4. Kapitel

SYNTAXGESTEUERTE ÜBERSETZUNG

Compilerbau
Prof. Dr. Wolfgang Schramm

Weitere Arbeitsschritte

Erledigt: Quellprogramm kann lexikalisch und syntaktisch analysiert werden, ein Ableitungsbaum kann erstellt werden.

Zu tun: Semantische Analyse und Erzeugen des Zielprogramms aus dem Ableitungsbaum.

Idee: Syntaxanalyse steuert semantische und andere Aktionen wie Codeerzeugung.

Beispiel Codeerzeugung

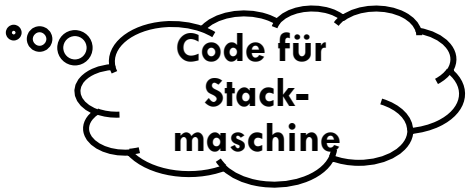
E	→	E + T	T	→	F
E	→	T	F	→	(E)
T	→	T * F	F	→	id

Syntaxanalyse erfolge Bottom-up

Reduktion/Produktion	Aktion/semantische Regel
F → id	erzeuge Befehl LOAD id
T → T * F	erzeuge Befehl MULT
E → E + T	erzeuge Befehl ADD

Für Eingabe (id + id) * id entsteht:

LOAD id
LOAD id
ADD
LOAD id
MULT



Beispiel Semantiküberprüfung

Reduktion/Produktion	Aktion/semantische Regel
F → id	bestimme id.typ aus der Symboltabelle F. typ := id. typ
F → (E)	F. typ := E. typ
T → F	T. typ := F. typ
T → T_1 * F	if (T_1. typ = INTEGER) and (F. typ = INTEGER) then T. typ := INTEGER else T. typ = REAL
E → T	E. typ := T. typ
E → E_1 + T	if (E_1. typ = INTEGER) and (T. typ = INTEGER) then E. typ := INTEGER else E. typ = REAL

Implementierung der Verzahnung von Syntaxanalyse und Aktionen

- Attribute „Wertemengen“

  z.B. Typ

  numerische Werte

  Einträge in die Symboltabelle

  Code

- Operationen (Funktionen) für die Auswertung von Attributen.

Es gibt 2 Arten von Attributen

- zusammengesetzte (synthetisierte Attribute)
- ererbte Attribute

Definition: Attributierte Grammatik

Definition: **Attributierte Grammatik**

Gegeben sei eine kontextfreie Grammatik $G = (N, T, P, S)$.

Eine **attributierte** Grammatik enthält

- für jedes Symbol $X \in (N \cup T)$ eine Menge von Attributen $A(X)$ und
- für jede Produktion $p: X_0 \rightarrow X_1 \ldots X_m$ $(X_i \in N \cup T)$ eine Menge von semantischen Regeln $R(p)$ der Form: $X_i.a := f(X_j.b, ..., X_k.c)$, wobei die X_i etc. i.d.R. vorkommende Symbole darstellen und $X_i.a$ deren Attribute.

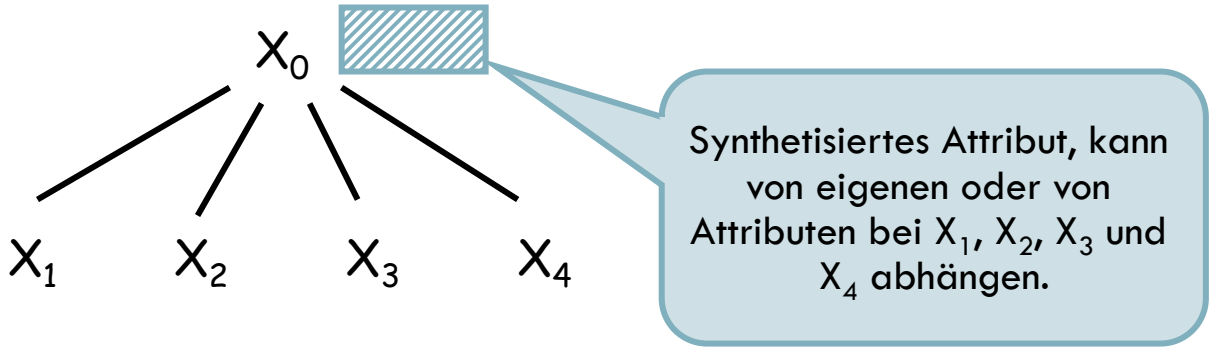
Für jedes Auftreten von X im Ableitungsbaum ist höchstens eine Regel anwendbar, um $X_i.a$ zu berechnen, " $a \in A(X)$.

Syntaxgesteuerte Definition: Die semantischen Regeln dürfen auch Seiteneffekte haben.

Zusammengesetze / Synthetisierte Attribute

- . . . hängen von den Attributen der Kinder ab, d.h. die Attributwerte des Elternknotens können erst dann berechnet werden, wenn die Attributwerte der Kinder bekannt sind.

- Sei p: $X_0 \rightarrow X_1 \ldots X_m$ die Produktion p, zu deren Regelmenge die Regel b := f($c_1 \ldots c_k$) gehört. Das Attribut b heißt synthetisiert, falls es Attribut von X_0 ist und die c_i ($1 \leq i \leq k$) Attribute der X_i ($1 \leq i \leq m$) sind.



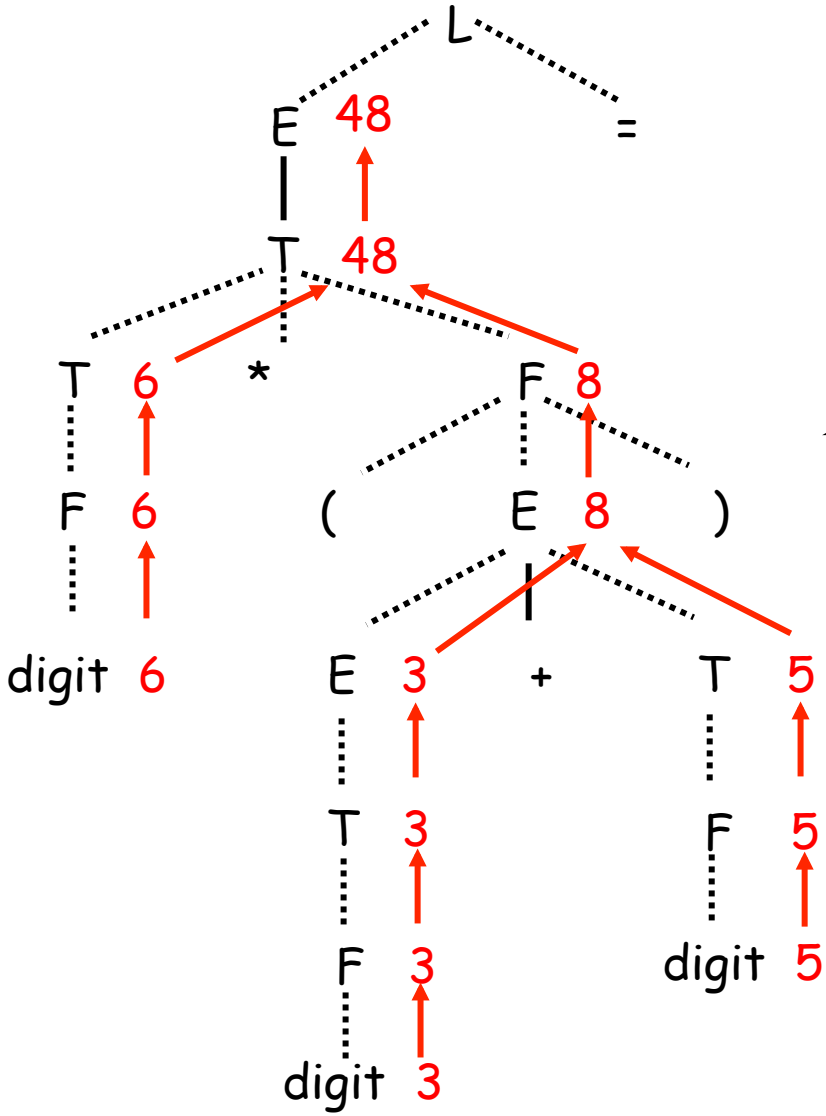
Synthetisierte Attribute – Beispiel 1/2

Taschenrechner für einfache arithmetische Ausdrücke:

Reduktion/Produktion			Aktion/semantische Regel
L	→	E=	output(E.val)
E	→	E_1 + T	E.val := E_1.val + T.val
E	→	T	E.val := T.val
T	→	T_1 * F	T.val := T_1 .val * F.val
T	→	F	T.val := F.val
F	→	(E)	F.val := E.val
F	→	digit	F.val := digit.lexval

S-attributierte Definition, wenn nur synthetisierte Attribute vorkommen

Synthetisierte Attribute – Beispiel 2/2



6 * (3 + 5)

Datenflussgraph

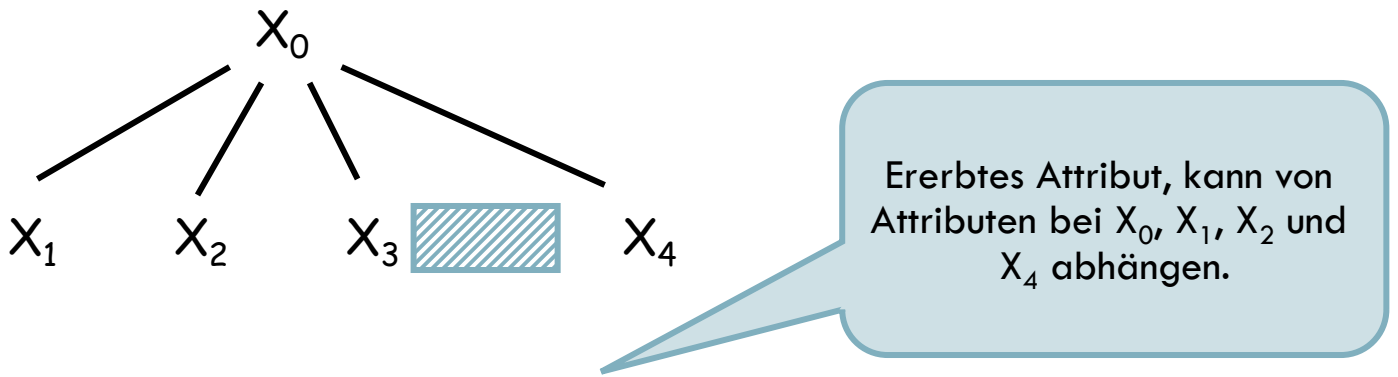
Datenfluss

Ableitungen

Ererbte Attribute

. . . hängen von den Attributen des Elternknotens oder der Geschwisterknoten ab, d.h. die Attributwerte eines Knotens werden nach denen des Elternknotens oder der Geschwisterknoten berechnet.

Sei p: $X_0 \rightarrow X_1 \ldots X_m$ die Produktion p, zu deren Regelmenge die Regel b := $f(c_1 \ldots c_k)$ gehört. Das Attribut b heißt ererbt, falls es Attribut eines der X_i $(1 \leq i \leq m)$ auf der rechten Seite von p ist und die c_i $(1 \leq i \leq k)$ Attribute der X_j $(0 \leq j \leq m, j \neq i)$ sind.



Ererbte Attribute erlauben die Erfassung von kontextsensitiven Konstruktionen in Programmiersprachen

Ererbte Attribute – Beispiel 1/2

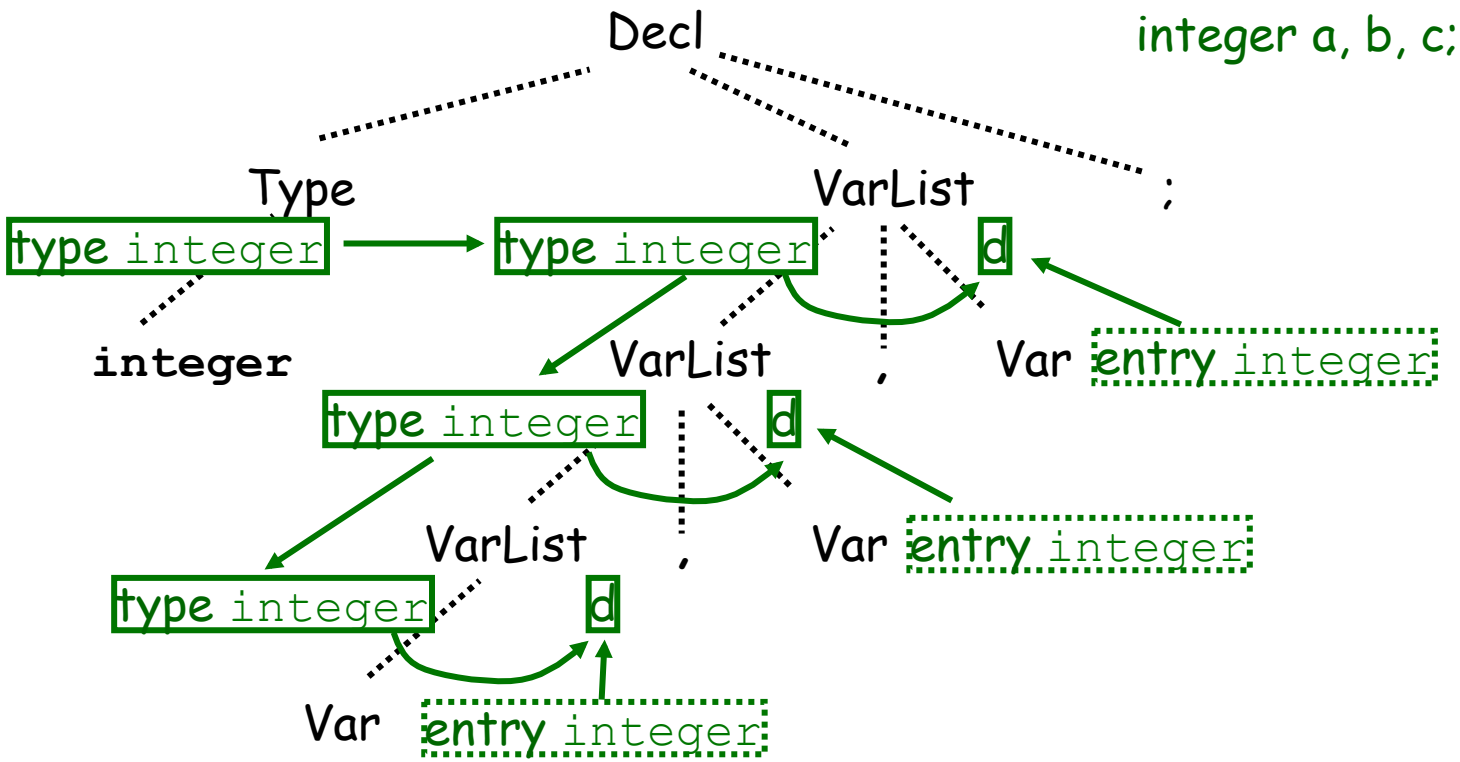
Beispiel: Typdeklarationen

```
integer a, b, c;

real e, f;
```

Produktion			semantische Regel
Decl	→	Type VarList ;	VarList.type := Type.type
Type	→	**integer**	Type.type := integer
Type	→	**real**	Type.type := real
VarList	→	$VarList_1$, Var	$VarList_1$.type := VarList.type addtype(Var.entry, VarList.type) trage in Symboltabelle bei Var als Typ den Wert von VarList.type ein
VarList	→	Var	addtype(Var.entry, VarList.type)

Ererbte Attribute – Beispiel 2/2



Abhängigkeitsgraph

for jeden Knoten n im Ableitungsbaum **do**

  for jedes Attribut a des Grammatiksymbols in n **do**

    erzeuge einen Knoten im Abhängigkeitsgraph für a;

for jeden Knoten n im Ableitungsbaum **do**

  for jede semantische Regel b := f(c_1 . . . c_k), die mit der Produktion in n verknüpft ist **do**

    for i := 1 **to** k **do**

      erzeuge eine Kante vom Knoten für c_i zum Knoten für b;

Abhängigkeitsgraph - Beispiele

- Beispiel: Produktion A → XY
- Semantische Regel: A.a := f(X.x, Y.y)
  - synthetisches Attribut A.a,
  - hängt ab von den Attributen X.x und Y.y
  - es gibt 3 Knoten mit Kanten von X.x → A.a und Y.y → A.a
- Semantische Regel: X.x := g(A.a, Y.y)
  - ererbtes Attribut X.x,
  - hängt ab von den Attributen A.a und Y.y
  - es gibt 3 Knoten mit Kanten von A.a → X.x und Y.y → X.x

Auswertung der Attribute

Wann werden die Attribute ausgewertet?

- Konstruktion des Syntaxbaums und anschließende Ausrechnung.
- Parallel zur Erstellung des Syntaxbaums.

Offen: In welcher Reihenfolge werden die Attribute ausgewertet?

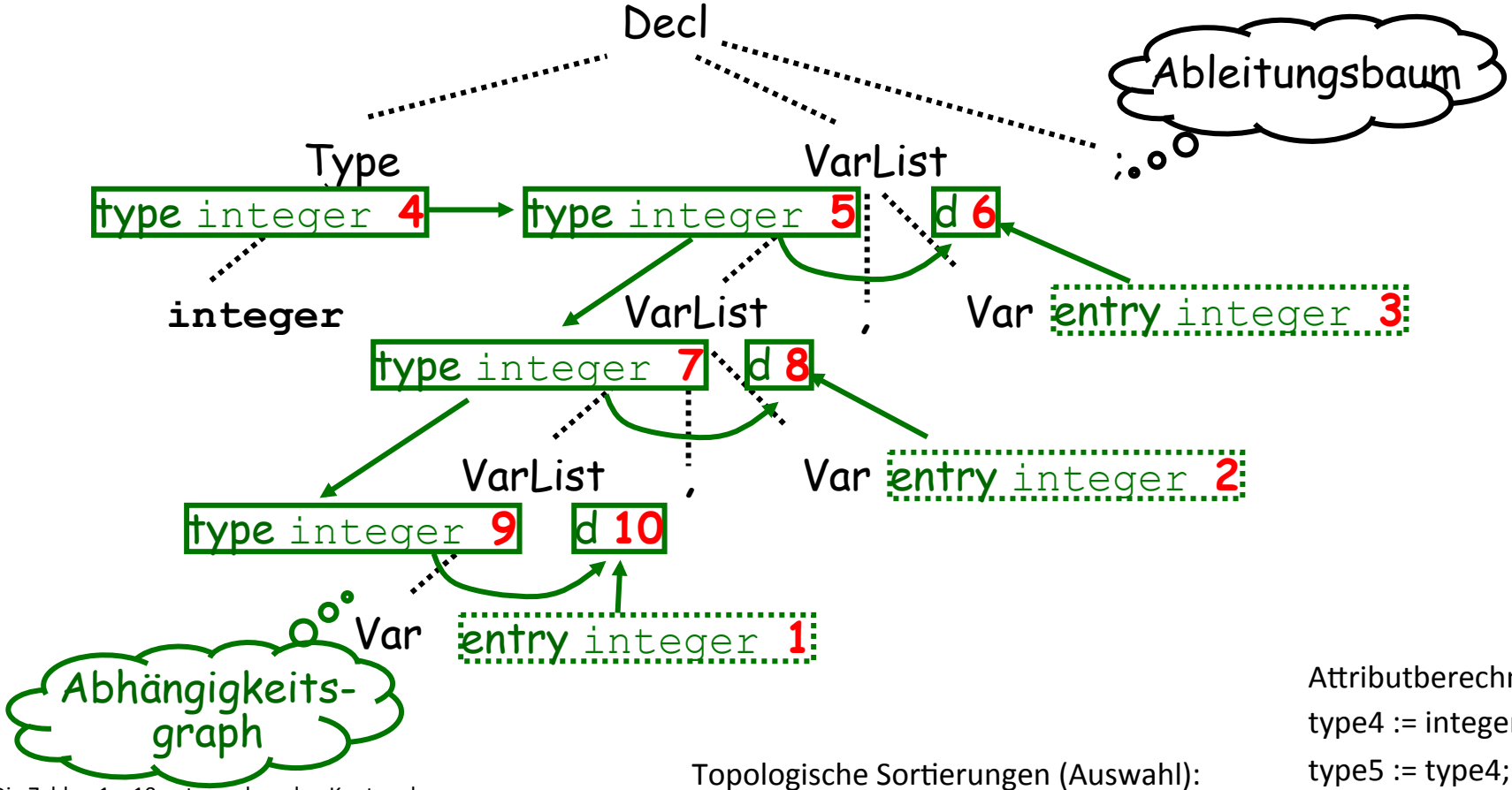
Abhängigkeitsgraph muss so sortiert werden, dass man eine geeignete Reihenfolge für die Auswertung der semantischen Regeln erhält.

➔ Topologisches Sortieren der Knoten des azyklischen Abhängigkeitsgraphen:

Zu komplex für die Praxis

Knoten werden in eine Reihenfolge $n_1, \ldots, n_k$ gebracht, so dass für jede Kanten (n_i , n_j) im Graphen gilt: $i \leq j$.
Damit ist garantiert, dass Knoten mit kleinerer Ordnungszahl vor einem mit einer größeren ausgewertet wird.

Auswertungsreihenfolge der Attribute (Abhängigkeitsgraph) - Beispiel



Die Zahlen 1 – 10 entsprechen den Knoten des Abhängigkeitsgraphen.

1-3: Attribut entry, das mit jedem mit id bezeichneten Blatt verbunden ist.

6, 8, 10: dummy-Attribute → stellen Anwendung der Fkt. addtype auf einen Typ einen einen der entry-Werte dar.

4: Attribut Type.type – hier beginnt die Attributauswertung tatsächlich.

Type.type wird an 5, 7, 9 übergeben und stellen VarList.type dar.

Topologische Sortierungen (Auswahl):

- 1 , 2, 3, 4, 5, 6. 7, 8. 9, 10
- 4, 5, 3, 6, 7, 2, 8, 9, 1, 10
- 4, 5, 7, 9, 1, 2, 3, 10, 8, 6
- 4, 5, 3, 6, 7, 8, 2, 9, 1, 10

Attributberechnung:

type4 := integer;

type5 := type4;

addtype (entry3, type5);

type7 := type5;

addtype (entry2, type7);

type9 := type7;

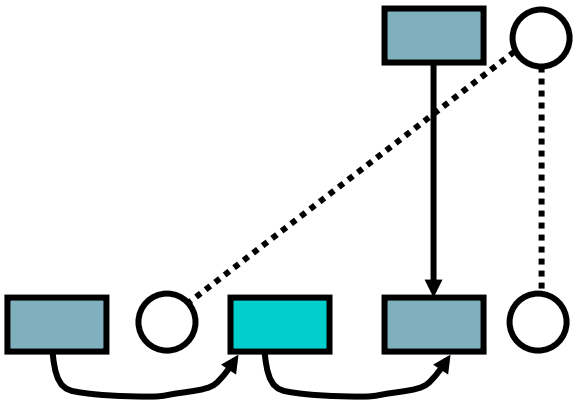
addtype (entry1, type5);

Auswertungsreihenfolge festlegen

Bisher: Syntaxgesteuerte Definition legt keine Reihenfolge bei der Attributauswertung fest.

Jetzt: Reihenfolge festlegen durch

- L-attributierte Definition (eingeschränkte Klasse syntaxgesteuerter Definitionen).
- Übersetzungsschema statt syntaxgesteuerter Definition.



Idee: Attribute von oben-nach-unten und von links-nach-rechts auswerten.

Synthetisierte Attribute (rechts vom Knoten)

Ererbte Attribute (links vom Knoten)

L-Attributierung

Definition: **L-attributierte syntaxgesteuerte Definition**

Eine syntaxgesteuerte Definition heißt L-attributiert, wenn jedes ererbte Attribut eines X_j auf der rechten Seite einer Produktion $X_0 \rightarrow X_1 \ldots X_m$ nur abhängt von

(i) Attributen der Symbole $X_1 \ldots X_{j-1}$ und

(ii) ererbten Attributen von X_0.

L steht für „von links nach rechts".

Die L-Attributierung reicht für die Beschreibung der meisten Konstruktionen in Programmiersprachen aus.

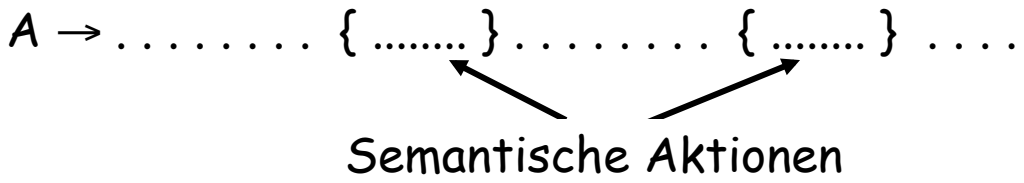
Reihenfolge der L-Attribut-Berechnung


```
procedure attr_eval (n: node) {
    Seien m_1, ....., m_k (von links nach rechts) die Söhne von n.
    for i := 1 to k do
        berechne die ererbten Attribute von m_i;
        attr_eval (m_i);
    od;
    Berechne die synthetisierten Attribute von n;
}
```

Übersetzungsschemata

... bestehen aus

- Kontextfreier Grammatik („Basis“).
- Attributen für Grammatiksymbole.
- Produktionen mit in die rechte Seite eingebetteten semantischen Aktionen.



In .... → . . . . . X { } Y . . . .

wird die semantische Aktion

- nach der Behandlung von X
- vor der Behandlung von Y

durchgeführt.

Übersetzungsschema - Beispiel

E → TR

R → +T {write ("+")} R

| -T {write ("-")} R

| ε

T → **num** {write (**num**.value)}

Wie sieht der Ableitungsbaum für das Übersetzungsschema für 17 + 6 - 9 aus?

Wie kommt man zu einem Übersetzungsschema?

Ausgangspunkt: L-attributierte Definition

- Der Wert eines ererbten Attributs eines Symbols auf der rechten Seite einer Produktion muss in einer semantischen Regel vor diesem Symbol zugewiesen werden.
- Der Wert eines synthetisierten Attributs wird in einer semantischen Regel zugewiesen, die am Ende der rechten Seite steht.

Beispiel:

Decl → Type {VarList.type := Type.type} VarList ;

Type → **integer** {Type.type := integer}

Type → **real** {Type.type := real}

VarList → {$VarList_1$.type := VarList.type} $VarList_1$,
Var {addtype(Var.entry,VarList.type)}

VarList → Var {addtype(Var.entry, VarList.type)}

Top-Down-Übersetzungsschema

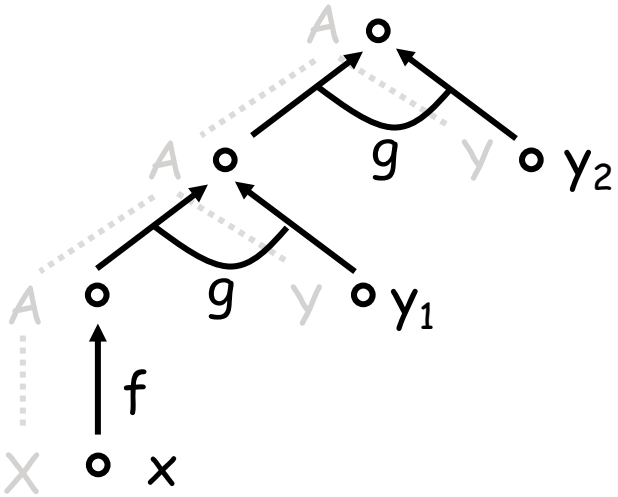
Ausgangspunkt (Grammatik mit Linksrekursion):

$A \rightarrow A\alpha \mid \beta$ $\qquad$ $A \rightarrow \beta A'$ und $A' \rightarrow \alpha A' \mid \varepsilon$

Übersetzungsschema:

$A \rightarrow A_1 Y \{A.a := g(A_1.a, Y.y)\}$

$\quad \mid X \{A.a := f(X.x)\}$



Problem: Linksrekursion

Ableitungsbaum ohne Linksrekursion

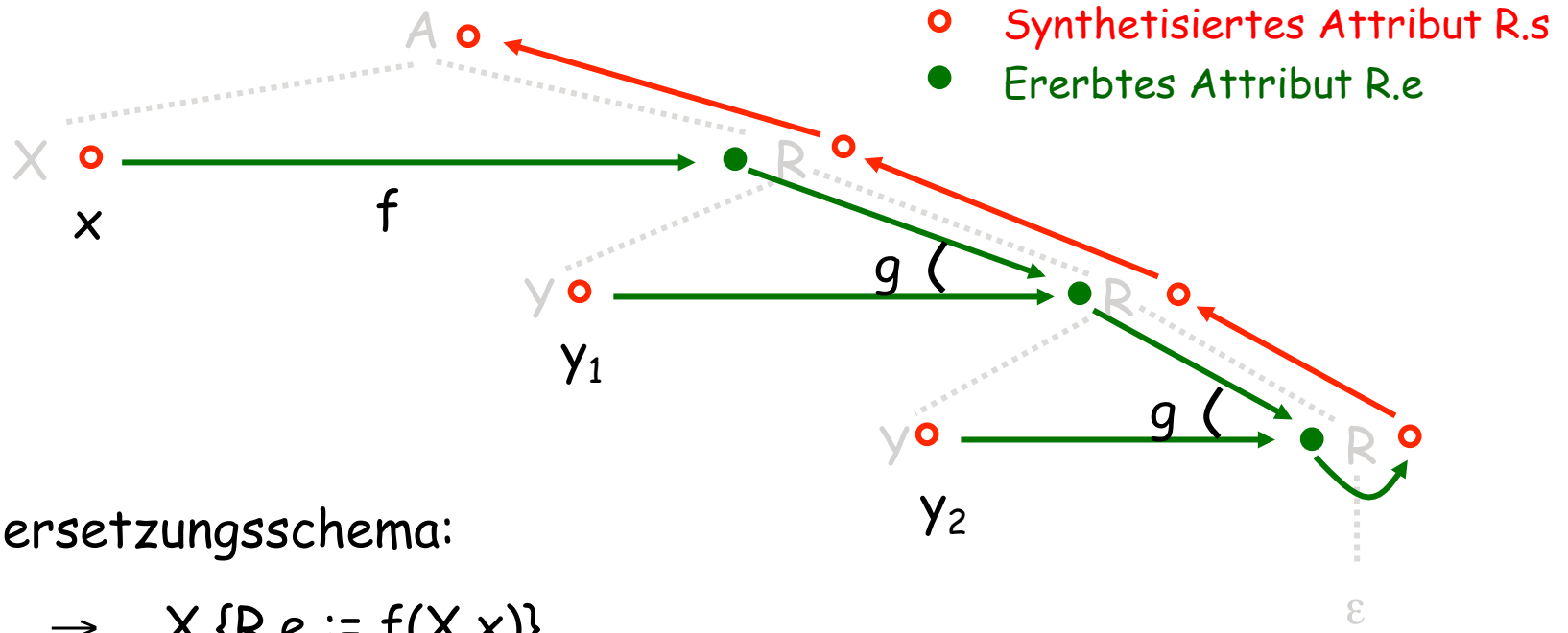
$A \rightarrow XR$

$R \rightarrow YR \mid \varepsilon$

Frage: Wie sieht jetzt das neue Übersetzungsschema aus?

Idee: Das neue Symbol R erhält 2 Attribute – ein ererbtes und (R.e) ein synthetisiertes (R.s).

Datenfluss nach Beseitigung der Linksrekursion



Übersetzungsschema:

A	→	X {R.e := f(X.x)}
		R {A.a := R.s}
R	→	Y {R_1.e := g(R.e, Y.y)}
		R_1 {R.s := R_1.s}
R	→	ε {R.s := R.e)}

Top-Down-Übersetzungsschema - Beispiel 1/2

Ausgangspunkt (Grammatik mit Linksrekursion):

E	→	E_1 + T	{E.val := E_1.val + T.val}
	→	T	{E.val := T.val}
T	→	T_1 * F	{T.val := T_1 .val * F.val}
	→	F	{T.val := F.val}
F	→	(E)	{F.val := E.val}
	→	digit	{F.val := digit.val}

E	→	TR
R	→	+TR | ε
T	→	FS
S	→	* FS | ε
F	→	(E) | digit

Weiter mit Grammatik ohne Linksrekursion

Top-Down-Übersetzungsschema - Beispiel 2/2

E	→	T	{R.e := T.val}
		R	{E.val := R.s}
R	→	+	
		T	{R_1.e := R.e + T.val}
		R_1	{R.s := R_1.s}
R	→	ε	{R.s := R.e}
T	→	F	{S.e := F.val}
		S	{T.val := S.s}
S	→	*	
		F	{S_1.e := S.e * F.val}
		S_1	{S.s := S_1.s}
S	→	ε	{S.s := S.e}
F	→	(	
		E	
		)	{F.val := E.val}
	→	digit	{F.val := digit.val}

Erzeugung von LL(1) Übersetzungsschemata 1/5

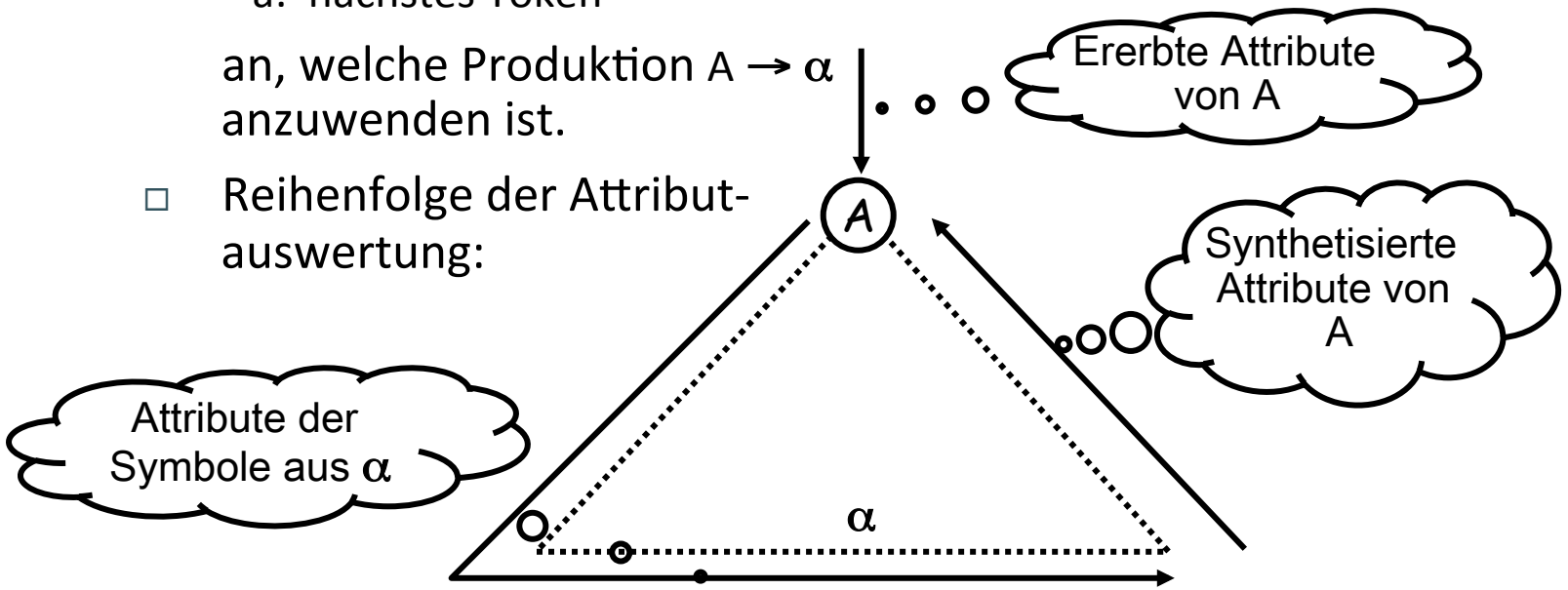
Erzeugung eines syntaxgesteuerten Übersetzungsschemas auf der Basis einer LL(1)-Grammatik.

- Erweitere die Methode des rekursiven Abstiegs um semantische Aktionen für die Auswertung von Attributen.
- Die LL(1)-Steuertabelle gibt in jeder Situation

  A: NTS

  a: nächstes Token

  an, welche Produktion $A \rightarrow \alpha$ anzuwenden ist.
- Reihenfolge der Attribut-auswertung:



Erzeugung von LL(1) Übersetzungsschemata 2/5

Aussehen von $A \rightarrow \alpha$ mit $\alpha = \alpha_1 \alpha_2 \ldots \alpha_n$

$$A \rightarrow \begin{array}{l} \alpha_1 \\ \alpha_2 \\ \ldots \\ \alpha_n \end{array}$$

Dabei sind die α_i die Elemente der rechten Seiten der Übersetzungs-schemata.

- Erzeuge für jedes NTS A eine Funktion des Musters:

function A (Parameter für die ererbten Attribute von A): synthetisierte Attribute von A

Lokale Variable für jedes Attribut, das einem Symbol der rechten Seite α einer Produktion $A \rightarrow \alpha$ zugeordnet ist.

Erzeugung von LL(1) Übersetzungsschemata 3/5

- Funktionsrumpf:

  Bestimme die anzuwendende Produktion $A \rightarrow \alpha$, wobei für jedes der α_i, $1 \leq i \leq n$, drei Fälle möglich sind:

  (i) α_i, ist TS
  (ii) α_i, ist NTS
  (iii) α_i, ist T semantische Aktion { .... }

  Setze Rumpf nach dem Muster

  Behandlung von α_1;
  Behandlung von α_2;
  . . . .
  Behandlung von α_n;

  zusammen.

Erzeugung von LL(1) Übersetzungsschemata 4/5

- α_i ist TS X

TS besitzen nur synthetisierte Attribute, die vom Scanner geliefert werden.

Aktionen:
if TS X mit dem momentan betrachteten Token (look-ahead) übereinstimmt **then**
    speichere die Attribute von X in den dafür vorgesehenen lokalen Variablen der Funktion für A (X.x := symbol.x);
    Beschaffe nächstes Token (match (x));
else beende die Funktion mit einer Fehlermeldung;

Erzeugung von LL(1) Übersetzungsschemata 5/5

- α_i ist NTS X

  Aktionen:
  Berechne die synthetisierten Attribute Xs von X durch die Zuweisung:
  Xs := X(Xe_1, ..., Xe_n)

- α_i ist semantische Aktion b := f (c_1, . . ., c_k)

  Aktionen:
  Kopiere die Aktionen in den Programmtext ein, wobei jedes Auftreten eines Attributs durch den Namen der entsprechenden lokalen Variablen ersetzt wird.

Erzeugung von LL(1) Übersetzungsschemata – Beispiel

S	→	*	{ * }
		F {$S_1.e$:= S.e * F.val}	
		S_1 {S.s := S_1.s}	
	→	ε {S.s := S.e}	{), $, + }

Steuermengen

```
function S (Se: integer): integer;
var Fval, S1e, Ss, S1s: integer;
if symbol = *_sym then
    match(*_sym);
    Fval := F();
    S1e := Se * Fval;
    S1s := S(S1e);
    Ss := S1s;
    return Ss;
elsif symbol = )_sym or symbol = $_sym or symbol = +_sym
    Ss := Se; return Ss;
else error;
fi;
```

Bottom-Up-Implementierung von L-attributierten Übersetzungsschemata

Übersetzungsaktionen sind in die rechte Seite eingebettet. Sie sind zwar an Reduktionen gekoppelt, müssen aber ausgeführt werden bevor die komplette rechte Seite auf dem Stack liegt. D.h. die Ausführung muss vor der Reduktion erfolgen.

$A \rightarrow \ldots\ldots \{ \text{action}_1 \} \ldots. \{\text{action}_2 \} \ldots.$

Einführung von Markierungsvariablen für semantische Aktionen.

$A \rightarrow \ldots\ldots M_1 \ldots. M_2 \ldots.$

$M_1 \rightarrow \varepsilon \{ \text{action}_1 \}$

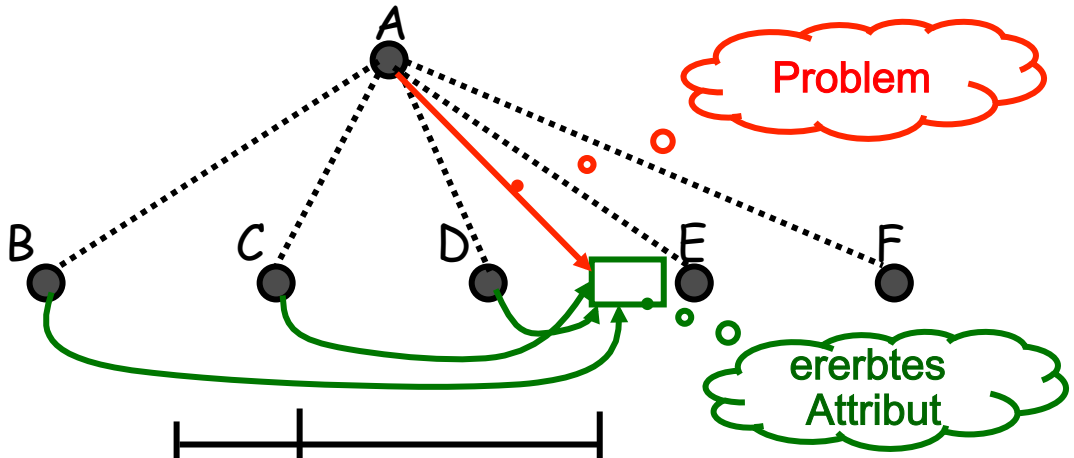
$M_2 \rightarrow \varepsilon \{ \text{action}_2 \}$

$\ldots\ldots$

S	→	*	
		F	$\{S_1.e := S.e * F.val\}$
		S_1	$\{S.s := S_1.s\}$
S	→	$*FMS_1$	$\{S.s := S_1.s\}$
M	→	ε	$\{S_1.e := S.e * F.val\}$

Ererbte Attributierung und Bottom-Up-Parsing 1/5

Den Attributen der Symbole auf der rechten Seite einer Produktion werden Werte zugewiesen. Diese Werte kommen von Attributen von links davon stehenden Symbolen und von ererbten Attributen der linken Seite der Produktion.



D	D.Attribute
C	C.Attribute
B	B.Attribute

E	E.Attribute
D	D.Attribute
C	C.Attribute
B	B.Attribute

Attribute, die von den Geschwistern ererbt werden, sind bekannt, wenn E an der Spitze des Parser-Stapels auftaucht.

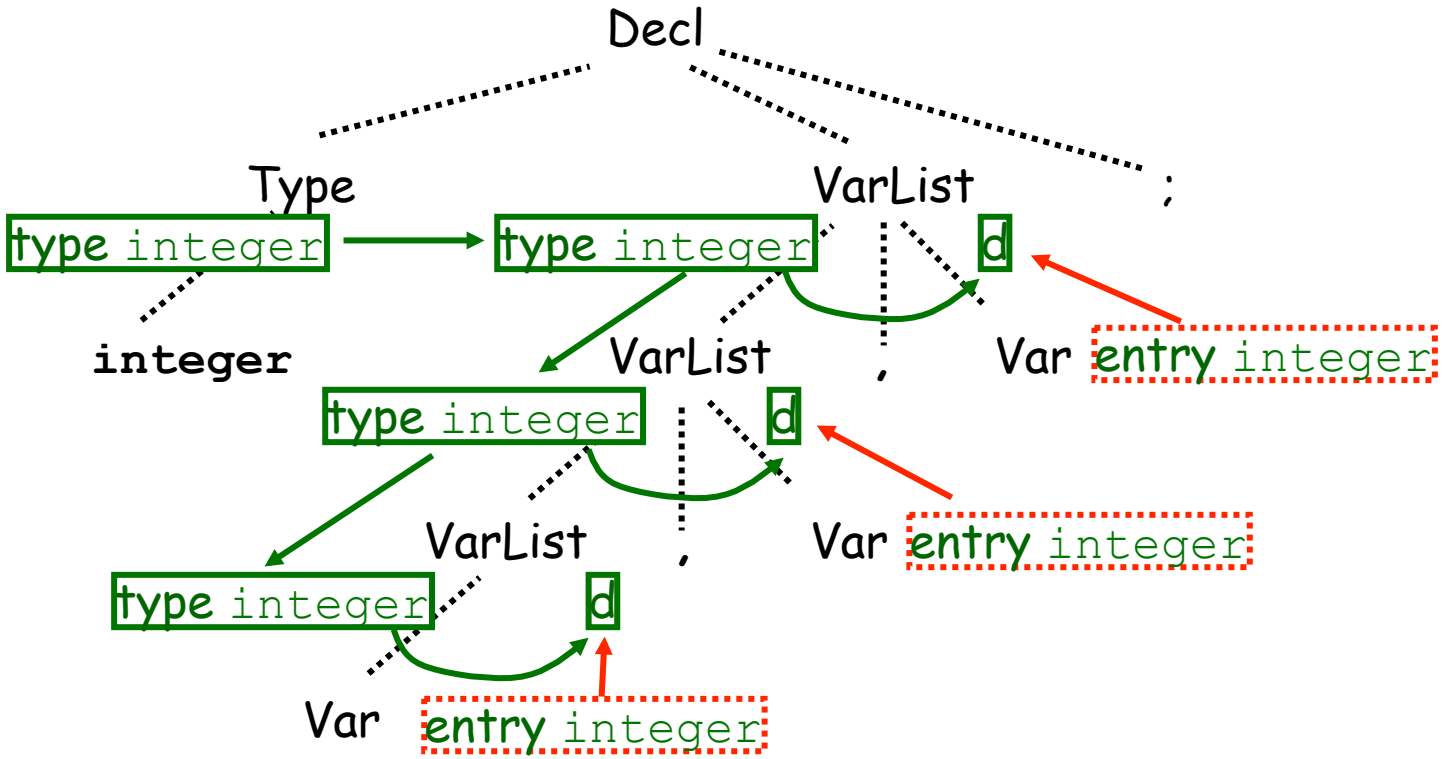
Ererbte Attributierung und Bottom-Up-Parsing - Übersetzungsschema Beispiel 1/4

Beispiel:

Decl	→	Type {VarList.type := Type.type} VarList ;
Type	→	**integer** {Type.type := integer}
Type	→	**real** {Type.type := real}
VarList	→	{$VarList_1$.type := VarList.type} $VarList_1$, Var {addtype(Var.entry, VarList.type)}
VarList	→	Var {addtype(Var.entry, VarList.type)}

```
integer a, b, c;
```

Ererbte Attributierung und Bottom-Up-Parsing - Übersetzungsschema Beispiel 2/4



Ererbte Attributierung und Bottom-Up-Parsing - Übersetzungsschema Beispiel 3/4

Aktionen des Bottom-up-Parsers

Stack	Eingabe	Reduktion
	integer a, b, c$	
$ integer	a, b, c$	Type → integer
$ Type	a, b, c$	
$ Type a	, b, c$	VarList → Var
$ Type VarList	, b, c$	
$ Type VarList ,	b, c$	
$ Type VarList , b	, c$	
$ Type VarList ,	, c$	VarList → VarList, Var

etc.

Ererbte Attributierung und Bottom-Up-Parsing - Übersetzungsschema Beispiel 4/4

Decl → Type VarList ;

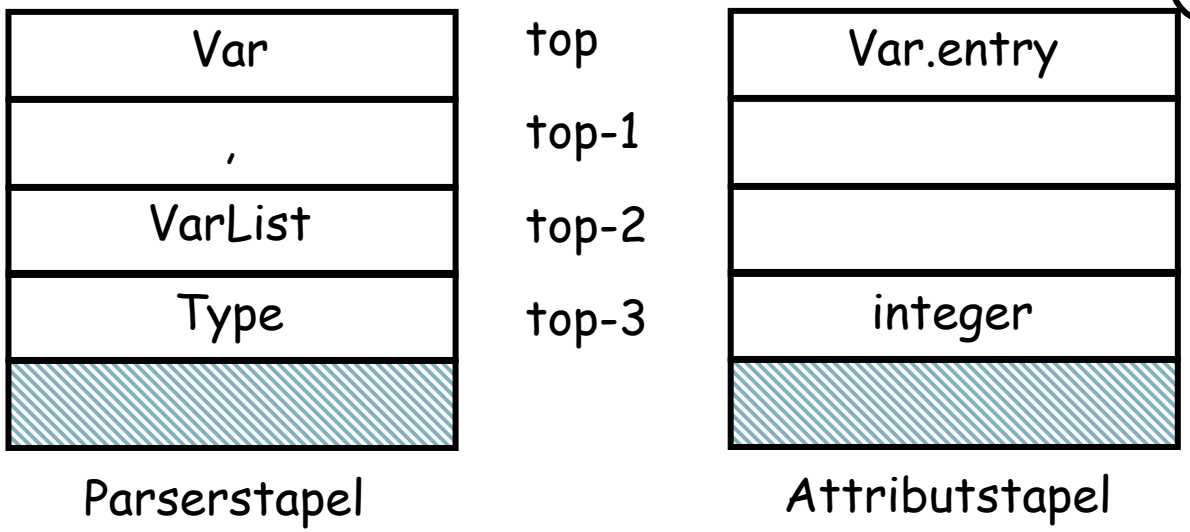
Type → **integer** {Attr[top] := integer}

→ **real** {Attr[top] := real}

VarList → $VarList_1$, Var
{addtype(Attr[top], Attr[top-3])}

VarList → Var
{addtype(Attr[top], Attr[top-1])}

ACHTUNG: Position von Attributen im Stapel muss bekannt sein!

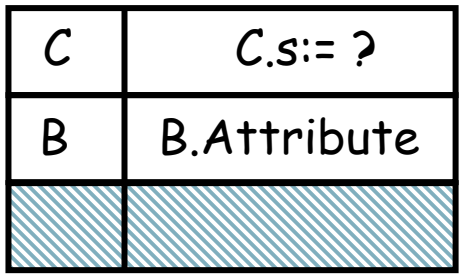


Nicht eindeutige Stack-Positionen

A → BC {C.e := B.s}
D → BEC {C.e := B.s}
C → c {C.s := f(... C.e ...)}

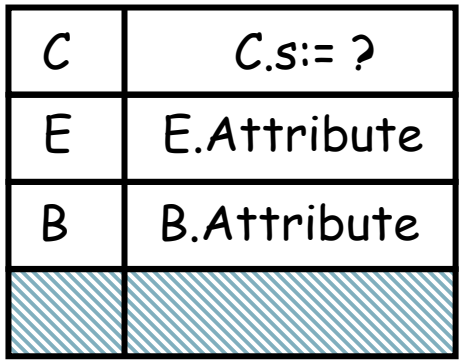
2 Situationen sind bei Reduktion mit C → c möglich

1



Benötigte Attribute liegen bei top-1

2



Benötigte Attribute liegen bei top-2

Einsatz von Markierungsvariablen

Ererbte Attribute der linken Seite der Produktion bzw. von weiter links stehenden Symbolen der rechten Seite weitergeben mit Markierungsvariablen M_i

$A \rightarrow X_1 \ldots . X_n \quad \Rightarrow \quad A \rightarrow M_1 X_1 \ldots . M_n X_n$

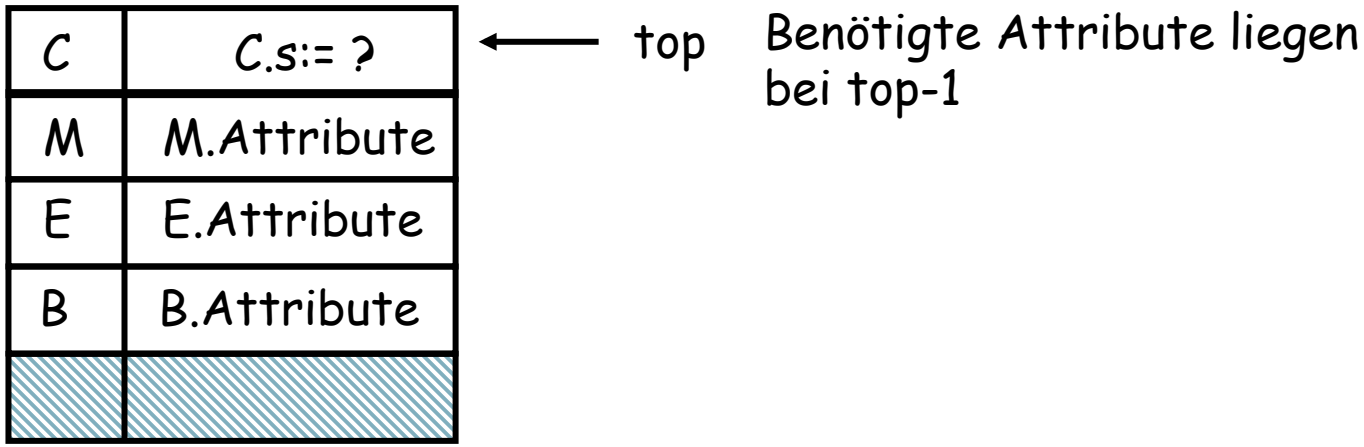
Allgemeines Schema

$A \rightarrow BC \ \{C.e := B.s\}$

$D \rightarrow BEMC \ \{M.e := B.s;\ C.e := M.s\}$

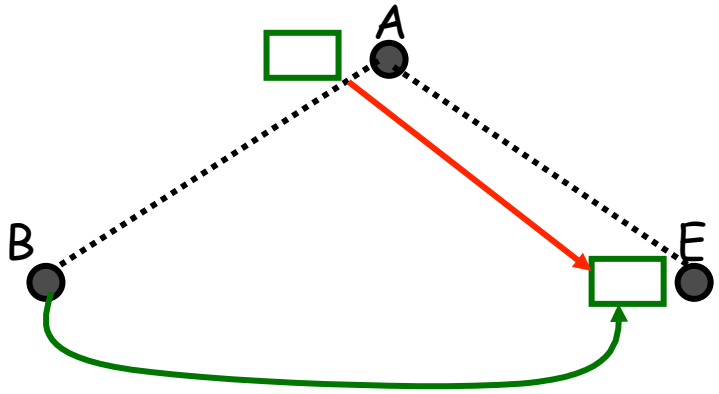
$C \rightarrow c \ \{C.s := f(\ldots C.e \ldots)\}$

$M \rightarrow \varepsilon \ \{M.s := M.e\}$



Ererbte Attribute bei der Bottom-up-Analyse – Zusammenfassung 1/2

Ererbte Attribute von A liegen direkt unterhalb von B (evtl. wurde eine Markierung vorgenommen: X → .... MA ... , M → ε



Fall1: Die ererbten Attribute C.e müssen lediglich von anderen Attributen links von C kopiert werden.
➔ Zugriffe über die bekannten Positionen dieser Attribute leicht möglich.

Fall2: Ererbte Attribute C.e müssen erst errechnet werden, d.h. C.e := f(... D.a ...).
➔Markierung setzen:

A → .... MC ..... mit semantischen Aktionen:
M.e := D.a
C.e := M.s
M → ε mit semantischen Aktionen:
M.s := f (.... M.e ....)

Ererbte Attribute bei der Bottom-up-Analyse – Zusammenfassung 2/2

Markierung vorgenommen: $A \rightarrow \ldots M \ldots$, $M \rightarrow \varepsilon$ spielen eine wichtige Rolle.

Sie erlauben folgende Normierungen:

- Semantische Aktionen werden nur im Zusammenhang mit Reduktionen ausgeführt.
- Die Position von Attributen kann relativ zur Spitze des Attribut-Stacks vorab bestimmt werden.
- Ererbte Attribute können rechtzeitig berechnet werden.

Aber .....

die Einführung von Markierungen zerstört evtl. die LR(1)-Eigenschaft.