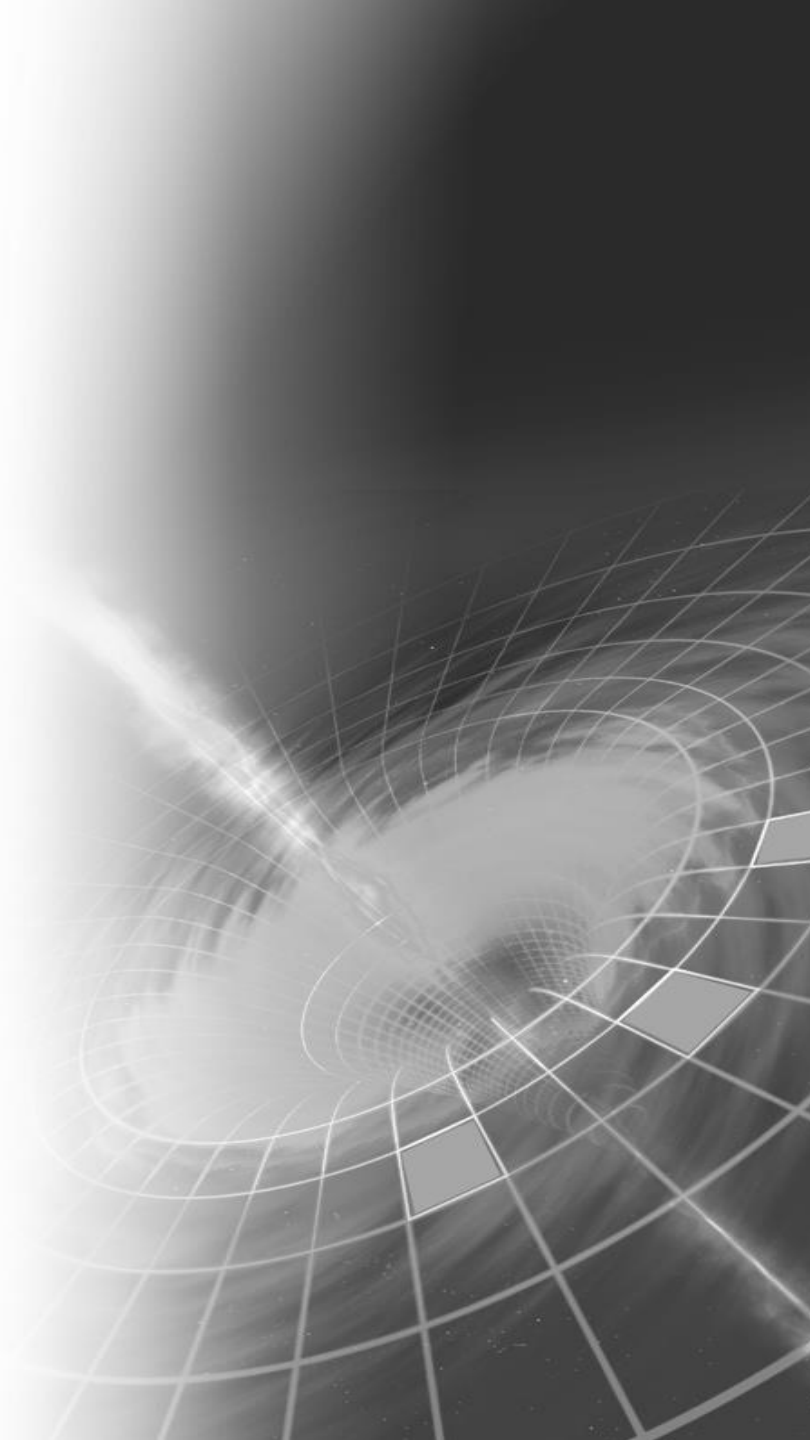
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbesch
- Kapitel 3
  Lexikalische Analyse: der Sca
- Kapitel 4
  Syntaktische Analyse: der Pa
- Kapitel 5
  Parser-Generatoren: Yacc, Bis
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

6.1
Überblick: Grammatik-basierte Übersetzung

6.2
Attribut-Grammatiken

6.3
S-attributierte Syntaxdefinitionen

6.4
L-attributierte Syntaxdefinitionen

6.5
Syntaxgesteuerte Übersetzungen im Überblick

6.6
Entwurf syntaxgesteuerter Übersetzungen

6.7
Symboltabelle

Synthetisierte und ererbte Attribute (Wdh.)

… nach Unterscheidung der Art der Attributberechnung

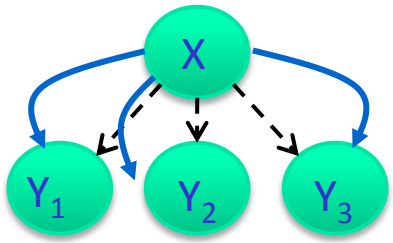
$X \rightarrow Y_1 Y_2 Y_3$

X. a

Y_1.a
Y_2.a
Y_3.a

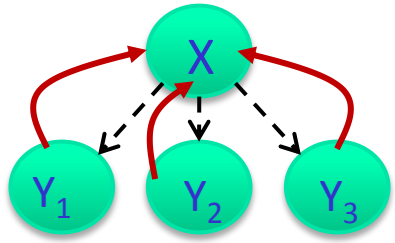
ererbte Attribute

$Y_1.a = f(X.a)$
$Y_2.a = f(X.a)$
$Y_3.a = f(X.a)$



Werte ererbter Attribute
werden von oberen Knoten
an untere zugewiesen (top-down)

synthetisierte Attribute



$X.a = f(Y_1.a, Y_2.a, Y_3.a)$

Werte synthetisierter Attribute
werden von unten nach oben „gereicht“
(bottom-up)

- **Token** (Terminalsymbole)
  - haben nur synthetisierte Attribute
  - ihre Werte werden i.allg. vom Scanner geliefert

- **Startsymbol**
  - hat keine ererbten Attribute

Implementierung syntaxgesteuerter Definitionen (Wdh.)

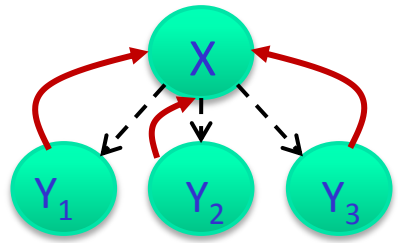
- **schwierig**
  Implementierung eines Compilers, der einer beliebigen syntaxgesteuerten Definition folgt

- **Klassifizierung** syntaxgesteuerter Definitionen

S-attributierte Definitionen
(enthalten nur synthetisierte Attribute)

können durch **LL**- und **LR-Parser**
on-the-fly
ausgewertet werden

synthetisierte Attribute



$X.a = f(Y_1.a, Y_2.a, Y_3.a)$

- Parser kann Attribute der Metasymbole (RS)
  synchron zu den Metasymbolen auf dem Stack festhalten
- bei einer Reduktion stehen dann diese Werte zur Verfügung, um die Attribute (LS) zu bestimmen

Beispiel: Typdeklaration von Bezeichern

syntaxgerichtete Definition

Produktion	semantische Regel
$D \rightarrow T\ L$	
$T \rightarrow$ **int**	
$T \rightarrow$ **real**	
$L \rightarrow L_1$, **id**	
$L \rightarrow$ **id**	

synthetisierte und ererbte Attribute

Aufgabe der Regeln:
Übernahme der Typinformation eines Bezeichners
aus einer Typdeklaration in die Symboltabelle

- Prüfen nicht, ob Bezeichner mehr als einmal deklarariert worden ist
- Reihenfolge der semantischen Aktionen ist zu bestimmen

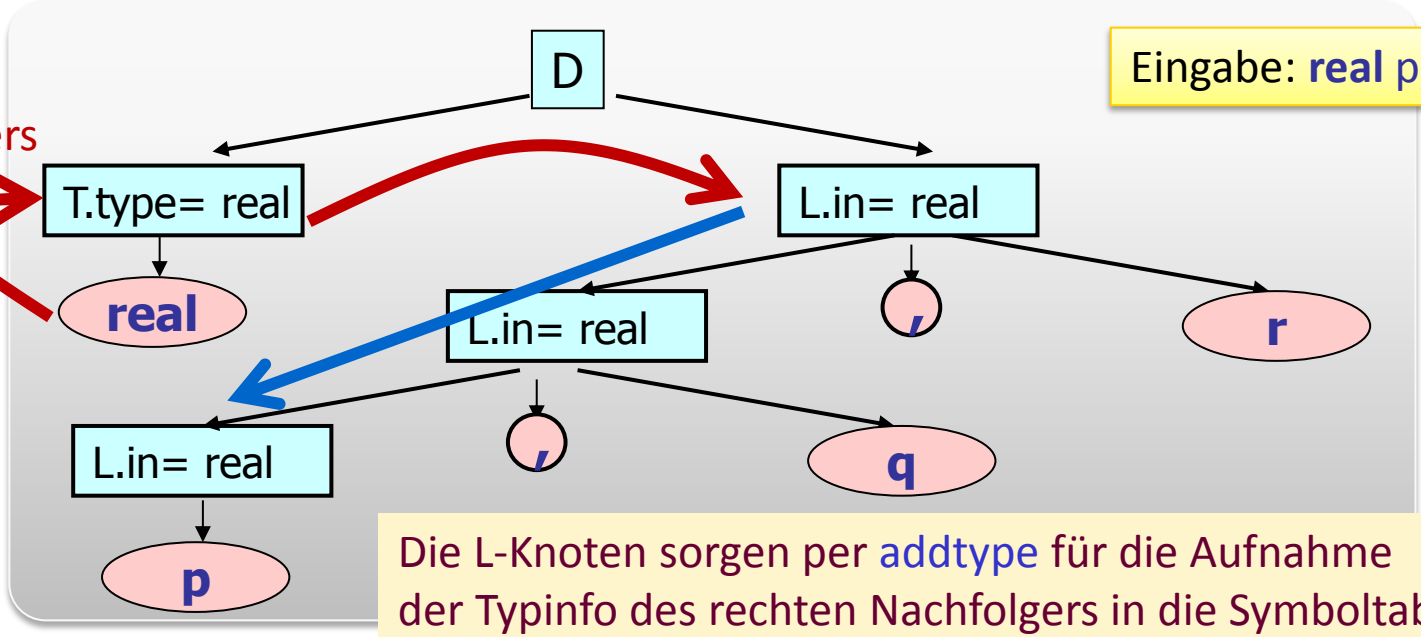
Beispiel: Typdeklaration von Bezeichern

Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

als geerbtes Attribut in wird die Typinformation top-down-artig im Baum verbreitet

Hinzufügen der Typinformation zum vorhandenen Bezeichner **id** in der Symboltabelle im Blattknoten

Bewegung des Parsers

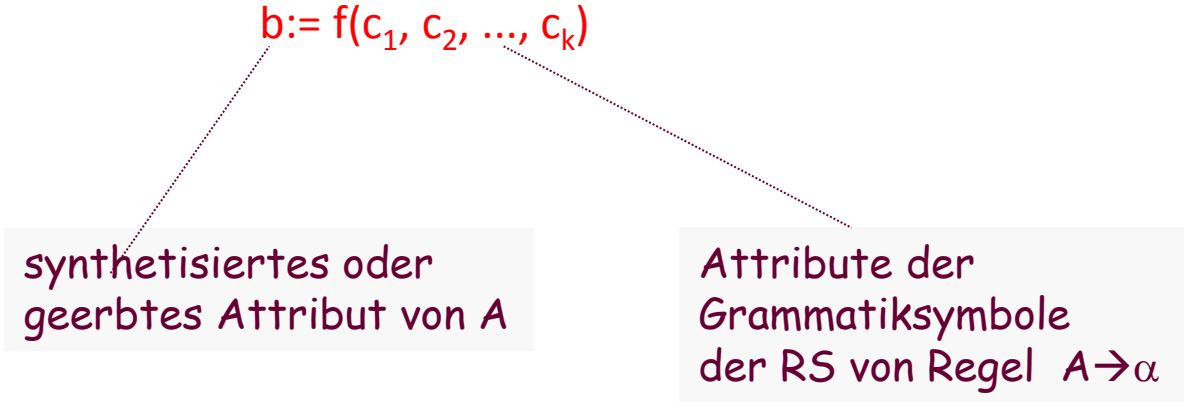


Eingabe: **real** p, q, r

Die L-Knoten sorgen per addtype für die Aufnahme der Typinfo des rechten Nachfolgers in die Symboltabelle

Konstruktion des Abhängigkeitsgraphen

- **Vorbereitung**
  - jede Semantikregel wird in folgende Funktions-Form gebracht:



  - der (Abhängigkeits-) Graph hat für jedes Attribut
    1. einen Knoten sowie
    2. eine Kante
       von Knoten c_i zum Knoten b, wenn b von c_i abhängt

Konstruktion des Abhängigkeitsgraphen

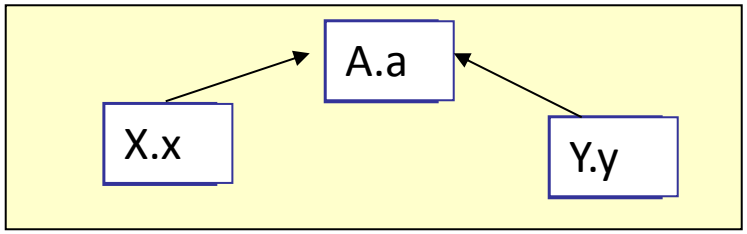
- **Beispielhafte Annahme**:

synthetisiertes Attribut

Produktion	semantische Regel
$A \rightarrow X\ Y$	A.a:= f (X.x, Y.y)
...	...

- **Idee**: wird eine solche Regel beim Aufbau des Syntaxbaums benutzt, dann gibt es zwei gerichtete Verbindungen

entsprechend der Datenflussrichtung



Algorithmus zur Konstruktion des Abhängigkeitsgraphen

```
for each Knoten k in syntaxBaum do
    for each Attribut a of GrammatikSymbol in k do
                konstruiereKnoten for a in graph;

for each Node k in syntaxTree do
    for each semantischeRegel b:= f(c1, c2, ..., cn) accociated with k do
        for i:= 1 to n do
                konstruiereKante from Knoten ki of graph to Knoten b;
```

danach:

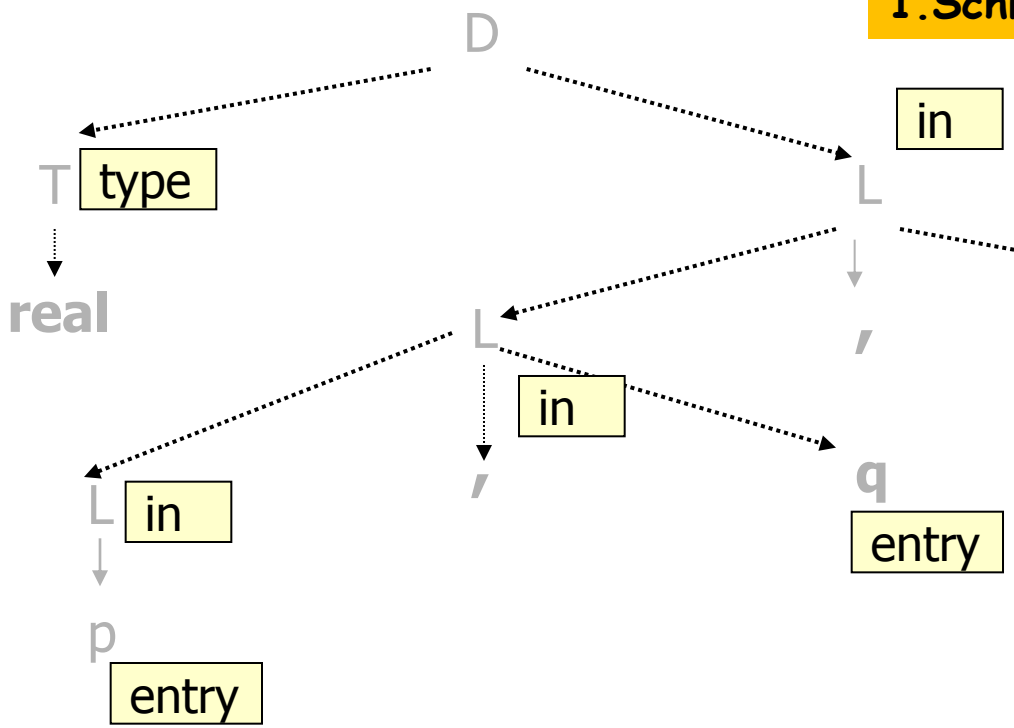
Durchführung einer topologischen Sortierung der Knoten

→ Reihenfolge der Knoten bestimmt Reihenfolge der Attributberechnung

Beispiel: Konstruktion des Abhängigkeitsgraphen

geg.: *attributierter Syntaxbaum für Eingabe*: **real** p, q, r

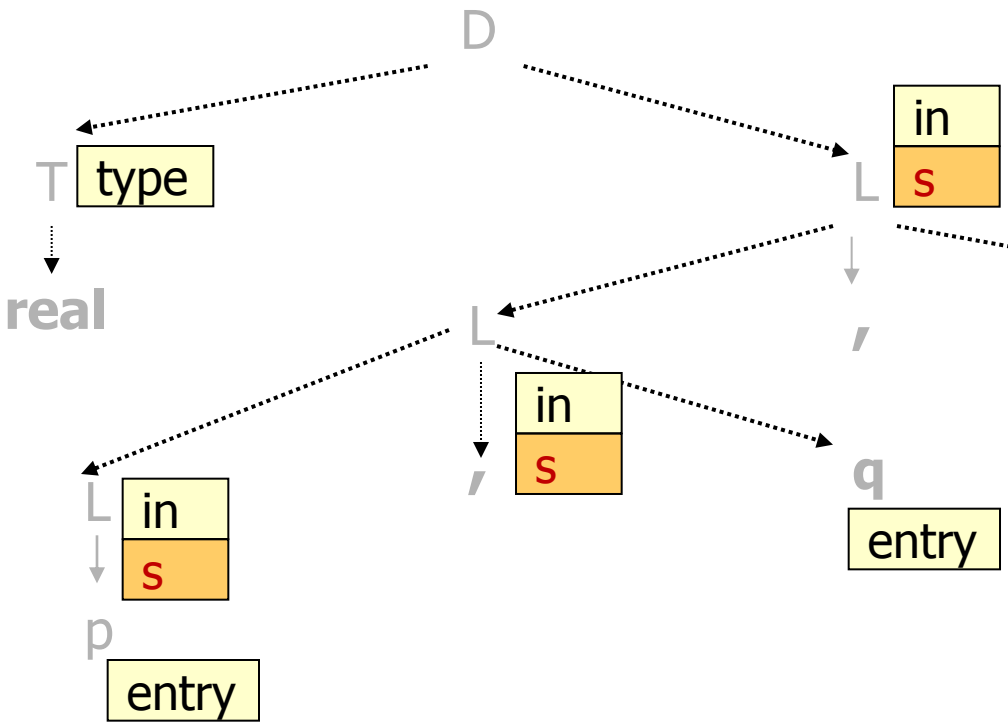
1.Schritt: Identifikation der Attribute im Baum



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

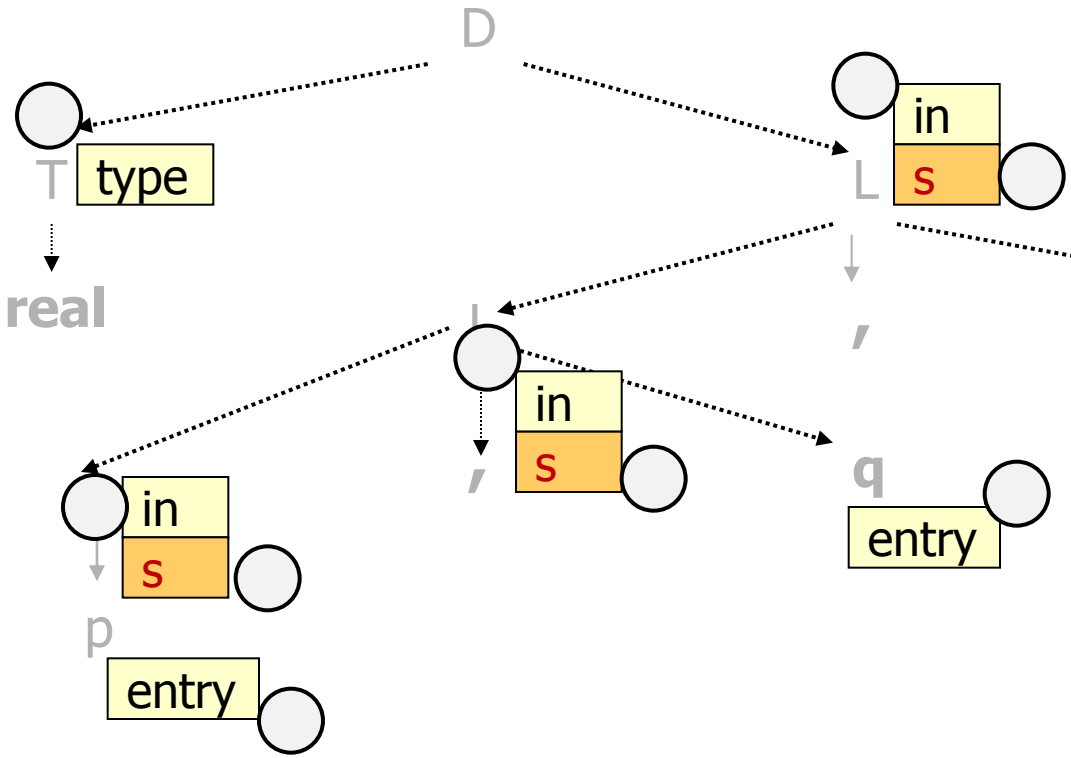
2.Schritt: Einführung zusätzlicher Scheinattribute *s*



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

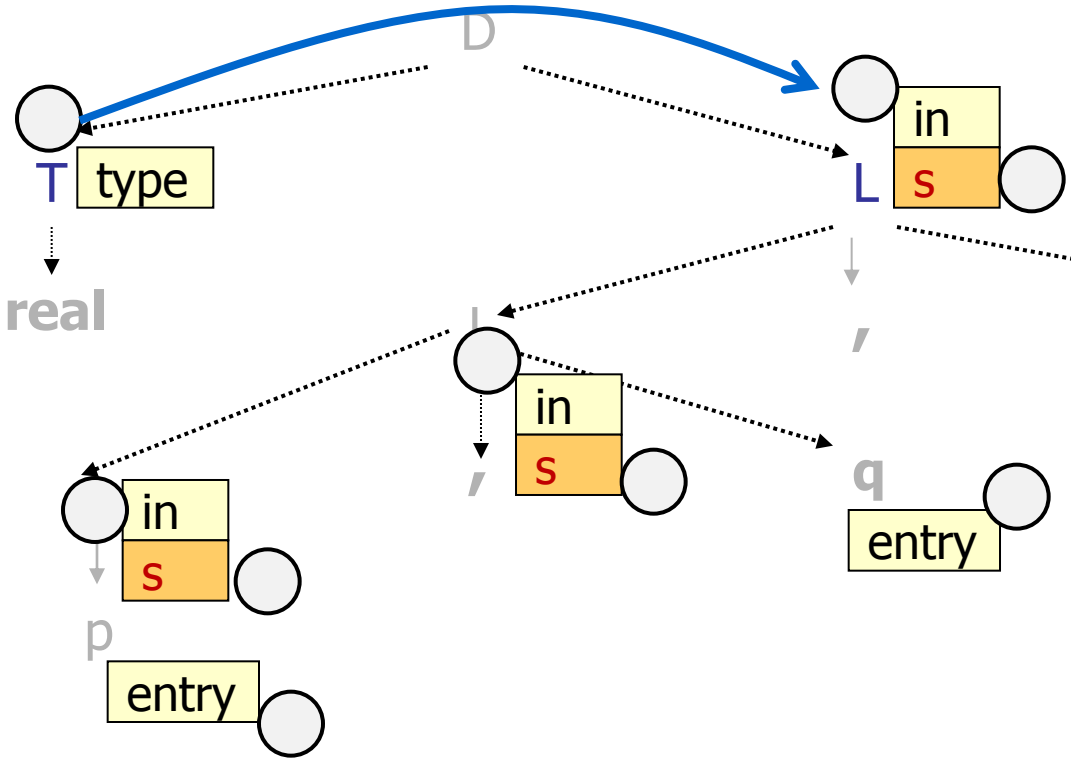
3.Schritt: Konstruktion von Knoten je Attribut des Abhängigkeitsgraphen



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
$L \rightarrow L_1$, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

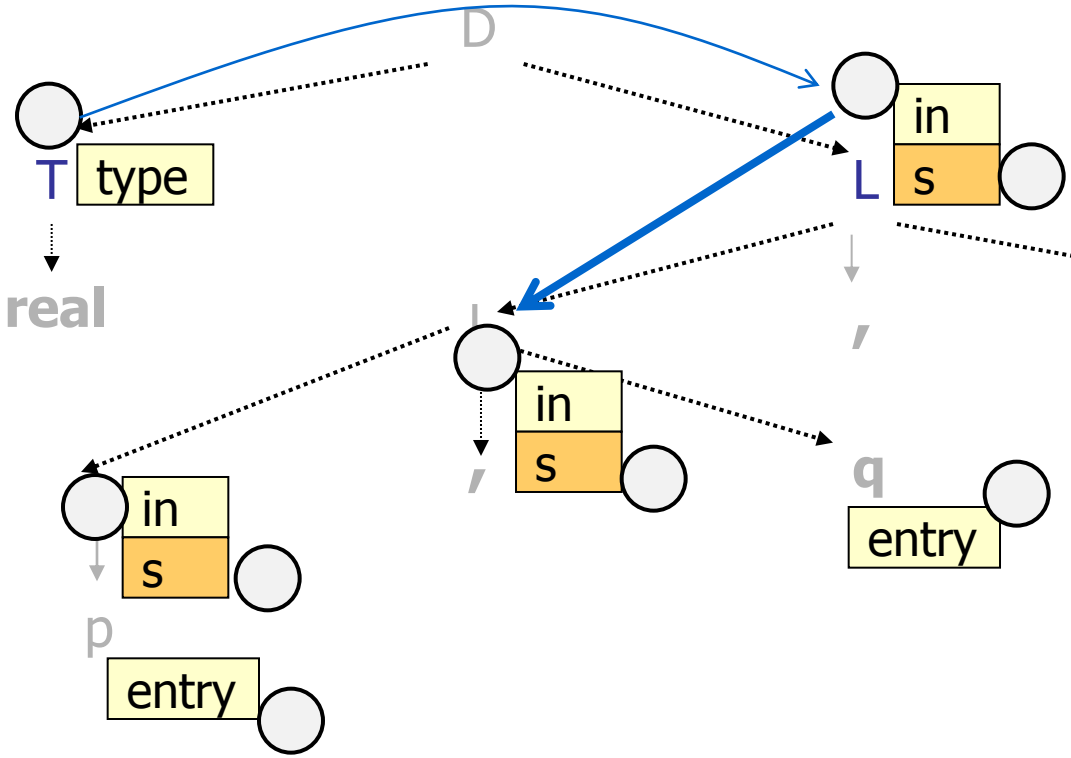
4. Schritt: Einführung von Kanten
(Richtung entspr. Abhängigkeitsbeziehung)



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

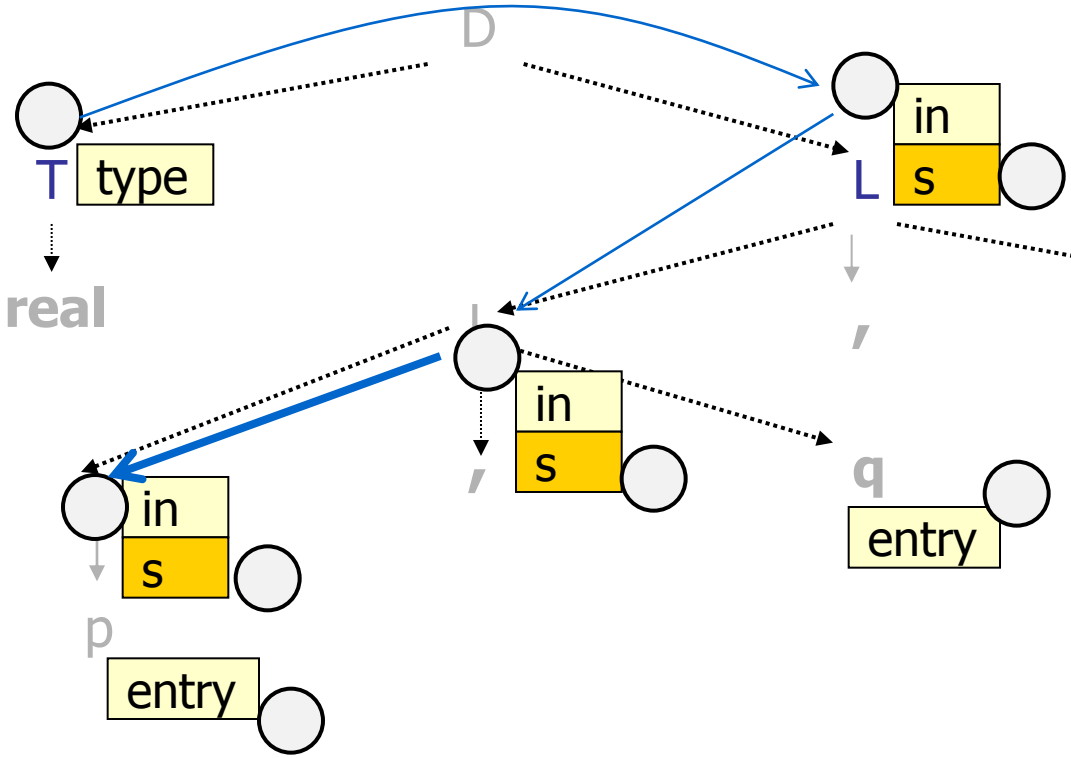
4. Schritt: Fortsetzung ...



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

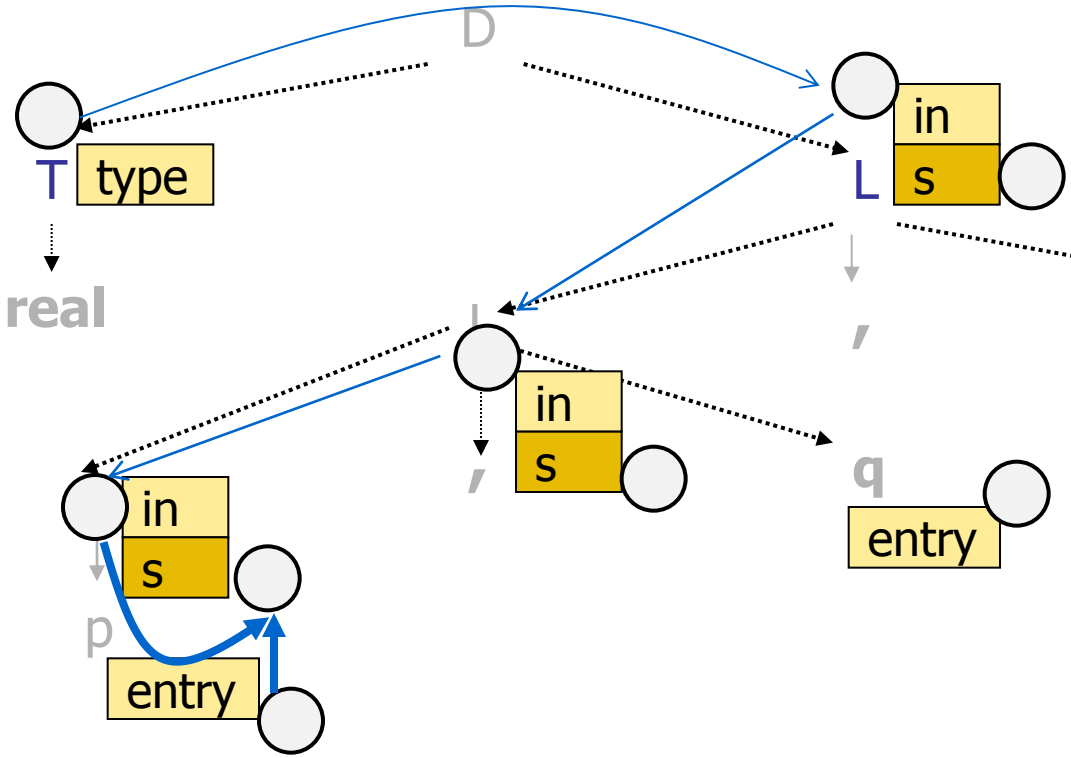
4. Schritt: Fortsetzung ...



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

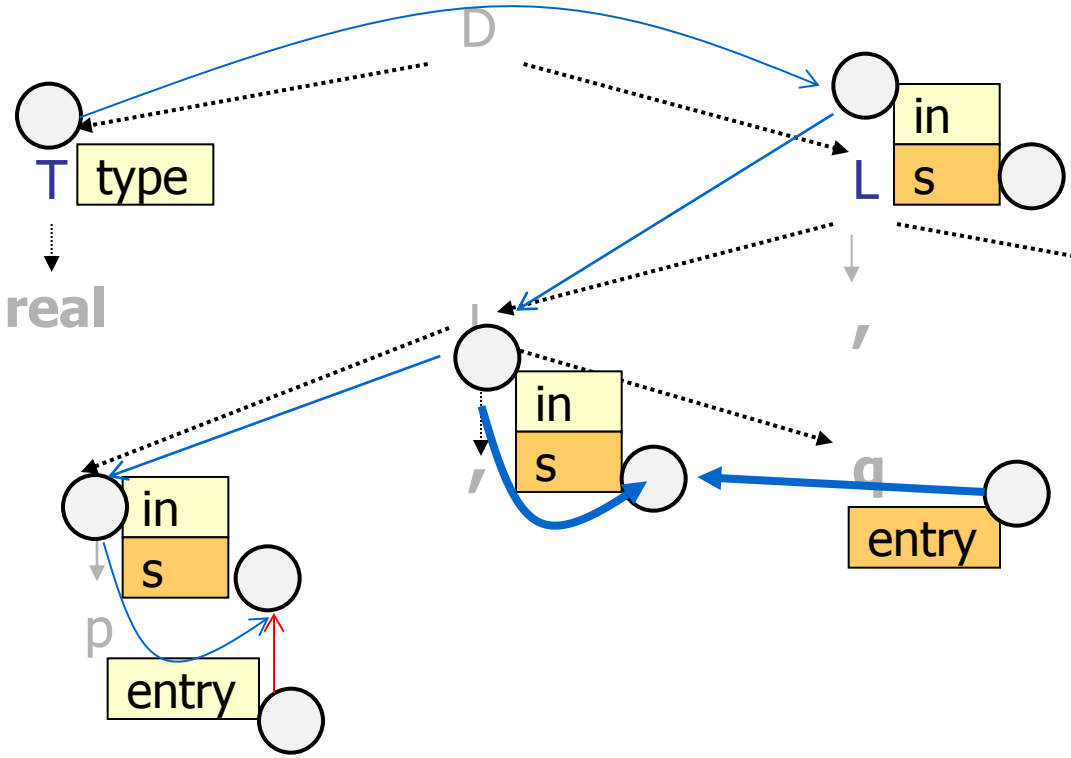
4. Schritt: Fortsetzung ...



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

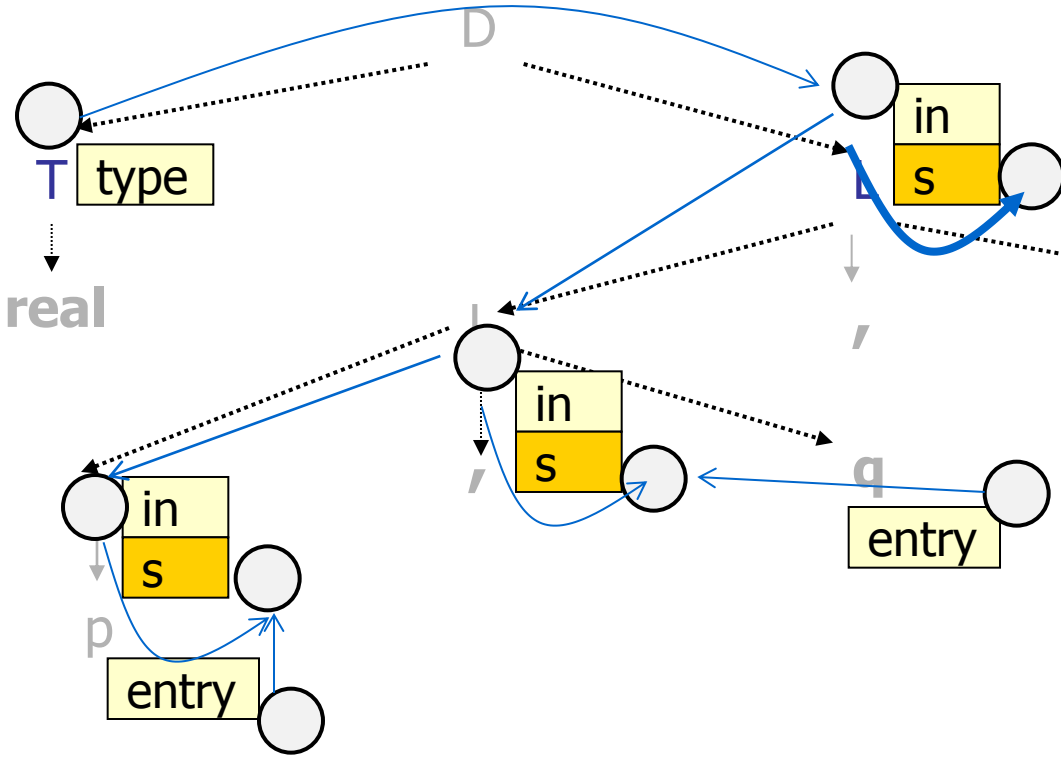
4. Schritt: Fortsetzung ...



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

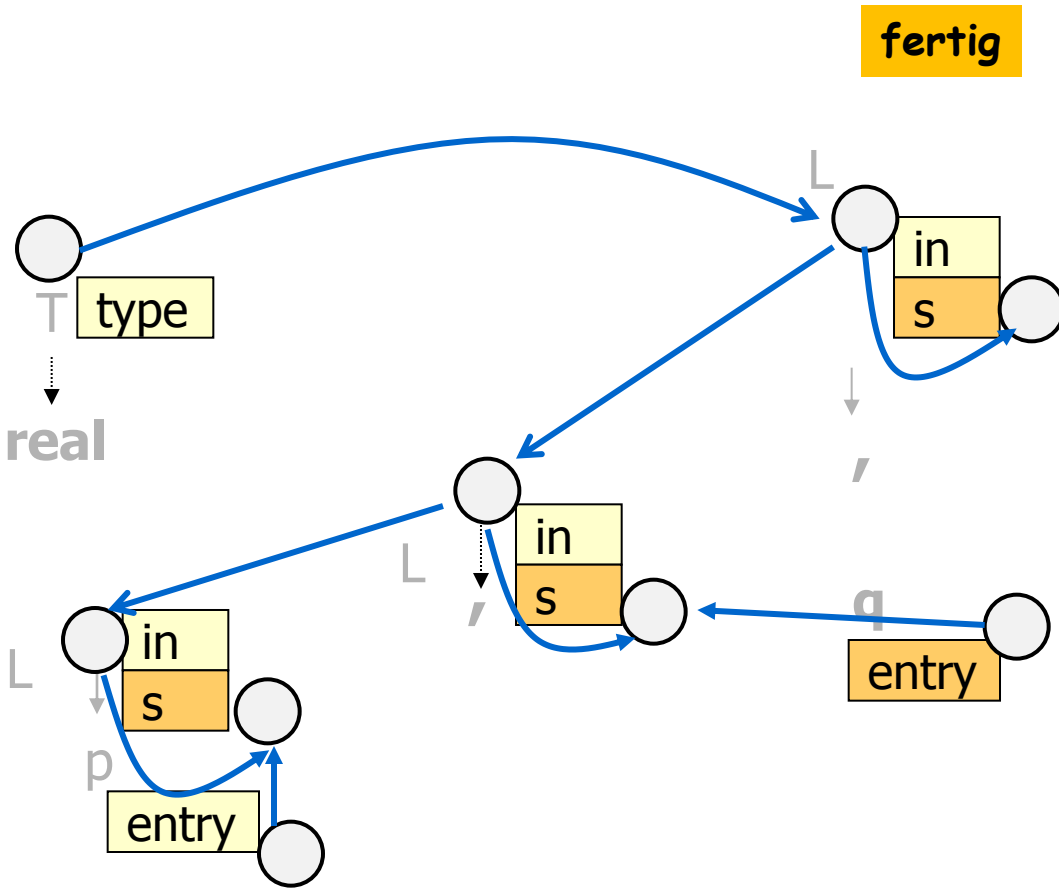
Beispiel: Konstruktion des Abhängigkeitsgraphen

4. Schritt: Fortsetzung ...



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen



Produktion	semantische Regel
D → T L	L.in := T.type
T → **int**	T.type := integer
T → **real**	T.type := real
L → L_1, **id**	L_1.in := L.in
	addtype (**id**.entry, L.in)
L → **id**	addtype (**id**.entry, L.in)

Beispiel: Konstruktion des Abhängigkeitsgraphen

5.Schritt: Berechnungsreihenfolge der Attribute nach topologischer Sortierung

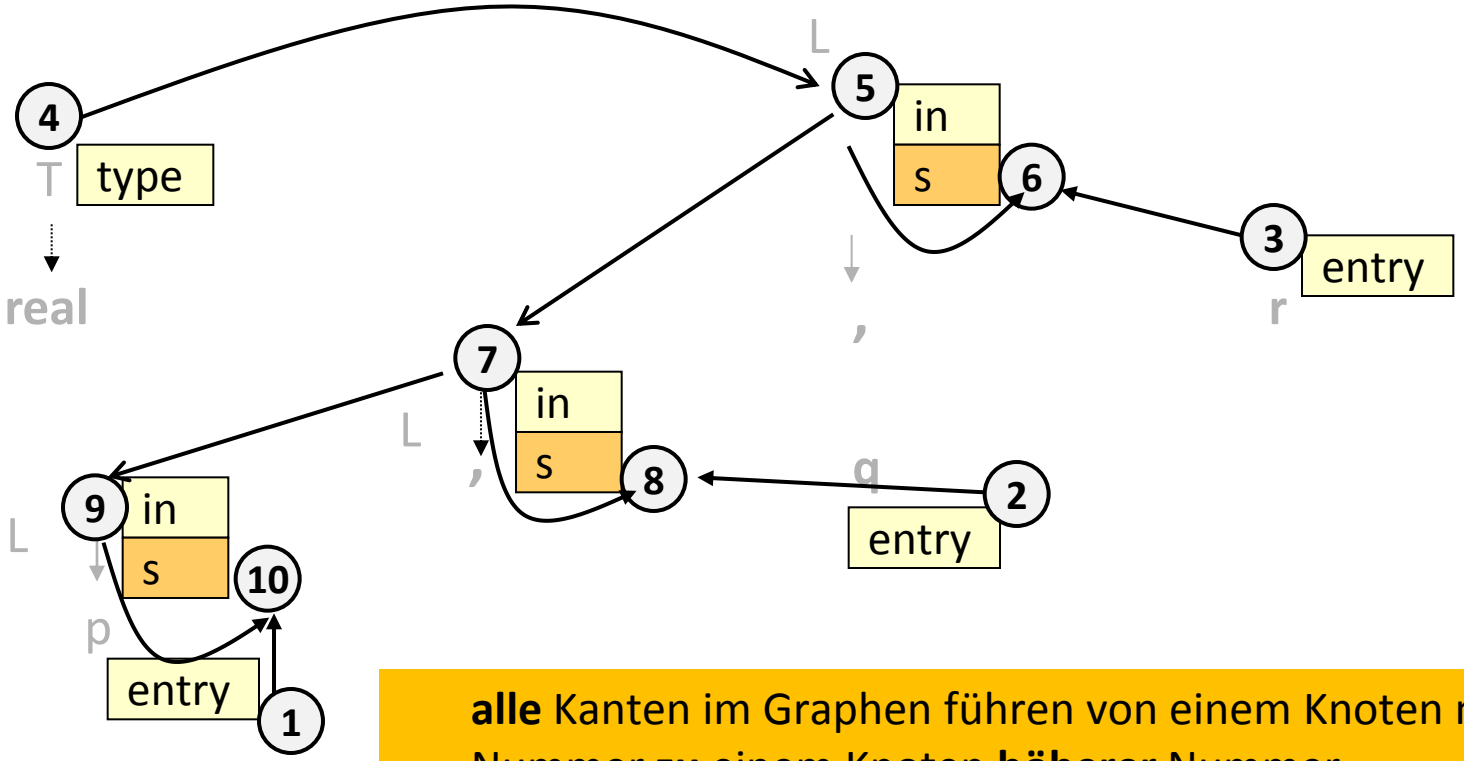
eine **topologische Sortierung** eines gerichteten azyklischen Graphen

ist

- irgendeine Reihenfolge $m_1, m_2, \ldots, m_k$ der Knoten m_i des Graphen,
- so dass alle Kanten von Knoten ausgehen,
  die früher in der Reihenfolge auftreten,
- und zu Knoten führen, die später vorkommen

$k_i \rightarrow k_j$, **mit** k_i **vor** k_j,

Beispiel: Konstruktion des Abhängigkeitsgraphen



alle Kanten im Graphen führen von einem Knoten mit **kleinerer** Nummer **zu** einem Knoten **höherer** Nummer

Abhängigkeitsgraph ist topologisch sortiert

Beispiel: Konstruktion des Abhängigkeitsgraphen

Ergebnis: Ausführungsreihenfolge semantischer Aktionen

(1) a_1:= "p"
(2) a_2:= "q"
(3) a_3:= "r"

} Attributwerte vom Scanner

(4) a_4:= real
(5) a_5:= a_4
(6) (a_6:=) addtype(r.entry, a_5)
(7) a_7:= a_5
(8) (a_8:=) addtype(r.entry, a_5)
(9) a_9:= a_7
(10) (a_{10}:=) addtype(r.entry, a_5)

Attribut a_i ist mit Knoten **i** im Abgängigkeitsgraphen assoziiert

mit Ausführung der Aktionen:

Typ **real** wird in Symboltabelleneinträgen für p, q, r übernommen

Zusammenfassung: Attribut-Abhängigkeitsgraph

Graph-Syntax

- Knoten repräsentieren Attribute
- Kanten repräsentieren Fluss der Werte

wichtige Erkenntnis:
Klassifizierung der Attribute induziert Abhängigkeitsgraphen

Eigenschaften

- Graph ist spezifisch für jeden Parse-Baum
- Größe steht in fester Beziehung zur Größe des Parse-Baumes
- **Abhängigkeitsgraph kann zusammen mit Parse-Baum erzeugt werden**
  Forderung: der Graph muss azyklisch sein;
  Testverfahren brauchen exponentielle Zeit (~ Größe der syntaxgesteuerten Definition)

Ausführungsreihenfolge

- Topologische Sortierung auf dem Abhängigkeitsgraphen, um die Attribute zu ordnen
- in dieser Ordnung werden die semantischen Regeln ausgewertet
- Ordnung hängt sowohl von der Grammatik aus auch von der Eingabe ab

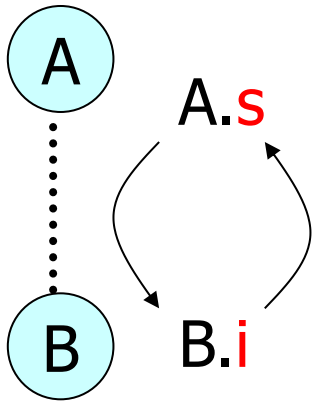
Problem mit beliebigem Mix geerbter und synthetisierter Attribute (Wdh.)

- keine Garantie für Eindeutigkeit der Reihenfolge der Attributberechnung

Beispiel:

A → B

A.s:= B.i
B.i:= A.s + 1



Regeln können i.allg. zirkulär sein

Ziel: Formulierung von Einschränkungen
bzgl. der Abhängigkeiten von Attributberechnungen,
die eine Zirkelfreiheit garantieren

Zwischenfazit

S-Attributgrammatik
zwar effizient berechenbar,
unterliegt aber gewissen Einschränkungen
(für die Praxis nicht mächtig genug)

- Suche nach Möglichkeit zur einfachen Implementation
  von solchen syntaxgesteuerten Definitionen,
  die neben synthetisierten auch gewisse ererbte Attribute zulassen

einfache Implementation wäre:

natürliche Navigation durch Baumstrukturen:
Tiefe-Zuerst (angewendet auf die Baumwurzel)

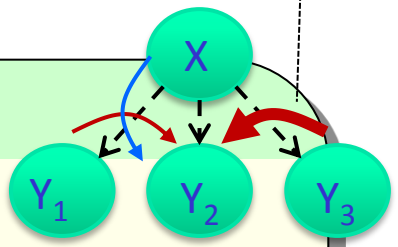
FRAGE: Welche Art von Attributabhängigkeit wird dabei vorausgesetzt?

ANTWORT: führt zur Definition von L-attributierten syntaxgesteuerten Definitionen

L-Attributgrammatiken

Bei Verletzung der Linksorientierung
➔ keine L-Attributgrammatik

$X \rightarrow Y_1 Y_2 Y_3$



Definition: L-Attributgrammatik

Sei $X \rightarrow Y_1 Y_2 \ldots Y_n$ eine Produktionsregel,

dann hängen die **ererbten** Attribute von Y_i ($inh(Y_i)$) mit $1 \leq \mathbf{i} \leq n$

nur von

(1) den ererbten Attributen des Nicht-Terminals X (inh(X))

und

(2) synthetisierten Attributen von $Y_1 \ldots Y_{i-1}$ (also nur von denen, die **links** von Y_i stehen)

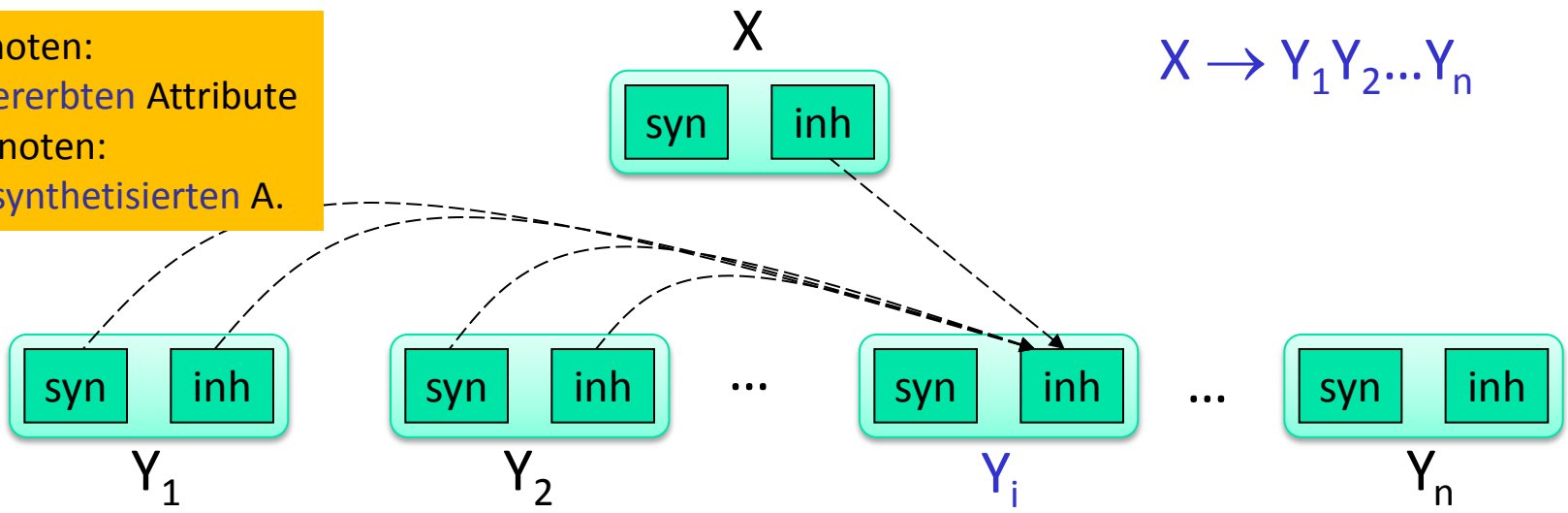
ab

Spezialfall einer L-Attributgrammatik
ist eine S-Attributgrammatik

L-Attribut-Grammatiken und LL-Parser

impliziert folgende (kanonische) Berechnungsreihenfolge für Top-Down-Parser:
inh(X), inh(Y_1), syn(Y_1), …, inh(Y_n), syn(Yn), syn(X)

bei Betreten eines Knoten:
Berechnung der ererbten Attribute
bei Verlassen eines Knoten:
Berechnung der synthetisierten A.



und dies stimmt mit der Auswertungsreihenfolge eines LL-Parser (Top-Down-Parsing) überein !!

L steht für Links
bedeutet: die Attributinformation »fließt« von links nach rechts
(als ein natürliches Prinzip)

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerier

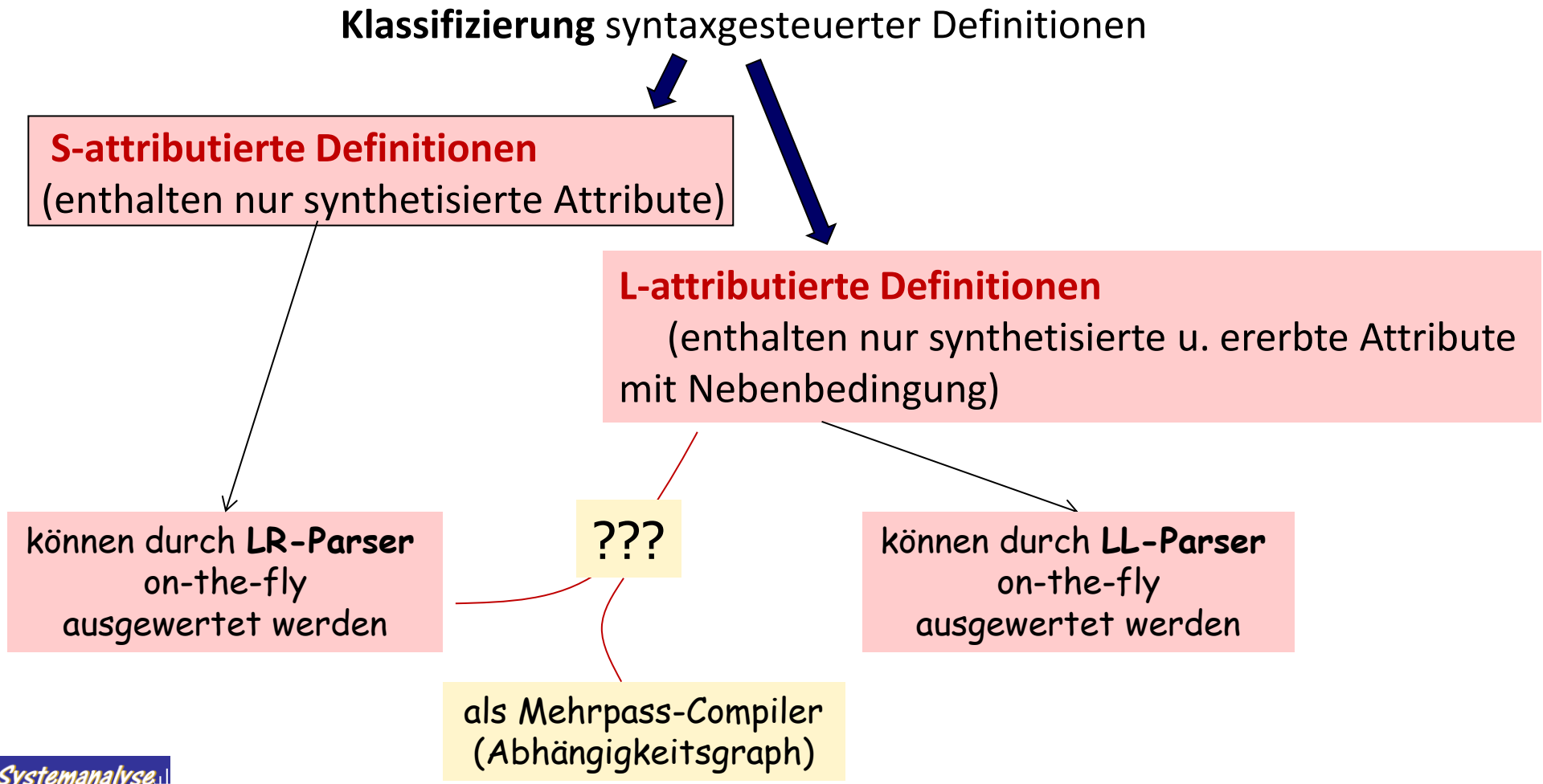
- 6.1
  Überblick: Grammatik-basierte Übersetzung
- 6.2
  Attributgrammatiken
- 6.3
  S-attributierte Syntaxdefinitionen
- 6.4
  Attributierte Syntaxdefinitionen mit synthetisierten und ererbten Attributen
- 6.5
  L-attributierte Syntaxdefinitionen
- 6.6
  Verfahren syntaxgesteuerter Übersetzungen im Überblick
- 6.7
  Entwurf syntaxgesteuerter Übersetzungen
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle

Erkenntnis-1: Bedeutung Attributgrammatiken

- Zentrales Problem bei der Übersetzung/statischen Analyse:
  - Assoziation von Informationen zu Teilstrukturen des Ableitungsbaums
  - Verwaltung der Informationen beim Aufbau des Baumes

- Attributgrammatiken als geeignetes Konzept
  - Attribute als Info-Behälter für jedes Grammatiksymbol
    Beispiel: Typinfos, Ausdruckswerte (als Zwischenresulte eines Interpreters)
  - jeder Produktion der Grammatik
    kann eine Menge von Funktionen als semantische Regel zugeordnet werden
  - Berechnung der Attribute erfolgt aus den Werten von Attributen anderer,
    in der Produktion vorkommenden Symbole

Erkenntnis-2:
Implementierung syntaxgesteuerter Definitionen ~ Art der Attributierung



Jetzt genauer
Parsertyp – Grammatiktyp – Attributierungstyp

Realisierungsmöglichkeiten /Probleme

Übersetzungsmethoden: on-the-fly

- **Rekursiv-absteigender Parser**
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Tabellengesteuerter Top-Down-Parser**
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LR-Grammatik mit einer S-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

Methode:

Funktionen des rekursiv absteigenden Parsers werden erweitert:

- **ererbte Attribute** werden zu Funktionsargumenten ihrer Nichtterminalsymbole
  (die aufgerufene Funktion erbt Informationen ihres Aufrufers)

- **synthetisierte Attribute** werden von diesen Funktionen als Kollektion zurückgegeben.

Methode

rekursiv absteigender Parser

```
void A () {
    /*Choose an A-production, A → X1X2…Xk;*/
    for ( i:= 1 to k) {
        if ( Xi is a_nonterminal )
            call procedure Xi();
        else if (Xi = current input symbol a)

            advance input to the next symbol;
        else /* error */
    }
}
```

jetzt:
Sematik-Aktionen: on-the-fly

Vorbereitung

- Argumente von A: geerbte Attribute vom A-Caller
- Return-Wert von A: Kollektion der synthetisierten Attribute von A
- Funktion hält für **jedes** Attribut aller Nichtterminalsymbole der A-Produktion eine lokale Variable

Prinzip der Verarbeitung

Betrachtung der Symbole/Aktionen X der rechten Seite der A-Produktion von links nach rechts:

- Sei X ein **Terminalsymbol** mit dem synthetisierten Attribut x, dann wird der Wert X.x in einer entsprechenden lokalen Variable gespeichert anschließend:
  Match von X und Aktualisierung der Eingabe
- Sei X ein **Nichtterminal** B, erfolgt eine Zuweisung
  $c := B(b_1, b_2, \ldots b_n)$
  mit
  b_i als Variable der geerbten Attribute für B
  c als lokale A-Variable zur Aufnahme des synthetisierten Attributs von B
- Sei X eine **Aktion**, wird der Code kopiert und jede Referenz auf ein Attribut wird durch die lokale Variable für dieses Attribut ersetzt

Beispiel

Konstruktion eines rekursiv-absteigenden Parsers
mit Auswerter der semantischen Regeln
kann nun nach dem „Kochrezept“ umgesetzt werden

Produktion	Semantische Regel
$E \rightarrow E_1 + T$	E.val:= E_1.val + T.val
$E \rightarrow E_1 - T$	E.val:= E_1.val - T.val
$E \rightarrow T$	E.val:= T.val
$T \rightarrow (E)$	T.val:= E.val
$T \rightarrow$ **number**	T.val:= **number**.val

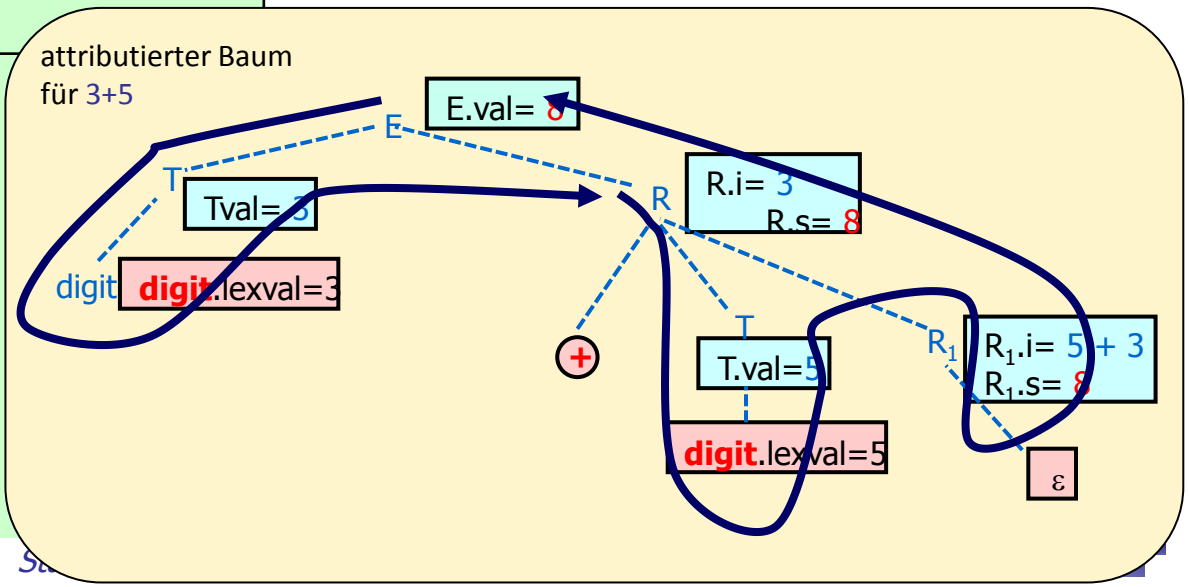
linksrekursiv, S-attributiert

Bei Entfernung der Linksrekursion
ändert sich die Attributierung:

neues Symbol R
mit zwei Attributen: i, s

rechtsrekursiv, L-attributiert

Produktion
$E \rightarrow T$ {R.i:= T.val} R {E.val:= R.s}
$R \rightarrow + T$ {R_1.i:= R.i + T.val} R_1 {R.s:= R_1.s}
$R \rightarrow - T$ {R_1.i:= R.i - T.val} R_1 {R.s:= R_1.s}
$R \rightarrow \varepsilon$ {R.s:= R.i}
$T \rightarrow$ (E {T.val:= E.val})
$T \rightarrow$ **number** {T.val:= number.val}



Anwendung der Methode

E → T {R.i:= T.val} R {E.val:= R.s}

```
procedure E(): int  {
        int Tval, Eval, Rs, Ri;
        Tval:= T();
        Ri:= Tval;
        Rs:= R(Ri);
        Eval:= Rs;
        return (Eval);
}
```

Vorbereitung

- Argumente von A: geerbte Attribute vom A-Caller
- Return-Wert von A: Kollektion der synthetisierten Attribute von A
- Funktion hält für **jedes** Attribut aller Nichtterminalsymbole der A-Produktion eine lokale Variable

Anwendung der Methode

E → T {R.i:= T.val} R {E.val:= R.s}

```
procedure E(): int  {
        int Tval, Eval, Rs, Ri;
        Tval:= T();
        Ri:= Tval;
        Rs:= R(Ri);
        Eval:= Rs;
        return (Eval);
}
```

T hat keine ererbten Attribute

Vorbereitung

- Argumente von A: geerbte Attribute vom A-Caller
- Return-Wert von A: Kollektion der synthetisierten Attribute von A
- Funktion hält für **jedes** Attribut aller Nichtterminalsymbole der A-Produktion eine lokale Variable

Prinzip der Verarbeitung

Betrachtung der Symbole/Aktionen X der rechten Seite der A-Produktion von links nach rechts:

- Sei X ein **Terminalsymbol** mit dem synthetisierten Attribut x, dann wird der Wert X.x in einer entsprechenden lokalen Variable gespeichert anschließend:
  Match von X und Aktualisierung der Eingabe
- Sei X ein **Nichtterminal** B, erfolgt eine Zuweisung
  $c:= B(b_1, b_2, \ldots b_n)$
  mit
  b_i als Variable der geerbten Attribute für B
  c als lokale A-Variable zur Aufnahme des synthetisierten Attributs von B
- Sei X eine **Aktion**, wird der Code kopiert und jede Referenz auf ein Attribut wird durch die lokale Variable für dieses Attribut ersetzt

Anwendung der Methode

$E \rightarrow T \{R.i := T.val\}\ R \{E.val := R.s\}$

```
procedure E(): int  {
        int Tval, Eval, Rs, Ri;
        Tval:= T();
        Ri:= Tval;
        Rs:= R(Ri);
        Eval:= Rs;
        return (Eval);
}
```

Vorbereitung

- Argumente von A: geerbte Attribute vom A-Caller
- Return-Wert von A: Kollektion der synthetisierten Attribute von A
- Funktion hält für **jedes** Attribut aller Nichtterminalsymbole der A-Produktion eine lokale Variable

Prinzip der Verarbeitung

Betrachtung der Symbole/Aktionen X der rechten Seite der A-Produktion von links nach rechts:

- Sei X ein **Terminalsymbol** mit dem synthetisierten Attribut x, dann wird der Wert X.x in einer entsprechenden lokalen Variable gespeichert anschließend:
  Match von X und Aktualisierung der Eingabe
- Sei X ein **Nichtterminal** B, erfolgt eine Zuweisung
  $c := B(b_1, b_2, \ldots b_n)$
  mit
  b_i als Variable der geerbten Attribute für B
  c als lokale A-Variable zur Aufnahme des synthetisierten Attributs von B
- Sei X eine **Aktion**, wird der Code kopiert und jede Referenz auf ein Attribut wird durch die lokale Variable für dieses Attribut ersetzt

Anwendung der Methode

$E \rightarrow T \{R.i := T.val\}\ R\ \{E.val := R.s\}$

```
procedure E(): int  {
        int Tval, Eval, Rs, Ri;
        Tval:= T();
        Ri:= Tval;
        Rs:= R(Ri);
        Eval:= Rs;
        return (Eval);
}
```

Vorbereitung

- Argumente von A: geerbte Attribute vom A-Caller
- Return-Wert von A: Kollektion der synthetisierten Attribute von A
- Funktion hält für **jedes** Attribut aller Nichtterminalsymbole der A-Produktion eine lokale Variable

Prinzip der Verarbeitung

Betrachtung der Symbole/Aktionen X der rechten Seite der A-Produktion von links nach rechts:

- Sei X ein **Terminalsymbol** mit dem synthetisierten Attribut x, dann wird der Wert X.x in einer entsprechenden lokalen Variable gespeichert anschließend:
  Match von X und Aktualisierung der Eingabe
- Sei X ein **Nichtterminal** B, erfolgt eine Zuweisung
  $c := B(b_1, b_2, \ldots b_n)$
  mit
  b_i als Variable der geerbten Attribute für B
  c als lokale A-Variable zur Aufnahme des synthetisierten Attributs von B
- Sei X eine **Aktion**, wird der Code kopiert und jede Referenz auf ein Attribut wird durch die lokale Variable für dieses Attribut ersetzt

Anwendung der Methode

$E \rightarrow T \{R.i := T.val\}\ R \{E.val := R.s\}$

```
procedure E(): int  {
        int Tval, Eval, Rs, Ri;
        Tval:= T();
        Ri:= Tval;
        Rs:= R(Ri);
        Eval:= Rs;
        return (Eval);
}
```

Vorbereitung

- Argumente von A: geerbte Attribute vom A-Caller
- Return-Wert von A: Kollektion der synthetisierten Attribute von A
- Funktion hält für **jedes** Attribut aller Nichtterminalsymbole der A-Produktion eine lokale Variable

Prinzip der Verarbeitung

Betrachtung der Symbole/Aktionen X der rechten Seite der A-Produktion von links nach rechts:

- Sei X ein **Terminalsymbol** mit dem synthetisierten Attribut x, dann wird der Wert X.x in einer entsprechenden lokalen Variable gespeichert anschließend:
  Match von X und Aktualisierung der Eingabe
- Sei X ein **Nichtterminal** B, erfolgt eine Zuweisung
  $c := B(b_1, b_2, \ldots b_n)$
  mit
  b_i als Variable der geerbten Attribute für B
  c als lokale A-Variable zur Aufnahme des synthetisierten Attributs von B
- Sei X eine **Aktion**, wird der Code kopiert und jede Referenz auf ein Attribut wird durch die lokale Variable für dieses Attribut ersetzt

```
procedure T(): int  {
        int Tval, Eval, numberval;
        if token == "(" then {
        nextToken();
                Eval:= E();
                Tval:= Eval();
                if not (token == ")" ) then
                        error();
                else {
                        nextToken();
                        return (Tval);
                }
        }
        else if token == number then {
                numberval:= lexval;
                nextToken();
                Tval:= numberval;
                return (Tval);
        }
        else error();
}
```

Produktion
$E \rightarrow T \{R.i := T.val\}\ R \{E.val := R.s\}$
$R \rightarrow + T \{R_1.i := R.i + T.val\}\ R_1 \{R.s := R_1.s\}$
$R \rightarrow - T \{R_1.i := R.i - T.val\}\ R_1 \{R.s := R_1.s\}$
$R \rightarrow \varepsilon \{R.s := R.i\}$
$T \rightarrow (E \{T.val := E.val\}\ )$
$T \rightarrow$ **number** $\{T.val := number.val\}$

```
procedure R (int Ri): int  {
        int Rs, Tval, R1s, R1i;
        if token == "+" then {
                nextToken();
                Tval:= T();
                R1i:= Ri + Tval;
                R1s:= R (R1i);
                Rs:= R1s;
                return (Rs);
        }
        else if token == "-"  then
                nextToken();
                Tval:= T();
                R1i:= Ri - Tval;
                R1s:= R (R1i);
                Rs:= R1s;
                return (Rs);
        }
        else if token == ")" or token == "$" then {
                Rs:= Ri;
                return (Rs);
        }
        else error();
```

```
procedure E(): int  {
        int Tval, Eval, Rs, Ri;
        Tval:= T();
        Ri:= Tval;
        Rs:= R(Ri);
        Eval:= Rs;
        return (Eval);
}
```

Übersetzungsmethoden: on-the-fly

- **Rekursiv-absteigender Parser**
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich, „hausgemacht“)

- **Tabellengesteuerter Top-Down-Parser**
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LR-Grammatik mit einer S-Attributierung
  (direkte Interpretation möglich, „hausgemacht“)

- **Bottom-Up**-Parser
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

Methode:

- **Hinzunahme eines Werte-Kellers**
  (wie bei S-Attributierung und LR-Parser)
- **aber schwieriger zu verwalten**
  Wertekeller wächst und schrumpft nicht synchron zum Parser-Stack

Top-Down-Implementierung L-attributierter Syntaxschemata (2)

- Liegt der L-attributierten syntaxbasierten Definition eine LL-Grammatik zugrunde, kann die Implementation (Ausführung der Aktionen) mit einem **tabellengesteuerten LL-Parser** erfolgen.

- Rekords für synthetisierte Attribute eines Nichtterminalsymbols werden unterhalb des Nichtterminalsymbols/Zustand auf dem **Parser-Keller** abgelegt,

- die ererbten Attribute eines Nichtterminalsymbols werden **synchron** mit dem Nichtterminalsymbol/Zustand (im Werte-Keller) abgelegt werden.

- Auch die Aktionen werden im Keller platziert, um die Attribute wie gefordert zu berechnen.

wollen wir hier
aber nicht weiter vertiefen

Übersetzungsmethoden: on-the-fly

- **Rekursiv-absteigender Parser**
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Tabellengesteuerter Top-Down-Parser**
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LR-Grammatik mit einer S-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

Methode:

- Ein S-attributiertes Übersetzungsschema kann implementiert werden,
  wenn **alle** Aktionen **am Ende** von Produktionen stehen.
- Die Aktionen berechnen dabei
  die synthetisierten Attribute der RS aus den synthetisierten Attributen der LS der Regeln.
- Ist die Ausgangsgrammatik vom Typ **LR**, dann kann dieses Übersetzungsschema auf dem LR-Parser-Kellerspeicher umgesetzt werden.

LR-Parser mit S-attributierter Übersetzung

Bison-Notation

```
%{
#include <ctype.h>
%}

%token          DIGIT

%%
line:   expr '\n'   {printf("%d\n", $1); }
                    ;
expr:  expr '+' expr    {$$ = $1 + $3; }
                    | term
                    ;
term: term '*' factor     {$$ = $1 * $3; }
                    | factor
                    ;
factor:             '(' expr ')'       {$$ = $2; }
                    | DIGIT
                    ;
```

Produktion	Semantische Regel
$L \rightarrow E$ **n**	print(val[tos])
$E \rightarrow E_1 + T$	val[ntos] := val[tos-2] + val[tos]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[ntos]:= val[tos-2] * val[tos]
$T \rightarrow F$	
$F \rightarrow (E)$	val[ntos] := val[tos-1]
$F \rightarrow$ digit	

syntaxgesteuerte Definition (S-Attributgrammatik)

ermöglicht durch eine Erweiterung des Kellerspeichers,
bei natürlicher **Synchronität** von Attribut-Werte-Stack und Parser-Stack

Bottom-up-Analyse passt (intuitiv) zur S-Attributierung
„man arbeitet von unten nach oben"

Frage
Lässt sich L-Attributberechnung
und
Bottom-up-Syntaxanalyse trotzdem kombinieren?

Jetzt LR und L

Verallgemeinerung der Methode LR-Parser und S-Attributierung
Attribute werden parallel zum Zustand ein- bzw. ausgekellert

Diese wird mächtig genug sein, um
L-attributierte Definitionen , die auf einer LL(1)-Grammatik aufbauen,
on-the-fly zu behandeln

Übersetzungsmethoden: on-the-fly

- **Rekursiv-absteigender Parser**:
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Tabellengesteuerter Top-Down-Parser**:
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LR-Grammatik mit einer S-Attributierung
  (direkte Interpretation möglich)

- **Bottom-Up**-Parser
  LL-Grammatik mit einer L-Attributierung
  (direkte Interpretation möglich)

Methode

- Grammatik-Transformation:
  Einführung neuer Nichtterminalsymbole als Marker
- Simulation ererbter Attribute durch synthetische Attribute der Marker

Konvertierungsprinzip

Übersetzungsschema

A → {B.i = f(A.i);} B C

Einführung eines Markers M
mit zwei Attributen

M | i | Kopie von A.i
| s | Stellvertreter für B.i

Übersetzungsschema

A → M B C

M → ε {M.i=A.i; M.s= f(A.i);}

Wie kommt man an
diese Information ?

ACHTUNG

auf den ersten Blick problematisch!

- Aktionen für M→ ε haben **keinen** Zugriff auf Attribute,
  die zu Grammatiksymbolen gehören,
  die nicht zu dieser Produktion gehören

JEDOCH

Implementation der Aktionen auf einem **LR-Parser-Kellerspeicher**

→ werden sehen, dass diese Attribute tatsächlich noch auf dem Keller liegen

ALTERNATIVER ANSATZ

A müsste vorsorglich dafür sorgen,

dass für andere Symbole (hier M),
die die ererbten Attribute benötigen, diese in globalen Speichern bereit gestellt werden

Stack-Indizierung in Yacc/Bison (bereits bekannt)

- jedes Grammatiksymbol hat im Rahmen einer Regelanwendung einen semantischen Wert

  erreichbar über $i Notation

Beispiel

```
E →   E '+' E   {$$ = $1 + $3;}
  →   E '*' E   {$$ = $1 * $3; }
  →   num       {$$ = $1;}
;
```

- neben Parser-Keller (Zustandskeller) gibt es einen synchronen **Wertekeller**
- die Pseudovariable $i bezieht sich auf: tos – (|**RS**|-i) im Wertekeller

  $$ bezieht sich auf Wert des Nichtterminalsymbols der **LS**

Stack-Indizierung in Yacc/Bison (neu)

$i ist auch für i ≤ 0 definiert

- bezieht sich auf Wertekellerpositionen **bevor** die aktuelle Regel zur Anwendung gekommen ist
- ACHTUNG: natürlich muss der Kontext beim Zugriff bekannt sein

$0

```
F   :   E B '+' E  { $$ = ... }
        ;
B   :   /* empty */ { temp_val=  $0 ;}
```

$0 refers to value of F in the previous production

Zwischenfazit

zwei Möglichkeiten in Bison
- a) direkter Zugriff auf Kellerposition mit Index- u. Typunsicherheit
- b) Verwendung globaler Größen

geg.

L-attributierte syntaxgesteuerte Definition (auf Basis einer LL-Grammatik)

dann

lässt sich bei Grammatik-Adaption die Definition auch für eine LR-Analyse einsetzen

Methode

(1) Starte mit Übersetzungsschema, das eingebettete Aktionen **vor** dem Nichtterminal platziert, um ererbte Attribute zu berechnen: $A \rightarrow \alpha \{a\} \beta$

(2) Führe jeweils verschiedene Nichtterminalsymbole M zur **Markierung** der Position der zu verschiebenden eingebetteten Aktionen ein: $A \rightarrow \alpha M \beta$
Dabei ist jedes M nur mit einer Produktion auszustatten: $M \rightarrow \varepsilon$,
wobei die Aktionen a ans Ende der rechten Seite platziert werden: $M \rightarrow \varepsilon \{a\}$

(3) Modifiziere dabei die Aktion a zu a', so dass
- alle Attribute von A oder Attribute von Symbolen aus α, die von a benötigt werden, als **ererbte** Attribute von M **kopiert** werden
- Berechne die Attribute so wie in a,
aber mache diese zu **synthetisierten** Attributen von M

nachdem Syntaxbäume aufgebaut sind, kann man

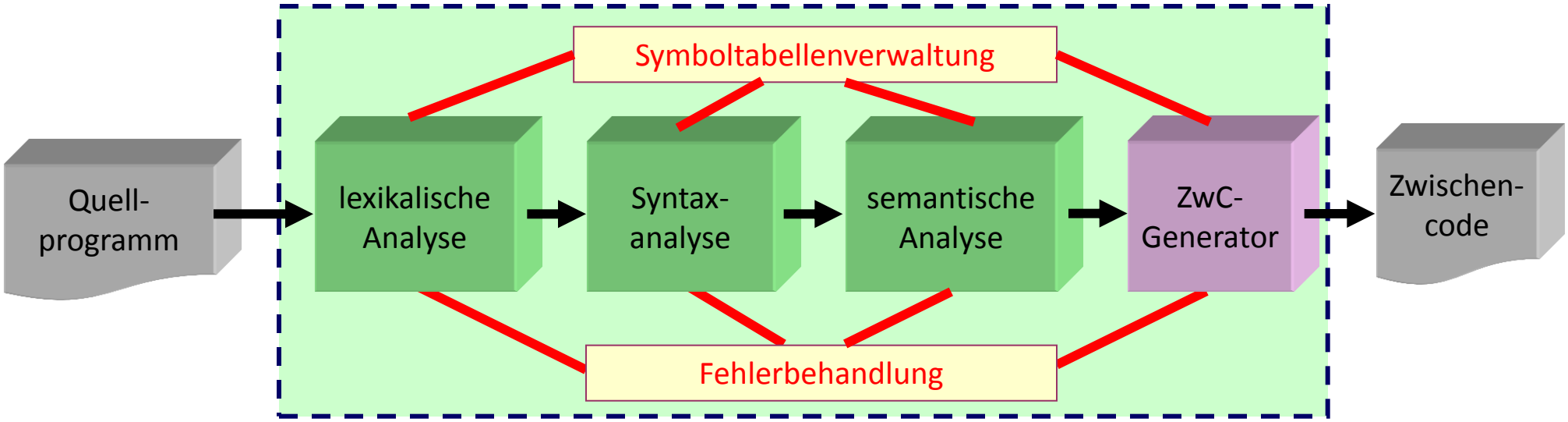
- sowohl Analyse- als auch
- Synthese-Aufgaben realisiert werden

Ziel des nächsten Unterkapitels:

semantische Aktionen zur Baumbearbeitung mit dem Nebeneffekt:

Ausgabe von 3-Adress-Code
(als beliebte Zwischencode-Form für Back-End-Tools)

Zwischencode-Erzeugung



Anforderungen an den Zwischencode

- leicht zu erzeugen
- leicht transformierbar in den letztendlichen Zielcode durch das Backend

```
temp1:= intToReal(60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1    := temp3
```

Zwischencode kann verschiedene Formen haben.

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerier

- 6.1
  Überblick: Grammatik-basierte Übersetzung
- 6.2
  Attributgrammatiken
- 6.3
  S-attributierte Syntaxdefinitionen
- 6.4
  Attributierte Syntaxdefinitionen mit synthetisierten und ererbten Attributen
- 6.5
  L-attributierte Syntaxdefinitionen
- 6.6
  Verfahren syntaxgesteuerter Übersetzungen im Überblick
- 6.7
  Entwurf syntaxgesteuerter Übersetzungen
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle

Drei-Adressbefehle (beliebte Zwischencodeform)

```
x = y op z          // op binärer Operator, x, y, z Adressen
x = op y            // op unärer Operator
```

allg. Format

```
x = y               // Kopierbefehl

goto L              // unbedingter Sprung zur „Marke“ L im Programm

if x goto L         // bedingte Sprünge
if False x goto L

if x relop y goto L // bedingter Sprung mit Vergleich mit relop als Vergleichsoperator

x   = y[i]          //Kopierbefehl mit Indizes
y[i] = x

x = &y              // Kopierbefehl mit Adressrechnung
x = *y
*y= x
```

Marke kann jedem Befehl als Präfix L: hinzugefügt werden

Beispiel-1: DreiAdress-Code (FOR-Anweisung)

FOR-Schleife

```
a=3;
b=4;
for(i=0;i<n;i++){
        a=b+1;
        a=a*a;
}
c=a;
```

in 3-Adresscode

```
        a=3;
        b=4;
        i=0;
L1:
        VAR1=i<n;
        if(VAR1) goto L2;
        goto L3;
L4:     i++;
        goto L1;
L2:     VAR2=b+1;
        a=VAR2;
        VAR3=a*a;
        a=VAR3;
        goto L4
L3:     c=a;
```

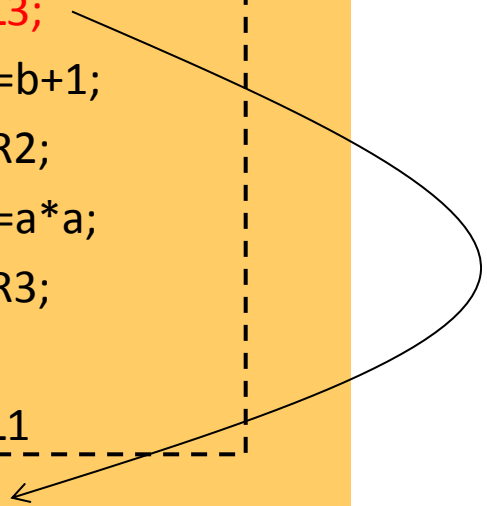
Beispiel-2: DreiAdress-Code (WHILE-Anweisung)

WHILE-Schleife

```
a=3;
b=4;
i=0;
while(i<n){
        a=b+1;
        a=a*a;
        i++;
}
c=a;
```

in 3-Adresscode

```
        a=3;
        b=4;
        i=0;
L1:
        VAR1=i<n;
        if(VAR1) goto L2;
        goto L3;
L2:     VAR2=b+1;
        a=VAR2;
        VAR3=a*a;
        a=VAR3;
        i++;
        goto L1
L3:     c=a;
```



Vorbereitung zur Übersetzung *in DreiAdress-Code*

jedes Nichtterminalsymbol der WHILE-Anweisungs-Regel erhält für Code-Generierung folg. Attribute:

- code: enthält den generierten 3ACode
- true: enthält Marke für Sprung mit assoziiertem Boolean-Ausdruck (falls true)
- false: (falls false)
- next: enthält Marke für Statement, das aus dem Nichtterminalsymbol abgeleitet wird

benutzte Funktion:

- newLabel(): liefert bei jedem Aufruf neue Marke
- gen(…): generiert Code (als String), der als Parameter übergeben wird

Beispiel: WHILE-Statement-Übersetzung *in DreiAdress-Code*

S → **if** C **then** S1
S → **if** C **then** S1 **else** S2
S → **while** C **do** S1
S → *assign-stmt*
→ …

```
P → S {
        S.next:= newLabel ;
        P.code:= S.code ||gen (S.next ':' ) ;
}
```

Bem.: konzentrieren uns lediglich auf den Programmfluss,
(blenden hier Zwischencode für andere Teile aus)

lokaleVariable

```
S → while E do S1
{
        L1:= newLabel ;
        L2:= newLabel ;
        C.false := S.next ;
        S1.next := L1 ;
        C.true := L2 ;
        S.code := gen(L1, ':') || C.code || gen(L2, ':') || S1.code ;
}
```

- code
- true
- false
- next

Attributierung zur Code-Repräsentation

- **Ann.:**
  Unsere code-Attribute der Meta-Symbole sind alle vom Typ **string**
  (anschaulich, aber praktisch ineffizient)

next
true
false
code

- eigentliche Anforderung:
  Werte von code müssen sich nur einfach verketten lassen
  (so wäre prinzipiell eine Liste beliebigen Typen denkbar)

  Bem.:
  praktisch interessantere Lösung wäre die sequentielle Ausgabe der Code-Stücke in eine Datei (zeigen wir später)

Implementierung attributierter Übersetzungsschemata

L1 : E-Code (next, L2)
L2 : S1-Code (L1)
next:

Problem: LR-Grammatik + **L**-attr.SyntaxDef

S → **while (C)** S1	L1= newLabel(); L2= newLabel(); S1.next= L1; C.false= S.next; C.true=L2; S.code= gen(L1 , ':') || C.code || gen(L2 , ':') || S1.code

Behandlung der Marken L1, L2
diese sind Variablen, keine Attribute
(so wie next)

Übersetzungsschema ist zu transformieren

S → **while (** { L1=newLabel(); L2= newLabel(); C.false= S.next; C.true=L2; }
C **)** {S1.next= L1; }
S1 {S.code= gen(L1, ':') || C.code || gen(L2 , ':') || S1.code }

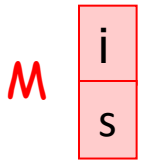
S → **while (** M C **)** N S1 {S.code= gen(L1 , ':') || C.code || gen(L2 , ':') || S1.code }
M → ε {L1=newLabel(); L2= newLabel(); C.false= S.next; C.true=L2; }
N → ε {S1.next= L1; }

Konvertierungsprinzip (Erinnerung)

Übersetzungsschema

$A \rightarrow \{B.i = f(A.i);\}\ B\ C$

Einführung eines Markers M
mit zwei Attributen



Kopie von A.i

Stellvertreter für B.i

Übersetzungsschema

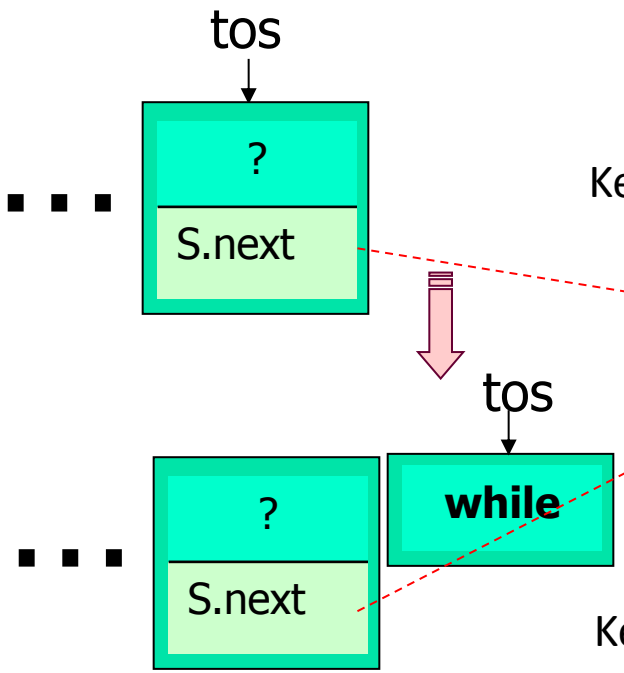
$A \rightarrow M\ B\ C$

$M \rightarrow \varepsilon\ \{M.i=A.i;\ M.s= f(A.i);\}$

Wie kommt man an
diese Information ?

Schritte des LR-Parsers (1)

Eingabe ... **while (C)** S1



Kellerbelegung unmittelbar vor der Erkennung von **while**

Sprungmarke als symbolisches Ende der aktuellen Anweisung S existiert bereits, wurde aber noch nicht im generierten Code verwendet!!!

Kellerbelegung sobald **while** aktuelles Eingabe-Token wird

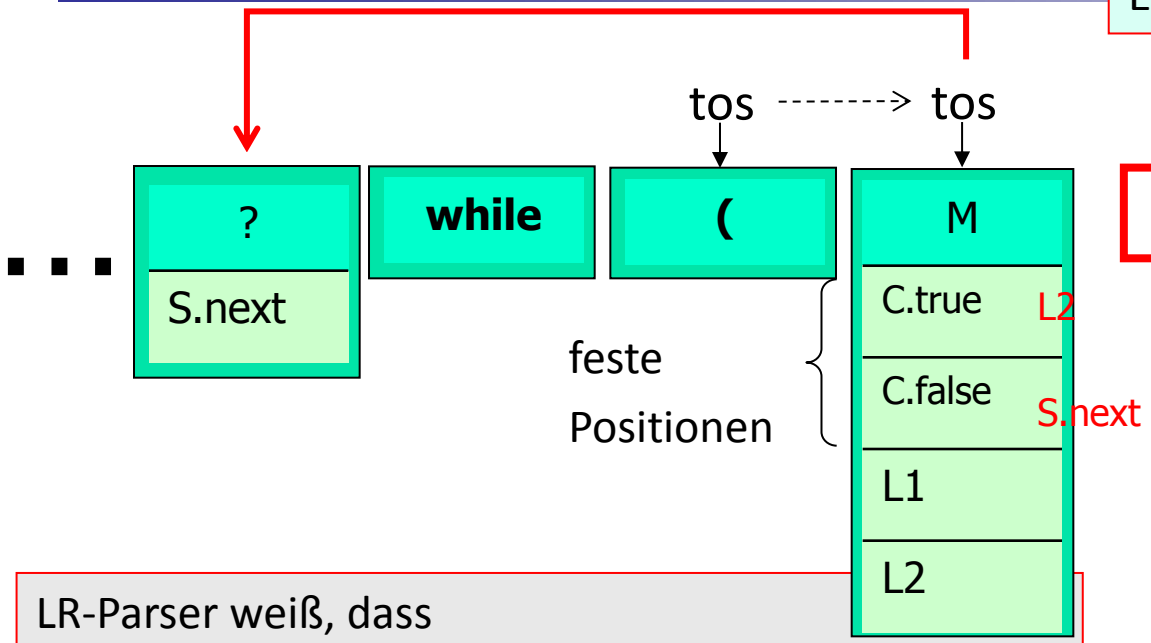
man kennt zwar **nicht** den darunter liegenden Zustand bzw. das entspr. Nichtterminalsymbol,

aber

es hat ein synchron gespeichertes Datenrekord,
das auf einer **festen** Position in Form von S.next vorliegt

Schritte des LR-Parsers (2)

Eingabe ... **while (C)** S1



$S \rightarrow$ **while** (M C) N S_1 { ... }

$M \rightarrow \varepsilon$ { ... }

$N \rightarrow \varepsilon$ { ... }

ursprüngliche Aktionen zur Berechnung der ererbten Attribute von C werden jetzt zu Aktionen von M

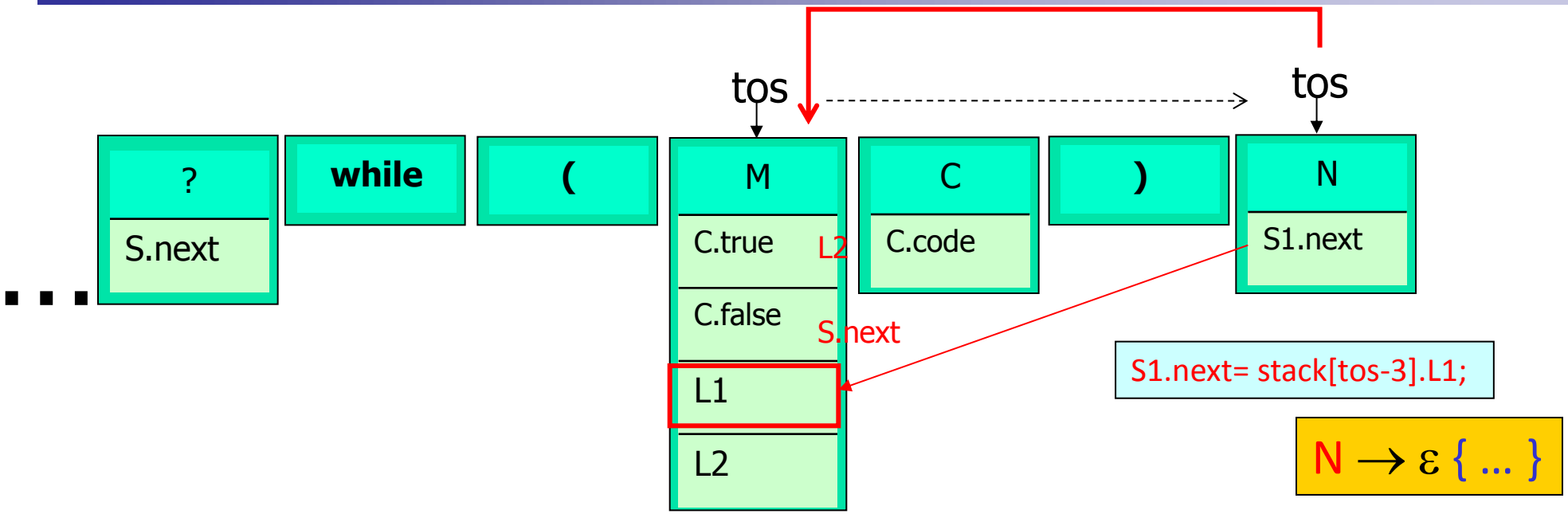
M-Aktionen

```
L1:= newLabel();
L2:= newLabel();
C.false:= stack[tos-3].next;        /*S.next*/
C.true:= L2;
```

LR-Parser weiß, dass
- es sich nach Lesen von (um eine while-Anweisung handelt
- und wird in dieser Situation eine Reduktion von ε nach M durchführen
- danach wird der Eingabecode, der später C ergibt, analysiert

L1 : C-Code (next, L2)
L2 : S1-Code (L1)
next:

Schritte des LR-Parsers (3)

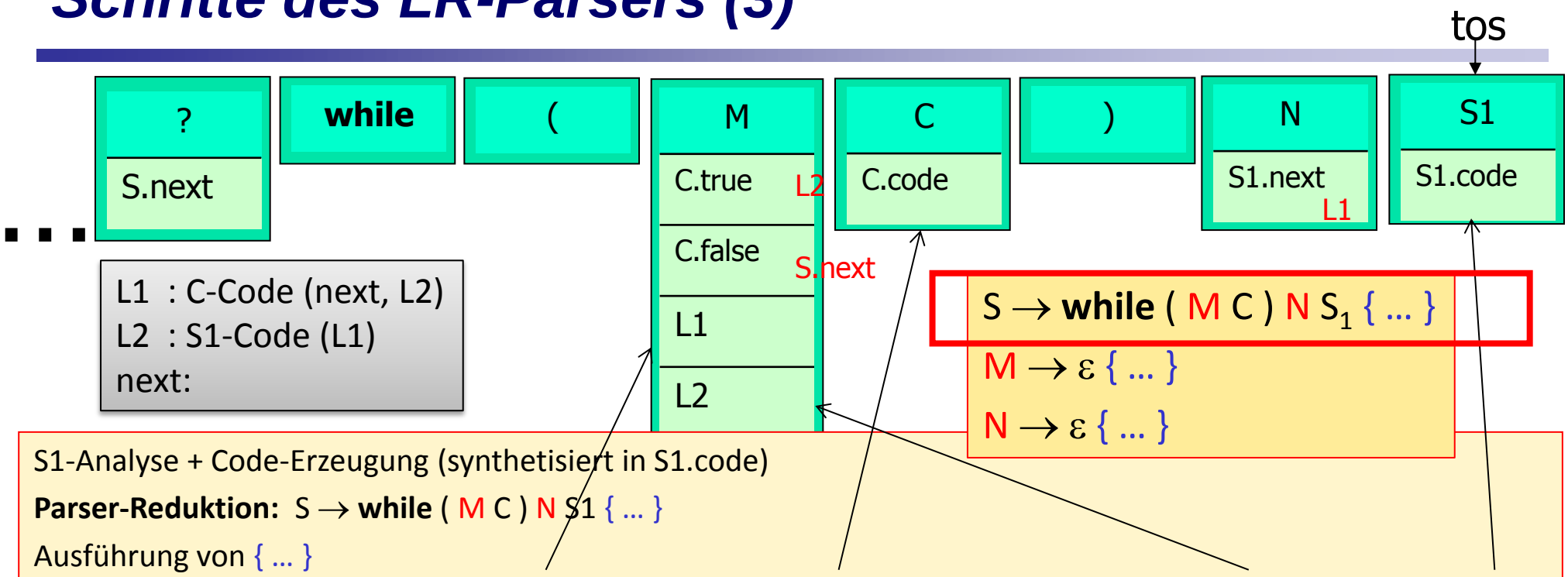


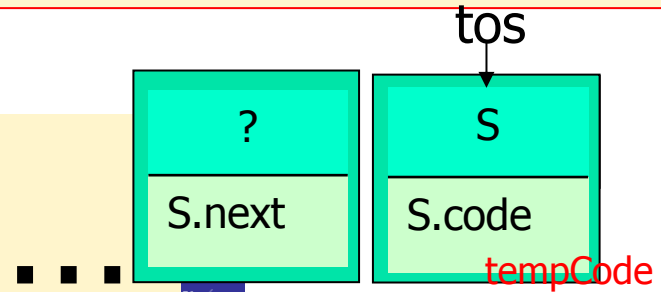
Ann.:

- nächste Eingabesymbole nach ((Bool-Ausdruckskomponenten) können zu C reduziert werden; das synthetisierte Attribut C.code ist im Datenrecord von C enthalten (wird erzeugt)
-) wird auf den Stack geschoben
- Parser weiß, dass er noch immer eine **while**-Anweisung bearbeitet (LL-Grammatik) und wird ε zu N reduzieren und die zugeordnete Aktion ausführen (hinterlegt für kommende S1-Analyse L1 als Rücksprungmarke)

Schritte des LR-Parsers (3)

Eingabe ... **while** (C) S1





S.next= Label für Sprungmarke als symbolischer Anfang der nächsten Anweisung S (== Ende der while-Anweisung)