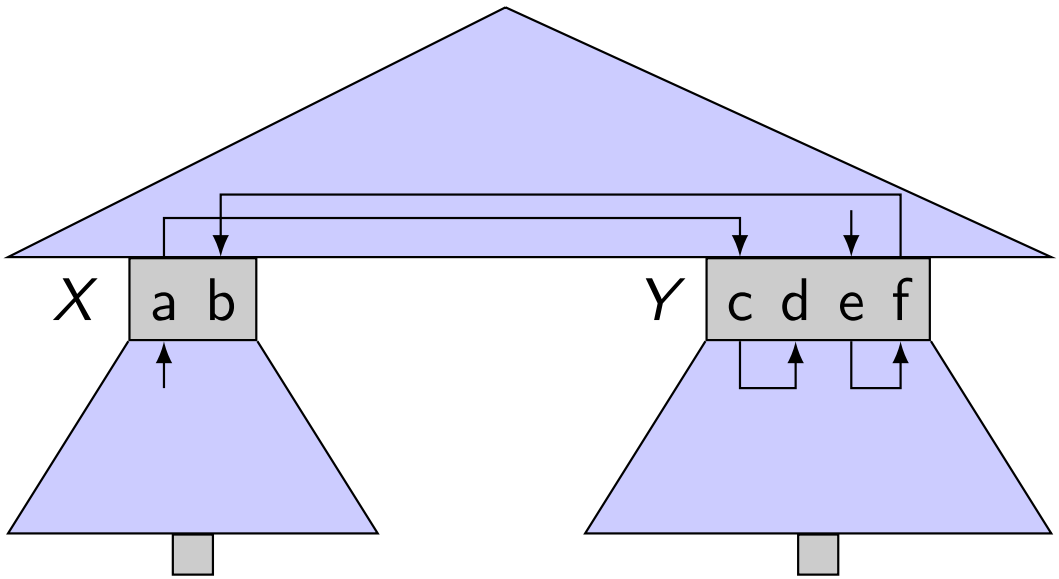
Definiere Partitionierung durch

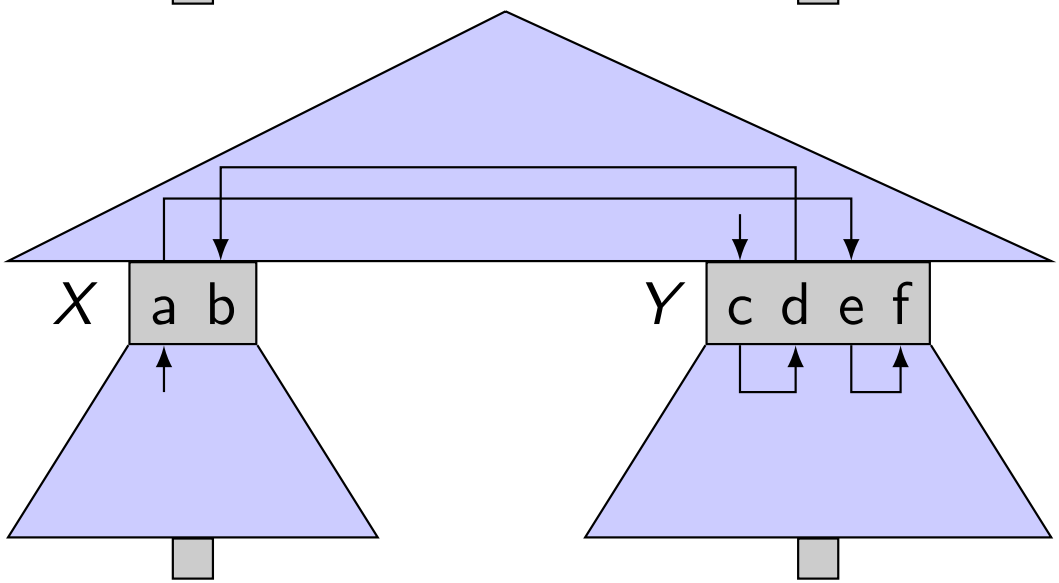
$$A_i(X) = T_{m-i+1}(X) \setminus T_{m-i-1}(X) \text{ für } i = 1, \ldots, m$$

- m ist minimal mit der Eigenschaft $T_{m-1}(X) \cup T_m(X) = A(X)$.
- m ist abhängig von X, einige $T_k(X)$ könnten leer sein.

Partitionierbar, aber nicht geordnet

Z → sXY | tXY, X → u, Y → v





Zerlegung (PAG):
$A_1(X) = \{a\}$,
$A_2(X) = \{b\}$, $A_3(X) = \emptyset$
$A_1(Y) = \{c, e\}$,
$A_2(Y) = \{d, f\}$

OAG konstruiert aber
$A_1(X) = \{b\}$,
$A_2(X) = \{a\}$ mit Zyklen
$b \to a \to \{c, e\} \to$
$\{d, f\} \to b$ in DP

Offenbar gibt es eine Zerlegung, aber der OAG Algorithmus findet sie nicht.

Erweiterung $\mathrm{PAG} \rightarrow \mathrm{OAG}$

Satz: Jede PAG kann durch Zufügen zusätzlicher Abhängigkeiten zu einer geordneten AG gemacht werden.

Beweisidee: OAG bedeutet „Berechnung so spät wie möglich". Wenn die gegebene Partitionierung ein Attribut zu früh berechnet, so kann man durch Zufügen einer an sich nicht vorhandenen Abhängigkeit erzwingen, dass das Attribut später berechnet wird.

Beispiel: füge auf der vorangehenden Folie eine Abhängigkeit a $\rightarrow$ b hinzu. Dann wird b nach a berechnet.

Besuchssequenzen

Attribute werden berechnet im Kontext der

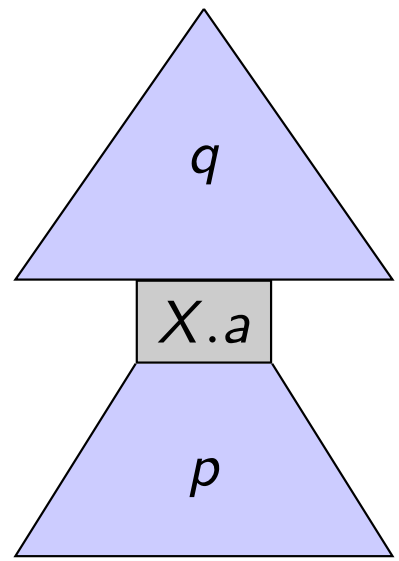
- Unterproduktion: abgeleitetes Attribut
- Oberproduktion: ererbtes Attribut

Berechnungen im Kontext p und q interagieren:

- Wenn für ererbtes (abgeleitetes) X.a := f(...) Argumente aus der anderen Produktion erforderlich, zuerst die andere Produktion besuchen, um Argumente zu berechnen

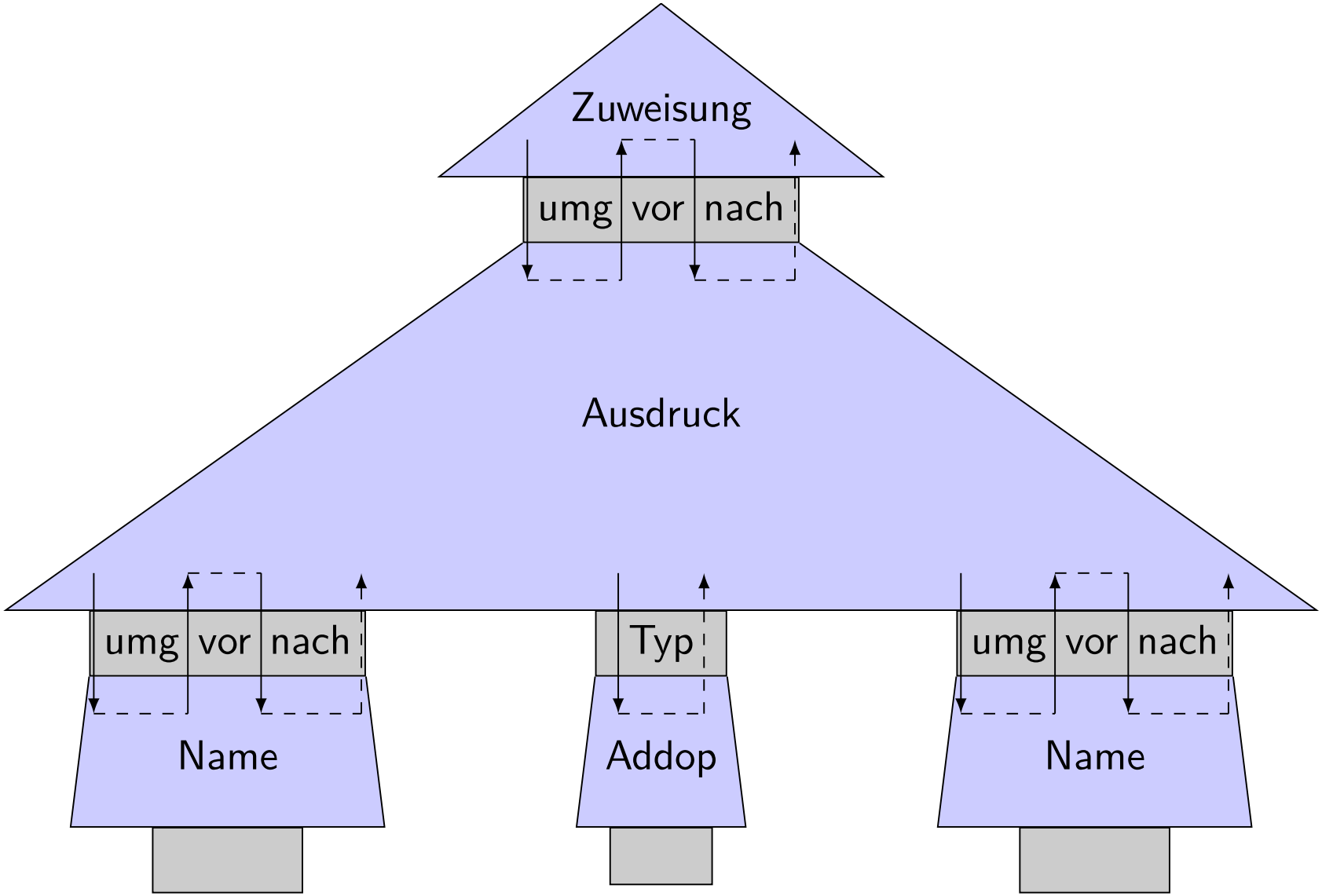
Aktivitäten an der Schnittstelle X:

- In q:
  - berechne ererbtes $X.a$
  - besuche Sohn p
- In p:
  - berechne abgeleitetes $X.a$
  - Rückkehr zum Vater



Besuchssequenz: Interaktionsprotokoll *berechne*, *besuche Sohn*, *besuche Vater*

Besuchssequenzen



Beispiel Wiederholung

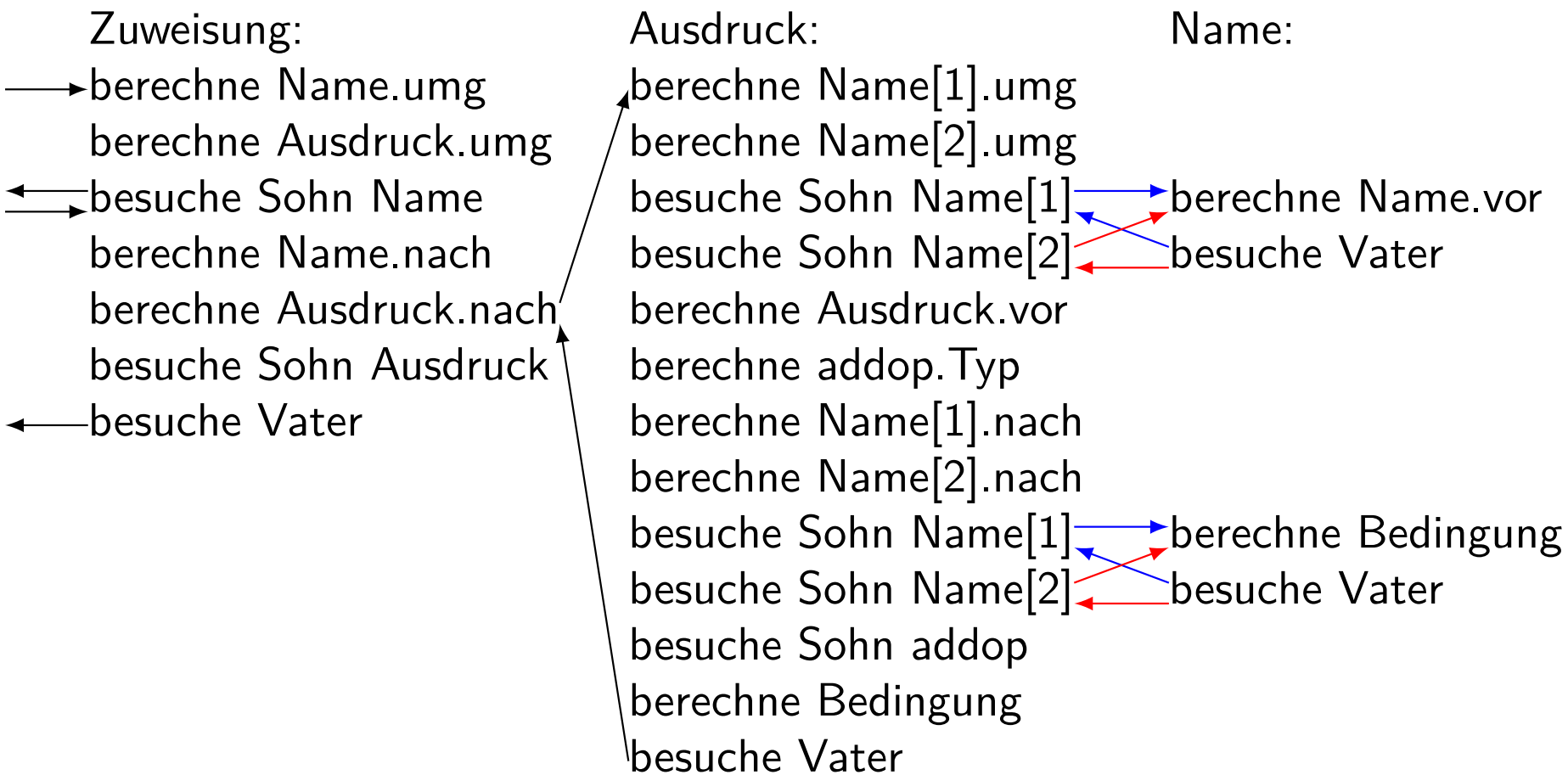
rule Zuweisung → Name ':=' Ausdruck .
attribution
Name.umg := Zuweisung.umg;
Ausdruck.umg := Zuweisung.umg;
Name.nach := Name.vor;
Ausdruck.nach := **if** Name.vor = int **then** int **else** float **end**;

rule Ausdruck → Name addop Name .
attribution
Name[1].umg := Ausdruck.umg;
Name[2].umg := Ausdruck.umg;
Ausdruck.vor := **if** anpassbar(Name[1].vor, int) ∧ anpassbar(Name[2].vor, int)
  then int **else** float **end**;
addop.Typ := Ausdruck.vor;
Name[1].nach := Ausdruck.vor;
Name[2].nach := Ausdruck.vor;
condition anpassbar(Ausdruck.vor, Ausdruck.nach);

rule addop → '+' .
attribution addop.operation := **if** addop.Typ = int **then** int_add **else** float_add **end**;

rule Name → Bezeichner .
attribution Name.vor := definiert(Bezeichner.Symbol, Name.umg);
condition anpassbar(Name.vor, Name.nach);

Besuchssequenzen als Kommunikationsprotokoll



Kapitel 4: Attributierte Grammatiken

Attributierte Grammatik für **while**-Anweisungen

rule statement → 'while' '(' **condition** ')' statement .
attribution
  l1 := new_label();
  l2 := new_label();
  statement[2].next = l1;
  condition.false = statement[1].next
  condition.true = l2;
  statement[1].code = label || l1 || **condition**.code || label || l2 || statement[2].code

Bemerkungen:

- statement.next ist das Label des nächsten Statements
- **condition**.false ist das Sprungziel bei falscher Bedingung
- **condition**.true ist das Sprungziel bei wahrer Bedingung
- || konkateniert Codefragmente

Als semantische Aktionen

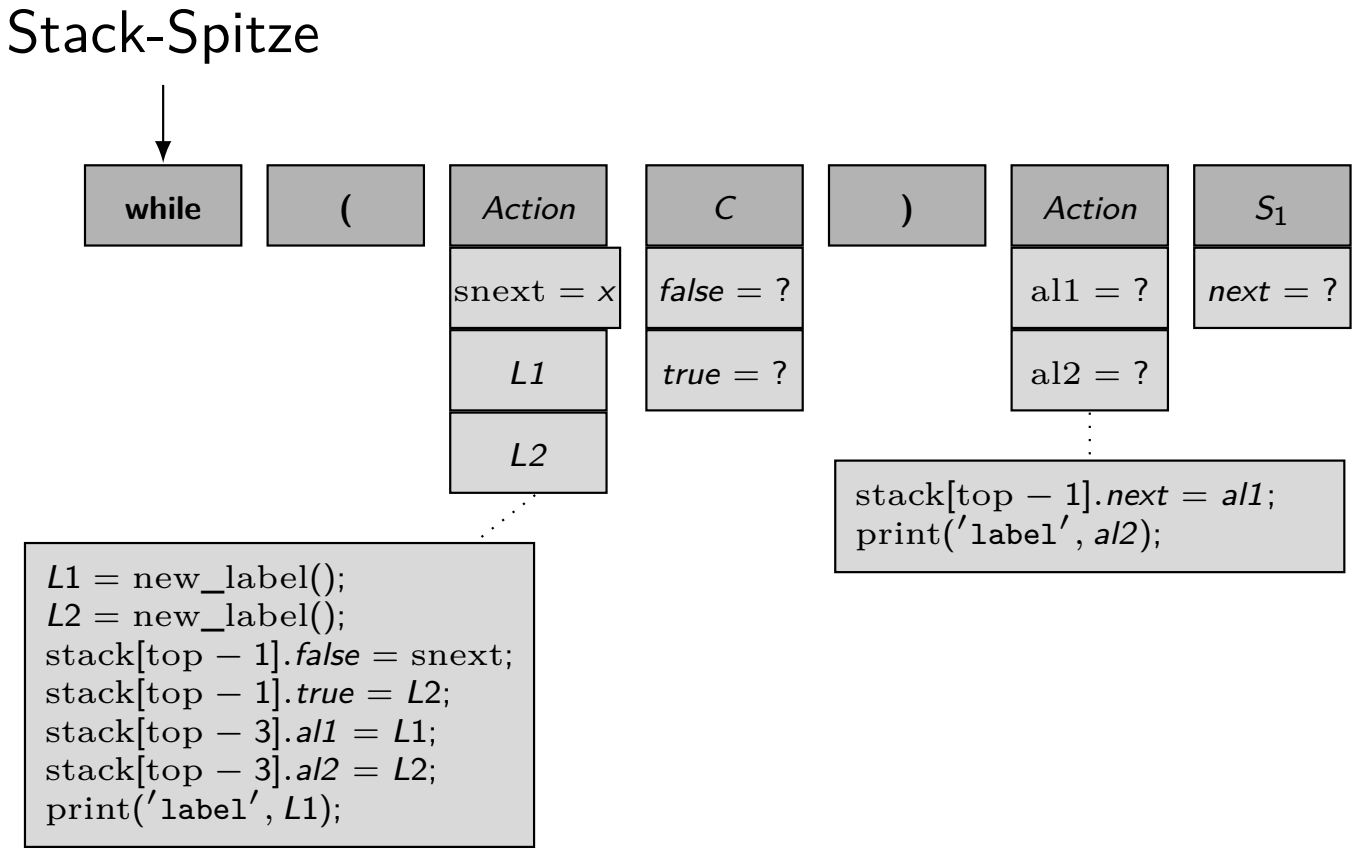
$$
\begin{array}{lll}
S \rightarrow & \textbf{while (} & \{\ \text{L1} = \text{new\_label}();\ \text{L2} = \text{new\_label}(); \\
 & & \text{C.false} = \text{S.next};\ \text{C.true} = \text{L2}; \\
 & & \text{print("label", L1);}\ \} \\
 & C\ \textbf{)} & \{\ \text{S1.next} = \text{L1};\ \text{print("label", L2);}\ \} \\
 & S1 &
\end{array}
$$

rekursiver Abstieg mit direkter Codeerzeugung

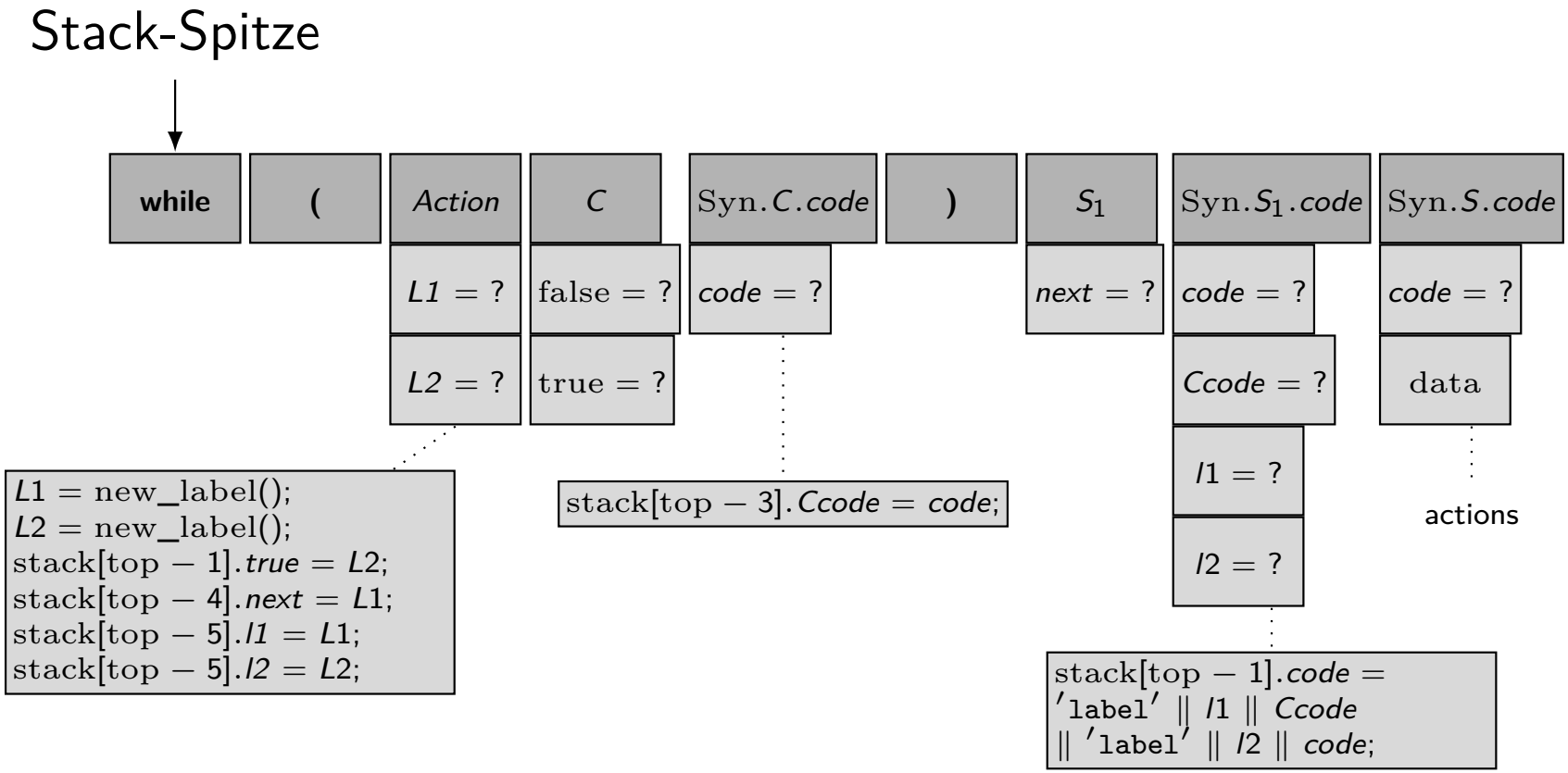
statement → **while (** *condition* **)** *statement*

```
void parse_statement(label next) {
    if (token == T_while) {
        next_token();
        if (token == '(') next_token(); else error(...);
        label L1 = new_label();
        label L2 = new_label();
        print("label", L1);
        /* parse and print condition. Jump to first arg if true,
           jump to 2nd arg if false */
        parse_condition(L2, next);
        if (token == ')') next_token(); else error(...);
        print("label", L2);
        parse_statement(L1);
    } else {
        /* other statements */
    }
}
```

Expandierung von S entsprechend der **while**-Produktion



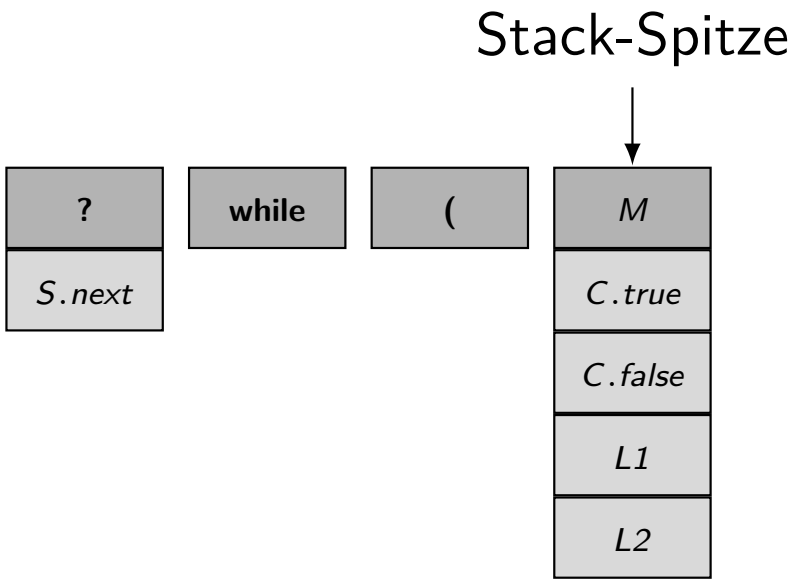
Konstruieren der synthetisierten Attribute



Erweiterte Produktionen mit Dummy-Nonterminals zur Attributberechnung

$$S \to \textbf{while (}\ M\ C\ \textbf{)}\ B\ S_1$$
$$M \to \varepsilon$$
$$N \to \varepsilon$$

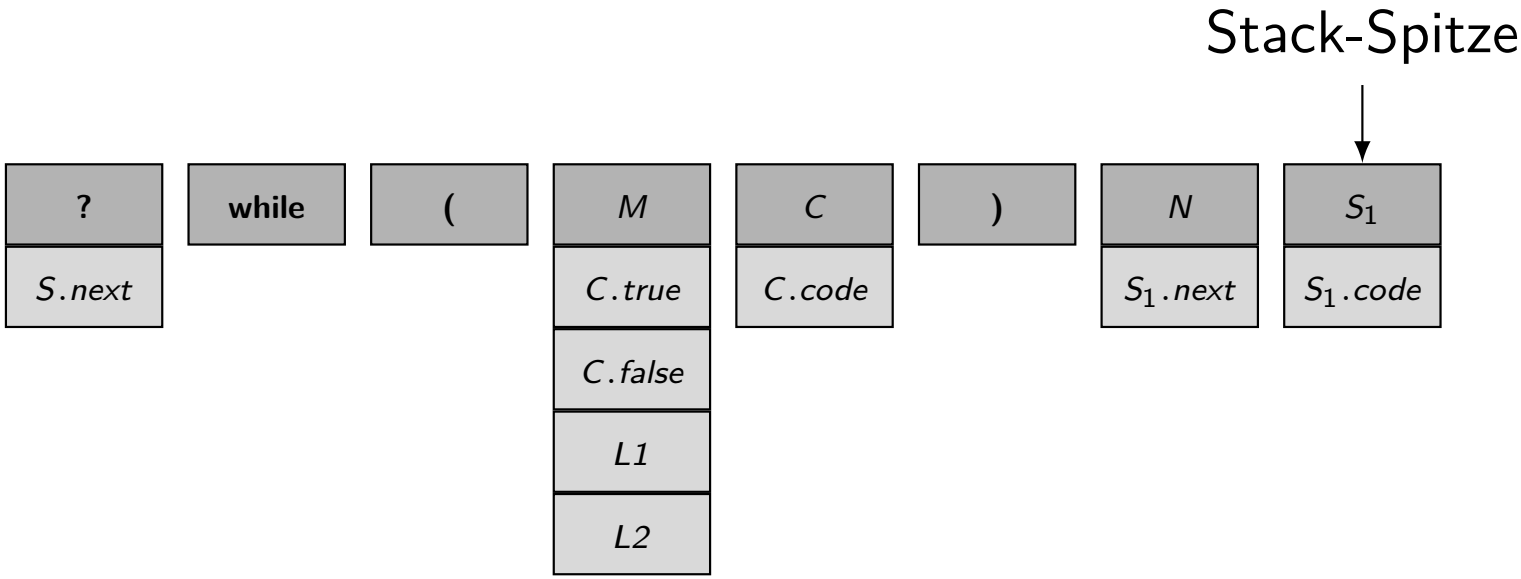
LR-Parserstack nach der Reduktion von ε zu M



Code, der während der Reduktion von ε zu M ausgeführt wird:

$L1 = \text{new\_label}();$
$L2 = \text{new\_label}();$
$C.true = L2;$
$C.false = \text{stack}[\text{top} - 3].next$

LR-Parserstack mit Attributberechnung



Ausgeführte Aktionen

$$
\begin{aligned}
\text{tempCode} = \textbf{label} \parallel \text{stack}[\text{top} - 4].\mathit{L1} \parallel \text{stack}[\text{top} - 3].\mathit{code} \parallel \\
\textbf{label} \parallel \text{stack}[\text{top} - 4].\mathit{L2} \parallel \text{stack}[\text{top}].\mathit{code};
\end{aligned}
$$

$$
\text{top} = \text{top} - 5;
$$

$$
\text{stack}[\text{top}].\mathit{code} = \text{tempCode};
$$