



# 4. Kapitel

## Semantische Analyse Teil 1

### Attributierte Grammatiken



# Kapitel 4: AGs

## 0. Einbettung

1. Grundbegriffe

2. Hierarchie

WAG, ANCAG, PAG, OAG, LAG, RAG, AAG

Besuchssequenzen

Induzierte Abhängigkeiten

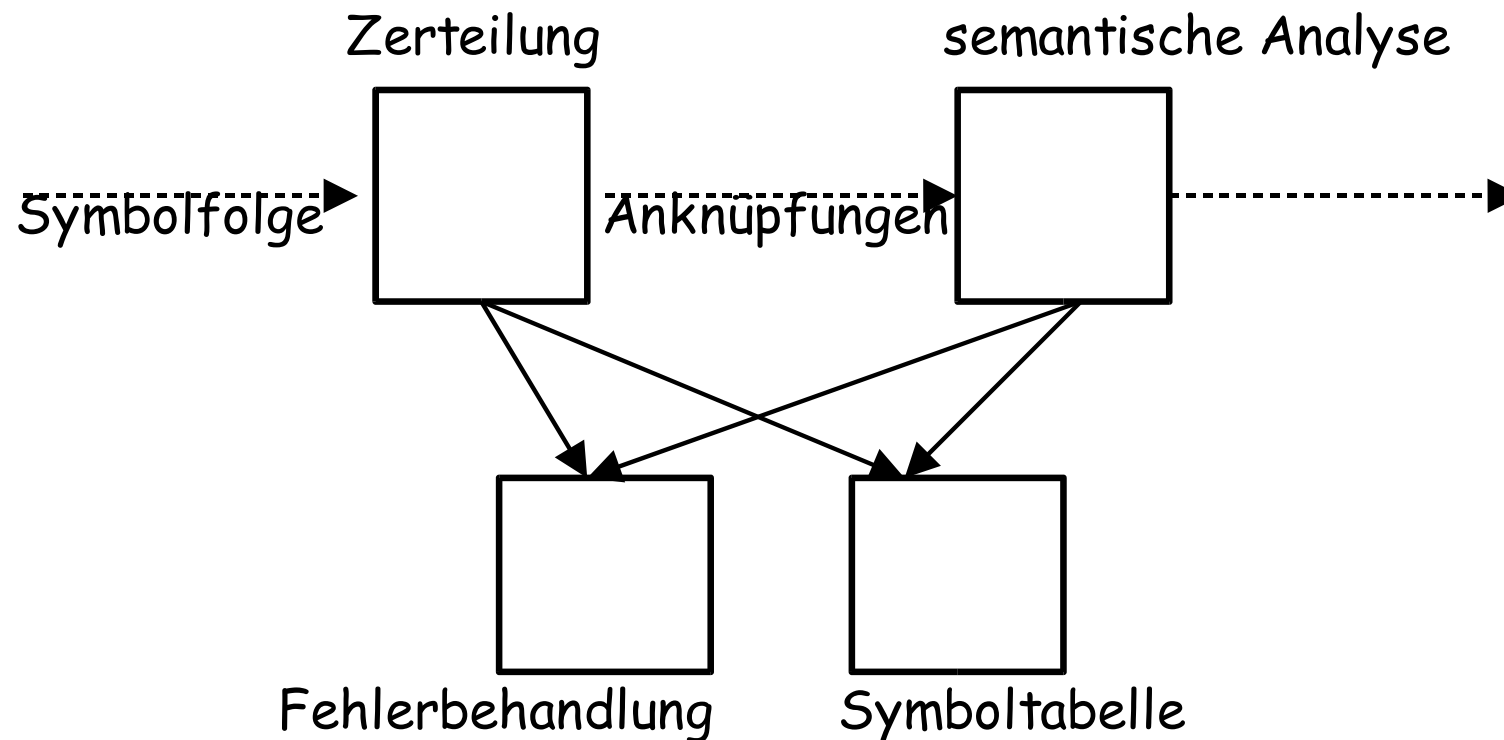
3. Implementierung

4. Attributspeicherung

5. Zusammenfassung

# 4.0 Schnittstelle

## Zerteiler - semantische Analyse



# 4.0 Konkrete und abstrakte Syntax

Konkrete Syntax  $G_k$  der zu übersetzenden Sprache

(Datenstruktur linear: Symbolfolge). Enthält:

1. Klammer ( ), begin end, etc.
2. Kettenproduktionen
3. Schlüsselwörter

Abstrakte Syntax  $G_a$  des Strukturbaums (Datenstruktur: Baum).

1. Klammerung durch Zerteilung bereits eindeutig
2. Kettenproduktionen überflüssig, wenn keine semantische Bedeutung
3. Schlüsselwörter dienen der eindeutigen Zerteilung, jetzt immer überflüssig, werden weggelassen.

Abbildung konkrete auf abstrakte Syntax durch Zerteiler (Verarbeitung von Anknüpfungen), eventuell weitere Transformation während semantischer Analyse

# 4.0 Anknüpfungen

## Struktur-Anknüpfungen

Wann immer ein Nichtterminal der  $G_k$  hergeleitet wird, wird ein Konstruktor der entsprechenden Ecke im Ableitungsbaum für  $G_a$  generiert, so es eine gibt.

## Symbol-Anknüpfungen

Den Constructoren werden gegebenenfalls Merkmale von Symbolen übergeben.

# 4.0 Abstrakter Strukturbaum (AST)

```
public class AnweisungsListe implements Node{
    Anweisung a;
    AnweisungsListe as;
    AnweisungsListe(Anweisung a,AnweisungsListe as) {
        this.a = a
        this.as = as;
    }
}
public class Anweisung {...}
public class IfAnweisung implements Anweisung {
    Ausdruck e;
    AnweisungsListe if;
    AnweisungsListe then;
    IfAnweisung(Ausdruck e,AnweisungsListe if,AnweisungsListe then)
    {
        this.e = e;
        this.if = if;
        this.then = then;
    }
}
```

# 4.0 Abstrakter Strukturbaum (AST)

```
public class Ausdruck implements Node {
    Operator o;
    Ausdruck e1;
    Ausdruck e2;
    Ausdruck ( Operator o , Ausdruck e1 , Ausdruck e2 ) { . . . }
}
public class Bezeichner implements Ausdruck {
    SymboltabellenEintrag st ;
    Bezeichner ( SymboltabellenEintrag st ) {
        this.st = st ;
    }
}
public class Konstante implements Ausdruck {
    SymboltabellenEintrag st;
    konstante (symboltabelleneintrag st) {
        this.st = st;
    }
}
```

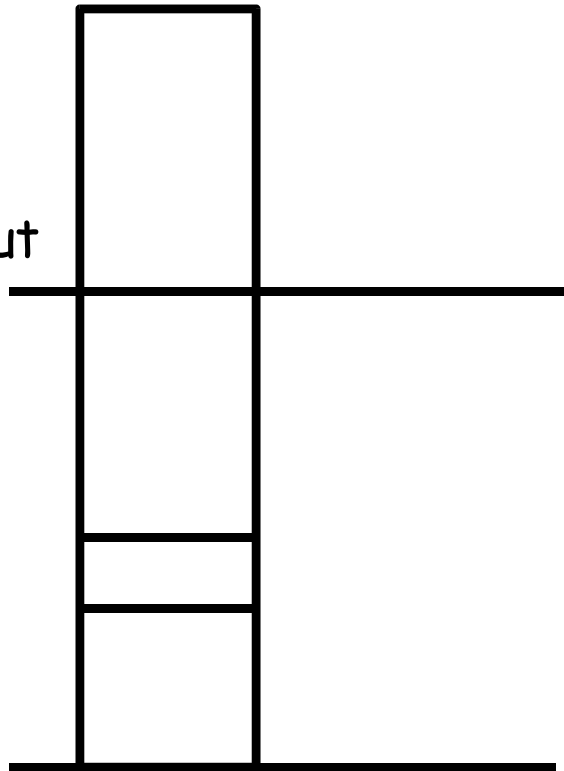
# 4.0 *obstack*

`obstack_grow`

Objekte werden aufgebaut

`obstack_finish`

Objekte mit fester  
Größe und Adresse





# Kapitel 4: AGs

0. Einbettung

1. Grundbegriffe

2. Hierarchie

WAG, ANCAG, PAG, OAG, LAG, RAG, AAG

Besuchssequenzen

Induzierte Abhängigkeiten

3. Implementierung

4. Attributspeicherung

5. Zusammenfassung

# 4.1 Attributgrammatiken (AG)

Verfahren zur Spezifikation der Eigenschaften von Bäumen

- Baum beschrieben durch kontextfreie Grammatik

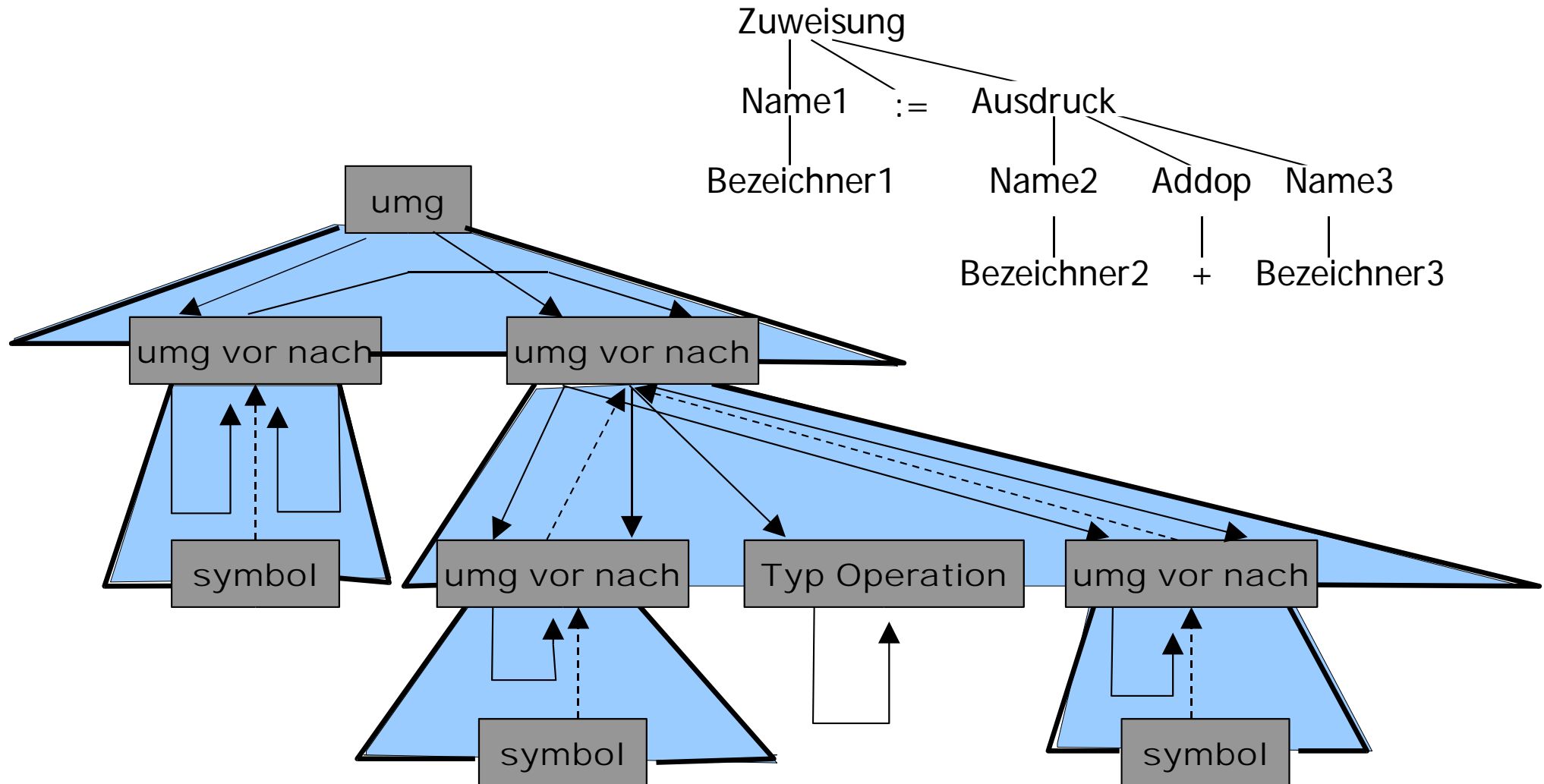
wird eingesetzt zur Spezifikation der semantischen Analyse

auch bei vielen anderen *software engineering* zu gebrauchen

Denkschema:

- jede Ecke im Baum besitzt ein oder mehrere **Attribute**
- Attribute können vorbesetzt sein, z.B. Positionsangaben, Bezeichner: Verweis in Symboltabelle
- Attribute verschiedener Ecken 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

# 4.1 Beispiel



# 4.1 Beispiel Fortsetzung

**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** real **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)  $\wedge$  *anpaßbar*(*Name*[2].*vor*,int) **then** int  
**else** real **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** real\_add  
**end**;

**rule** *Name* ::= *Bezeichner* .

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

**condition** *anpaßbar*(*Name.vor*,*Name.nach*);

**rule** *Name* ::= *Bezeichner* .

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

**condition** *anpaßbar*(*Name.vor*,*Name.nach*);



# 4.1 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 von  
 $p: X_0 \rightarrow X_1 \dots X_n$  gehören,  $0 \leq i, j \leq n$
- $AF(p) := \{X_i.a \mid p: X_0 \rightarrow X_1 \dots X_n, 0 \leq i \leq n, X_i.a := f(\dots) \in R(p)\}$  heißt Menge der  
in  $p$  definierten Attribute
- Jedes Attribut  $X.a \in A(X)$  einer Ecke  $X$  im Strukturbaum eines Satzes  
der Sprache  $L(G)$  mit maximal einer Regel aus  $R$  berechenbar.

# 4.1 Synthetisierte und ererbte Attribute

Die folgenden Mengen sind disjunkt für alle  $X$  im Vokabular von  $G$ :

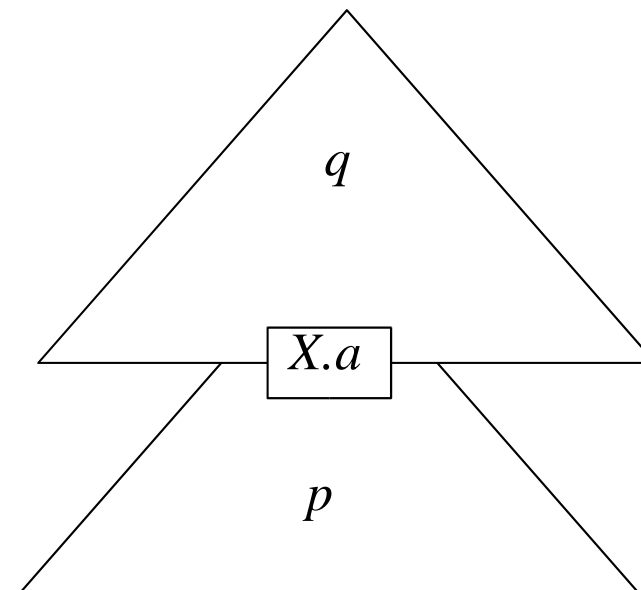
$$AS(X) = \{X_0.a \mid \exists p: X_0 \rightarrow X_1 \dots X_n \in P \wedge X_0.a \in AF(p)\}$$

$$AI(X) = \{X.a \mid \exists q: Y \rightarrow uXv \in P \wedge X.a \in AF(q)\}$$

Grund: Für alle  $a \in A(X)$  nur eine Berechnungsregel  $X.a \leftarrow f(\dots)$  in  $R$ .

Attribute in  $AS(X)$  heißen  
synthetisiert oder abgeleitet

Attribute in  $AI(X)$  heißen  
erbt (*inherited*)



# 4.1 Vollständigkeit einer AG

Eine AG heißt **vollständig**, wenn

- $\forall p: X \rightarrow x \in P: AS(X) \subseteq AF(p)$
- $\forall q: Y \rightarrow \mu X \nu \in P: AI(X) \subseteq AF(q)$
- $AS(X) \cup AI(X) = A(X)$
- $AI(Z) = \emptyset$ , wenn  $Z$  das Axiom der  $kg$  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

# 4.1 Korrekte Attributierung

Ein Strukturbaum heißt **korrekt attributiert**, wenn 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 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: AGs

0. Einbettung

1. Grundbegriffe

2. Hierarchie

WAG, ANCAG, PAG, OAG, LAG, RAG, AAG

Besuchssequenzen

Induzierte Abhängigkeiten

3. Implementierung

4. Attributspeicherung

5. Zusammenfassung

## 4.2 AG Hierarchie

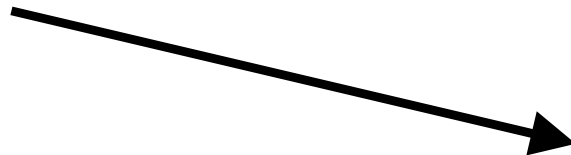
*WAG* (wohldefiniert)



*ANCAG* (absolut azyklisch)



*PAG* (partitioniert, zerlegbar)



*OAG* (geordnet)



*LAG, RAG, AAG* (links/rechts/alternierend)

## 4.2 Wohldefiniertheit einer AG: WAG

Eine AG ist wohldefiniert gdw. sie vollständig ist und für jeden Strukturbaum alle Attribute **effektiv** berechenbar sind.  
Also: für jedes Attribut im Baum gibt es genau eine Regel. Attribute sind nicht zyklisch voneinander abhängig.

**Wohldefiniertheit ist nur mit exponentiellem Aufwand nachprüfbar!**

Intuitives Schema (Fang 1972): Ordne jedem Attribut  $X.a$  im Baum einen Prozeß zu. Ein Prozeß wird gestartet, wenn alle Attribute  $X'.b$  der Attributierungsregel  $X.a := f(\dots, X'.b, \dots)$  bekannt sind (Synchronisierungsbedingung); nach Berechnung von  $X.a$  teilt er dies allen Prozessen mit, die  $X.a$  verwenden. Eine AG ist wohldefiniert gdw. dieses Prozeßsystem verklemmungsfrei ist.

## 4.2 Kriterium für WAG

Menge der **direkten Attributabhängigkeiten** einer Produktion

$$p : X_0 \rightarrow X_1 \dots X_n \in P: DDP(p) = \{(X_i.a, X_j.b) \mid X_j.b := f(\dots X_i.a \dots) \in R(p)\}$$

AG ist **lokal azyklisch**, wenn  $DDP(p)$  azyklisch ist für alle  $p \in P$ .

$S$  sei ein attributierter Strukturbaum zu einem Satz aus  $L(G)$  und

$K_0 \dots K_n$  seien Ecken im Strukturbaum zu einer Regel  $p : X_0 \rightarrow X_1 \dots X_n$ .

$K_i.a \rightarrow K_j.b$  wenn  $(X_i.a, X_j.b) \in DDP(p)$ .

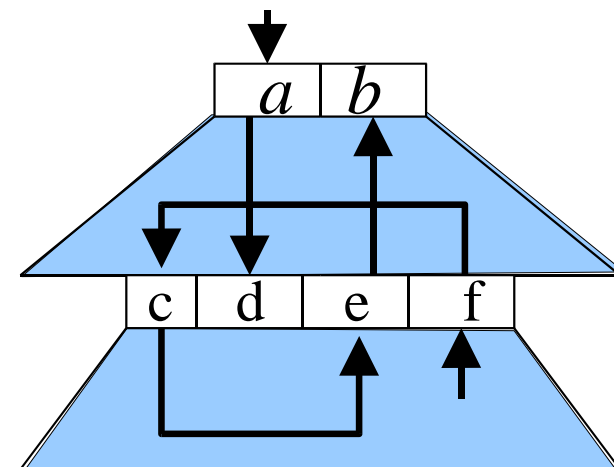
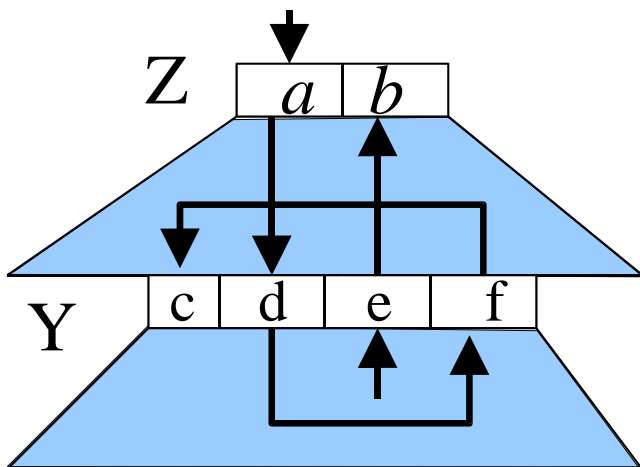
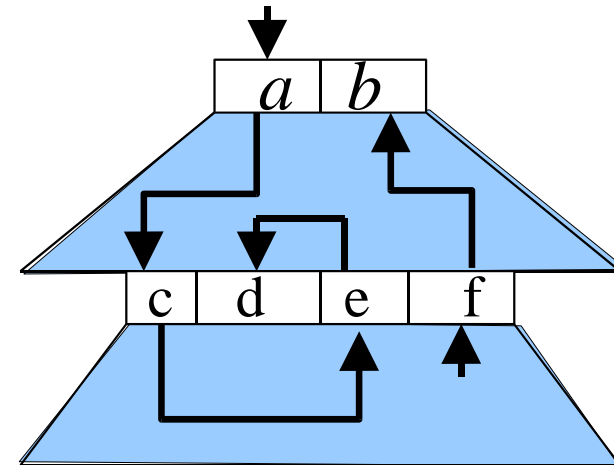
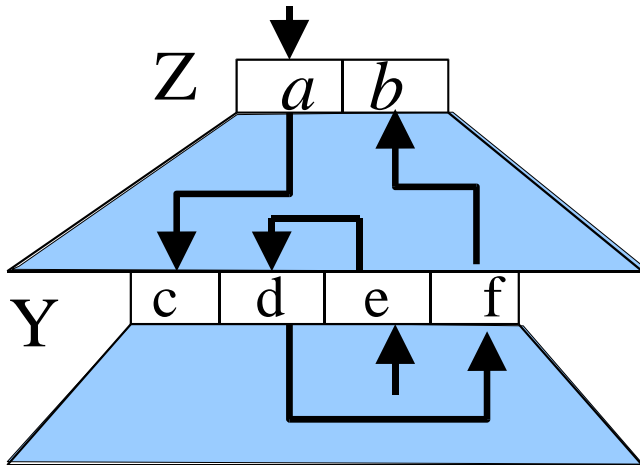
**Abhängigkeiten**  $DT(S)$  über dem Baum  $S : DT(S) = \{K_i.a \rightarrow K_j.b\}$

wobei alle Anwendungen von Produktionen in  $S$  berücksichtigt werden.

AG ist WAG gdw. sie vollständig und  $DT(S)$  azyklisch ist **für alle**  $S$ .

# 4.2 WAG mit 4 Ableitungsbäumen

$S ::= Z$   
 $Z ::= s Y / y Y$   
 $Y ::= u / v$



## 4.2 Besuchssequenzen

Attribute werden berechnet im Kontext der

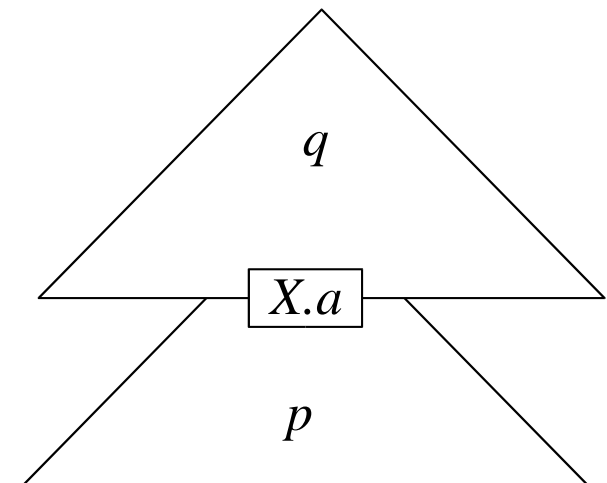
- Oberproduktion: abgeleitetes Attribut
- Unterproduktion: ererbtes Attribut

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

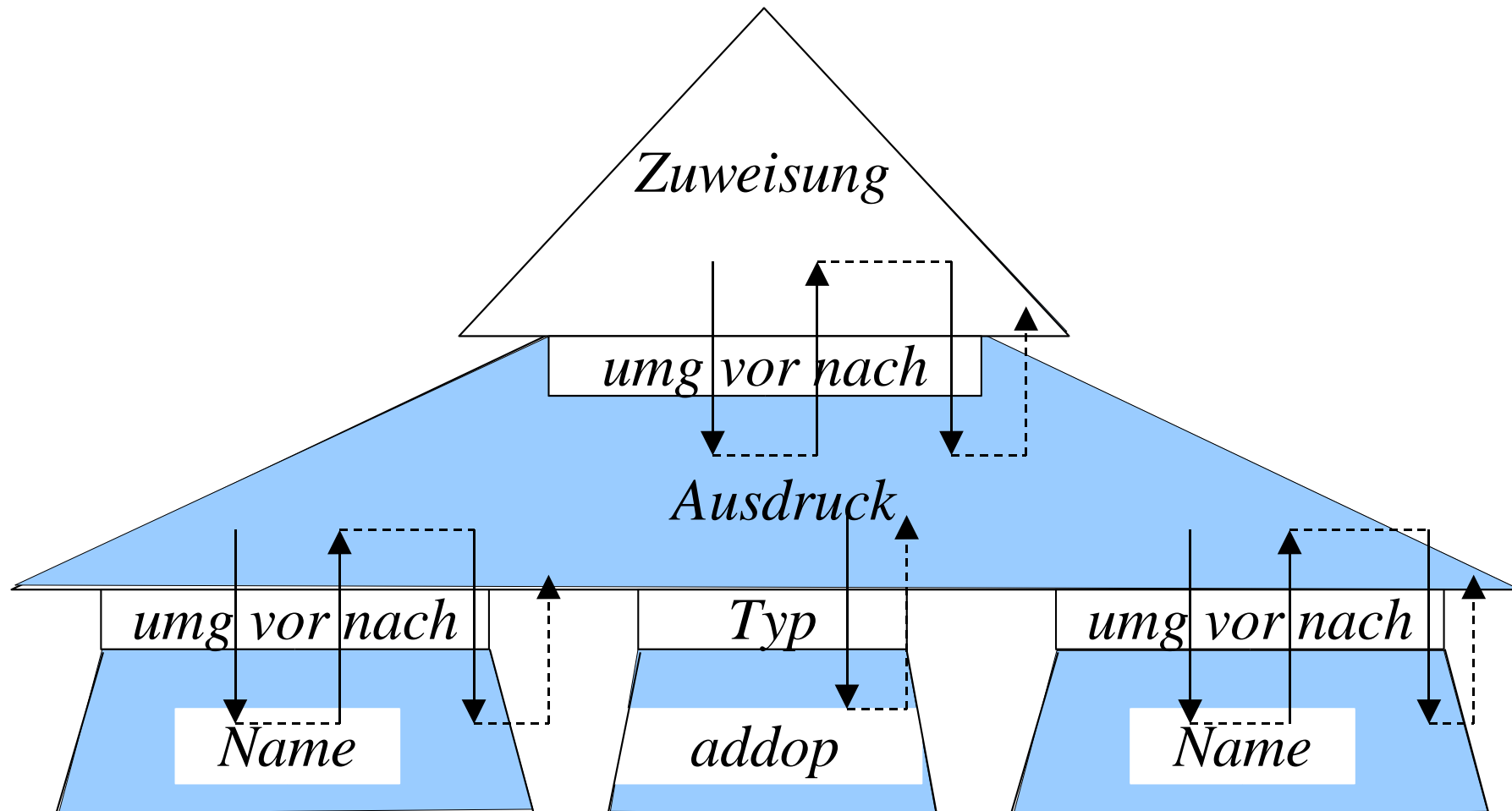
Aktivitäten an der Schnittstelle  $X$ :

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



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

## 4.2 Besuchssequenzen



## 4.2 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** real **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)  $\wedge$  *anpaßbar*(*Name*[2].*vor*,int) **then** int  
**else** real **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** real\_add  
**end**;

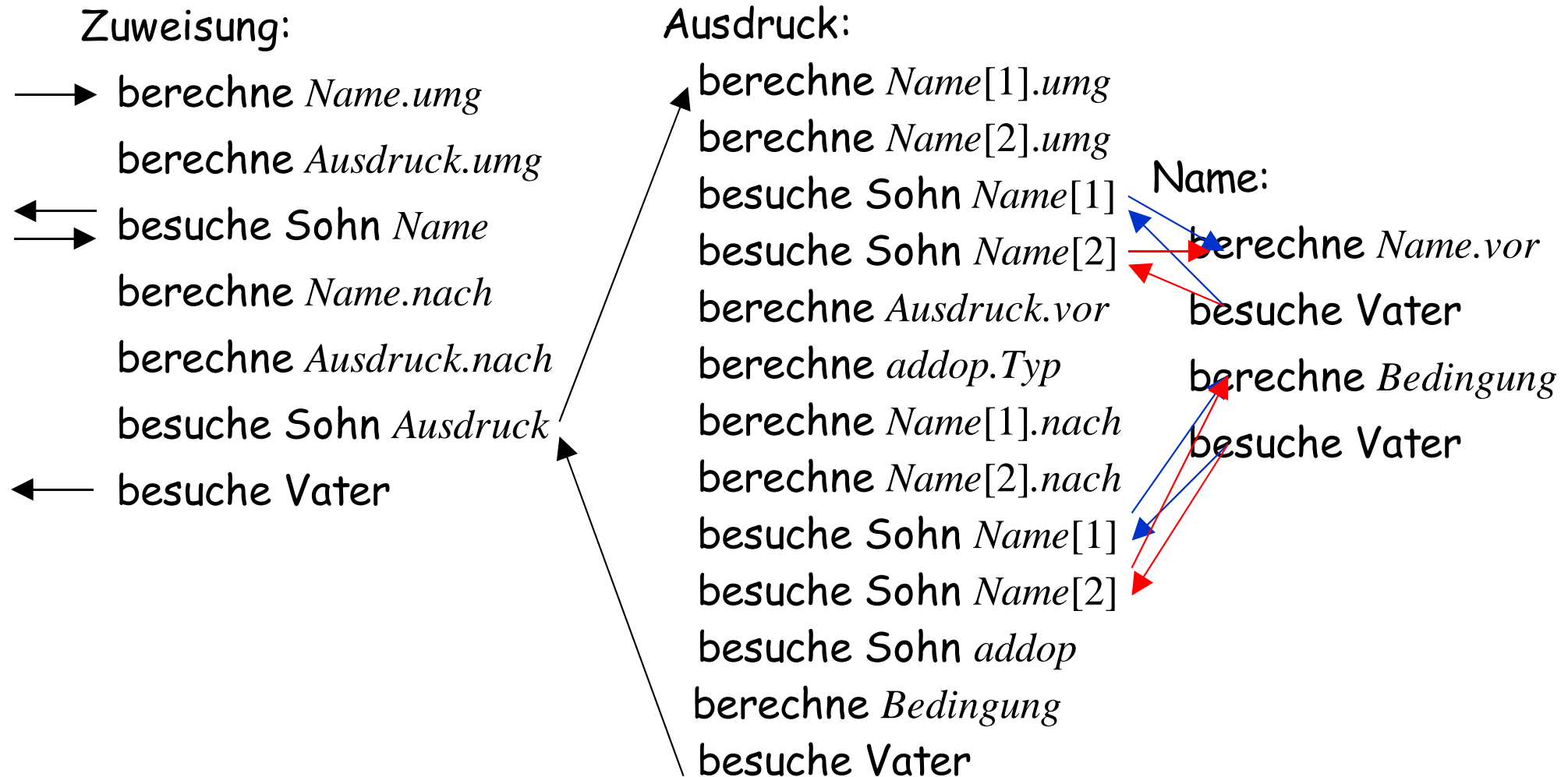
**rule** *Name* ::= *Bezeichner* .

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

**condition** *anpaßbar*(*Name.vor*,*Name.nach*);



# 4.2 Besuchssequenzen als Kommunikationsprotokoll



## 4.2 Induzierte Abhängigkeiten

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

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

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

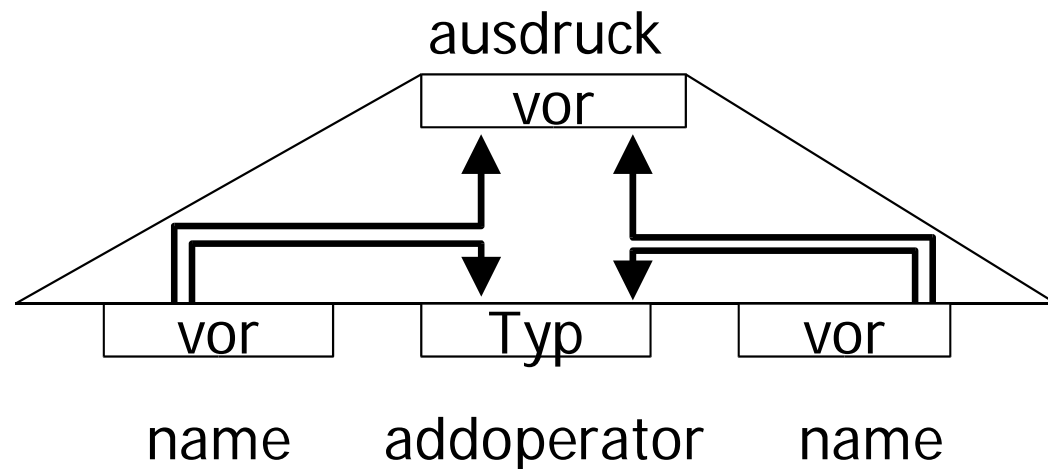
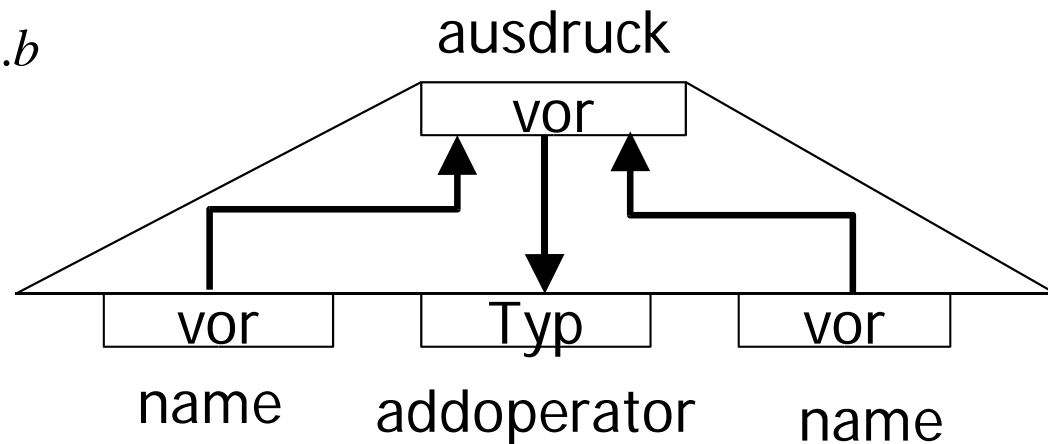
$IDP$  und  $IDS$  heißen **induzierte Abhängigkeiten über Produktionen bzw. Symbole**.

Unterschied  $NDDP(p)$ ,  $DDP(p)$ : Wenn  $X_i.a$  in der Produktion berechnet wird:

$X_i.a := g(\dots) \in R(p)$ , dann ersetze überall die Verwendung von  $X_i.a$  durch  $g(\dots)$  und streiche  $X_i.a$  sowie die Abhängigkeit  $(X_i.a, X_j.b)$ .

## 4.2 NDDP und DDP

- Sei  $X_j.b := f(\dots, X_i.a, \dots) \in R(p)$  und  $X_i.a := g(\dots) \in R(p)$ , .d.h.  $X_i.a, X_j.b \in AF(p)$ . Dann kann man  $X_j.b$  auch mit  $X_j.b := f(\dots, g(\dots), \dots)$  berechnen.
- $NDDP(p)$  beseitigt die Abhängigkeit  $(X_i.a, X_j.b)$  und verlangt diese Substitutionen.



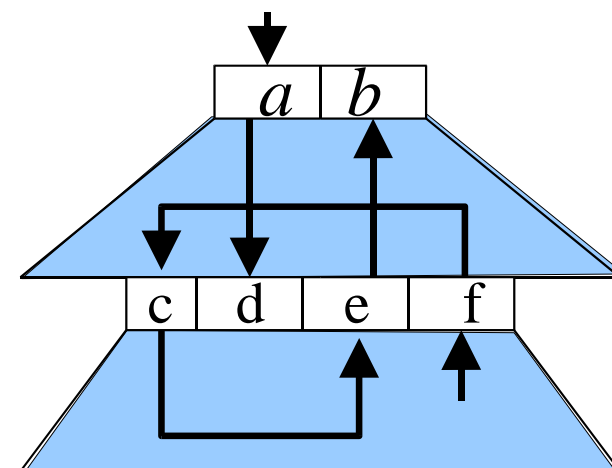
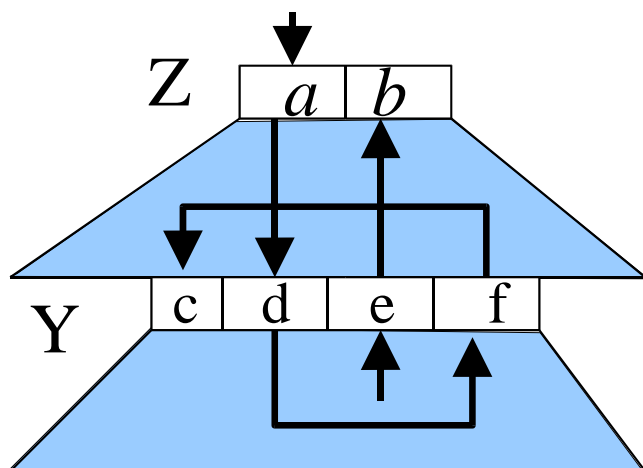
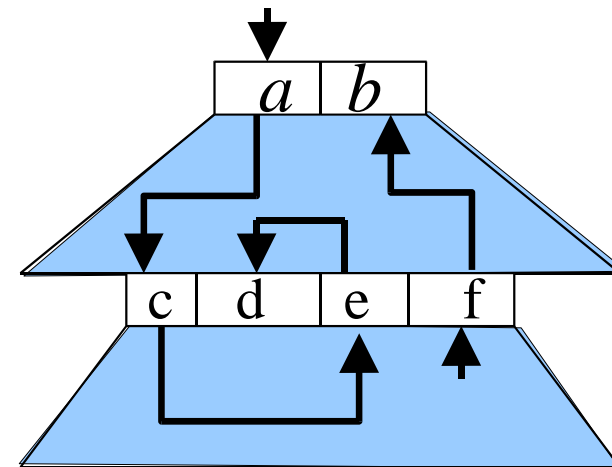
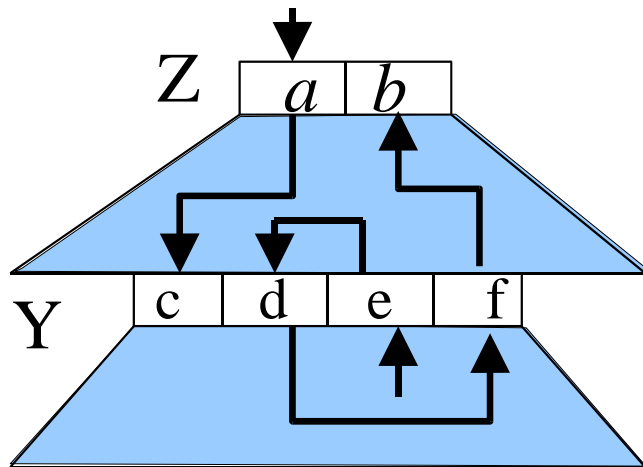
## 4.2 *IDP* und *IDS* intuitiv

- $IDS(X)$  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!

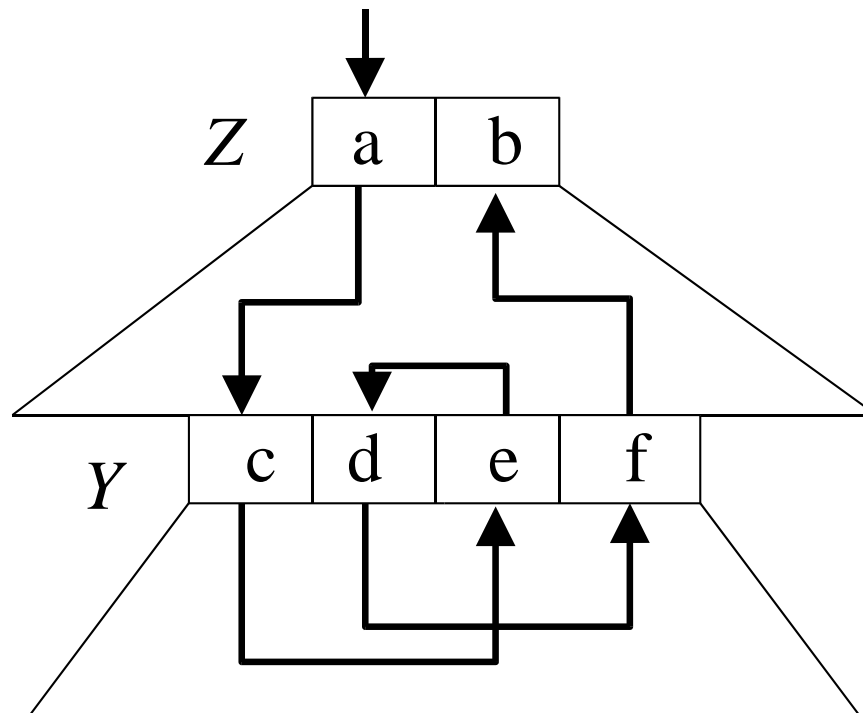
# 4.2 WAG mit 4 Ableitungsbäumen

$S ::= Z$   
 $Z ::= s Y / y Y$   
 $Y ::= u / v$



## 4.2 IDP und IDS

für Z:



$IDS(Z) = \{a \rightarrow b\}$  obwohl diese Abhängigkeit in keinem Ableitungsbaum vorhanden.

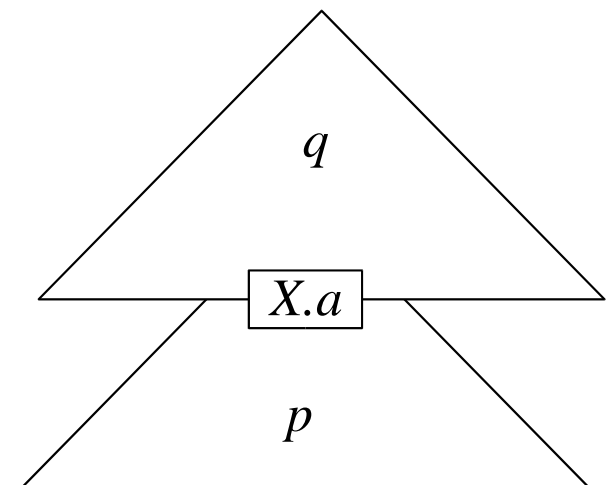
# 4.2 Absolut Azyklische AGs: *ANCAG*

Eine Attributgrammatik heißt **absolut azyklisch** (eine *ANCAG*), wenn

- $IDS(X)$  azyklisch für alle Symbole  $X \in T \cup N$
- $IDP(p)$  azyklisch für alle Produktionen  $p \in P$ .

Bei absolut azyklischen Attributgrammatiken hängt die Besuchssequenz vom Paar  $p, q$  ab, das über ein Nichtterminal  $X$  verbunden ist.

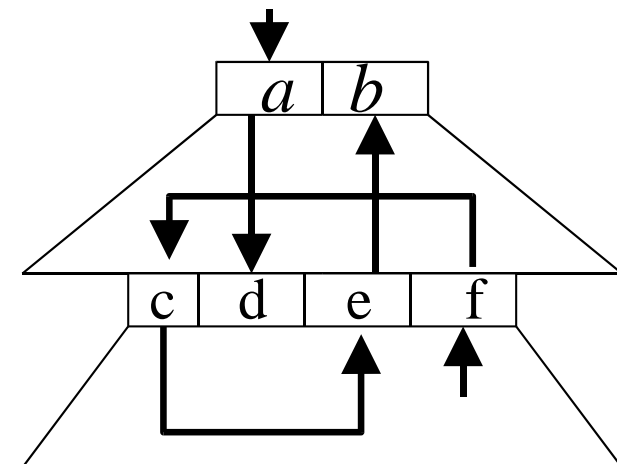
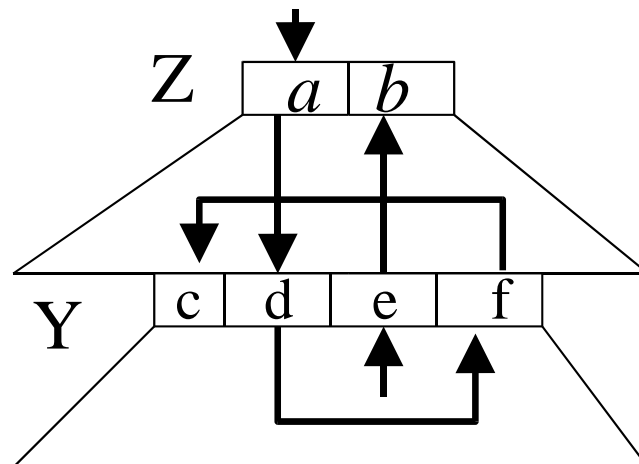
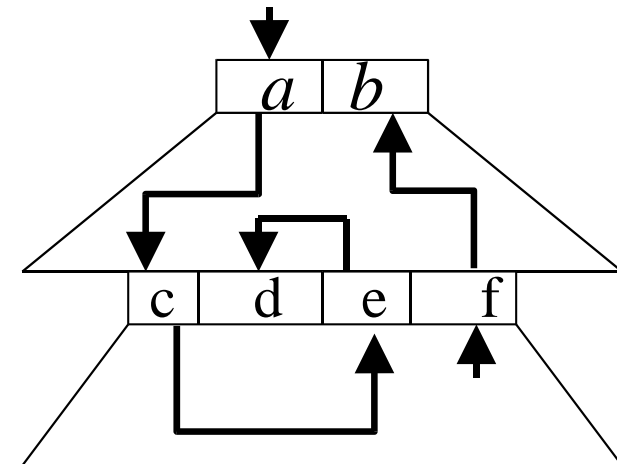
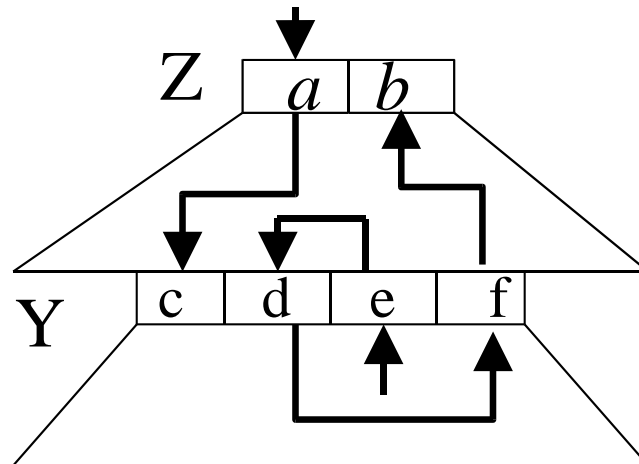
Eigenschaft mit **polynomiell**em Aufwand prüfbar!



# 4.2 Wohldefiniert, aber nicht ANCAG

Hier noch mal unser Beispiel. Obwohl jeder Ableitungsbaum azyklische Abhängigkeiten hat, ist die AG nicht ANCAG. Siehe nächste Folie.

$S ::= Z$   
 $Z ::= s Y / y Y$   
 $Y ::= u / v$

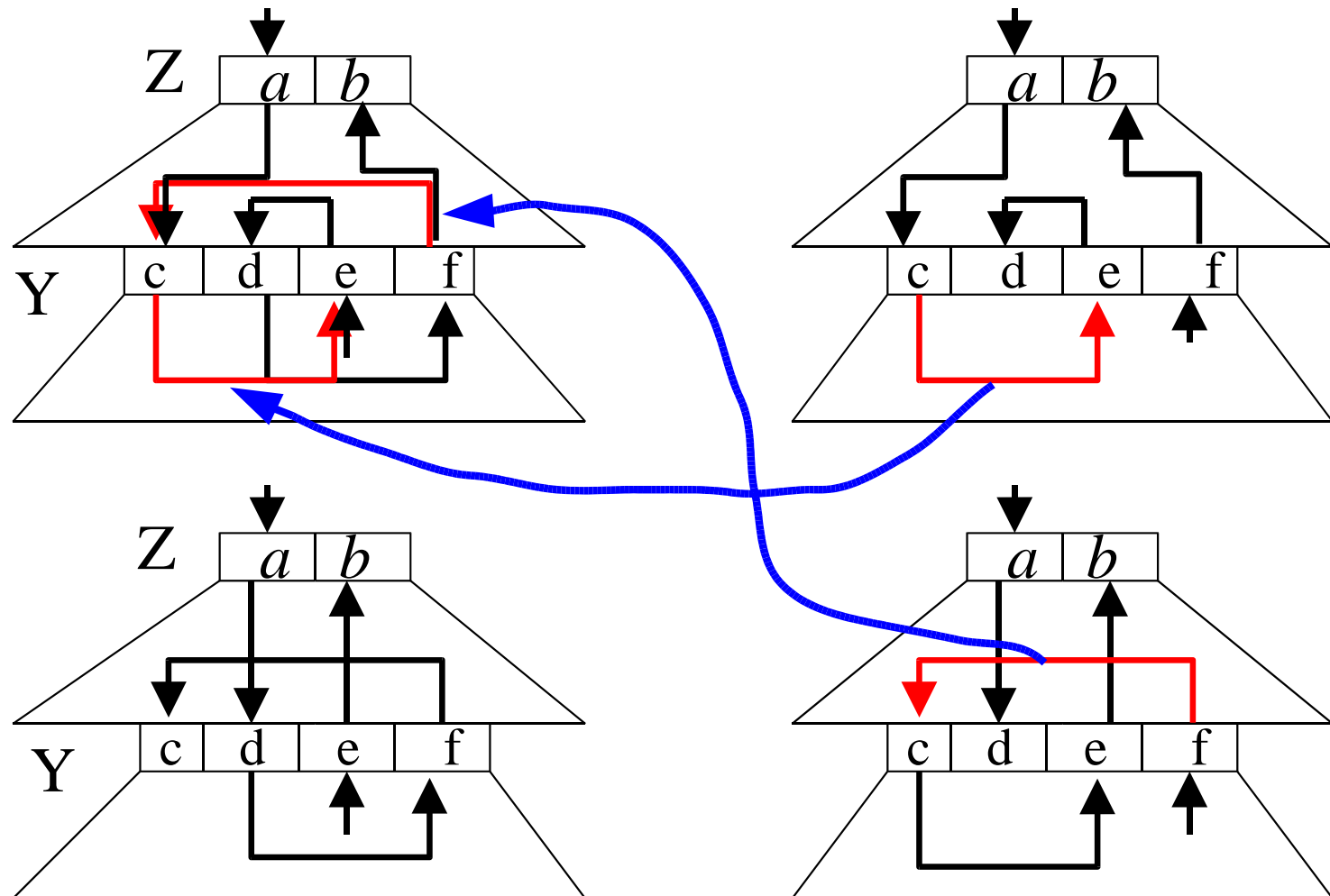




# 4.2 Wohldefiniert, aber nicht ANCAG

$IDS(Y)$  zyklisch, obwohl in keinem Ableitungsbaum eine zyklische Abhängigkeit vorhanden:

$S ::= Z$   
 $Z ::= s Y / y Y$   
 $Y ::= u / v$



# 4.2 Partitionierbare (zerlegbare) AGs: *PAG*

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

- $A_i(X) \in AS(X)$  für  $i = m, m - 2, \dots$ , und
- $A_i(X) \in AI(X)$  für  $i = m - 1, m - 3, \dots$

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!).

**Beachte:**  $m = m(X)$  ist von dem Nichtterminal  $X$  abhängig,  $A_m(X) \subseteq AS(X)$  !

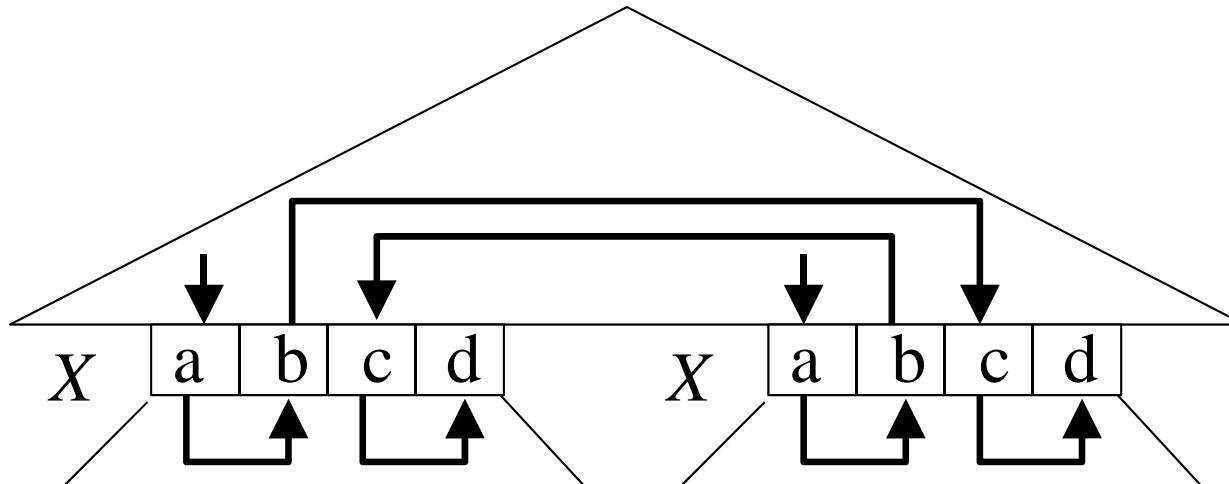
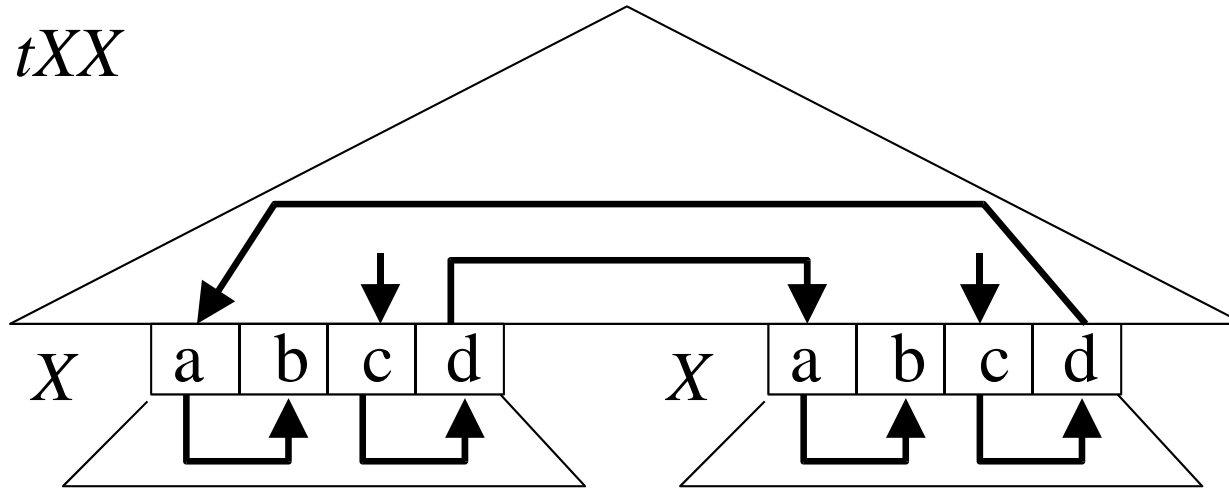
Bei *PAGs* ist die Berechnungsreihenfolge unabhängig vom Strukturbaum:  
Attributauswerter statisch konstruierbar.

**Aber: Prüfung der Eigenschaft *PAG* ist *NP*-vollständig!**

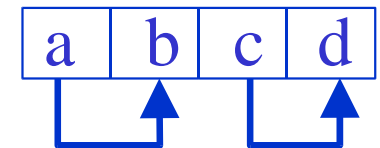
## 4.2 ANCAG, aber nicht PAG

$Z ::= sXX / tXX$

$X ::= u$



$IDS(X)$



aber:

Berechnungs-  
reihenfolge

$(a,b)/(c,d)$

unterschiedlich

## 4.2 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

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

die **Abhängigkeitsrelation über der Produktion  $p$** .

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

# 4.2 Geordnete Attributgrammatiken: OAG

Eine AG heißt **geordnet**, wenn **faule Auswertung** eine zulässige Partitionierung liefert:

Sei  $T_{-1}(X) = T_0(X) = \emptyset$  und für  $k > 0$

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

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

Setze  $A_i(X) = T_{m-i+1}(X) - T_{m-i}(X)$  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.

**Die OAG-Eigenschaft ist mit polynomiellern Aufwand nachprüfbar.**

Konstruktionsprinzip: in  $T_h(X)$  werden alle Attribute  $a$  aufgenommen,

die nur von Attributen  $b \in T_j(X)$ ,  $j \leq h$ , abhängen. Also:

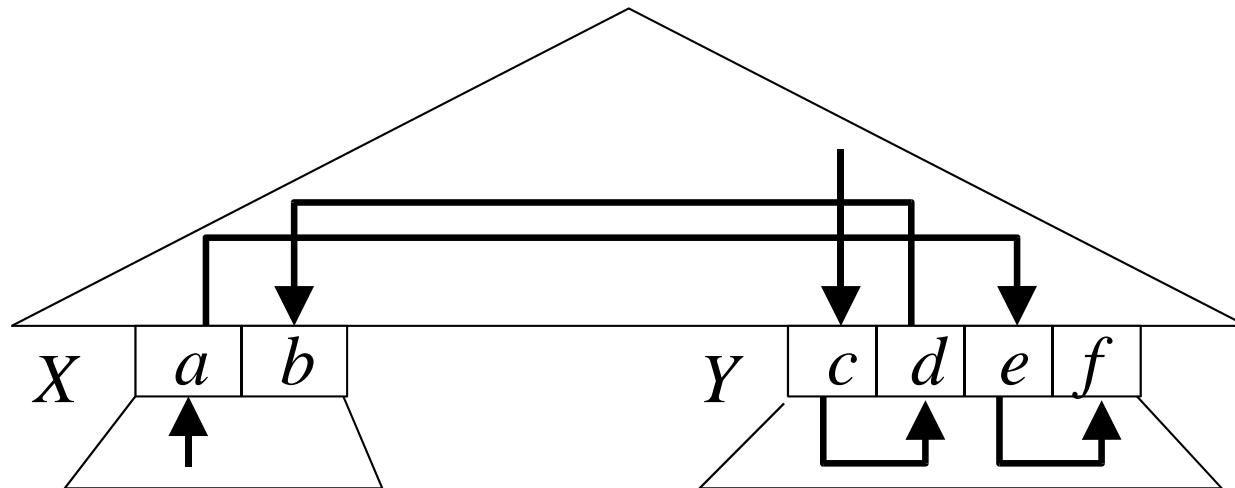
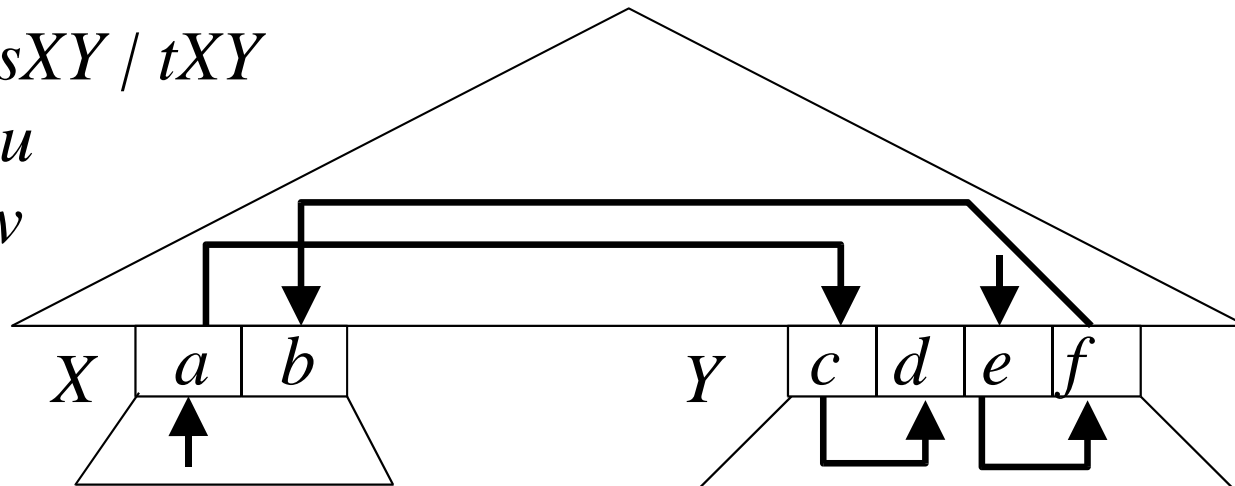
$A_i(X)$  enthält alle Attribute, die zu keinem  $A_j(X)$ ,  $i < j$ , gehören können.

# 4.2 Zerlegbar, aber nicht geordnet

$Z ::= sXY / tXY$

$X ::= u$

$Y ::= v$



Zerlegung:

$A_1(X) = \{a\}, A_2(X) = \{b\}, A_3(X) = \{\}$

$A_1(Y) = \{c, e\}, A_2(Y) = \{d, f\}$

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

## 4.2 Erweiterung $PAG \rightarrow OAG$

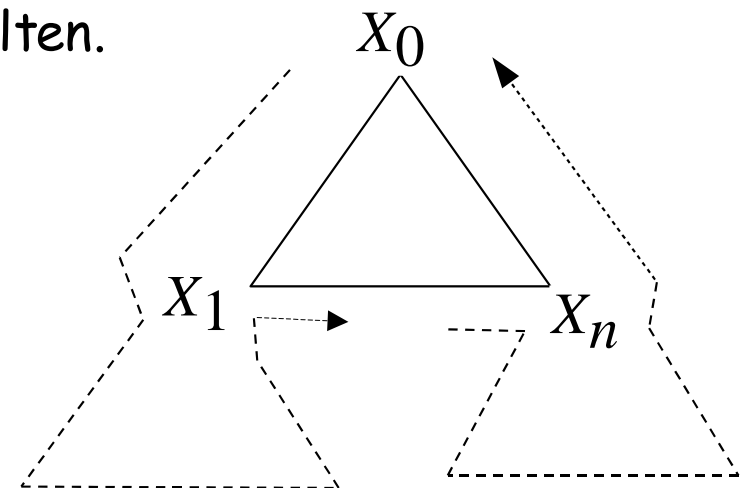
**Satz:** Jede  $PAG$  kann durch Zufügen zusätzlicher Abhängigkeiten zu einer geordneten  $AG$  gemacht werden.

**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.

## 4.2 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  $AI(X_0), AI(X_1), AS(X_1), AI(X_2), \dots, AS(X_n), AS(X_0)$  berechnet werden können.
- Eine LAG(1) ist während der LL-Zerteilung 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.





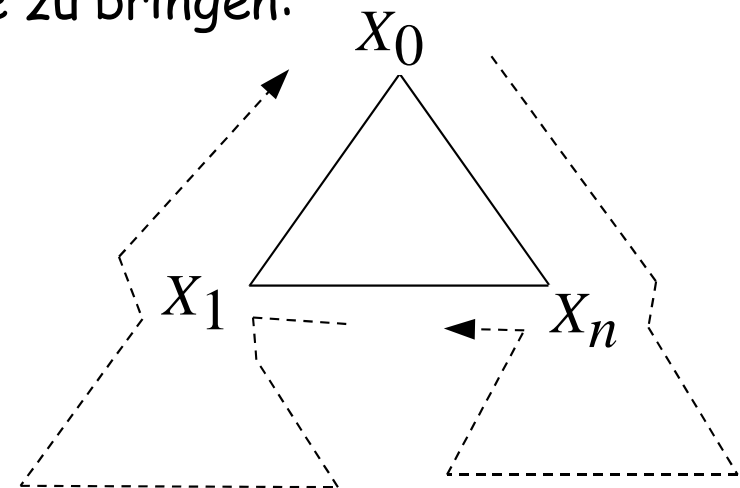
## 4.2 RAG(1) - Attributgrammatik

Eine AG ist **rechts abwärts** berechenbar, eine **RAG(1)**, wenn für jede Produktion  $p : X_0 \rightarrow X_1 \dots X_n \in P$  die Attribute in der Reihenfolge  $AI(X_0), AI(X_n), AS(X_n), AI(X_{n-1}), \dots, AS(X_1), AS(X_0)$  berechnet werden können.

Wichtiges Prinzip, wenn die Daten nicht vollständig im Speicher gehalten werden können.

Methode, um Information von hinten nach vorne zu bringen:

- früher wichtig mit Magnetbändern als Zwischenspeicher



## 4.2 $LAG(k)$ - Attributgrammatik

Eine  $AG$  ist  $LAG(k)$ , wenn für jede Produktion

$p : X_0 \rightarrow X_1 \dots X_n \in P$  die Attribute in Gruppen  $G_1, \dots, G_k$  so 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.

## 4.2 *RAG(k)* - Attributgrammatik

Eine *AG* ist *RAG(k)*, wenn für jede Produktion

$p : X_0 \rightarrow X_1 \dots X_n \in P$  die Attribute in Gruppen  $G_1, \dots, G_k$  so zerlegt werden können, daß die Gruppen nacheinander und die Attribute jeder Gruppe nach dem *RAG(1)*-Schema berechnet können.

# 4.2 Alternierende Attributgrammatiken, $AAG(k)$

Eine  $AG$  ist eine **alternierende  $AG$** , eine  $AAG(k)$ , wenn für jede Produktion  $p : X_0 \rightarrow X_1 \dots X_n \in P$  die Attribute in Gruppen  $G_1, \dots, G_k$  so zerlegt werden können, daß die Attribute jeder Gruppe abwechselnd nach dem  $LAG(1)$ -Schema und dem  $RAG(1)$ -Schema berechnet können, also  $G_1$   $LAG(1)$ ,  $G_2$   $RAG(1)$ , usw.

- früher: vorwärts auf Band schreiben, rückwärts lesen und vorwärts schreiben, usw.
- $AAG(k)$  ist die allgemeinste Form der  $AG(k)$ s. Bei wahlfreiem Zugriff simuliert die Berechnung von mehreren Gruppen von Attributmengen  $AI(X_0), AI(X_n), AS(X_n), AI(X_{n-1}), \dots, AS(X_1), AS(X_0)$  in beliebiger Reihenfolge eine  $AAG(k)$
- $AAG(k)$  ist von Hand programmierbar.

## 4.2 Beispiele

Ein Einpaß-Übersetzer, der an die Syntaxanalyse 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.

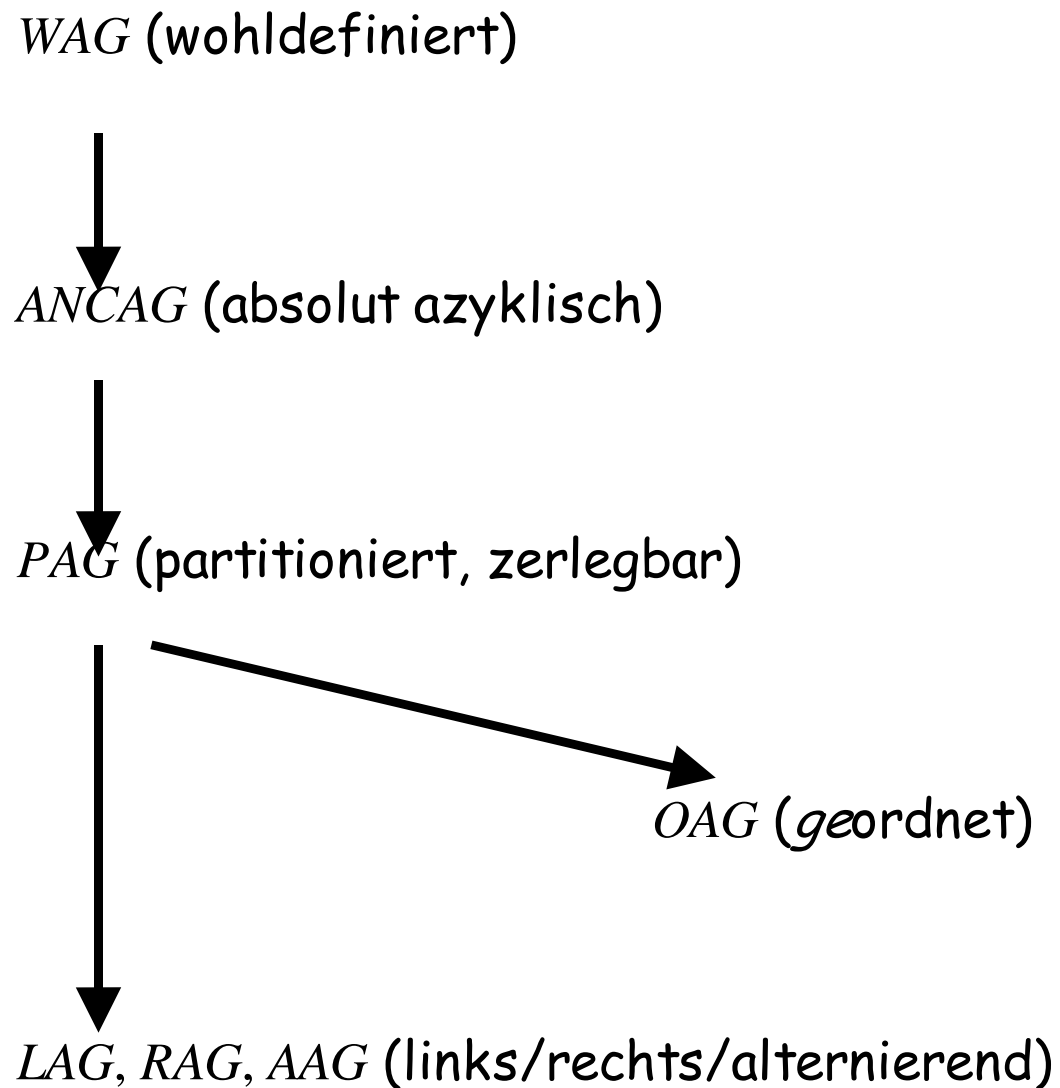
fast alle Sprachen benötigen  $k=2,3$  oder 4. Höheres  $k$  immer nur lokal für einzelne Sprachelemente nötig

- Daher ist eine OAG meist kostengünstiger: sie spart Baumdurchläufe. Sie ist auch systematischer zu entwerfen:

Denken in  $LAG(k)$  oder  $AAG(k)$  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.

## 4.2 AG Hierarchie



# Kapitel 4: AGs

0. Einbettung

1. Grundbegriffe

2. Hierarchie

WAG, ANCAG, PAG, OAG, LAG, RAG, AAG

Besuchssequenzen

Induzierte Abhängigkeiten

3. Implementierung

4. Attributspeicherung

5. Zusammenfassung

## 4.3 Entwurfsprobleme

- AG korrekt entwerfen: systematischer Ansatz mit Begutachtung Spezifikation gegen Sprachbeschreibung nötig.
- AG mit geeigneten Werkzeugen als Prototyp implementierbar
  - Test der AG billiger, Testaufwand Implementierung geringer
- Spezifikation sehr umfangreich (> 100 S. für große Sprachen). Viele Vereinfachungen möglich (hier nicht behandelt)
- In Datenabhängigkeiten, nicht in Ablaufreihenfolgen denken: OAG statt LAG( $k$ ) benutzen.
- Speicheraufwand sehr groß: Zu viele Bytes/Ecken des Strukturbaums. Dazu Anzahl Attribute reduzieren:
  - zuerst die für die Weiterverarbeitung (Transformation, Codeerzeugung) unbedingt nötigen Attribute bestimmen, erst dann die für die semantische Analyse zusätzlich nötigen Attribute.

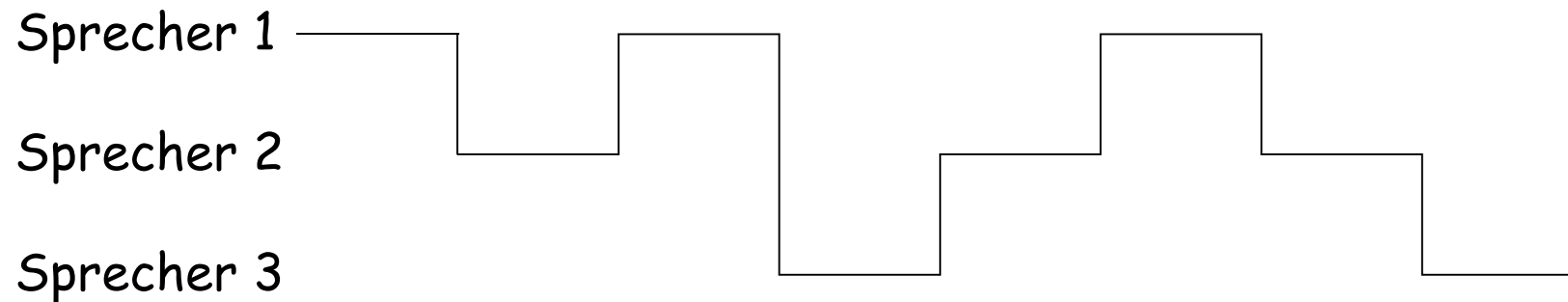


# 4.3 Implementierungsprobleme

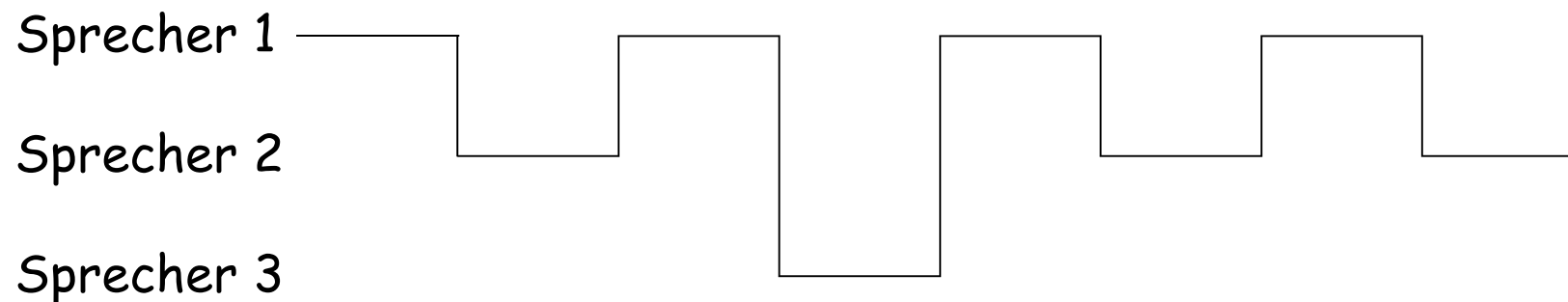
- Interaktion zwischen den Attributierungen verschiedener Produktionen: Implementierung des Besuchsprotokolls durch Koroutineninteraktion
- Speicherprobleme:  
Gegenmaßnahmen in der Implementierung:
  - Attribute möglichst lokal und nicht auf Dauer im Baum speichern. Benutzung von Kellern und globalen Variablen.
  - Kontextinformationen in globale Tabellen statt in den Baum.
  - Kopieren von Attributen zwischen Nachbarecken vermeiden.

## 4.3 Exkurs Koroutinen (1)

Symmetrische Koroutinen (strenges Wechselgespräch):



Asymmetrische Koroutinen (strenges Wechselgespräch mit Dirigent):



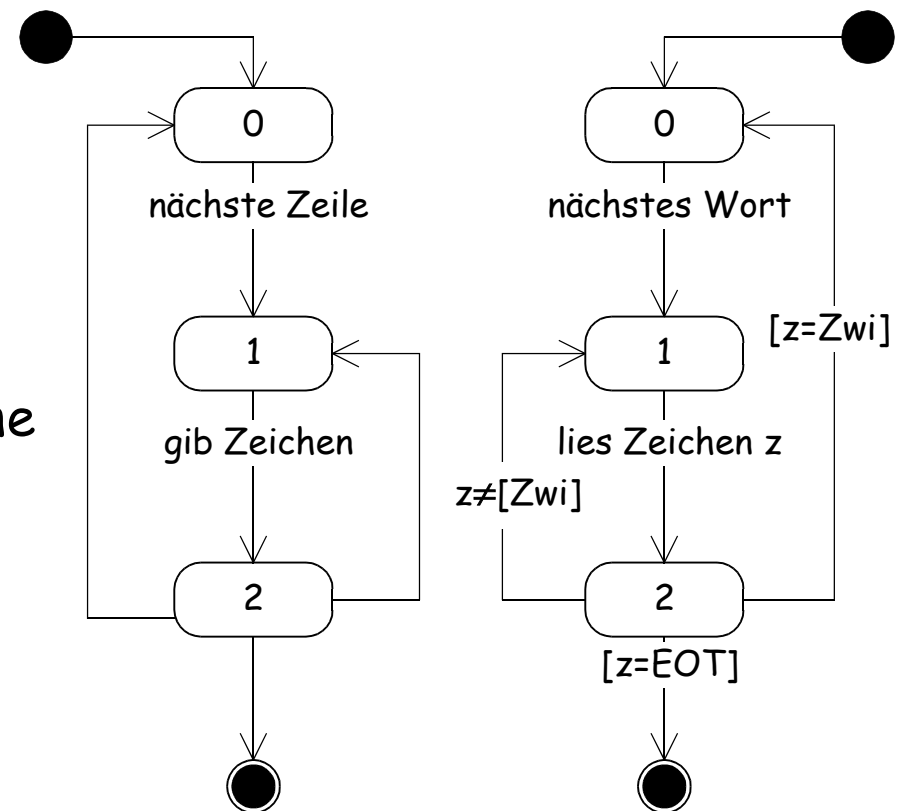
## 4.3 Exkurs Koroutinen (2)

Gegeben: Text mit  $n \geq 0$  Zeilen zu 80 Zeichen;  
die letzte Zeile kann kürzer sein.

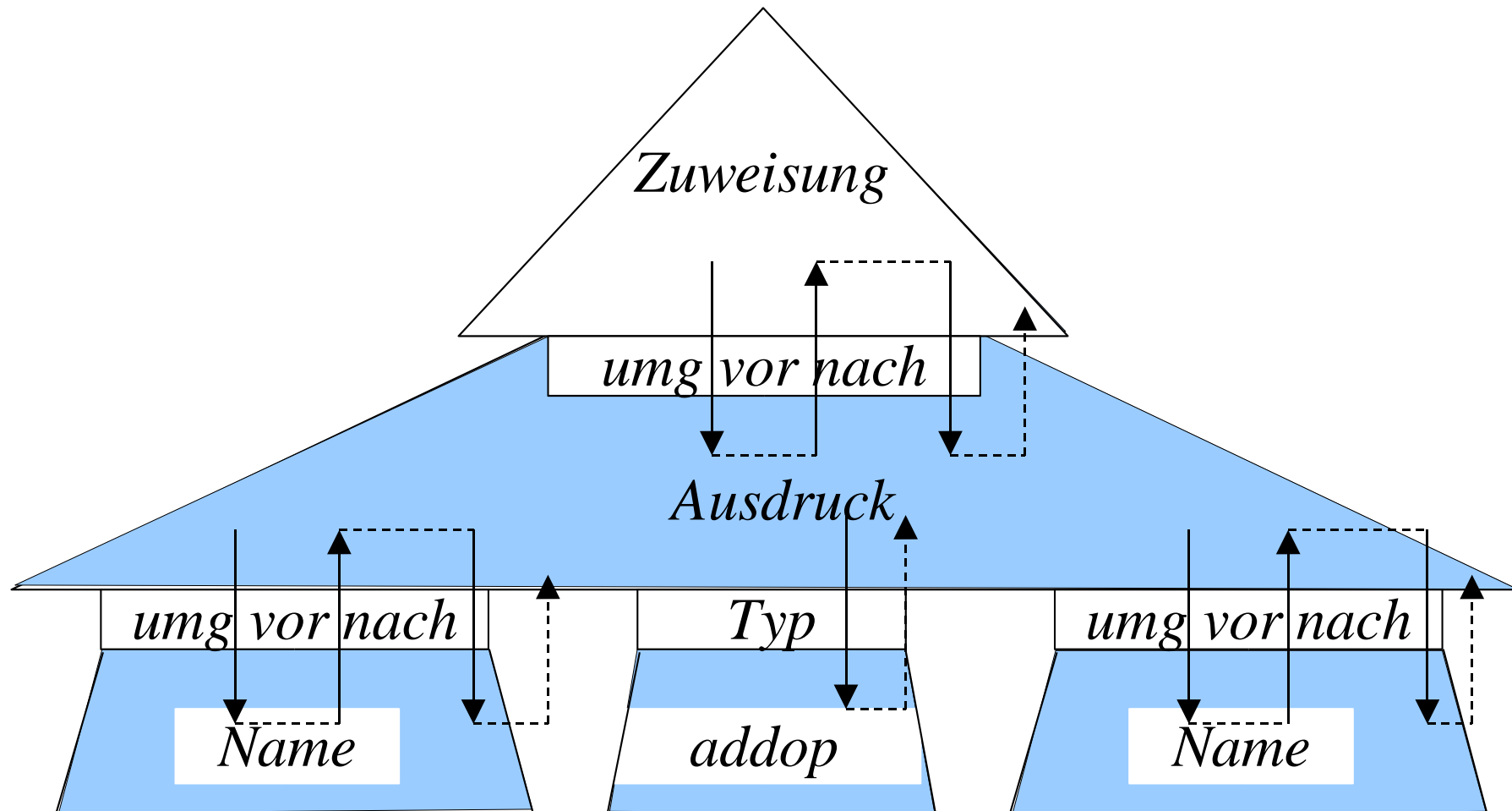
Der Text besteht aus Wörtern,  
die durch einen Zwischenraum  
getrennt sind.

Die Zeilen sind vollgeschrieben; Wörter,  
die nicht vollständig passen, werden ohne  
Trennzeichen auf der nächsten  
Zeile fortgesetzt.

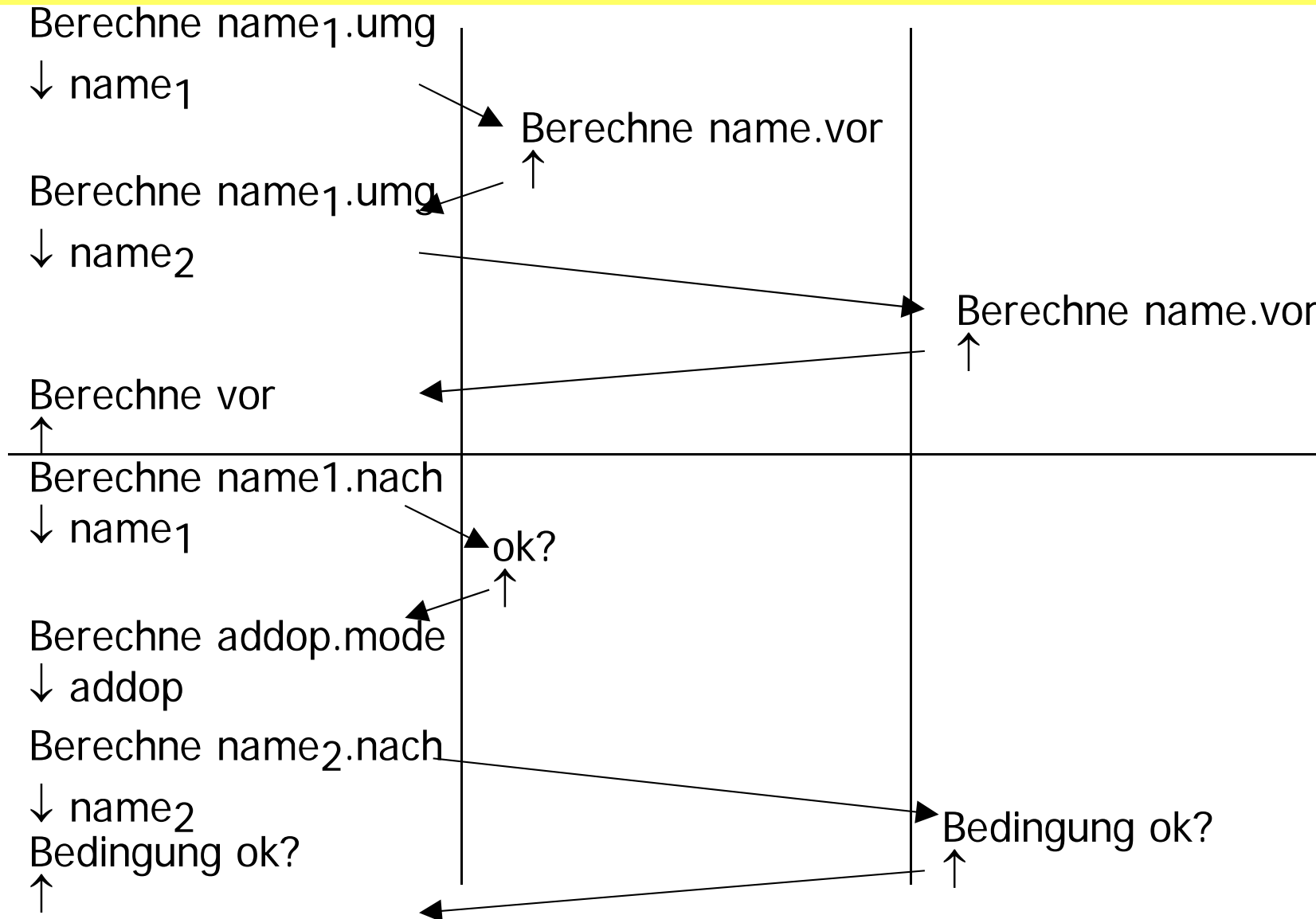
Wir wollen die Anzahl der Wörter zählen.



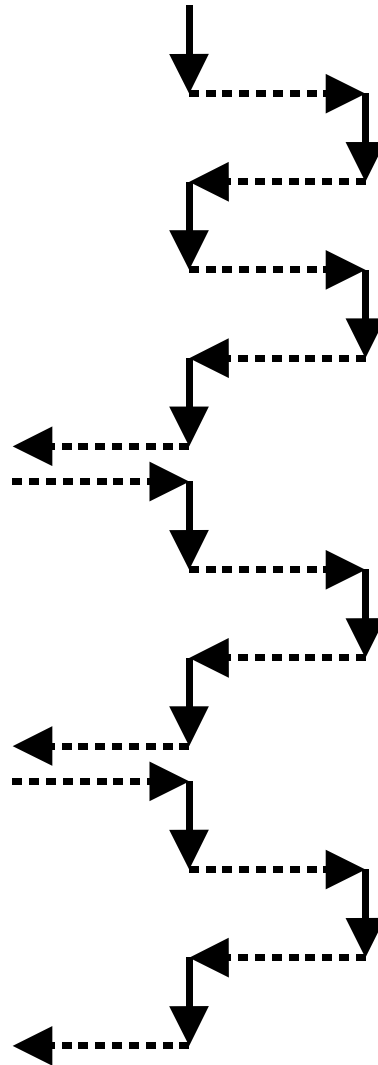
# 4.3 Besuchssequenzen



# 4.3 Besuchssequenzen

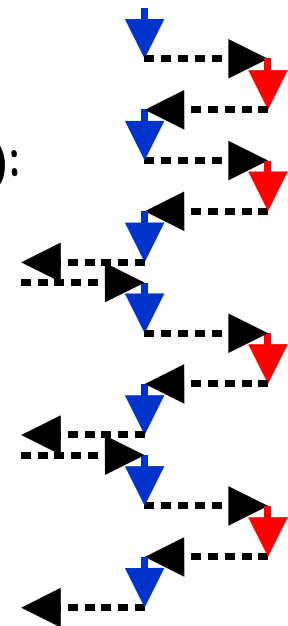


# 4.3 Steuerfluß als Koroutinen



# 4.3 Implementierung von Koroutinen

- Grundverfahren in Simula 67: Koroutineninteraktion durch Aktion *resume*: Unterprogramm sprung auf Rückkehradresse
- Implementierung durch ereignisgesteuerte Fäden (*threads*):
  - Bei *resume* Fortsetzungsereignis an den betroffenen Nachbarfäden schicken.
  - **Nachteil**: In vielen Sprachen nicht möglich, bei weitem zu viele Fäden (pro Produktion im Strukturbaum einer)
- Implementierung durch **Programm inversion** (Michael Jackson):
  - Jede Koroutine an den *resume*-Punkten aufschneiden. Aus jedem Stück eine Prozedur machen (oder eine Prozedur mit Fallunterscheidung)
  - **Programm inversion auch für andere Anwendungen einsetzbar**



## 4.3 Implementierung AST Ausdruck

```
public class Expression extends Thread {
    Name n1, n2;    /*AST Ecken*/
    Addop a;
    Node parent;
    Type vor;      /*Attribute*/
    Postmode nach;
    Environment umg;
    MyThread t=null;
    /*AST Initialisierung*/
    Expression (Name n1, n2, Addop a, Node parent){
        this.N1 = n1;
        this.N1 = n2;
        this.a = a;
        this.parent = parent;
    }
}
```



## 4.3 Implementierung resume

```
/*Dezentrales resume*/

public void resume_computation () {
    if ( MyThread == null ) start();
        /*Erstmaliger Aufruf startet run()*/

    else t.resume();           /*Jeder weitere Aufruf*/

    currentThread.Suspend(); /*legt Aufrufer schlafen*/
}                             /*Aufrufer: n1,n2,a,oder parent*/
```

## 4.3 Implementierung Attributberechnung

```
/*Aktivität des Thread*/
public void run() {
    t=currentThread();           /*Merkt sich die ThreadId*/
    n1.umg=umg;                  /*Berechne n1.umg*/
    n1.resume_computation ();    /*Startet n1 Thread,
                                legt sich selbst schlafen*/
    n2.umg=umg;                  /*etc. */
    n2.resume_computation ();
    vor = bestimme_Grundtyp(n1.vor, n2.vor);
    parent.resume_computation ();
    n1.nach=vor;
    n1.resume_computation ();
    a.Typ=vor;
    a.resume_computation ();
    n2.nach=vor;
    n2.resume_computation ();
    parent.resume_computation ();
}
```

## 4.3 Programminversion

Ersetze `node.resume()` durch `node.evaluate-attributes(K)`.

Ersetze `parent.resume()` durch `return()`.

Für jedes Symbol  $X$

```
public class x extends Node{
node n1, n2, ... , parent; -- AST
Attributes a1, a2, ... ; --Attribute von x
evaluate-attributes(int k){
if (k=0){
    /* Attributauswertung bis zum 1. parent.resume()*/
    return ();
}
else if (k=1){
    /* Attributauswertung bis zum 2. parent.resume()*/
    return ();
}
...
else if (k=n){
    /* Attributauswertung bis zum letzten parent.resume()*/
    return ();
```

```
}}}
```

## 4.3 Anwendung für Ausdrücke

```
public class AddExpression extends Expression {
    Expression e1, e2; Addop a; Node parent; Primode vor; Postmode nach;
    Environment umg;
    public void evaluate-attributes(int k) {
        if (k==0){
            e1.umg=umg;
            e1.evaluate-attributes(0);
            e2.umg=umg;
            e2.evaluate-attributes(0);
            vor = bestimmeGrundtyp(e1.vor, e2.vor);
            return();
        }
        else if (k==1)
            e1.nach=vor;
            e1.evaluate-attributes(1);
            a.mode=vor;
            a.evaluate-attributes(0);
            e2.nach=vor;
            e2.evaluate-attributes(1);
            return ();
    }
}
```



# 4.3 Verallgemeinerung: oo Entwurf

- **Anforderungsanalyse**: alle Objekte im oo-Modell sind aktiv, d.h. selbständige Prozesse
  - nur private, keine öffentlichen Methoden
- **Entwurf**: konstruiere die Prozesse algorithmisch
- **Implementierungsentwurf**: bestimme alle Objekttypen (Klassen), die passiv werden sollen, d.h. nur bei Aufruf aus anderen Objekten aktiv
- **Programminversion**: schneide alle Prozesse passiver Klassen an den Kommunikationsstellen auf, die Stücke ergeben die öffentlichen Methoden
- der ursprüngliche Prozeß enthält ein **Ablaufprotokoll**: Reihenfolge der zulässigen Methodenaufrufe ist vorgeschrieben
  - kontrolliere das Ablaufprotokoll mit geeigneten Datenstrukturen (allgemeinster Fall: Implementierung eines **endlichen Automaten**)
- ersetze alle Kommunikationsoperationen mit passiven Objekten durch **Methodenaufrufe**

# Kapitel 4: AGs

0. Einbettung

1. Grundbegriffe

2. Hierarchie

WAG, ANCAG, PAG, OAG, LAG, RAG, AAG

Besuchssequenzen

Induzierte Abhängigkeiten

3. Implementierung

4. Attributspeicherung

5. Zusammenfassung

# 4.4 Attributspeicherung

- (i) explizit im *AST* - teuer, immer möglich
- (ii) im Prozedurkeller - effizienter als (i) , aber nicht immer möglich
- (iii) in separaten Attributkeller - u.U. effizienter als (ii) , immer möglich wenn (ii) möglich
- (iv) als globale Variable - effizienter als (iii) , aber selten möglich
- (v) durch globale Tabelle: praktisch das wichtigste Verfahren, aber nicht systematisch aus *AG* ableitbar

- (i) *AST*
- (ii) Prozedurkeller
- (iii) Attributkeller
- (iv) globale Variable
- (v) globale Tabelle

Frage: Wie kann man die Attribute für alle *ASTs* einheitlich implementieren?

## 4.4 Explizit im *AST* (i)

Immer nötig, wenn

- Attribut als Ergebnis der Attributierung für folgende Übersetzerphasen benötigt und keine globale Tabelle benutzt
- Speicheranordnung von Objekten

- (i) *AST*
- (ii) Prozedurkeller
- (iii) Attributkeller
- (iv) globale Variable
- (v) globale Tabelle

- Attribut ist Zwischenergebnis, jedoch (ii) - (iv) nicht möglich



## 4.4 LAG (1)

(ii) Immer möglich, weil für jedes Attribut  $X.a$ :

1. Definition einer Instanz von  $X.a$
2. Definition von Instanzen  $X'.a$
3. Letzte Verwendung von Instanzen  $X'.a$  und
4. Letzte Verwendung von Instanz  $X.a$

Dadurch umschließende Lebenszeit von Attributen garantiert.

(iii) Auch immer möglich

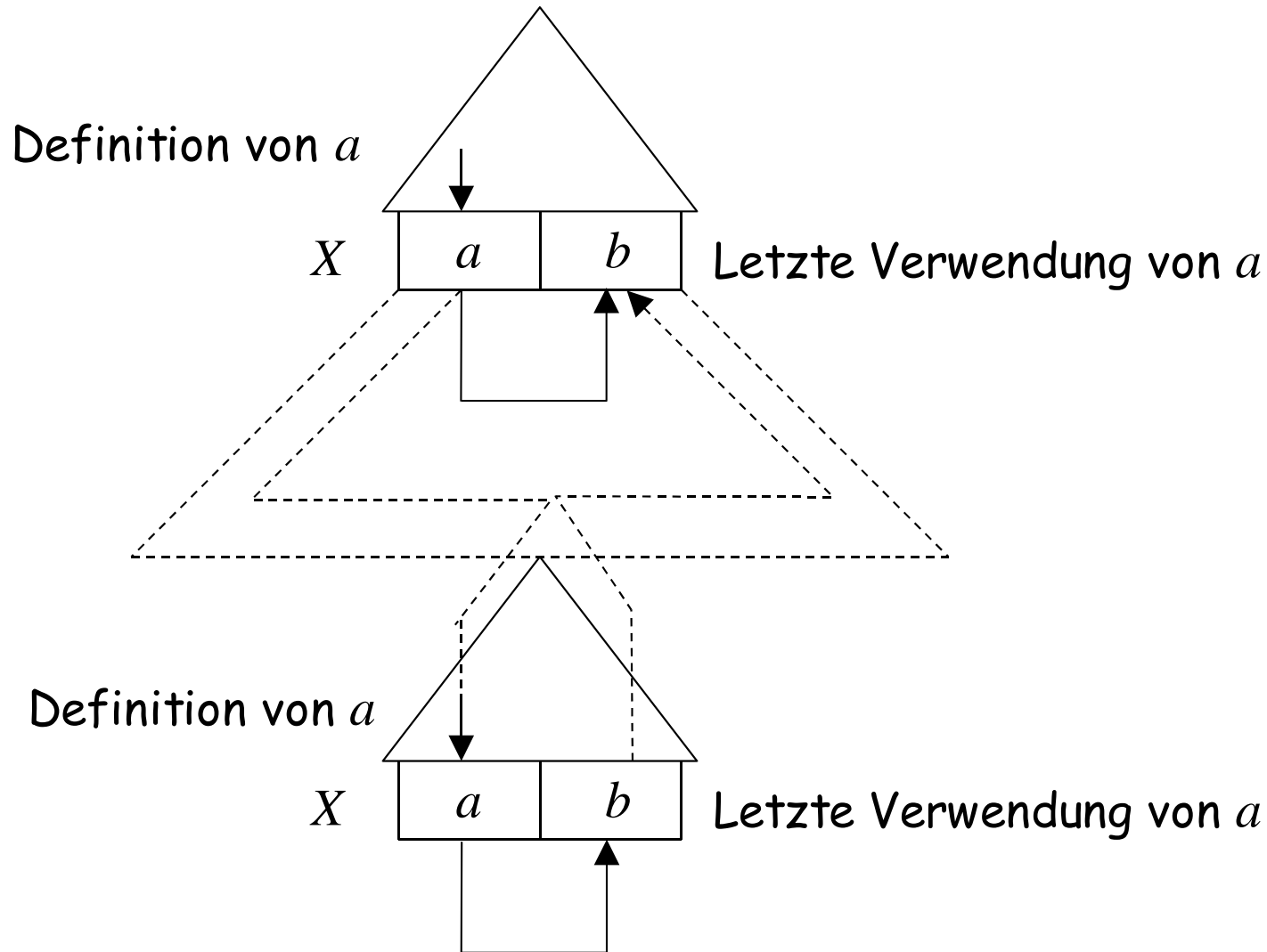
(iv) Nicht immer möglich, weil nicht garantiert werden kann:

1. Definition einer Instanz von  $X.a$
2. Letzte Verwendung von Instanz  $X.a$
3. Definition von Instanz  $X'.a$
4. Letzte Verwendung von Instanz  $X'.a$  und

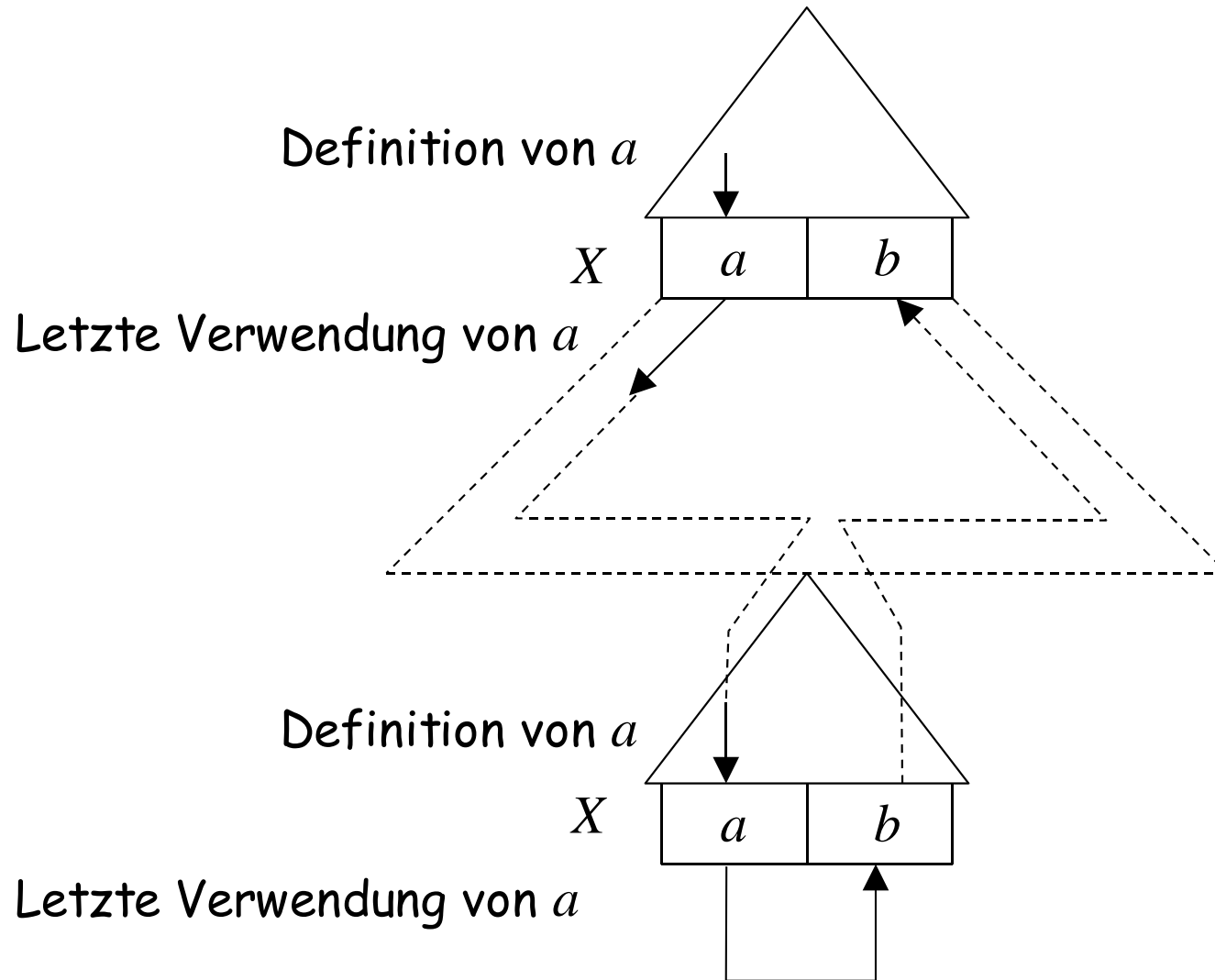
nicht überlappende Lebenszeit von Attributen notwendig.

- (i) AST
- (ii) Prozedurkeller
- (iii) Attributkeller
- (iv) globale Variable
- (v) globale Tabelle

# 4.4 Maximale Lebenszeit von $LAG(1)$ Attributen



# 4.4 Nicht überlappende Lebenszeit

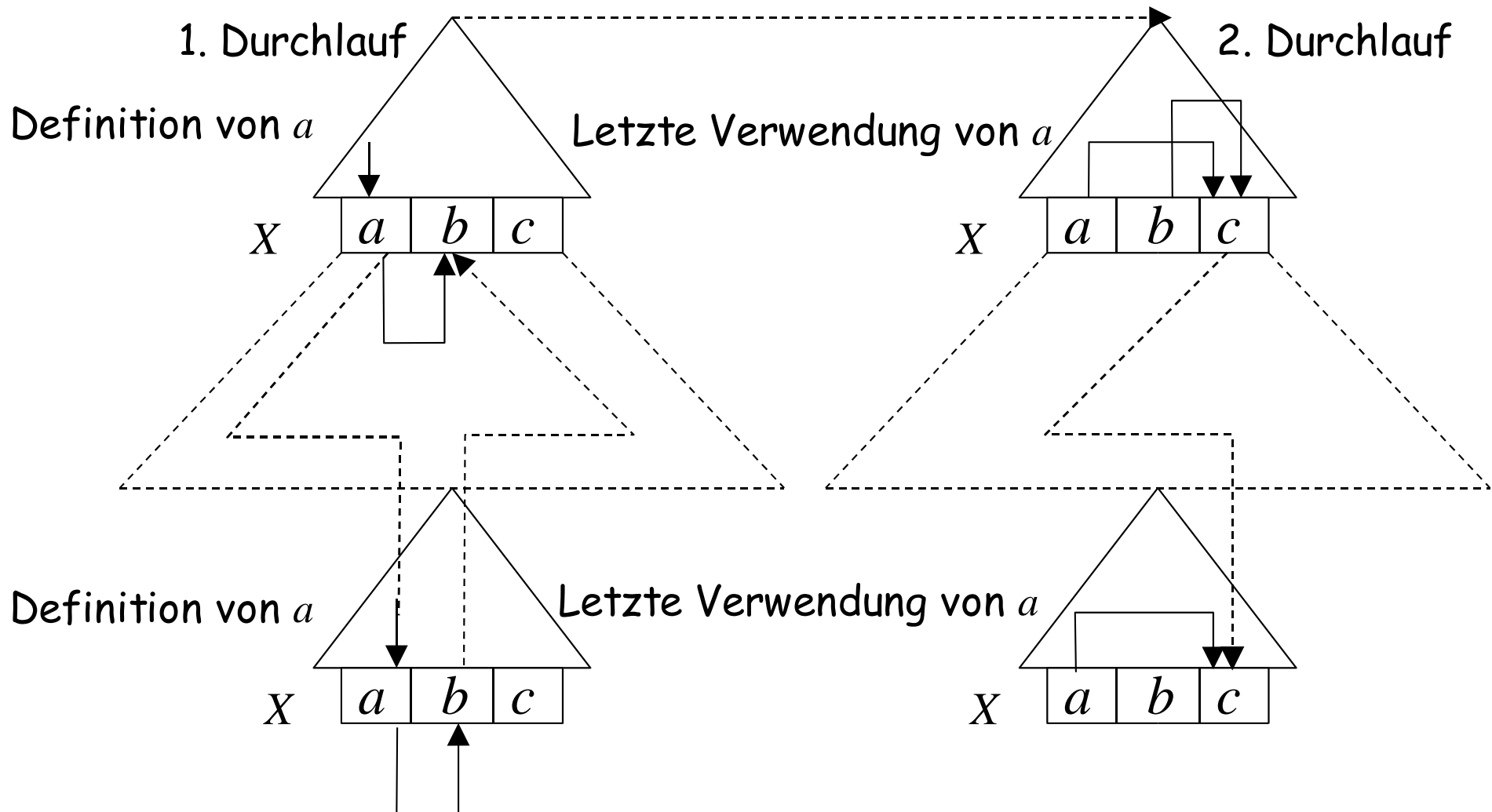


## 4.4 Verallgemeinerung für $LAG(k)$

- (ii) nicht immer möglich, weil für jedes Attribut  $X.a$ : Überlappende aber nicht umschließende Lebenszeit von Attributen möglich.
- Beispiel: Umgebungsattribut ( $umg$ ) bei Sprachen mit Namensverwendung vor Namensbenutzung.
- Einziges Beispiel von praktischer Bedeutung
- Merken von Kellerinhalten: eigene Datenstruktur notwendig.

- (i) AST
- (ii) Prozedurkeller
- (iii) Attributkeller
- (iv) globale Variable
- (v) globale Tabelle

# 4.4 Beispiel *LAG(2)*



# 4.4 Hinreichendes Verfahren für Entscheidung (ii)

1. Bestimme Grammatik  $G_B$  der Besuchssequenzen bezüglich Attribut  $X.a$  mit

- $R$  für return zum Vater
- $D_a$  für Definition  $D_a$  für Attribut  $X.a$  in einer Prozedur evaluate-attribute
- $L_a$  für letzte Verwendung  $L_a$  für Attribut  $X.a$  in evaluate-attribute

2. Stelle fest, ob zwischen  $D_a$  und  $L_a$  kein  $R$  vorhanden.

- (i) AST
- (ii) Prozedurkeller
- (iii) Attributkeller
- (iv) globale Variable
- (v) globale Tabelle

Verallgemeinerbar für *PAG* und *OAG*.

# 4.4 Hinreichendes Verfahren für Entscheidung (iv)

1. Bestimme Grammatik  $G_B$  der Besuchssequenzen bezüglich Attribut  $X. a$
2. Wenn  $L(G) \in (D_a L_a)^*$ , d.h. abwechselnde Definition und Verwendung, dann globale Variable möglich.

- (i) AST
- (ii) Prozedurkeller
- (iii) Attributkeller
- (iv) globale Variable
- (v) globale Tabelle

Verallgemeinerbar für *PAG* und *OAG*.

# 4.4 Grammatik $G_B$ für Besuchssequenzen bzgl. $X.a$

1.  $T = \{D_a, R, L_a\}$
2. Für jedes Nichtterminal  $X$  der Grammatik - Nichtterminale  $X(1) \dots X(k)$  in  $G_B$ .
3. Für jede Besuchssequenz von Produktion  $P: X_0 \rightarrow X_1, \dots, X_n$ , werden  $k$  Produktionen für  $G_B$  generiert.  
(gemäß den Prozeduren evaluate-attribute)
4.  $k$ -te Besuchssequenz von  $P$  sei:  $anweisung_1; \dots; anweisung_m$   
wobei  $anweisung_i \in \{X_i.a_j := f(X_l.a_k \dots), N.call(k), return\}$   
Generiere Produktion  $X_0(k) = p(anweisung_1); \dots; p(anweisung_m)$  wobei:  
 $p(anweisung_i) = \varepsilon$  wenn  $X.a$  nicht beteiligt  
 $p(anweisung_i) = D_a$  wenn  $X.a$  definiert  
 $p(anweisung_i) = L_a$  wenn  $X.a$  verwendet  
 $p(anweisung_i) = X_i(k)$  für  $N.call(k)$  mit Typ  $N = X_i$ .  
 $p(return) = R$



# Kapitel 4: AGs

0. Einbettung

1. Grundbegriffe

2. Hierarchie

WAG, ANCAG, PAG, OAG, LAG, RAG, AAG

Besuchssequenzen

Induzierte Abhängigkeiten

3. Implementierung

4. Attributspeicherung

5. Zusammenfassung

# 4.5 Zusammenfassung AGs

- Attributierte Grammatiken sind allgemeines Hilfsmittel, um Eigenschaften in Bäumen zu berechnen
- AGs sind funktionale Sprachen, Turing-mächtig
- Klassifikation: *WAG*, *ANCAG*, *PAG (OAG)*, *LAG/RAG/AAG*
- Begriff des Übersetzerlaufs
- beim Entwurf: denke in Datenabhängigkeiten, nicht in Berechnungsreihenfolgen
  - benutze *PAGs/OAGs*, keine *LAGs*
- Implementierungsprobleme bei *PAGs*: Implementierung von Koroutinen
- Probleme bei der Attributspeicherung
  
- *OAGs* implementiert im Eli-System (Waite/Kastens/Sloan)  
<http://www.uni-paderborn.de/project-hp/eli.html>