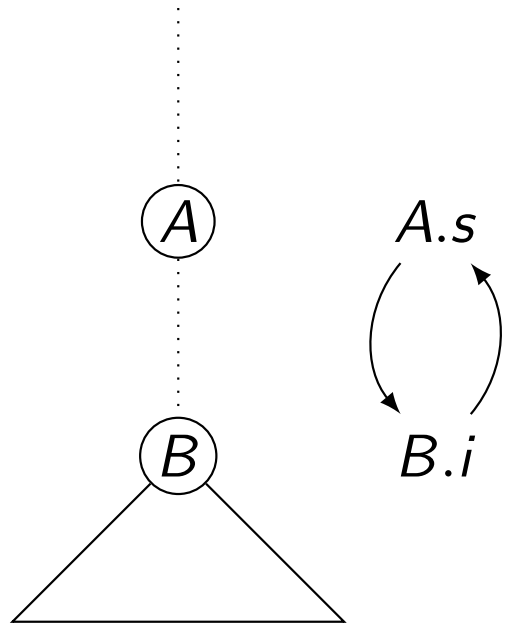
Beispiel: Taschenrechner mit Attributierter Grammatik

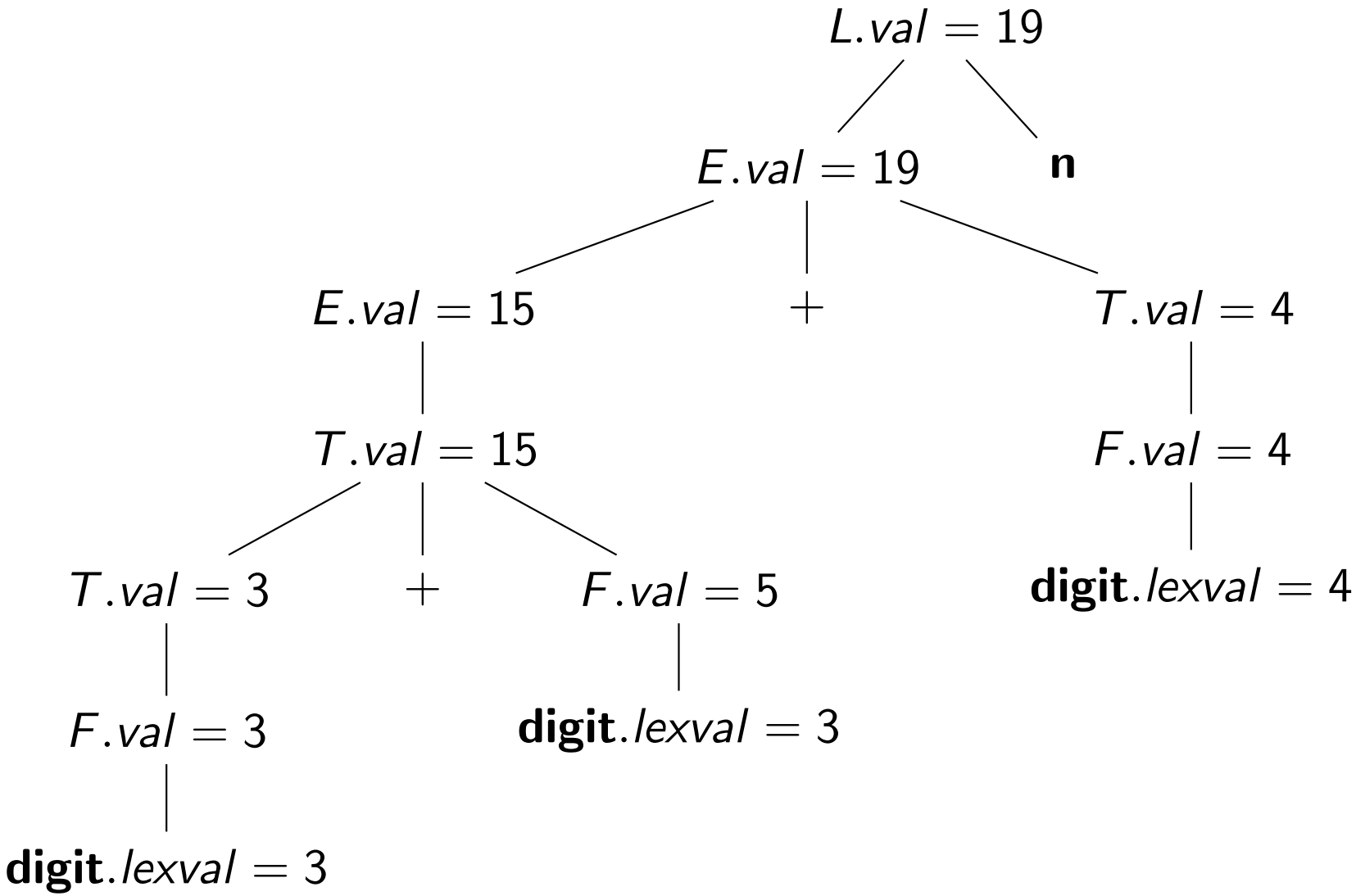
	Produktion	**Semantische Regeln**
1)	$L \rightarrow E\,\mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Zirkuläre Abhängigkeit

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



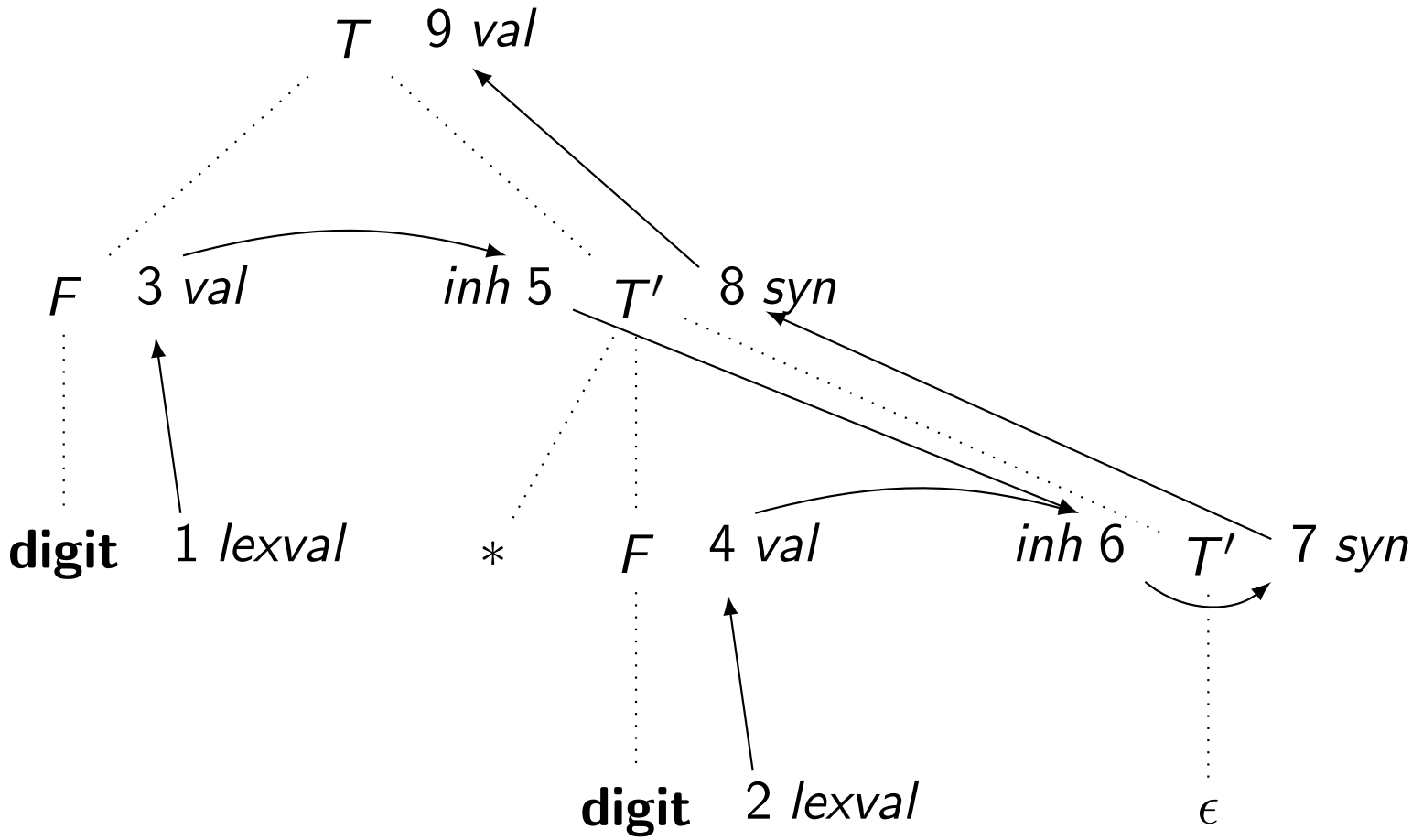
Attributierter Parsebaum für **3 * 5 + 4n**



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)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Abhängigkeitsgraph für attributierten Parsebaum

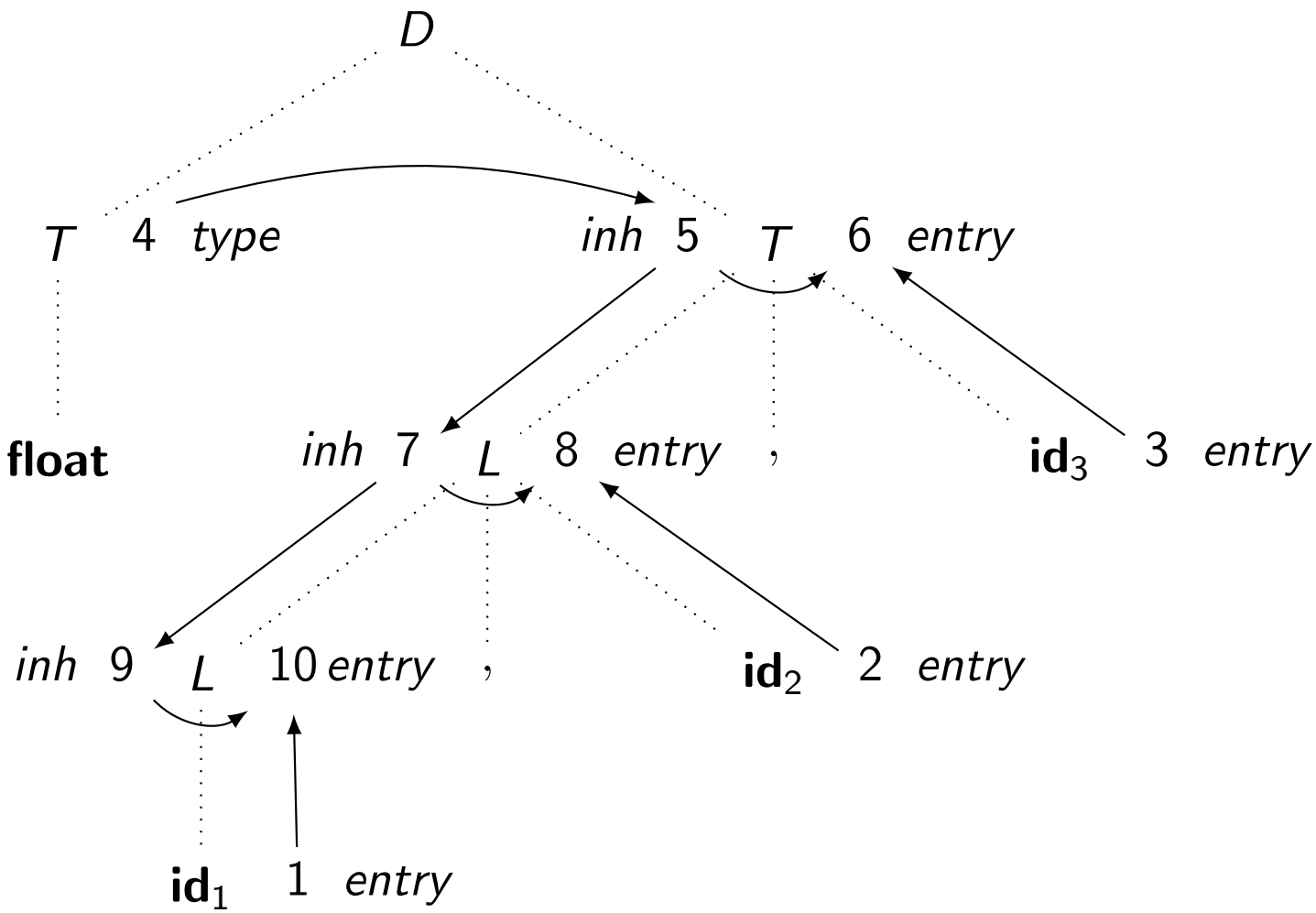


Parse-Baum für **digit * digit**.

Grammatik mit Attributierung für Typdeklarationen

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

Abhängigkeitsgraph für attributierten Parsebaum

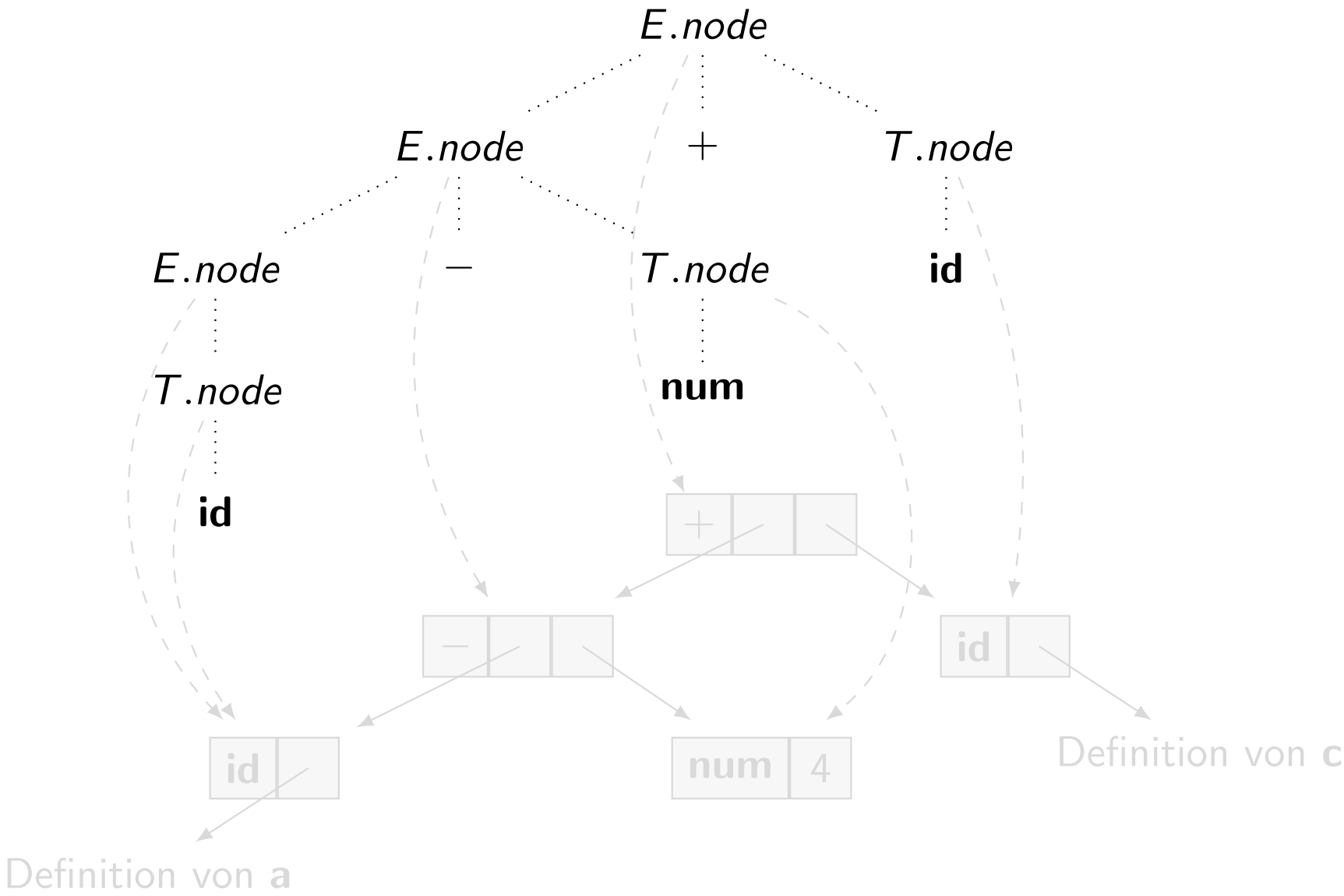


Parse-Baum für **float** $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$.

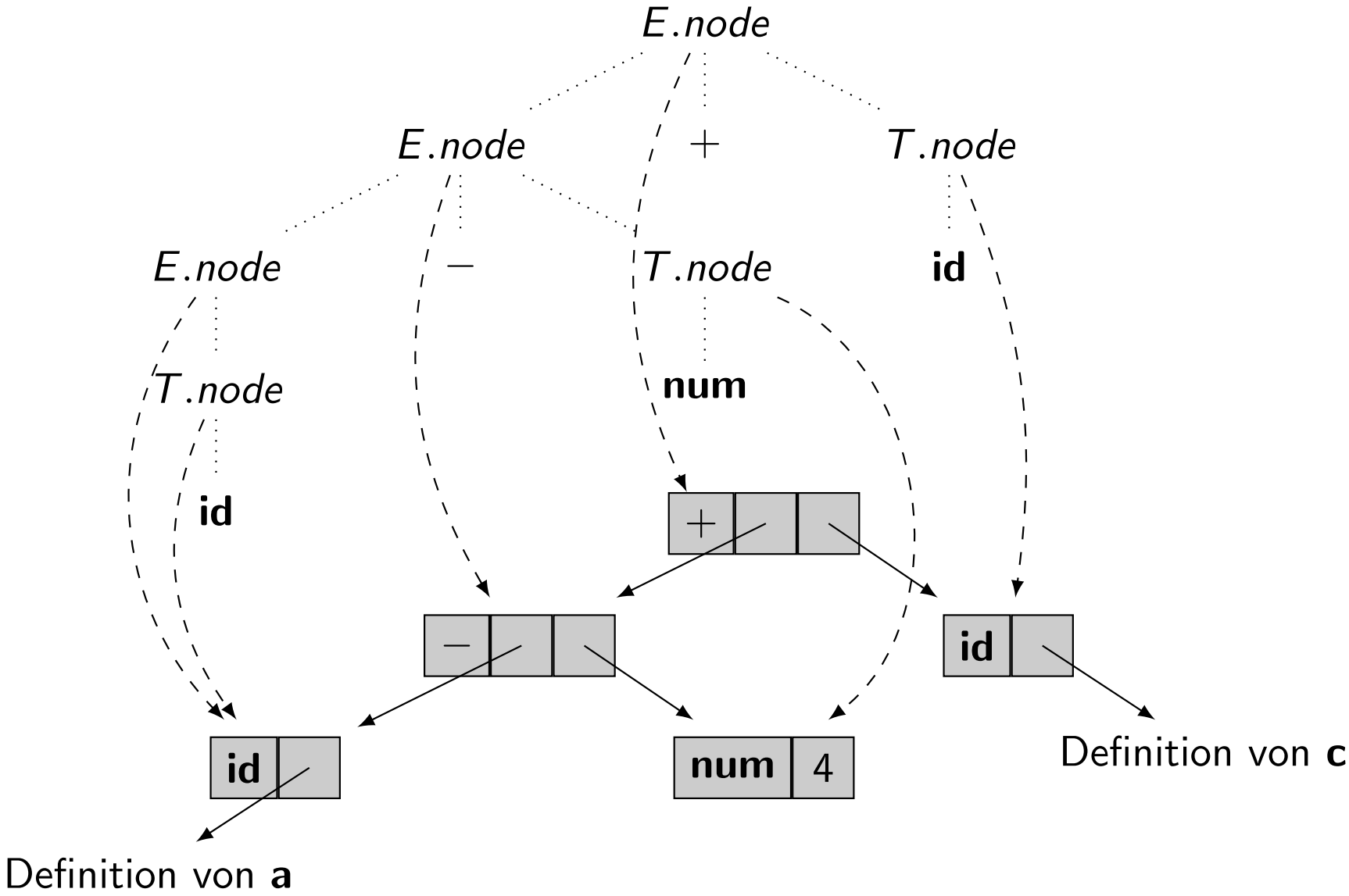
AST-Aufbau mit AGs

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

Beispiel AST-Aufbau



Beispiel AST-Aufbau



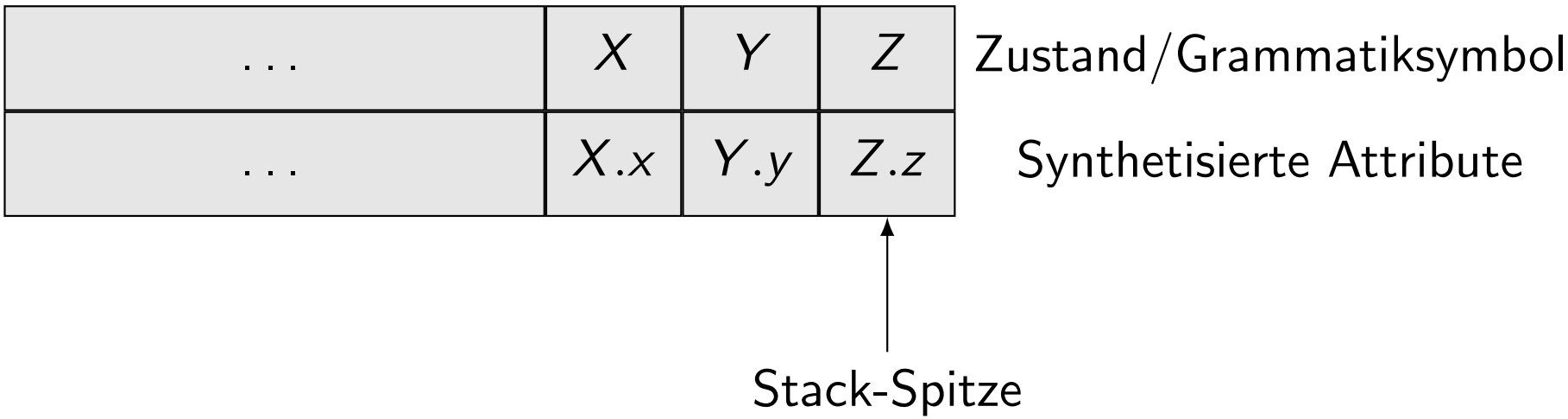
Attributierte Grammatik für Arraytypen

Produktion	**Semantische Regeln**
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = \text{integer}$
$B \rightarrow$ **float**	$B.t = \text{float}$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = \text{array}(\textbf{num}.val, C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Taschenrechner Implementierung

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

Parserstack mit synthetisierten Attributen



Beispiel Schriftsatz

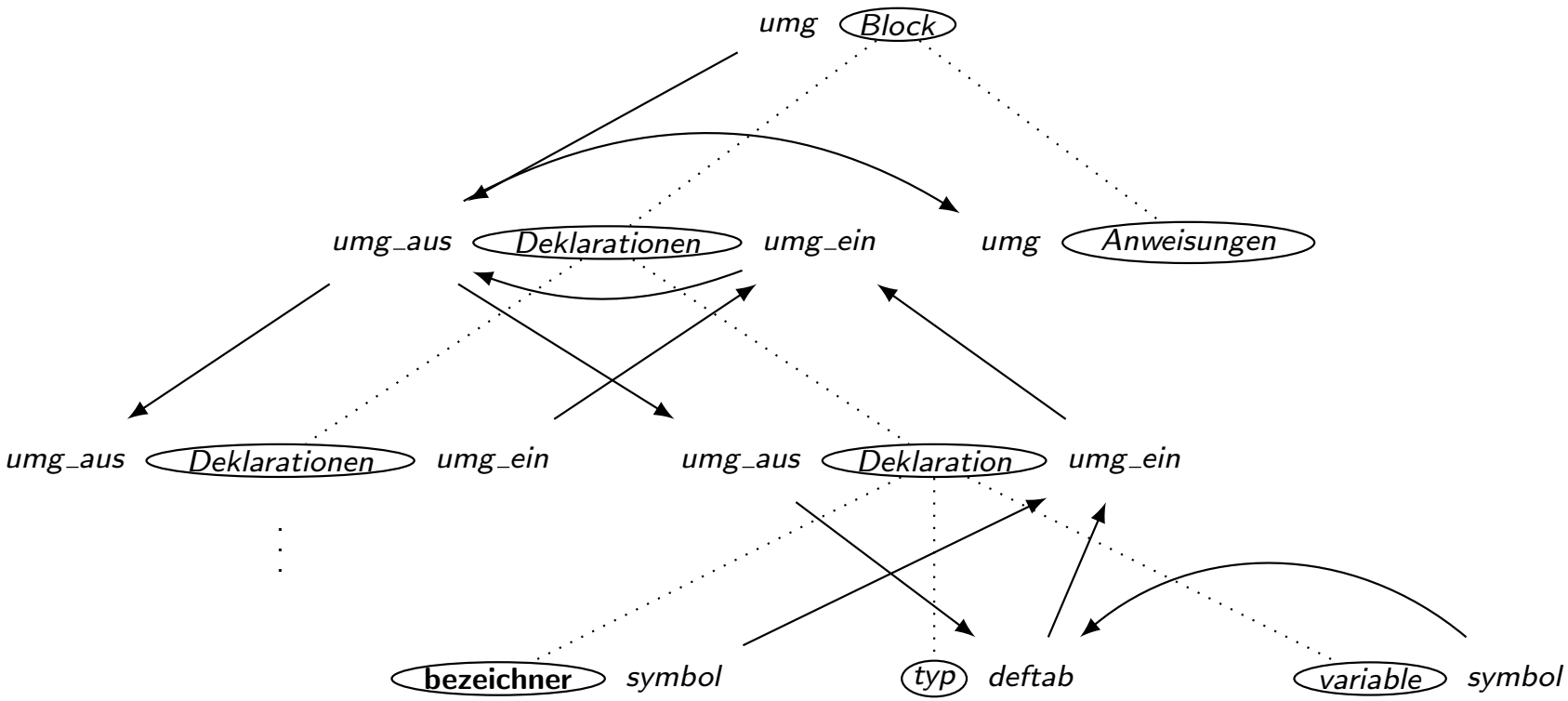
Abbildung 5.24: Konstruieren größerer Kästen aus kleineren

Attributierte Grammatik für den Schriftsatz

	Produktion	**Semantische Regeln**
1)	$S \rightarrow B$	$B.ps = 10$
2)	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$
		$B_2.ps = B.ps$
		$B.ht = \max B_1.ht, B_2.ht$
		$B.dp = \max B_1.dp, B_2.dp$
3)	$B \rightarrow B_1$ **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)	$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$
		$B.ht = B_1.ht$
		$B.dp = B_1.dp$
5)	$B \rightarrow$ **text**	$B.ht = \mathrm{getHt}(B.ps, \textbf{text}.lexval)$
		$B.dp = \mathrm{getDp}(B.ps, \textbf{text}.lexval)$

Erläuterung zur AG Blockschachtelung

siehe Folien Semantik Seite 12



Reparieren der zirkulären Abhängigkeit

rule block ::= deklarationen ; anweisungen .
attribution
anweisungen.umg := deklarationen.umg_aus;
deklarationen.umg_aus := **append**(block.umg,deklarationen.umg_ein)

rule deklarationen ::= deklarationen ';' deklaration.
attribution
deklarationen[1].umg_ein := **append**(deklarationen[2].umg_ein,
  deklaration.umg_ein);
deklarationen[2].umg_aus := deklarationen[1].umg_aus;
deklaration.umg_aus := deklarationen[1].umg_aus

rule deklaration ::= bezeichner ':' typ ':=' variable ';'.
attribution
typ.deftab := deklaration.umg_aus.search(typ.symbol);
variable.deftab:=
  deklaration.umg_aus.search(variable.symbol);
deklaration.umg_ein :=
  new Umg(bezeichner.symbol,typ.deftab,...);

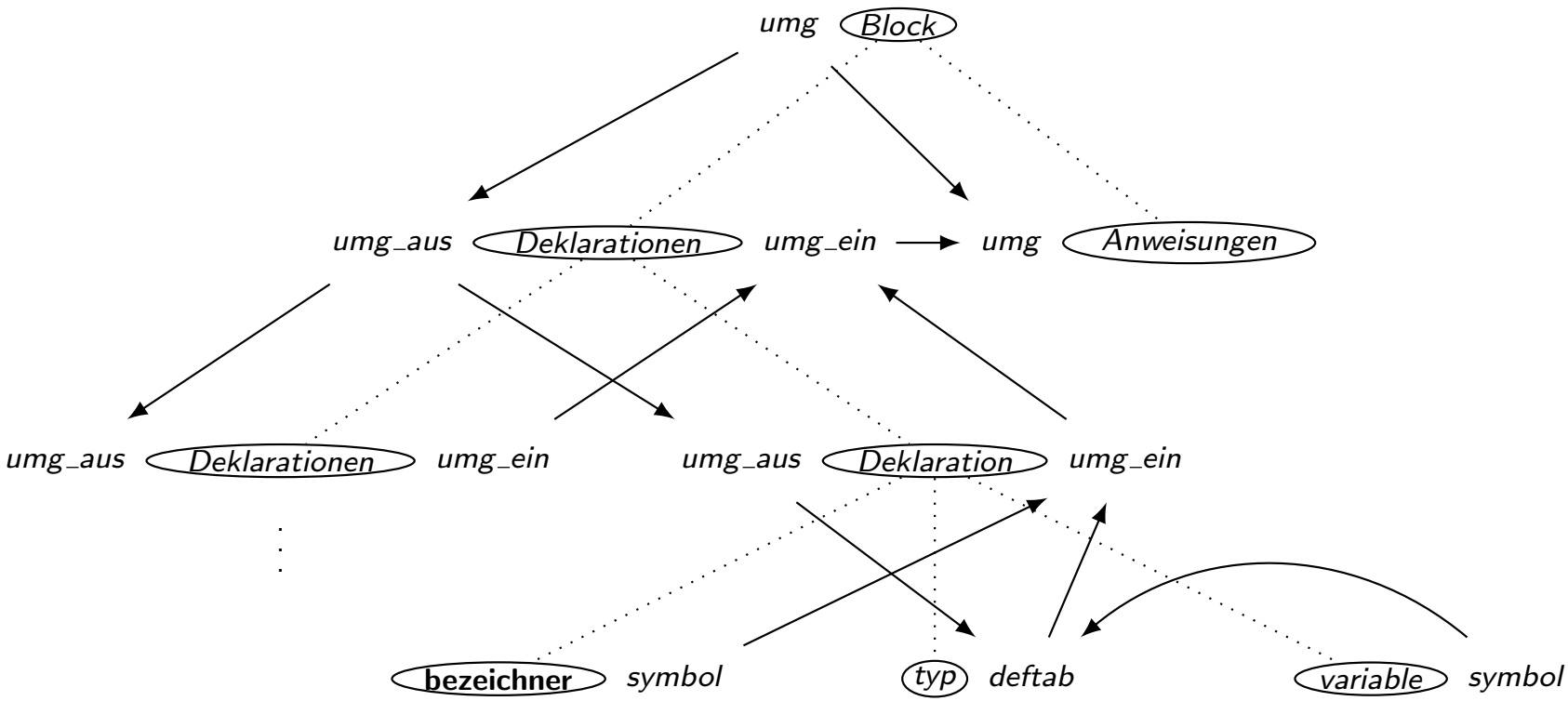
Reparieren der zirkulären Abhängigkeit

rule block ::= deklarationen ; anweisungen .
attribution
anweisungen.umg := **append**(block.umg, declarationen.umg_ein)
deklarationen.umg_aus := block.umg

rule deklarationen ::= deklarationen ';' deklaration.
attribution
deklarationen[1].umg_ein := **append**(deklarationen[2].umg_ein,
  deklaration.umg_ein);
deklarationen[2].umg_aus := deklarationen[1].umg_aus;
deklaration.umg_aus := deklarationen[1].umg_aus

rule deklaration ::= bezeichner ':' typ ':=' variable ';'.
attribution
typ.deftab := deklaration.umg_aus.search(typ.symbol);
variable.deftab:=
  deklaration.umg_aus.search(variable.symbol);
deklaration.umg_ein :=
  new Umg(bezeichner.symbol,typ.deftab,...);

Reparierte AG



LAG(1)-Attributierte Grammatiken

Eine Attributierte Grammatik gehört zur Klasse der LAG(1) wenn jedes Attribut entweder:

- synthetisiert ist
- ererbt ist, mit folgenden Einschränkungen:
  Sei $A \rightarrow X_1\ X_2 \ldots X_n$ die Produktion mit dem ererbten Attribut $X_i.a$. Die Berechnungsregeln für X_i dürfen nur folgende Attribute verwenden:
  - Attribute des Kopfes A der Produktion
  - Attribute der Symbole $X_1, X_2, \ldots, X_{i-1}$
  - Attribute von X_i, aber nur so, dass im Abhängigkeitsgraphen der Attribute von X_i keine Zyklen entstehen.

rekursiver Abstieg mit direkter Codeerzeugung

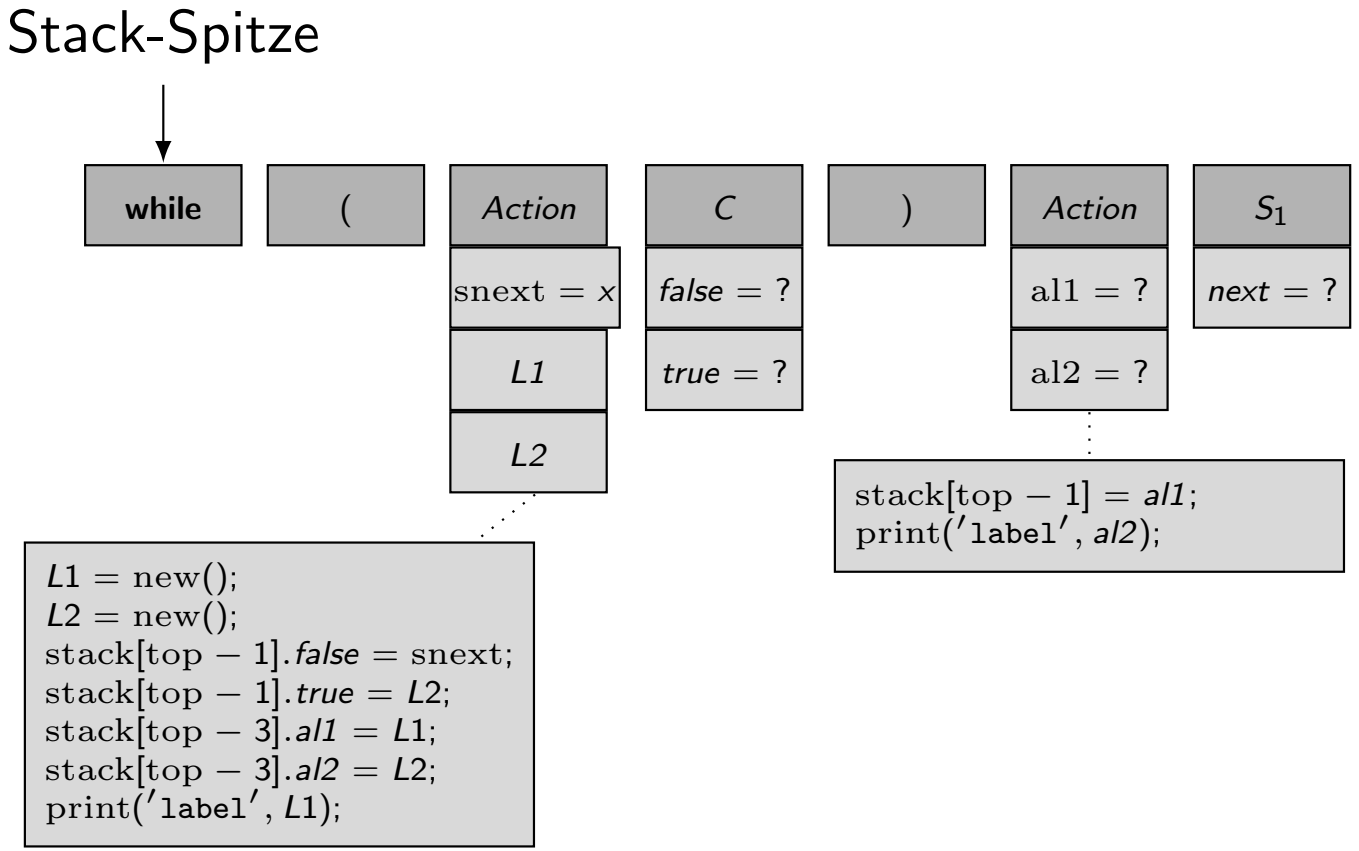
```
void parse_statement(label next) {
    label l1, l2;
    if (token == T_while) {
        next_token();
        if (token == '(') next_token(); else error(...);
        l1 = new();
        l2 = new();
        print("label", l1);
        parse_condition(next, l2);
        if (token == ')') next_token(); else error(...);
        print("label", l2);
        parse_statement(l1);
    } else {
        /* other statements */
    }
}
```


Attributierte Grammatik für **while**-Anweisungen

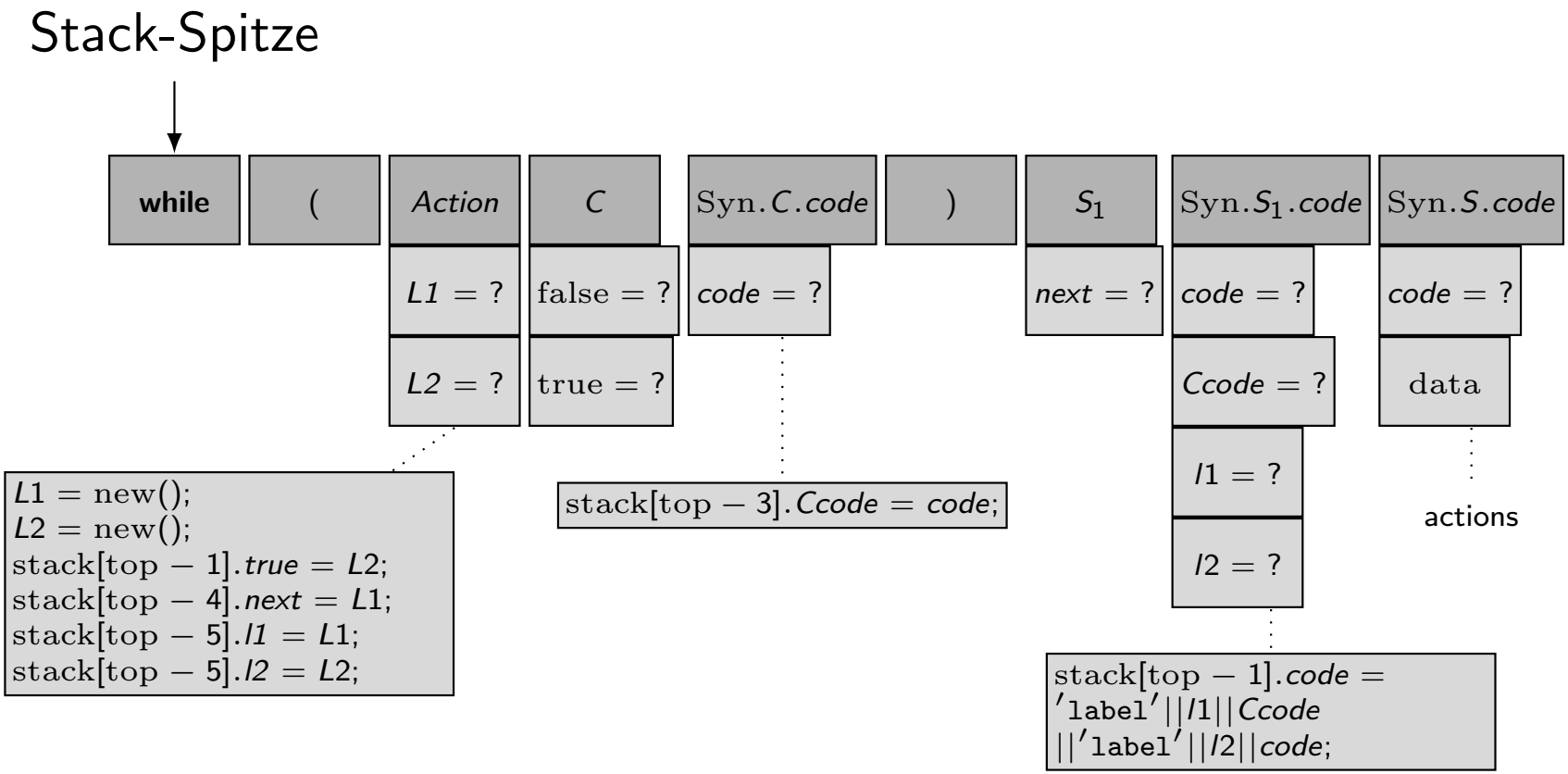
$S \rightarrow$ **while** (C) S_1

$L_1 = \text{new}();$
$L_2 = \text{new}();$
$S_1.next = L_1;$
$C.false = S.next;$
$C.true = L_2;$
$S.code =$ **label**$||L_1||C.code||$**label**$||L_2||S.code$

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



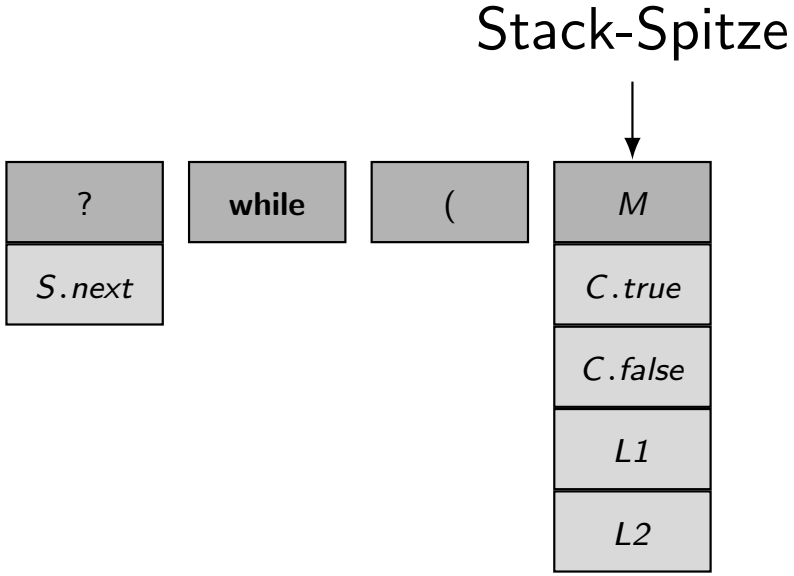
Konstruieren der synthetisierten Attribute



Erweiterte Produktionen mit Dummy-Nonterminals zur Attributberechnung

$$
\begin{aligned}
S &\rightarrow \textbf{while}\ (\ M\ C\ )\ B\ S_1 \\
M &\rightarrow \epsilon \\
N &\rightarrow \epsilon
\end{aligned}
$$

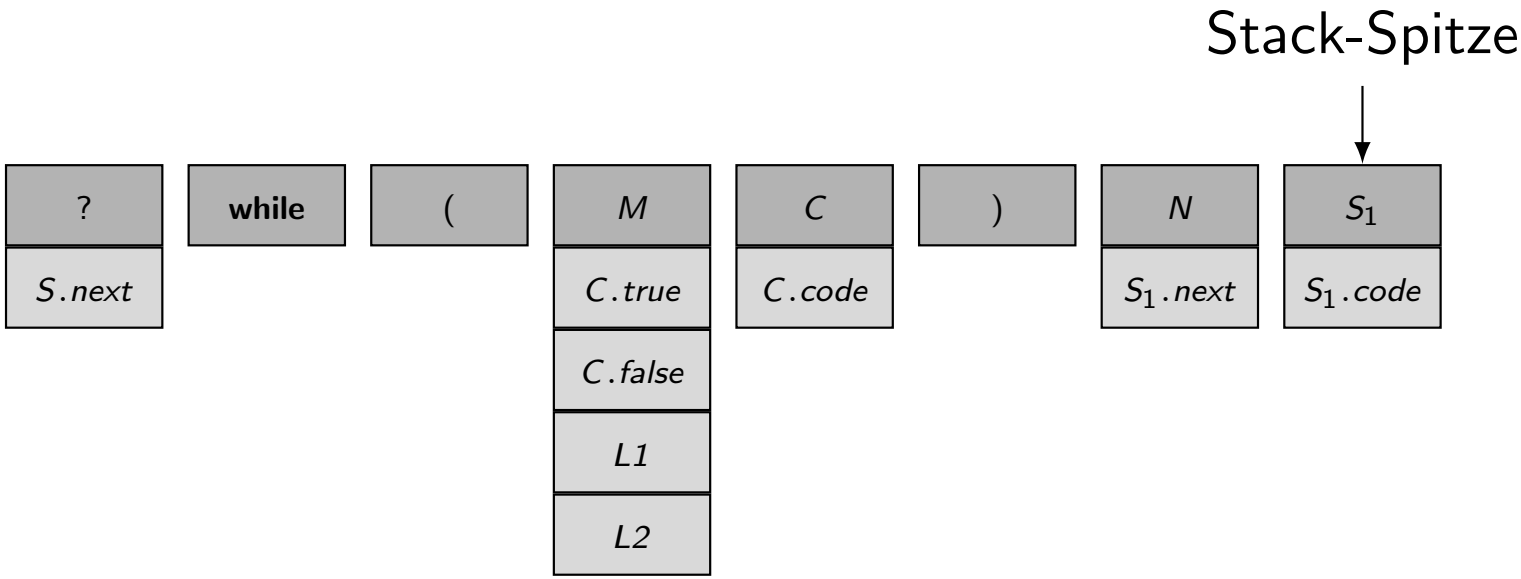
LR-Parserstack nach der Reduktion von ϵ zu M



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

$L = \text{new}();$
$L = \text{new}();$
$C.true = L_2;$
$C.false =$
$\text{stack}[\text{top} - 3].next$

LR-Parserstack mit Attributberechnung



Ausgeführte Aktionen

$$
\begin{aligned}
\text{tempCode} &= \textbf{label}||\text{stack}[\text{top}-4].\mathit{L1}||\text{stack}[\text{top}-3].\mathit{code}|| \\
&\quad\ \textbf{label}||\text{stack}[\text{top}-4].\mathit{L2}||\text{stack}[\text{top}].\mathit{code}; \\
\text{top} &= \text{top}-5; \\
\text{stack}[\text{top}].\mathit{code} &= \text{tempCode};
\end{aligned}
$$