

Kapitel 4

Attributierte Grammatiken

Kapitel 4: Attributierte Grammatiken

1 Einführung

2 Beispiele

- Taschenrechner
- AST-Aufbau
- Typdeklarationen
- Satzschluss
- Typanpassung

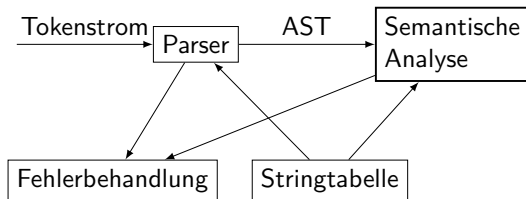
3 Grundbegriffe

4 Hierarchie

- LAG
- PAG
- OAG

5 Beispiel: Codeerzeugung mit AGs

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

1 Einführung

2 Beispiele

- Taschenrechner
- AST-Aufbau
- Typdeklarationen
- Satzumsatz
- Typanpassung

3 Grundbegriffe

4 Hierarchie

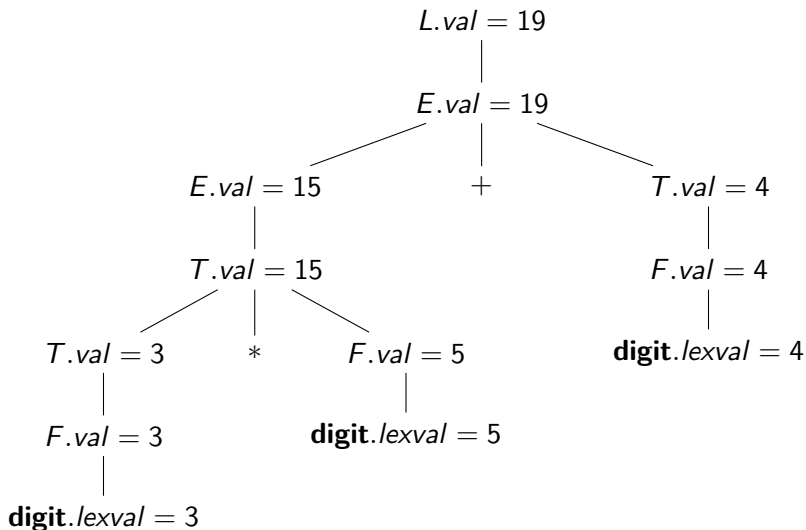
- LAG
- PAG
- OAG

5 Beispiel: Codeerzeugung mit AGs

Beispiel: Taschenrechner mit Attributierter Grammatik

	Produktion	Semantische Regeln
1)	$L \rightarrow E$	$L.val = E.val$
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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

Attributierter Parsebaum für $3 * 5 + 4$



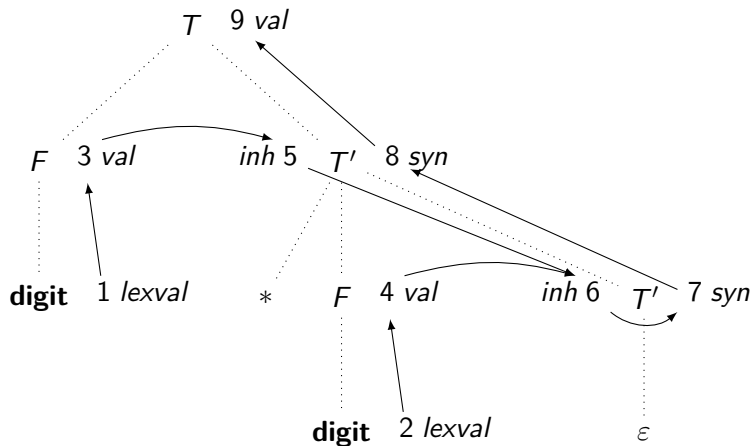
Taschenrechner Implementierung

$L \rightarrow E$	$\{ \text{print}(E.val); \}$
$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val; \}$
$E \rightarrow T$	$\{ E.val = T.val; \}$
$T \rightarrow T_1 * F$	$\{ T.val = T_1.val * F.val; \}$
$T \rightarrow F$	$\{ T.val = F.val; \}$
$F \rightarrow (E)$	$\{ F.val = E.val; \}$
$F \rightarrow \text{digit}$	$\{ F.val = \text{digit.lexval}; \}$

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)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

Abhängigkeitsgraph für attributierten Parsebaum

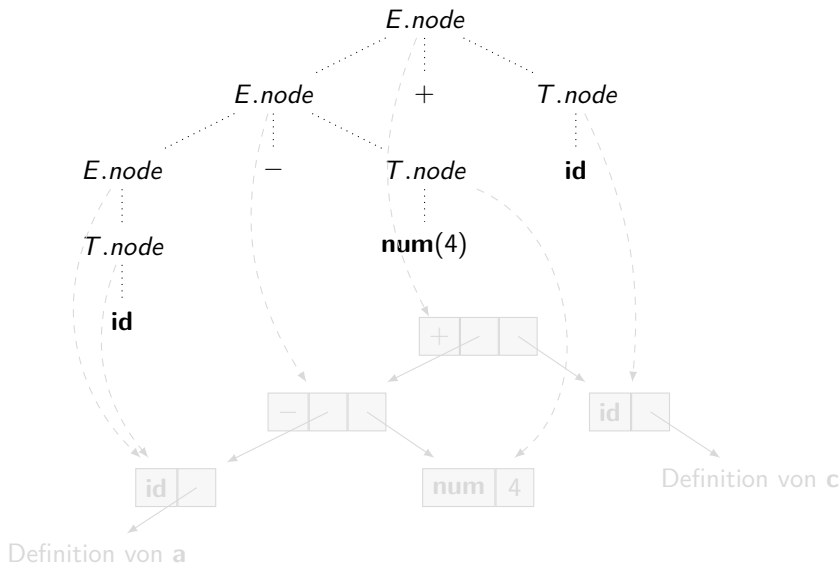


Parse-Baum für **digit** * **digit**.

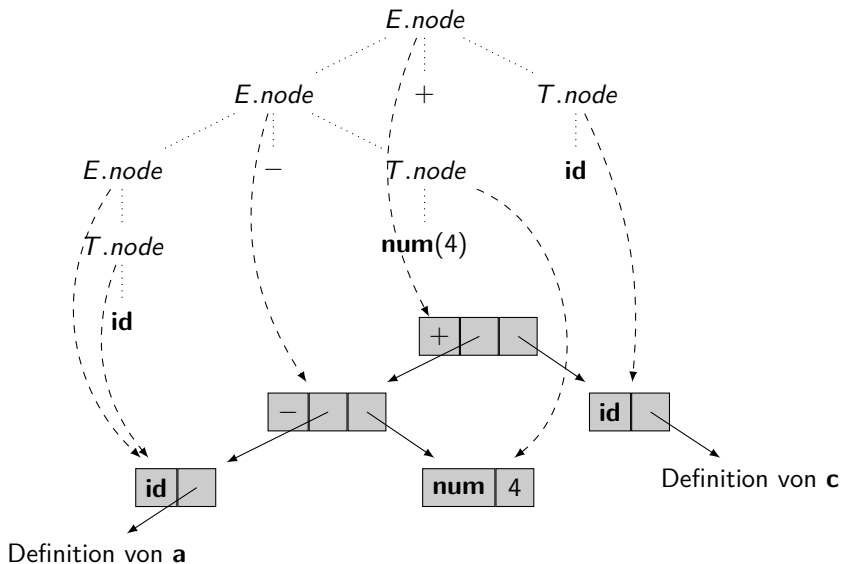
AST-Aufbau mit AGs

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

Beispiel AST-Aufbau



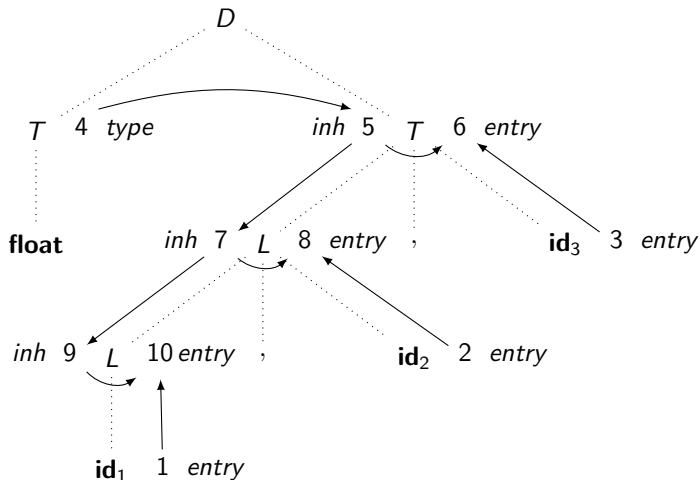
Beispiel AST-Aufbau



Grammatik mit Attributierung für Typdeklarationen

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

Abhängigkeitsgraph für attribuierten Parsebaum



Parse-Baum für **float id₁, id₂, id₃**.

Attributierte Grammatik für Arraytypen

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

Beispiel Schriftsatz

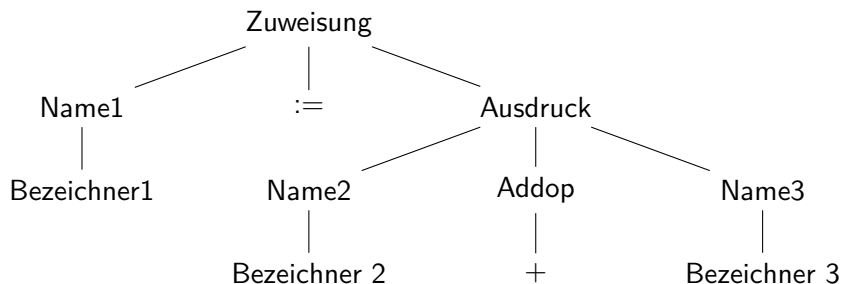


Abbildung 5.24: Konstruieren größerer Kästen aus kleineren

Attributierte Grammatik für den Satzsatz

	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)	$B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps = B.ps$ $B_2.ps = 0.7 * B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 * B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 * B.ps)$
4)	$B \rightarrow (B_1)$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \rightarrow \text{text}$	$B.ht = \text{getHt}(B.ps, \text{text.lexval})$ $B.dp = \text{getDp}(B.ps, \text{text.lexval})$

Beispiel Typanpassung - konkrete Syntax



Typanpassung - Beispiel AG

rule Zuweisung $\rightarrow$ 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 $\rightarrow$ Name addop Name .

attribution

Name[1].umg := Ausdruck.umg;

Name[2].umg := Ausdruck.umg;

Ausdruck.vor := **if** anpassbar(Name[1].vor, int) $\wedge$ anpassbar(Name[2].vor, int)
then int **else** float **end**;

addop.Type := Ausdruck.vor;

Name[1].nach := Ausdruck.vor;

Name[2].nach := Ausdruck.vor;

condition anpassbar(Ausdruck.vor, Ausdruck.nach);

rule addop $\rightarrow$ '+' .

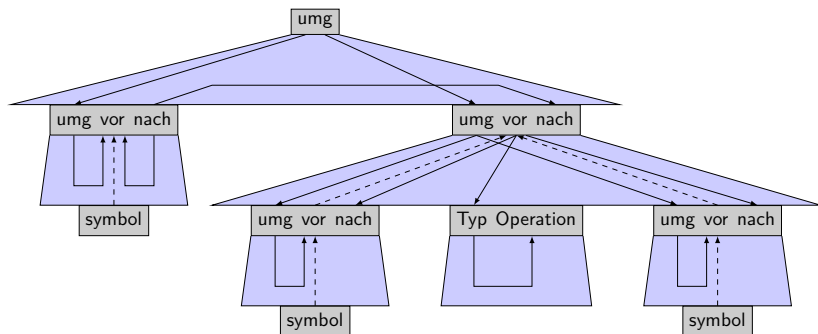
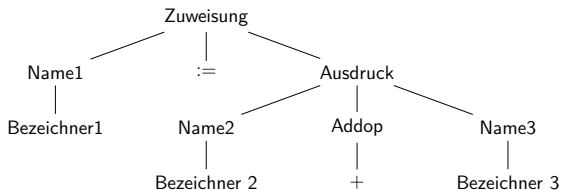
attribution addop.operation := **if** addop.Type = int **then** int_add **else** float_add **end**;

rule Name $\rightarrow$ Bezeichner .

attribution Name.vor := definiert(Bezeichner.Symbol, Name.umg);

condition anpassbar(Name.vor, Name.nach);

Typanpassung - Auswertung an konkretem Syntaxbaum



Kapitel 4: Attributierte Grammatiken

1 Einführung

2 Beispiele

- Taschenrechner
- AST-Aufbau
- Typdeklarationen
- Satzumsatz
- Typanpassung

3 Grundbegriffe

4 Hierarchie

- LAG
- PAG
- OAG

5 Beispiel: Codeerzeugung mit AGs

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

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

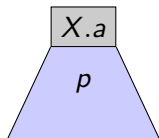
sind Attribute, die zu einem Nichtterminal der Produktion
 $p : X_0 \rightarrow X_1 \dots X_n$ gehören ($0 \leq i \wedge j \leq n$)

- $AF(p) := \{X.a \mid X.a := f(\dots) \in R(p)\}$
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

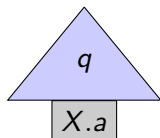
$$\text{syn}(X) = \{X.a \mid \exists p : X \rightarrow X_1 \dots X_n \in P \wedge X.a \in \text{AF}(p)\}$$

Attribute in $\text{syn}(X)$ heißen **synthetisiert** oder **abgeleitet**



$$\text{inh}(X) = \{X.a \mid \exists q : Y \rightarrow \mu X \omega \in P \wedge X.a \in \text{AF}(q)\}$$

Attribute in $\text{inh}(X)$ heißen **ererb**t (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 $X.a := f(\dots) \in R$.

Vollständigkeit einer AG

Eine AG heißt **vollständig**, wenn

- $\forall(p : X \rightarrow x \in P) : \text{syn}(X) \subseteq \text{AF}(p)$
- $\forall(q : Y \rightarrow \mu X \omega \in P) : \text{inh}(X) \subseteq \text{AF}(q)$
- $\text{syn}(X) \cup \text{inh}(X) = A(X)$
- $\text{inh}(Z) = \emptyset$, 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 attribuiert**, wenn

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

Die Bedingung $B(p)$ einer Produktion $p : X_0 \rightarrow X_1 \dots X_n$ 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

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

so ist ein Strukturbaum genau dann korrekt attribuiert, 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

1 Einführung

2 Beispiele

- Taschenrechner
- AST-Aufbau
- Typdeklarationen
- Satzumsatz
- Typanpassung

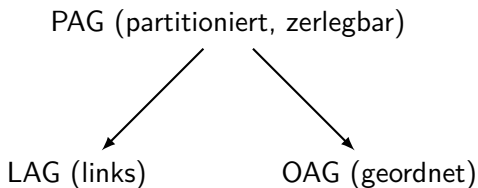
3 Grundbegriffe

4 Hierarchie

- LAG
- PAG
- OAG

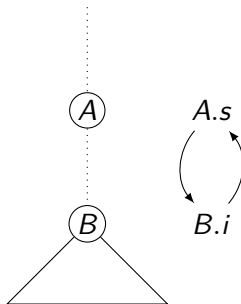
5 Beispiel: Codeerzeugung mit AGs

AG Hierarchie



Zyklische Abhängigkeit

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



Direkte Attributabhängigkeiten

Die Menge der **direkten Attributabhängigkeiten** einer Produktion $p : X_0 \rightarrow X_1 \dots X_n \in P$ ist gegeben durch

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

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

Beispiel DDP

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

$$\text{DDP}(D \rightarrow T L) = \{(T.type, L.inh)\}$$

$$\text{DDP}(T \rightarrow \mathbf{int}) = \{(\mathbf{int.integer}, T.type)\}$$

$$\text{DDP}(T \rightarrow \mathbf{float}) = \{(\mathbf{float.float}, T.type)\}$$

$$\text{DDP}(L \rightarrow L_1, \mathbf{id}) = \{(L.inh, L_1.inh), (L.inh, \mathbf{id.entry})\}$$

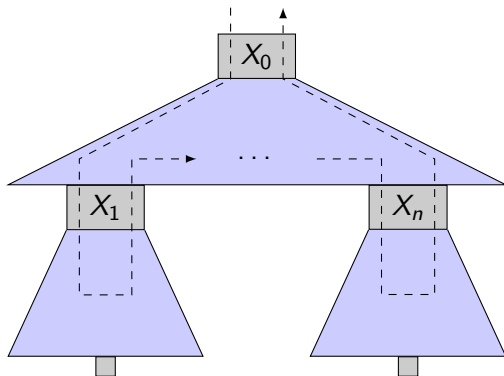
$$\text{DDP}(L \rightarrow \mathbf{id}) = \{(L.inh, \mathbf{id.entry})\}$$

LAG(1) - Attributgrammatik

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

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

berechnet werden können.



DDP bei LAG(1)

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

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

Bemerkung: CI^+ bezeichnet die transitive Hülle

Bemerkungen zu LAG(1)

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

LAG(k) - Attributgrammatik

Eine AG ist LAG(k), wenn für jede Produktion
 $p : X_0 \rightarrow X_1 \dots X_n \in P$ die Attribute so in Gruppen

$$G_1, \dots, G_k$$

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

- Entspricht dem intuitiven Begriff mehrerer Übersetzerpässe
- Erforderlich, da LAG(1) fast nie ausreicht

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
 $X \rightarrow X_1 \dots X_n$
- 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 vorliegt. 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 $M_e: M \rightarrow \varepsilon$
- 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.

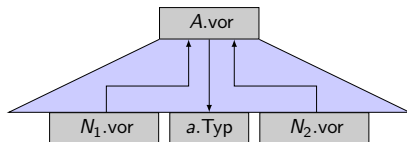
Normalisierte direkte Abhängigkeiten

Für alle $p : X_0 \rightarrow X_1 \dots X_n \in P$ definiere die **normalisierten direkten Abhängigkeiten**¹

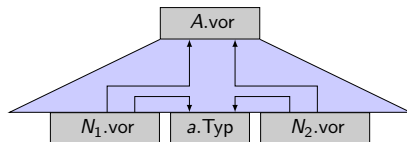
$$\text{NDDP}(p) = \text{DDP}(p)^+ \setminus \{(X_i.a, X_j.b) \mid X_i.a, X_j.b \in \text{AF}(p)\}$$

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

DDP



NDDP



¹DDP⁺ bezeichnet die transitive Hülle der direkten Abhängigkeiten

Induzierte Abhängigkeiten

Berechnung von IDP und IDS:

- 1 Für alle $p \in P$ setze $IDP(p) := NDDP(p)$
- 2 Für alle $X \in N \cup T$ setze
 $IDS(X) := \{(X.a, X.b) \mid \exists q \in P : (X.a, X.b) \in IDP(q)^+\}$
- 3 Für alle $p : X_0 \rightarrow X_1 \dots X_n \in P$ setze
 $IDP(p) := IDP(p) \cup IDS(X_0) \cup \dots \cup IDS(X_n)$
- 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!

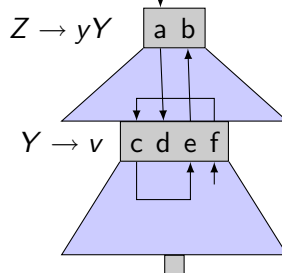
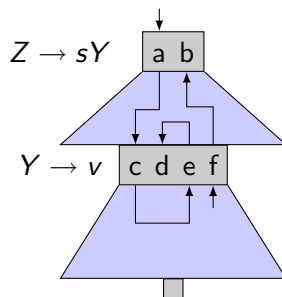
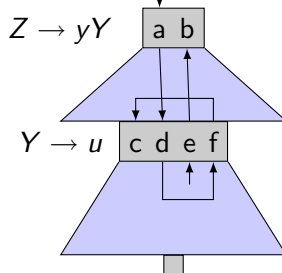
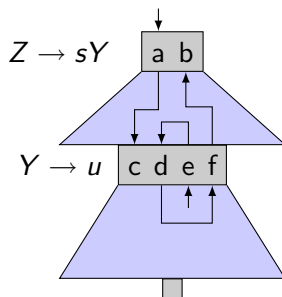
Anschaulich: Alle Abhängigkeiten aller möglichen Ableitungsbäume werden zur Gewinnung von IDS und IDP übereinandergelegt.

AG mit 4 Ableitungsbäumen

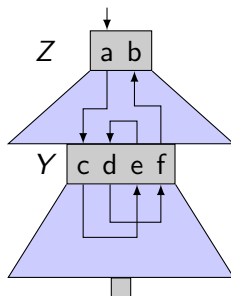
$S \rightarrow Z$

$Z \rightarrow sY \mid yY$

$Y \rightarrow u \mid v$



IDP und IDS



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

Partitionierung der Attribute

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

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

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

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 $A_1(X), \dots, A_m(X)$ 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 $A_1(X), \dots, A_m(X)$ gegeben.

Für alle $p : X_0 \rightarrow X_1 \dots X_n \in P$ ist

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

die **Abhängigkeitsrelation** über der Produktion p .

Satz: AG ist zerlegbar, gdw. $DP(p)$ 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 $k > 0$

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

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

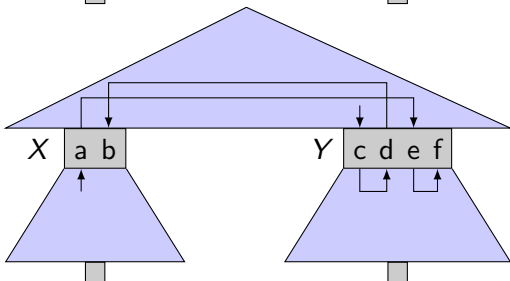
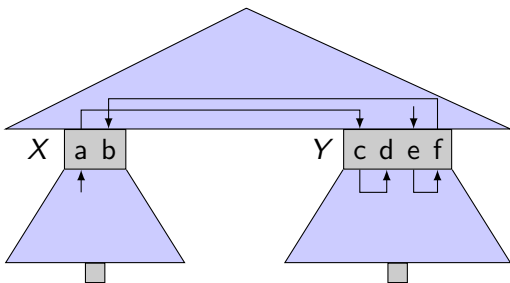
Definiere Partitionierung durch

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

- 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

$Z \rightarrow sXY \mid tXY, \quad X \rightarrow u, \quad Y \rightarrow v$



Attributierte Grammatiken

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

$b \rightarrow a \rightarrow \{c, e\} \rightarrow$

$\{d, f\} \rightarrow b$ in DP

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

Erweiterung PAG $\rightarrow$ 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.

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 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 $\text{LAG}(k)$ 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.

Besuchssequenzen

Attribute werden berechnet im Kontext der

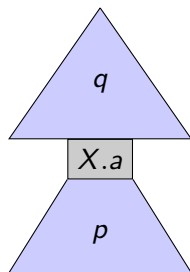
- Unterproduktion: abgeleitetes Attribut
- Oberproduktion: ererbtes Attribut

Berechnungen im Kontext p und q interagieren:

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

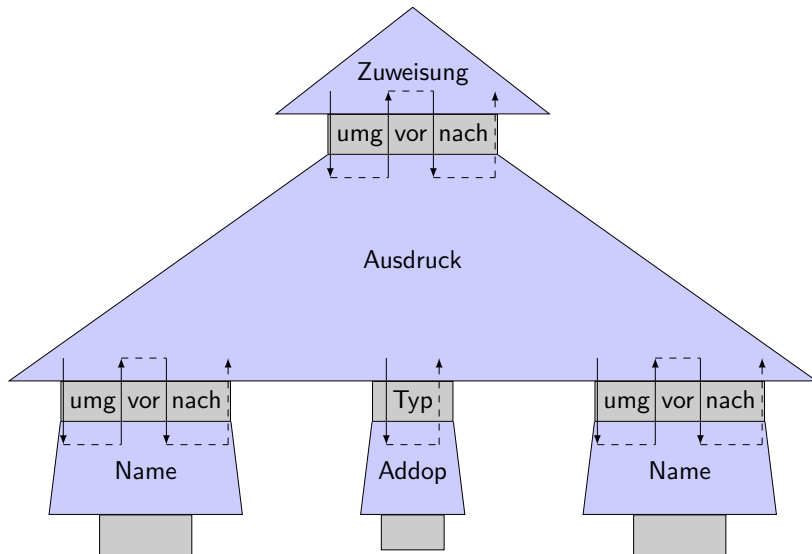
Aktivitäten an der Schnittstelle X :

- 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 $\rightarrow$ 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 $\rightarrow$ Name addop Name .

attribution

Name[1].umg := Ausdruck.umg;

Name[2].umg := Ausdruck.umg;

Ausdruck.vor := **if** anpassbar(Name[1].vor, int) $\wedge$ anpassbar(Name[2].vor, int)
then int **else** float **end**;

addop.Type := Ausdruck.vor;

Name[1].nach := Ausdruck.vor;

Name[2].nach := Ausdruck.vor;

condition anpassbar(Ausdruck.vor, Ausdruck.nach);

rule addop $\rightarrow$ '+' .

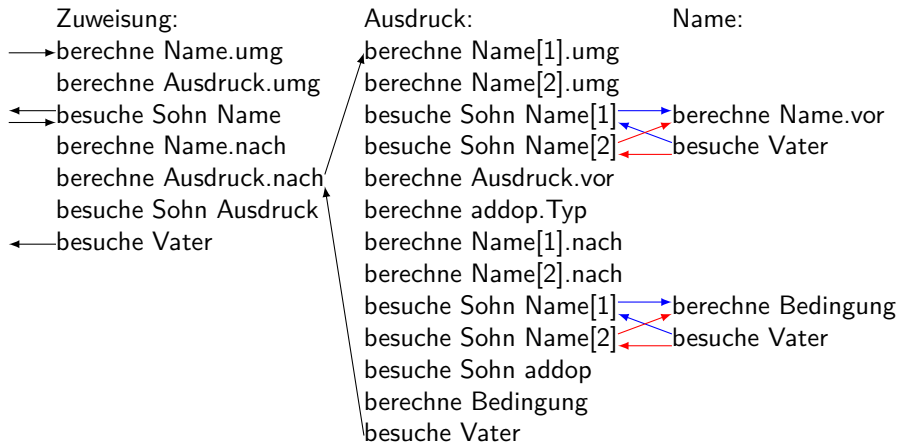
attribution addop.operation := **if** addop.Type = int **then** int_add **else** float_add **end**;

rule Name $\rightarrow$ Bezeichner .

attribution Name.vor := definiert(Bezeichner.Symbol, Name.umg);

condition anpassbar(Name.vor, Name.nach);

Besuchssequenzen als Kommunikationsprotokoll



Kapitel 4: Attributierte Grammatiken

1 Einführung

2 Beispiele

- Taschenrechner
- AST-Aufbau
- Typdeklarationen
- Satzschluss
- Typanpassung

3 Grundbegriffe

4 Hierarchie

- LAG
- PAG
- OAG

5 Beispiel: Codeerzeugung mit AGs

Attributierte Grammatik für **while**-Anweisungen

rule statement $\rightarrow$ '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

```
S → while ( { L1 = new_label(); L2 = new_label();  
              C.false = S.next; C.true = L2;  
              print("label ", L1); }  
  C )      { S1.next = L1; print("label ", L2); }  
  S1
```

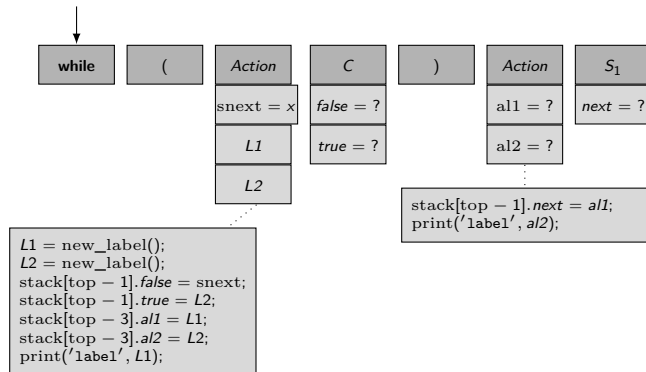
rekursiver Abstieg mit direkter Codeerzeugung

statement $\rightarrow$ **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 */  
    }  
}
```

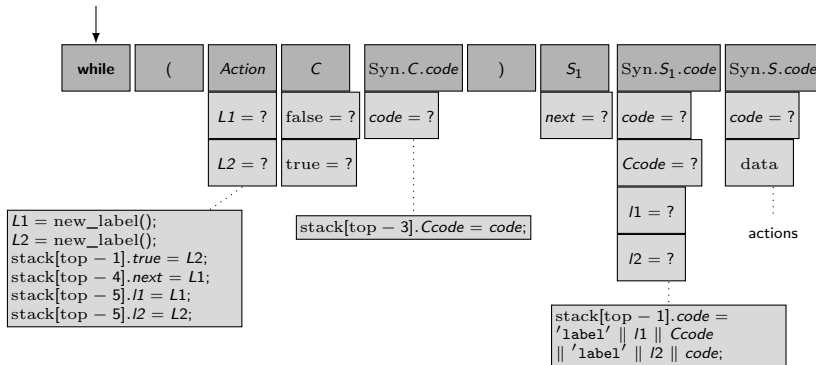
Expandierung von S entsprechend der **while**-Produktion

Stack-Spitze



Konstruieren der synthetisierten Attribute

Stack-Spitze



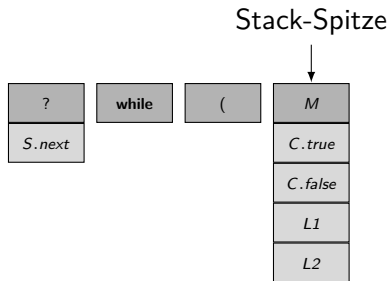
Erweiterte Produktionen mit Dummy-Nonterminals zur Attributberechnung

$$S \rightarrow \mathbf{while} (M \ C) \ B \ S_1$$

$$M \rightarrow \varepsilon$$

$$N \rightarrow \varepsilon$$

LR-Parserstack nach der Reduktion von ε zu M



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

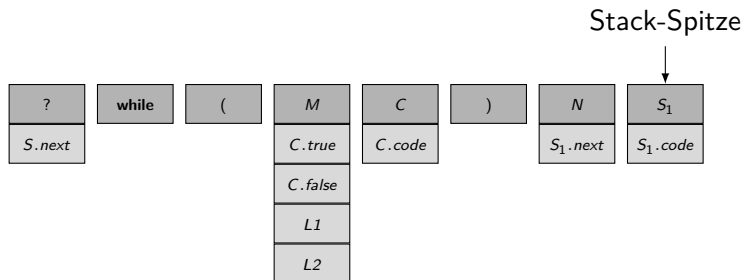
```
L1 = new_label();
```

```
L2 = new_label();
```

```
C.true = L2;
```

```
C.false = stack[top - 3].next
```

LR-Parserstack mit Attributberechnung



Ausgeführte Aktionen

```
tempCode = label || stack[top - 4].L1 || stack[top - 3].code ||  
           label || stack[top - 4].L2 || stack[top].code;  
top = top - 5;  
stack[top].code = tempCode;
```