

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Vorlesung Compilerbau: Semantische Bearbeitung

Vorlesung des Grundstudiums  
Prof. Johann Christoph Freytag, Ph.D.  
Institut für Informatik, Humboldt-Universität zu Berlin  
SoSe2018



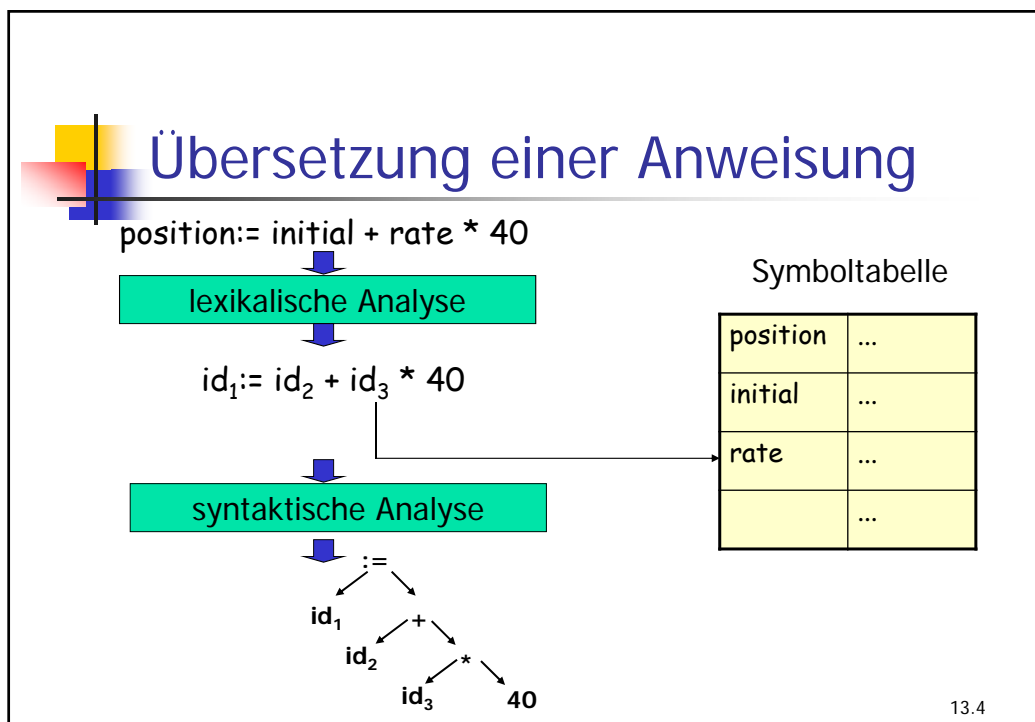
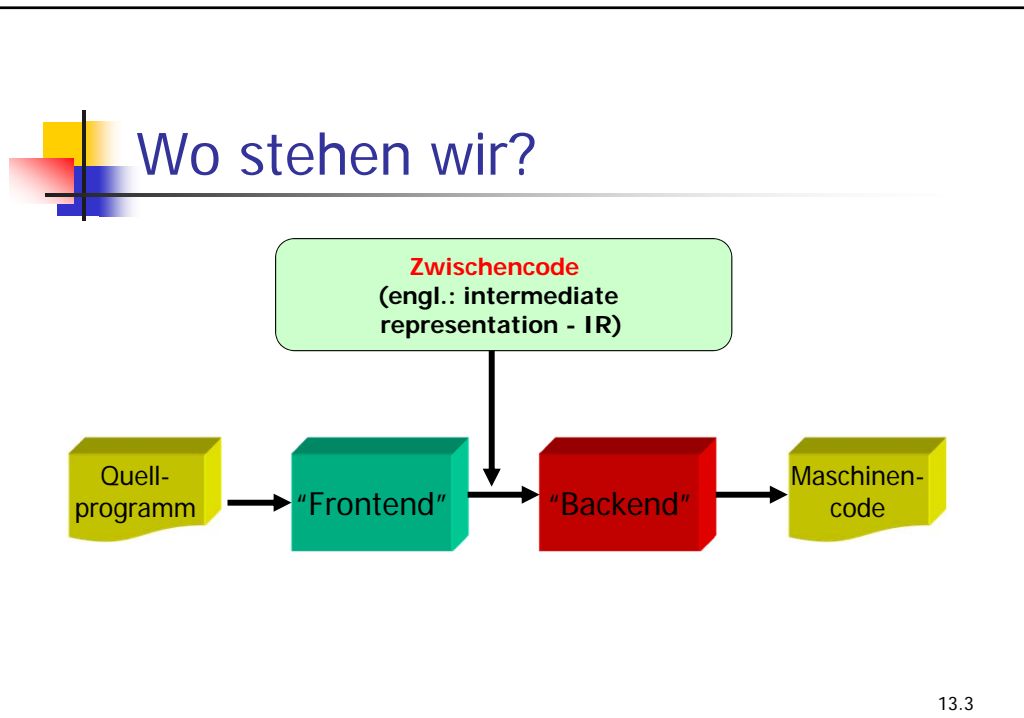
### Ziel des Kapitels

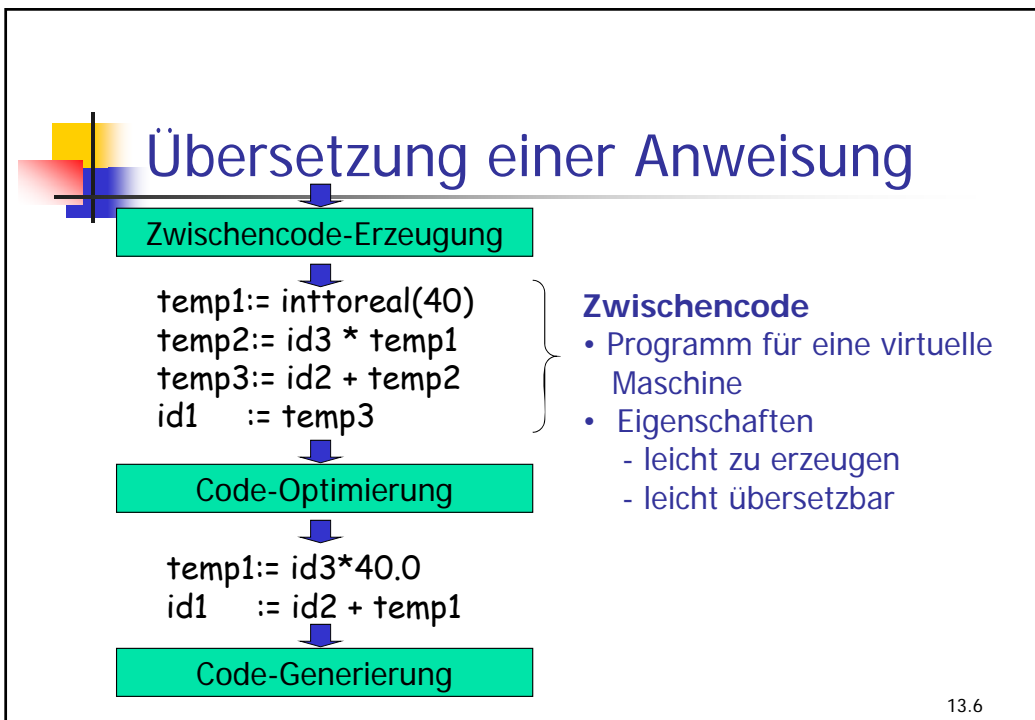
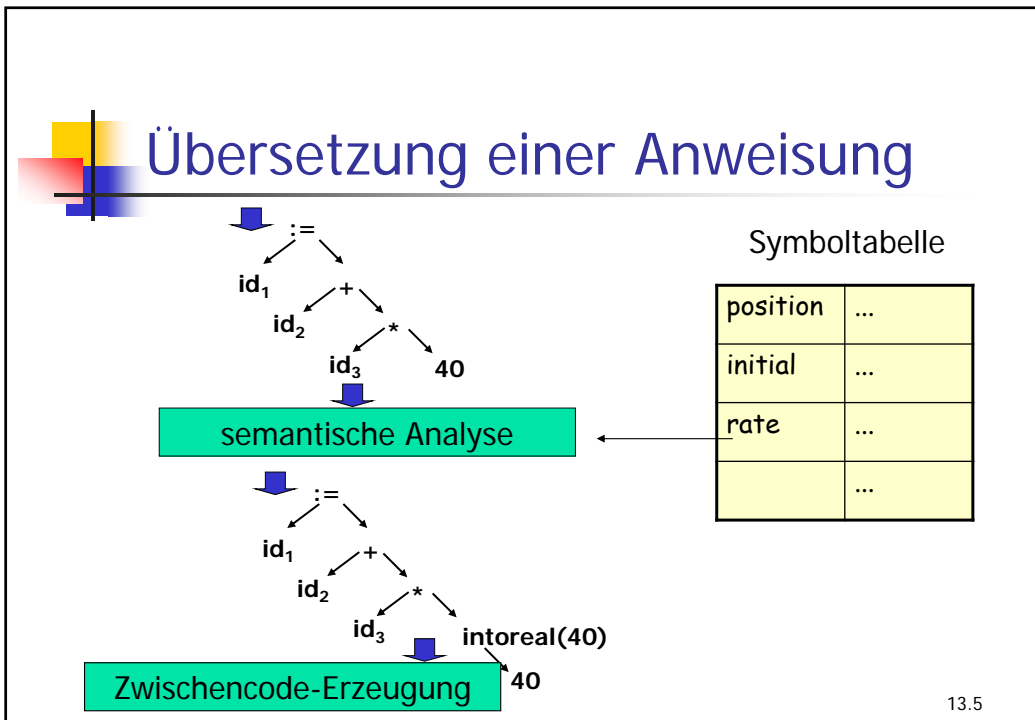
#### Semantische Analyse

- Phasenorientierte Bearbeitung
- Darstellungsformen für Programme
- Zwischencoderepräsentation
- Semantische Aktionen
- Symboltabelle

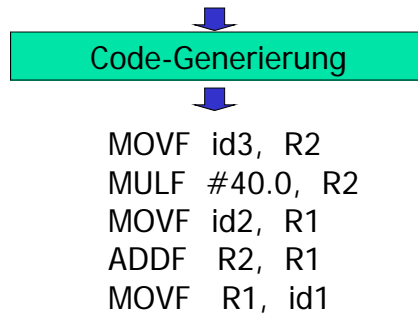
# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung





## Übersetzung einer Anw. (cont.)



13.7

## Semantische Analyse

- Überprüfung des Quellprogramms auf semantische Fehler
- Sammlung von Typ-Informationen für Codegenerierung
- Nutzung der hierarchischen Struktur der Syntaxanalyse (Operator- u. Operandenbestimmung von Ausdrücken u. Anweisungen)

13.8



## Semantische Analyse (Forts.)

**Übersetzungsprozess** wird durch die syntaktische Struktur des Programms getrieben, so wie sie vom Parser konstruiert wird

### Semantische Routinen

- interpretieren die Bedeutung des Programms auf der syntaktischen Struktur
- haben zwei Ziele:
  1. Vervollständigung der Analyse durch kontextsensitive Informationen
  2. Beginn der Synthese durch die Erzeugung der Zwischenrepräsentation bzw. des Zielcodes
- werden mit einzelnen Produktionsregeln der kontextfreien Grammatik oder mit einem (Teil-)Baum des Syntaxbaumes assoziiert

13.9



## Kontextsensitive Analyse

Was sind die kontextsensitiven Aufgaben?

- Ist *x* ein Skalar, ein Array oder eine Funktion?
- Wurde *x* vor seiner Nutzung definiert?
- Gibt es Namen, die nie benutzt wurden?
- Welche Deklaration von *x* ist mit dieser Referenz gemeint?
- Ist ein Ausdruck typkonsistent?
- Stimmt die Dimensionsangabe von *foo* mit der Deklaration überein?
- Wo soll *x* gespeichert werden (Keller, Heap)?
- Werden die Array-Grenzen eingehalten? (zumindest bei Konstanten)
- Produziert die Funktion *foo* einen konstanten Wert?

Diese Fragen können nicht durch eine cf Grammatik beantwortet werden

13.10

## Kontextsensitive Analyse (Forts.)

### Warum ist kontextsensitive Analyse schwer?

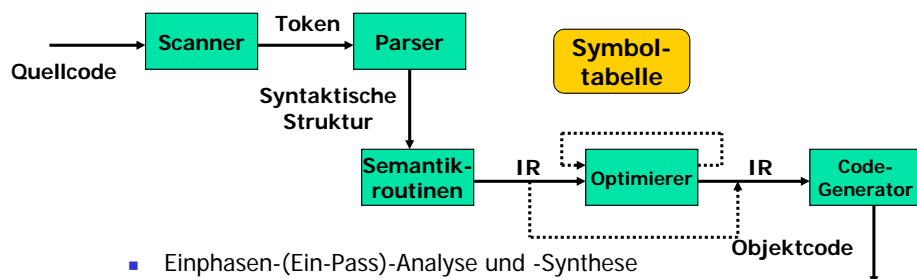
- Antwort ist abhängig von Werten (Inhalten) nicht von Struktur
- Fragen und Antworten betreffen nicht-lokale Informationen
- Antwort muss möglicherweise berechnet werden

### Verschiedene Alternativen & Strukturen

- **abstrakter Syntaxbaum**: spezifiziere nicht-lokale Berechnung
- **Attribut-Grammatik**: automatische Ausführung (später)
- **Symboltabelle**: Zentraler Speicher für »Fakten«

13.11

## Alternativen der Bearbeitung



- Einphasen-(Ein-Pass)-Analyse und -Synthese
- Einphasen-Compiler plus Guckloch-Optimierung
- Einphasen-Analyse & IR-Synthese + Codegenerierungsphase
- Mehrphasen-Analyse
- Mehrphasen-Synthese

13.12

# Vorlesung Compilerbau (SoSe2018)


## Teil 13: Semantische Bearbeitung



### Einphasen-Compiler

- Überlappung von Scannen, Parsen, Überprüfung und Übersetzung
- Kein expliziter Zwischencode (IR)
- Generiert Zielcode direkt (sofort)
  - Erzeugung von kleinen Code-Sequenzen für jede Parseraktion (Symbolerkennung für vorhersehbares Parsen/LR-Reduktion)
  - Konsequenz: Wenig Optimierung möglich (minimaler Kontext)
- Möglich: Phase der „**Peephole**“-Optimierung
  - Zusätzliche Phase zur Optimierung der generierten Codes über ein Fenster (Guckloch, engl. peephole) von Instruktionen
  - Glätten der »Bruchkanten« zwischen den Teilen des Codes, die unabhängig voneinander erzeugt wurden

13.13



### Einphasenanalyse/-synthese und Codegenerierung

#### Generierung von Zwischencode als Schnittstelle zum Codegenerator

- Lineare Anordnung in Form von »Tupeln«
- Alternativen der Codegenerierung
  - »one tuple at a time«
  - mehrere Tupel für überschaubaren Kontext und besseren Code

#### Vorteil:

- Backend ist unabhängig vom Frontend
  - Kann besser an neue HW-Umgebungen angepasst werden
  - IR muss ausdrucksfähig genug für unterschiedliche HW-Umgebungen sein
- später können Optimierungsphasen hinzugefügt werden (Mehrphasenanalyse und -optimierung)

13.14

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Mehrphasen-Analyse

Historische Motivation:

- Begrenzter Speicherplatz vorhanden

Mehrere Phasen, die ihre Ausgabe in eine Datei schreiben

1. Einlesen des Quellcodes generiert Token (**plus** Einfügen der Bezeichner und Konstanten in die Symboltabelle)
2. Token-Datei parsen
  - führe semantische Aktionen aus und linearisiere den Parse-Baum
3. Ausgabe des Parsers
  - Verarbeitung der Deklarationen, die eine Datei der Symboltabelle erzeugen
  - semantische Überprüfung mit Codeerzeugung oder linearem Zwischencode (Mehrphasensynthese)

13.15



### Mehrphasen-Analyse (Forts.)

Weitere Gründe für Mehrphasenanalyse  
(neben Notwendigkeit für Zwischenspeicherung)

- Sprache erfordert mehrere Phasen:
  - Beispiel: Deklarationen nach Nutzung von Bezeichnern
    - **scan** und **parse** zur Erzeugung der Symboltabelle
    - semantische Überprüfung und Code- oder IR-Erzeugung
- einfachere Realisierung/Implementierung
  - erst Scanning, Parsing und baumartige IR-Erzeugung
  - danach ein oder mehrere Phasen der semantischen Analyse und Codeerzeugung auf der Baumstruktur (Synthese)

13.16



# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Mehrphasensynthese

- Phasen arbeiten auf linearem oder baumartigem Zwischencode (IR)
  - Codegenerierung oder »Guckloch«-Optimierung
  - Mehrphasentransformation auf IR: maschinenunabhängige und maschinenabhängige Optimierungen
  - Übersetzung von »high-level«-maschinenunabhängigen IR zu »lower-level« IR vor der Codegenerierung
  - sprachenabhängiges Frontend: zunächst Übersetzung in »high-level«-IR
  - Backends für unterschiedliche HW-Umgebungen (zunächst Transformation in »low-level« IR)

13.17



### Zwischencode

#### Warum Zwischencode ?

- *Aufteilung* des Compilers in »handhabbare« Teile (gutes SW-Engineering)
- Ein vollständiger Durchlauf des gesamten Quellprogramms, ehe Zielcode ausgegeben wird  
Compiler hat möglicherweise mehr als eine Option für die Zielcodeerzeugung
- *Vereinfachung* bei der Portierung des Compilers auf eine neue HW  
trennt Backend vom Frontend
- Vereinfacht das Problem der »Poly-Architektur«  
m Sprachen, n HW-Umgebungen  $\Rightarrow$  m+n Komponenten
- erlaubt maschinenunabhängige Optimierungen

**Nochmals:** Zwischencode (IR = Intermediate Representation) ist eine Datenstruktur, die zur Übersetzungszeit erzeugt und genutzt wird

13.18



## Zwischencode (Forts.)

- Mögliche IR-Formen entsprechend der Literatur
  - Abstrakte Syntaxbäume (AST)
  - Linearisierte Form des (Operator-) Baumes
  - Gerichteter azyklischer Graph (DAG)
  - Kontrollfluss-Graph
  - Programm-Abhängigkeitsgraph
  - Drei-Address-Code
  - Hybride Kombinationen
- Unterschiedliche IR für unterschiedliche Programmteile

13.19



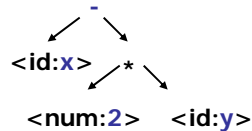
## Zwischencode (Forts.)

- Wichtige IR-Eigenschaften
  - einfache Generierung
  - einfache Manipulation (für Optimierung)
  - mögliche Abschätzung der Kosten für Manipulation
  - Abstraktionsebene: nicht zu detailliert, nicht zu einfach
  - »Freiheitsgrade« bei (verschiedenen) Darstellungsmöglichkeiten
- »kleine« Unterschiede bei IR-Entwurfsentscheidungen
  - haben weitreichende Konsequenzen für die Schnelligkeit und die Möglichkeiten des Compilers
  - Detaillierungsgrad ist eine wichtige Entwurfsentscheidung

13.20

### Abstrakter Syntaxbaum

- ein abstrakter Syntaxbaum (AST) ist ein komprimierter Parse-Baum, bei dem (fast alle) Knoten für Nicht-Terminalsymbole entfernt wurden



Operatoren und Schlüsselwörter kommen in einem AST nicht als Blätter vor

- Repräsentation für "**x - 2 \* y**"
- einfache Manipulation möglich
- Linearisierung möglich
  - Beispiel: "**Postfix Form**": **x 2 y \* -**

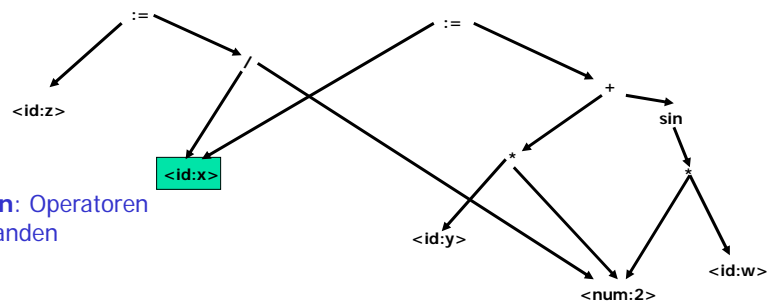
13.21

### Gerichteter azyklischer Graph (DAG)

Ein AST wird repräsentiert durch ein DAG (Directed Acyclic Graph) mit einem eindeutigen Knoten für jeden Wert/jede Variable

**Beispiel:**

- **x** := 2 \* y + sin(2 \* w)
- z := **x** / 2



innere Knoten: Operatoren  
Blätter: Operanden

13.22

# Vorlesung Compilerbau (SoSe2018)

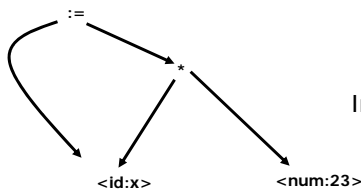
## Teil 13: Semantische Bearbeitung

### Gerichteter azyklischer Graph (Forts.)

Speicherung in einem Array;

**Beispiel:**

$x := x * 23$



1	id				
2	num	23			
3	*	1	2		
4	:=	1	3		
5	...				

zum  
Eintrag für x

Index ist Zeiger auf Knoten

- Knoten 3 hat das Label \*
- linker Nachfolger ist Knoten 1
- rechter Nachfolger ist Knoten 2

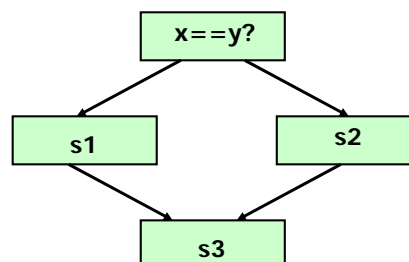
13.23

### Kontrollflussgraph

Der Kontrollflussgraph (CFG) modelliert die Abarbeitung von »Basisblöcken« innerhalb eines Programms

- Knoten des Graphen stellen »Basisblöcke« des Codes dar
- Kanten im Graph repräsentieren den Kontrollfluss, z.B. Sequenz, Schleifen, if-then-else, case und goto

if (x==y) then  
  s1  
else  
  s2  
s3



13.24

### Drei-Adress-Code

Drei-Adress-Code existiert in verschiedenen Formen

- im Allgemeinen sind dies Zuweisungen der Form  
 $x \leftarrow y \text{ op } z$   
mit einem einzigen Operator und maximal drei Namen
- Beispiel: Ausdruck  
 $x - 2 * y$   
wird transformiert in  
 $t1 \leftarrow 2 * y$       /\* t1 und t2 sind vom Compiler generierte Namen \*/  
 $t2 \leftarrow x - t1$
- Vorteil  
kompakte Form (mit expliziten Namen)  
Namen für Zwischenergebnisse
- ist linearisierte Form eines AST oder DAG

13.25

### Drei-Adress-Code (Forts.)

**Arten des Drei-Adress-Codes** (verwandt mit Assembler-Code)

- Zuweisung:  $x \leftarrow y \text{ op } z$
- Zuweisung:  $x \leftarrow \text{op } y$
- Zuweisung:  $x \leftarrow y[i], x[i] \leftarrow y$
- Zuweisung:  $x \leftarrow y$
- unbedingte Verzweigung: **goto L**
- bedingte Verzweigung: **if x relop y goto L** (relationaler Operator: <, =, ...)
- Prozeduraufruf: **param x, call p**  
und evtl. **return y**
- Adress- und Zeigerzuweisung:  
 $x \leftarrow \&y, x \leftarrow *y$  und  $*x \leftarrow y$

typische Verwendung:  $p(x_1, \dots, x_n)$

```
param x1
param x2
...
param xn
call p, n
```

13.26

### Drei-Adress-Code (Forts.)

#### Quadrupel

x - 2 * y				
(1)	load	t1	y	
(2)	load	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

#### Drei-Adress-Anweisung-Implementierungsstruktur

- auch für unäre Operatoren
- Sprünge setzen; Zielmarke im Ergebnis(result)-Feld
- arg1, arg2 sind Zeiger auf Symboltabelleneinträge

- einfache Record-Struktur mit 4 Feldern (op, result, arg1, arg2)
- einfache Umordnung möglich
- explizite Namen für Speicherzellen

13.27

### Drei-Adress-Code (Forts.)

#### Tripel

x - 2 * y			
(1)	load	y	
(2)	load	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

#### Ziel

- Verhinderung, dass temporäre Variablen in die Symboltabelle eingetragen werden

- Record-Struktur mit 3 Feldern (op, arg1, arg2)
- Nutzung des Tabellenindex als impliziten Namen (... ) sind Zeiger in die Tripel-Struktur
- schwieriger umzuordnen (alle Referenzen ändern sich)

13.28

## Drei-Adress-Code (Forts.)

### indirekte Tripel

x - 2 * y				
no	stmt	op	arg1	arg2
(1)	(100)	load	y	
(2)	(101)	load	2	
(3)	(102)	mult	(100)	(101)
(4)	(103)	load	x	
(5)	(104)	sub	(103)	(102)

- Adressen unabhängig von Reihenfolge
- Vereinfacht die Umordnung von Zuweisungen
- mehr Platz als für einfache Tripel notwendig
- implizite Namensvergabe und -verwaltung notwendig

13.29

## Hybride Darstellungen

- Mischung bisheriger Techniken
  - Graphnutzung, wo notwendig
  - linearer Code, wenn ausreichend
  - keine generelle Theorie/Übereinstimmung, wann was zu nutzen ist
- verschiedene Implementierungen mit unterschiedlichen Ergebnissen

13.30



## Zwischendarstellung

umfasst mehr als nur Codedarstellung:

- **Symboltabelle**
  - Bezeichner für Variablen, Prozeduren etc.
  - Größen-, Typangabe, Lokation (symbolische Adresse)
  - Angabe über Gültigkeitsbereich
- **Konstantentabelle** (typischerweise für Zeichenketten)
  - Repräsentation, Typangabe
  - Ort der Speicherung, Offsets
- **Übertragung (mapping) in den Speicher**
  - Speicherorganisation
  - (virtuelle) Registerallokation

13.31



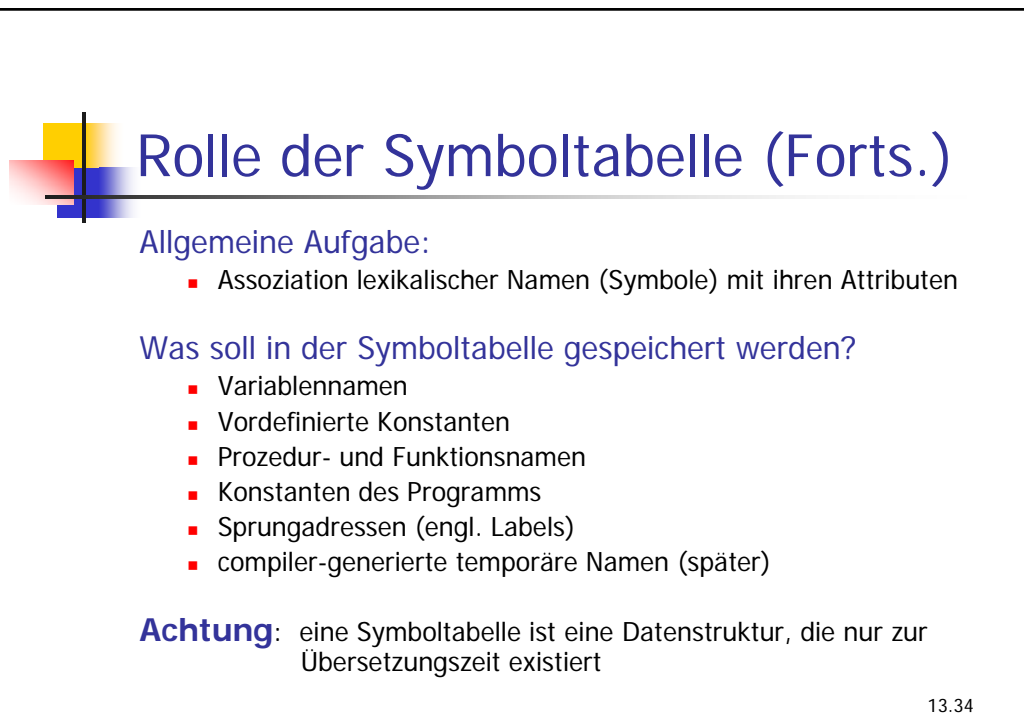
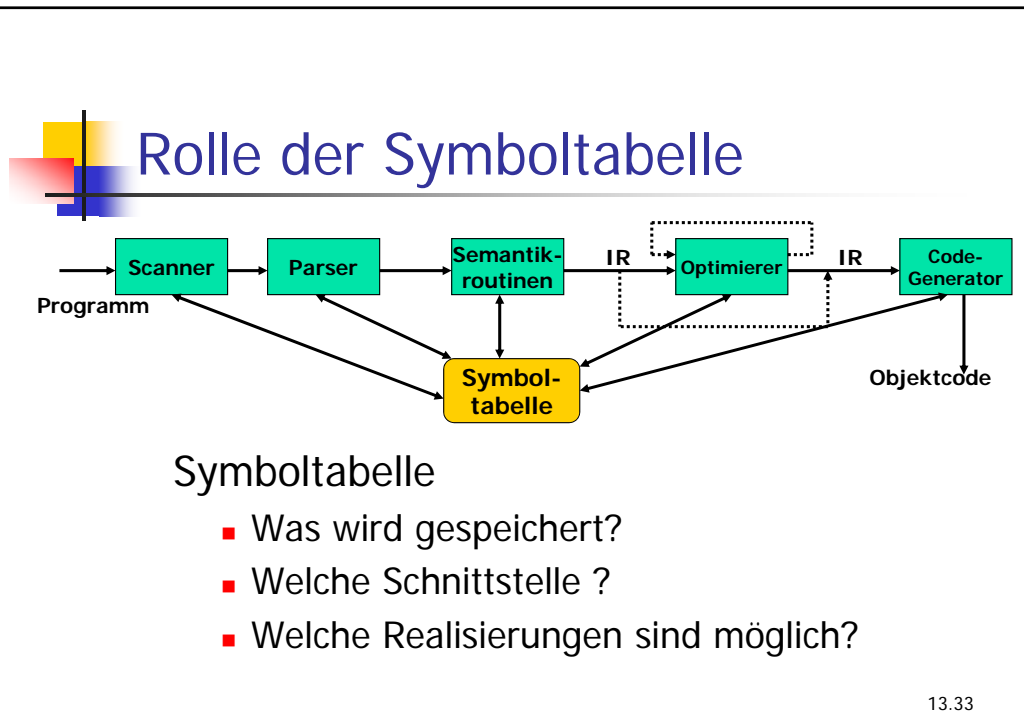
## Symboltabelle

13.32



# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung





## Rolle der Symboltabelle (Forts.)

Welche Art von Informationen braucht ein Compiler?

- Namen (Zeichenkette)
- Datentypen
- Dimensionsinformation (Felder)
- Prozedur-/ Funktionsdeklaration
- Anzahl und Typ der Argumente von Prozeduren/Funktionen
- Sichtbarkeitsbereich der Deklarationen
- Speicherklasse (static, extern, global, ...)
- Offset im Speicher
- falls Name eines Records: dann Strukturbeschreibung
- falls Parameter: call-by-value, call-by-reference

13.35




## Implementierung der Symboltabelle

### Alternativen

- Lineare Liste
  - Komplexität  $O(n)$  Tests pro Suchoperation
  - einfach zu erweitern, keine feste Größe
  - ein neuer Eintrag pro neu eingefügtes Element
- Geordnetes lineares Feld
  - Komplexität  $O(\log_2 n)$  Tests pro Suchoperation mit binärer Suche
  - Einfügen ist aufwendig, da Reihenfolge erhalten bleiben muss
- ...

13.36




## Implementierung der Symboltabelle (Forts.)

### Alternativen

- ...
- Binärer Baum
  - Komplexität  $O(n)$  Tests pro Suchoperation bei nicht-balancierten Bäumen
  - Komplexität  $O(\log_2 n)$  Tests pro Suchoperation bei balancierten Bäumen
  - einfach zu erweitern, keine feste Größe
  - ein neuer Eintrag pro eingefügtes Element
- Hash-Tabelle
  - Komplexität  $O(1)$  Tests pro Suchoperation (meistens, abh. vom Füllstand)
  - Einfügen ist unterschiedlich schwierig (abh. vom gewählten Verfahren)

13.37



## Inhalt der Symboltabelle

Was muss in der Symboltabelle zu finden sein?

- Informationen für Namen/Identifikatoren

Darüber hinaus....

- Verwaltung von Gültigkeitsbereichen!!
  - wodurch entstehen Gültigkeitsbereiche?
    - Definition eines Moduls/Programms, einer Funktion/Prozedur/ eines Blocks
  - können geschachtelt sein
  - Fragen zur Analysezeit nach einem Namen, muss die in diesem Gültigkeitsbereich gültige Deklaration mit ihren Informationen zurückgegeben werden, oder
  - Deklaration aus einem "äußeren" Gültigkeitsbereich
  - innerer Gültigkeitsbereich überschreibt möglicherweise Deklarationen in äußeren Bereichen

13.38

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Gültigkeitsbereich: Beispiel 1

```
program sort (input, output) ;
  var a : array [0..10] of integer;
  x : integer;

  procedure readarray;
    var i : integer;
    begin ..... a[i] ..... end {readarray} ;

  procedure exchange ( i, j : integer);
    begin {exchange }
      x := a[i];
      a[i] := a[j];
      a[j] := x
    end { exchange };
```

```
procedure quicksort (m, n : integer );
  var k, v : integer;

  function partition (y, z : integer): integer;
    var i, j : integer;

    begin ..... a .....
      ..... v .....
      ..... exchange (i, j); .....
    end {partition } ;

  begin {quicksort}
    .....
  end {quicksort };

begin {sort } ..... end {sort } .
```

13.39



### Gültigkeitsbereich: Beispiel 2

```
program foo (input, output);

  procedure b (function h (i: integer): integer);
    begin writeln (h(2)) end { b} ;

  procedure c;
    var m : integer;
    function f (n : integer): integer;
      begin f := m + n end { f };

    begin m := 0; b(f) end { c } ;

  begin
    c
  end { foo }
```

13.40



## Inhalt der Symboltabelle (Forts.)

### Eigenschaften von Gültigkeitsbereich (GB)

- Neue Deklarationen können nur im jetzigen GB definiert werden

### Welche Operationen?

- void put (Symbol key, Object value)  
Binden des Schlüssels (Namen) an einen (komplexen) Wert
- Object get (Symbol key)  
Auffinden des (komplexen) Wertes für einen gegebenen Schlüssel
- void beginScope ()  
Erzeuge neuen GB
- void endScope ()  
Schließe (und lösche) momentanen GB und setze den nächst äußeren GB als den jetzt gültigen

13.41



## Inhalt der Symboltabelle (Forts.)

Symboltabelle assoziiert Namen mit Attributen  
(komplexe Werten)

- Attribute beschreiben zur Übersetzungszeit Eigenschaften einer Deklaration
- Attribute unterscheiden sich je nach der Bedeutung des Namens
  - **Variablennamen:** Typ, Prozedurebene, Speicherinfo (Rahmen)
  - **Typen:** Beschreibung des Typs, Größe und Speicheranforderungen: Alignment
  - **Konstanten:** Typ, Wert
  - **Prozeduren/Funktionen:** Formale Namen und Typen, Ergebnistyp, Speicherinfos, Rahmengröße

13.42



---

## Semantische Aktionen

13.43



---

## Semantische Aktionen

- Parser
  - muss mehr als accept/reject als Ergebnis zurückgeben
  - initialisiert auch die Übersetzung durch **semantische Aktionen**
- semantische Aktionen:  
sind Routinen, die durch den Parser bei der Erkennung eines syntaktischen Symbols ausgeführt werden
  - jedes Symbol hat einen assoziierten semantischen Wert
  - unterschiedliche Handhabung für LL- und LR-Parser

13.44



## LL-Parser und Aktionen

Wie werden Aktionen vom LL-Parser ausgeführt?

- Regeln werden vor dem Scannen der rechten Seite einer Regel expandiert
- Daher werden Aktionen wie alle anderen Grammatiksymbole auf dem Keller gespeichert (push)
- Erscheinen diese als oberstes Kellerzeichen (tos), wird die entsprechende Aktion ausgeführt

13.45



## LL-Parser und Aktionen (Forts.)

```
push EOF;
push Start_Symbol;
token ← next_token();
repeat
  pop X;
  if X is a terminal or EOF then
    if X is_token then
      token ← next_token();
    else error()
  else if X is_an action then
    perform X
  else /* X is a non-terminal symbol */
    if M[X, token] == X → Y1Y2...Yn then
      push(ak, Yn, ak-1, Yn-1, ..., a1, Y1) ← Aktionen ai
    else error();
until X == EOF
```

13.46



### LR-Parser und Aktionen

Wie werden Aktionen vom LR-Parser ausgeführt?

- Scannen der gesamten rechten Seite einer Regel (RHS), ehe die Regel zur Reduktion angewandt wird
  - deshalb kann – *eigentlich* - Aktion erst nach dem Scannen der kompletten RHS der Regel angewandt werden
  - deshalb kann eine Aktion nur als letztes »Symbol« einer Produktion erscheinen
  - impliziert neue Nicht-Terminalsymbole, um diese Restriktion zu umgehen
    - aus:  $A \rightarrow w \text{ action } \beta$  wird:  $A \rightarrow M\beta$  und  $M \rightarrow w \text{ action}$
  - wird von YACC und BISON automatisch erzeugt

13.47



### Aktionsgesteuerter Keller

Ansatz zur Semantikaktionsausführung

- Semantikaktionen arbeiten auf (separatem) Semantikkeller
- Aktionen erhalten Argumente explizit vom Keller
- Aktionen platzieren Ergebnis wiederum auf dem Keller

Vorteil

- Aktionen haben Zugriff auf den gesamten Keller (ohne pop)

Nachteil

- Implementierung der Aktionen ist direkt an Keller orientiert
- Aktionen enthalten expliziten Code zur Handhabung des Kellers

Alternative

- Kellerimplementation wird hinter push/pop-Schnittstelle verborgen
- Zugriff auf den Keller erfordert explizite pop-Operation, d.h. Kopieren des Wertes und damit häufig Leistungsverlust
- Weiterhin expliziter Kellerzugriff bei Fehlerbehandlung notwendig

13.48



# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Attribut-Grammatiken

Ziel: **Implementierungsunabhängige** Spezifikation semantischer Aktionen

Idee:

- Definition von Attributen/Feldern für jeden Knoten im Syntaxbaum
- Spezifikation von Gleichungen zur Definition von (eindeutigen) Werten
- Attribute des Eltern- und der Kinderknoten können benutzt werden

Beispiel:

- Füge ein Typ- und ein Klassenattribut zu Knoten hinzu, die Ausdrücke darstellen
  - **type** ist der (abgeleitete oder definierte) Typ eines Wertes/Variablen/Ausdrucks
  - **class** spezifiziert, ob es sich um eine Variable/Konstante/Ausdruck/ ... handelt
- »Überprüfungsregeln« (Gleichungen) für den Zuweisungsknoten "!="
  - Überprüfe, dass LHS.class eine Variable ist (und keine Konstante)
  - Überprüfe, dass LHS.type und RHS.type konsistent in ihren Type-Werten sind

13.49

### Attribut-Grammatiken (Forts.)

**Idee** wird durch Attribut-Grammatiken formalisiert  
von *Donald Knuth* vorgeschlagen und formalisiert



#### Vorgehensweise

- Grammatikbasierte Spezifikation der Baumattribute
- Wertezuweisungen für Baumattribute werden mit den Produktionsregeln festgelegt
- Jedes Attribut erhält einen eindeutig festgelegten Wert
- Bezeichne identische Terme eindeutig

#### Beobachtung

- Attribut-Grammatiken können kontextsensitive Aktionen implementationsunabhängig definieren (unabh. ob LR- oder LL-Parser)

13.50

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel für Attribut-Grammatik

Produktion	Semantische Regel
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in$ $addtype(id.entry, L.in)$
$L \rightarrow \text{id}$	$addtype(id.entry, L.in)$

13.51

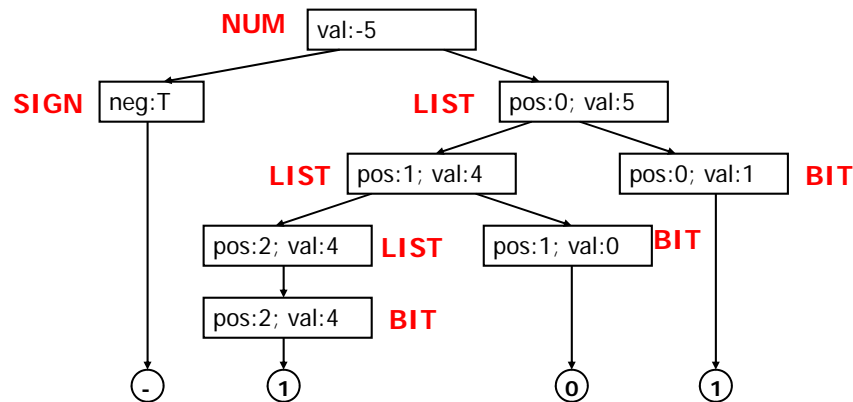
### Beispiel: Evaluierung von binären Zahlen mit Vorzeichen

Produktionsregel	Semantische Regel
$NUM \rightarrow SIGN LIST$	$LIST.pos := 0$ if $SIGN.neg$ then $NUM.val := - LIST.val$ else $NUM.val := LIST.val$
$SIGN \rightarrow +$	$SIGN.neg := \text{false}$
$SIGN \rightarrow -$	$SIGN.neg := \text{true}$
$LIST \rightarrow BIT$	$BIT.pos := LIST.pos$ $LIST.val := BIT.val$
$LIST \rightarrow LIST_1 BIT$	$LIST_1.pos := LIST.pos + 1$ $BIT.pos := LIST.pos$ $LIST.val := LIST_1.val + BIT.val$
$BIT \rightarrow 0$	$BIT.val := 0$
$BIT \rightarrow 1$	$BIT.val := 2^{BIT.pos}$

13.52

## Beispiel (Forts.)

### ■ Attribut-Parse-Baum für -101



13.53

## Abhängigkeiten von Attributen

### Abhängigkeitsbeziehungen zwischen Attributen

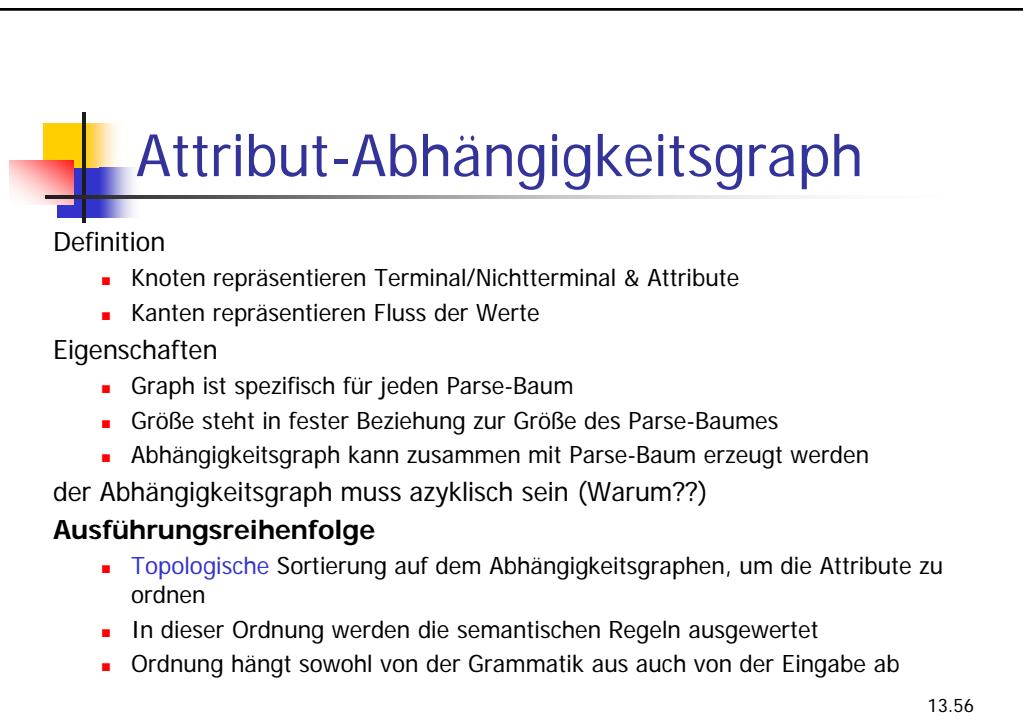
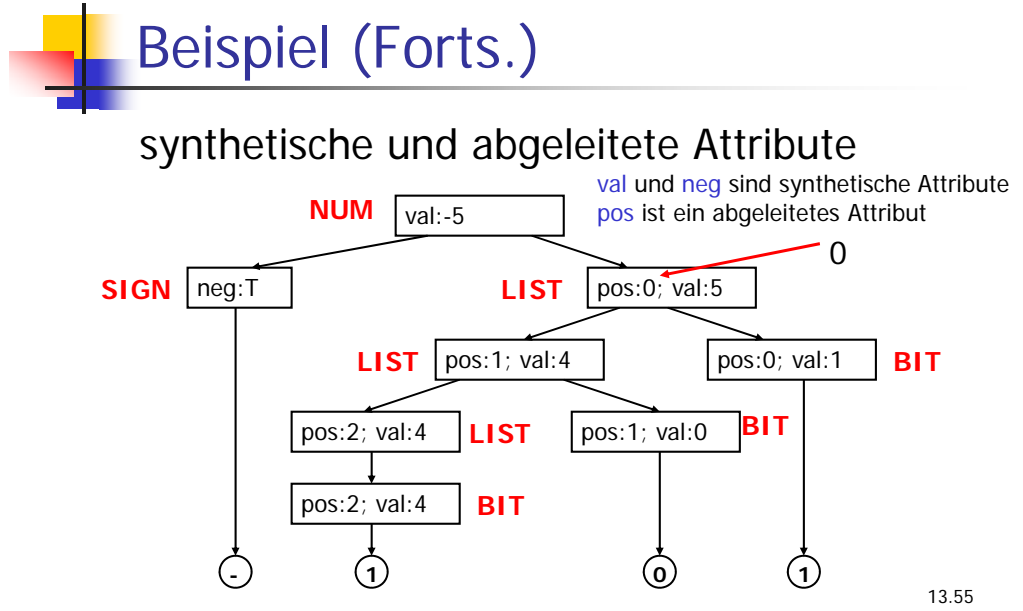
- Werte werden aus Konstanten und anderen Attributwerten berechnet
- **Synthetische Attribute:** Werte werden von *Kinderattributwerten* berechnet
- **Abgeleitete (engl. inherited) Attribute:** Werte werden von *Geschwister- oder Elternattributwerten* berechnet

Wichtig!!

### Wichtige Erkenntnis/Folgerung:

- Klassifizierung der Attribute induziert Abhängigkeitsgraphen

13.54



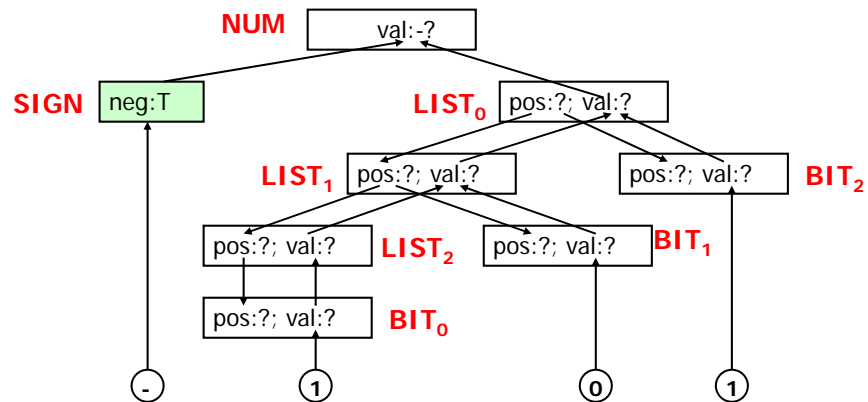
# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101

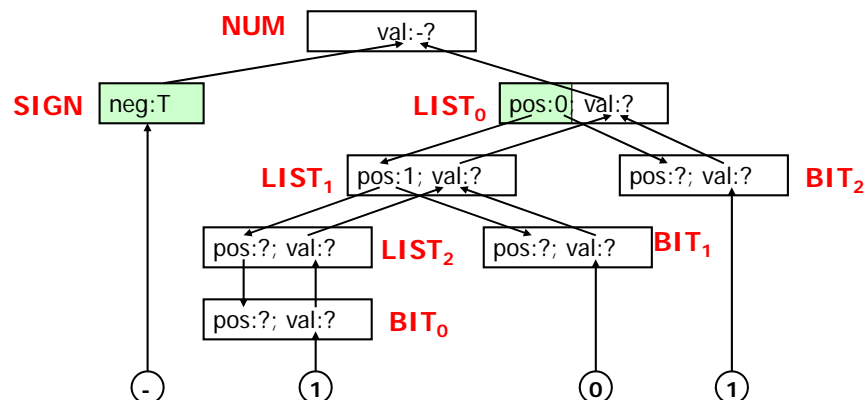


13.57



### Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101



13.58

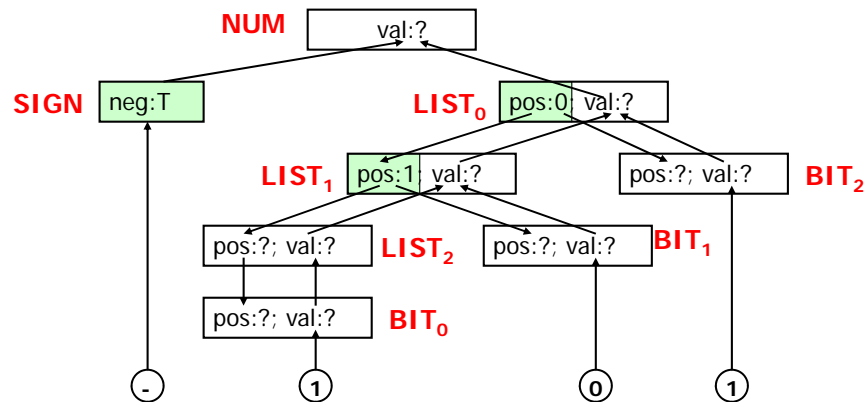
# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101

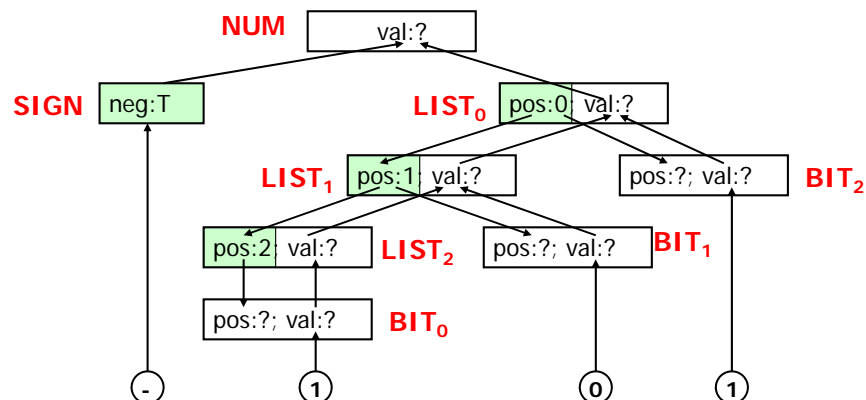


13.59



### Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101



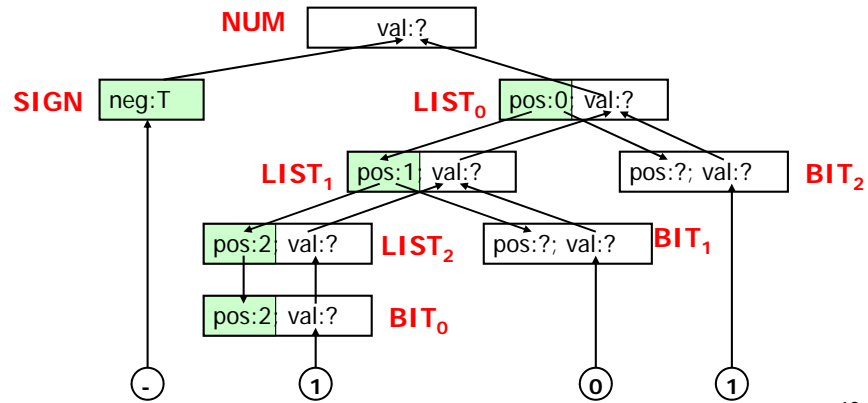
13.60

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Attributabhängiger Graph

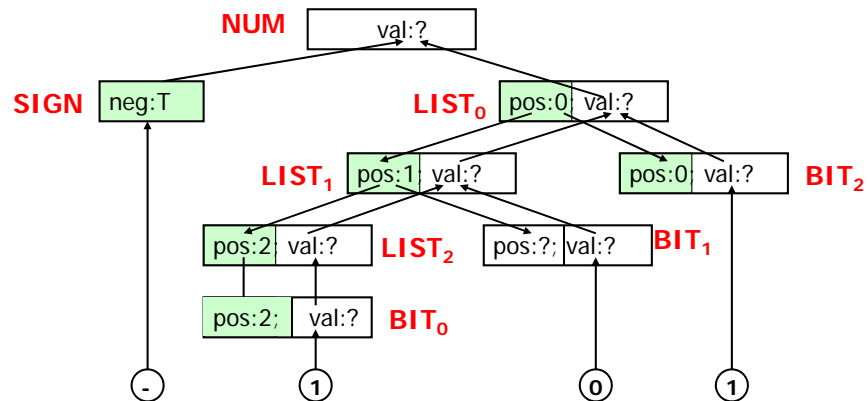
Abhängigkeitsgraph für -101



13.61

### Beispiel: Attributabhängiger Graph

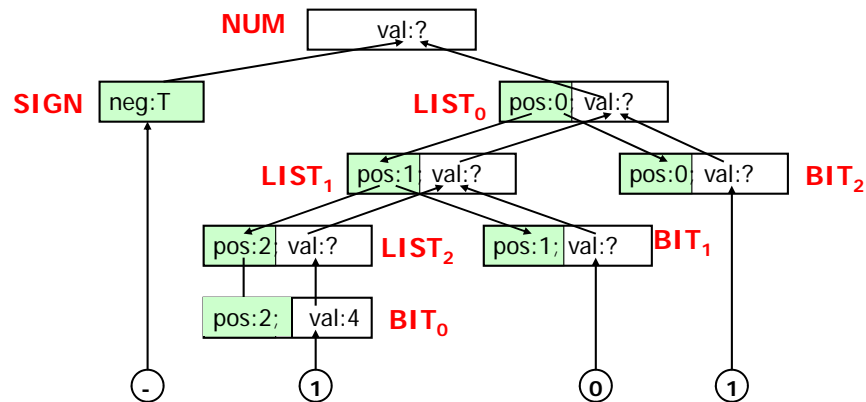
Abhängigkeitsgraph für -101



13.62

## Beispiel: Attributabhängiger Graph

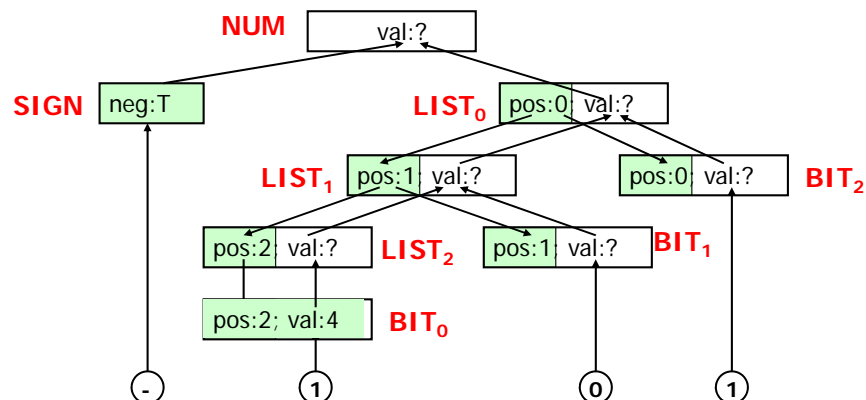
Abhängigkeitsgraph für -101



13.63

## Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101



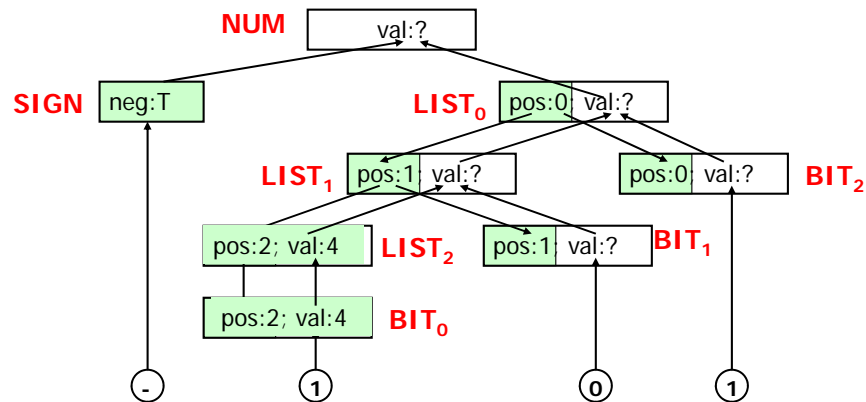
13.64





## Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101

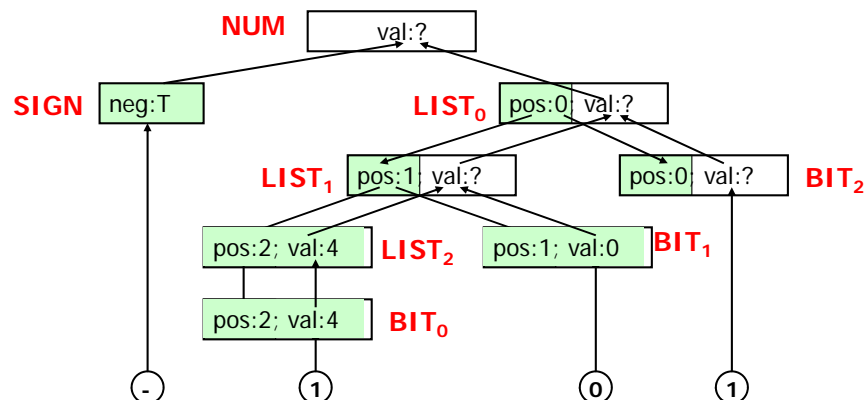


13.65



## Beispiel: Attributabhängiger Graph

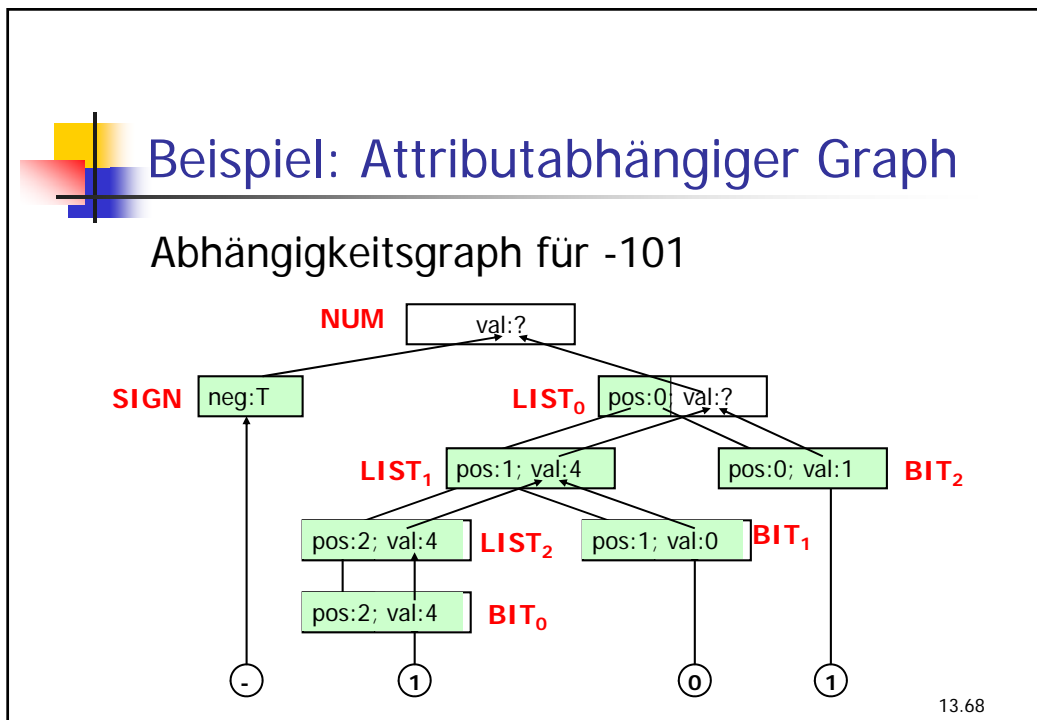
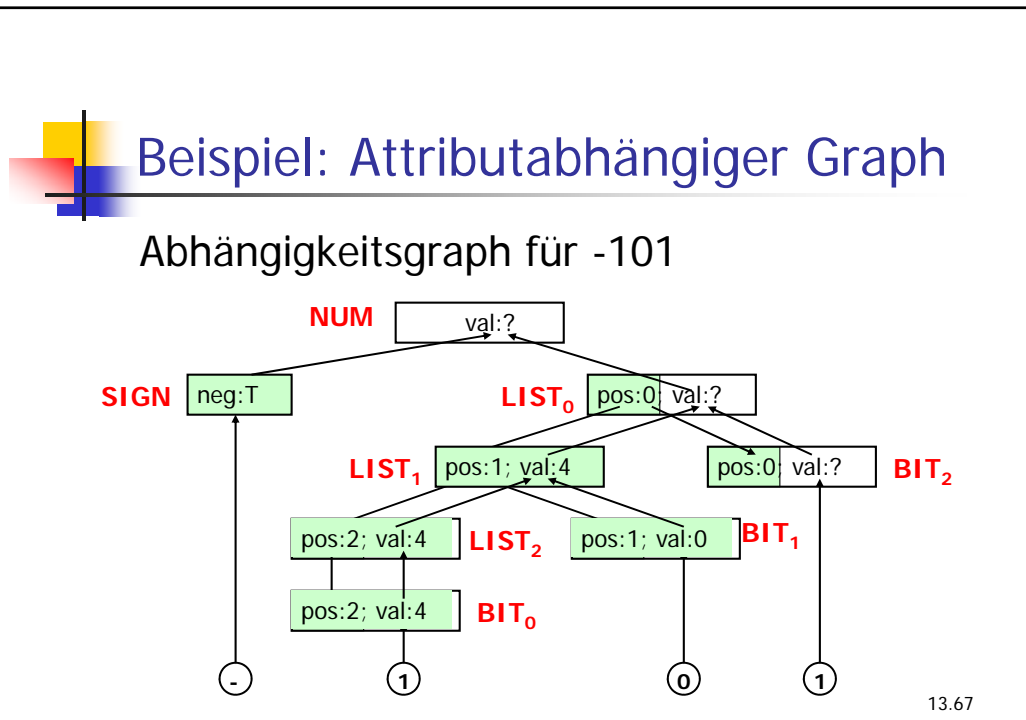
Abhängigkeitsgraph für -101



13.66

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

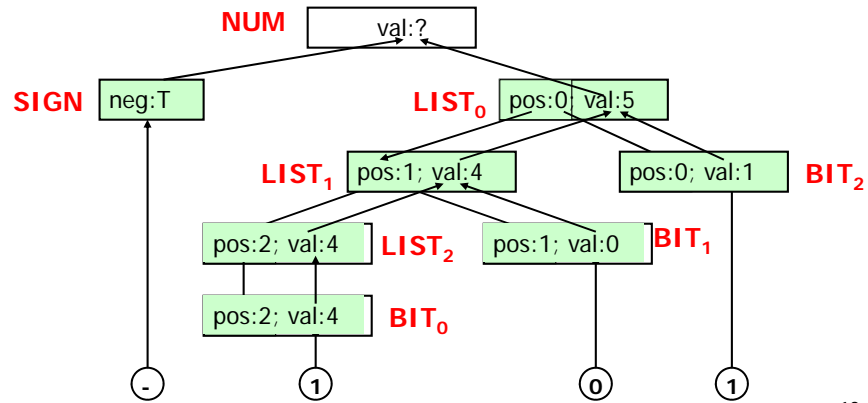


# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Attributabhängiger Graph

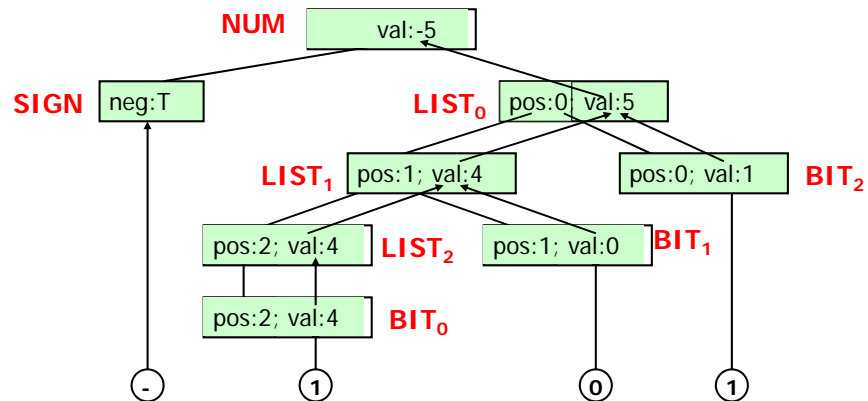
Abhängigkeitsgraph für -101



13.69

### Beispiel: Attributabhängiger Graph

Abhängigkeitsgraph für -101



13.70



## Beispiel: topologische Ordnung

1. SIGN.neg
2. LIST0.pos
3. LIST1.pos
4. LIST2.pos
5. BIT0.pos
6. BIT1.pos
7. BIT2.pos
8. BIT0.val
9. LIST2.val
10. BIT1.val
11. LIST1.val
12. BIT2.val
13. LIST0.val
14. NUM.val

Evaluierung in dieser Reihenfolge  
erzeugt den Wert NUM.val = -5  
(Entsprechend topologischer  
Sortierung)

13.71



## Topologische Sortierung

- Eine **topologische Sortierung** eines gerichteten azyklischen Graphen ist irgendeine Reihenfolge  $m_1, m_2, \dots, m_k$  der Knoten des Graphen,
  - so dass alle Kanten von Knoten ausgehen, die **früher** in der Reihenfolge auftreten,
  - und zu Knoten führen, die später vorkommen

$$m_i \rightarrow m_j, \quad \text{mit } m_i \text{ vor } m_j,$$

13.72

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Ziel bei Attribut-Grammatiken

- Berechnung von Attributwerten
- Wann wird berechnet?
  - während des Parsens der Eingabe oder
  - nachdem der Syntaxbaum komplett konstruiert worden ist
- Reihenfolge der Berechnung von Attributwerten
  - ist zu beachten (**unbekannte Reihenfolge** muss berechnet werden)
  - Topologie ergibt sich aus Attributabhängigkeiten (synthetisierte Attribute, geerbte Attribute)
- Berechnungsvorschrift für Attribute an einem Knoten (Semantikregel)
  - kann auch Seiteneffekte haben: drucken
  - jeder Grammatikproduktion wird eine Menge von Semantikregeln zugeordnet
  - sind meist nur Ausdrücke

13.73

### Beispiel für S-Attribut-Grammatik

Tischrechner: Eingabezeile (enthält Ausdruck) wird mit n abgeschlossen, berechneter Wert wird ausgegeben

Produktion	Semantische Regel
$L \rightarrow E \$$	<code>print(E.val)</code>
$E \rightarrow E_1 + T$	<code>E.val := E<sub>1</sub>.val + T.val</code>
$E \rightarrow T$	<code>E.val := T.val</code>
$T \rightarrow T_1 * F$	<code>T.val := T<sub>1</sub>.val * F.val</code>
$T \rightarrow F$	<code>T.val := F.val</code>
$F \rightarrow (E)$	<code>F.val := E.val</code>
$F \rightarrow \text{digit}$	<code>F.val := digit.lexval</code>

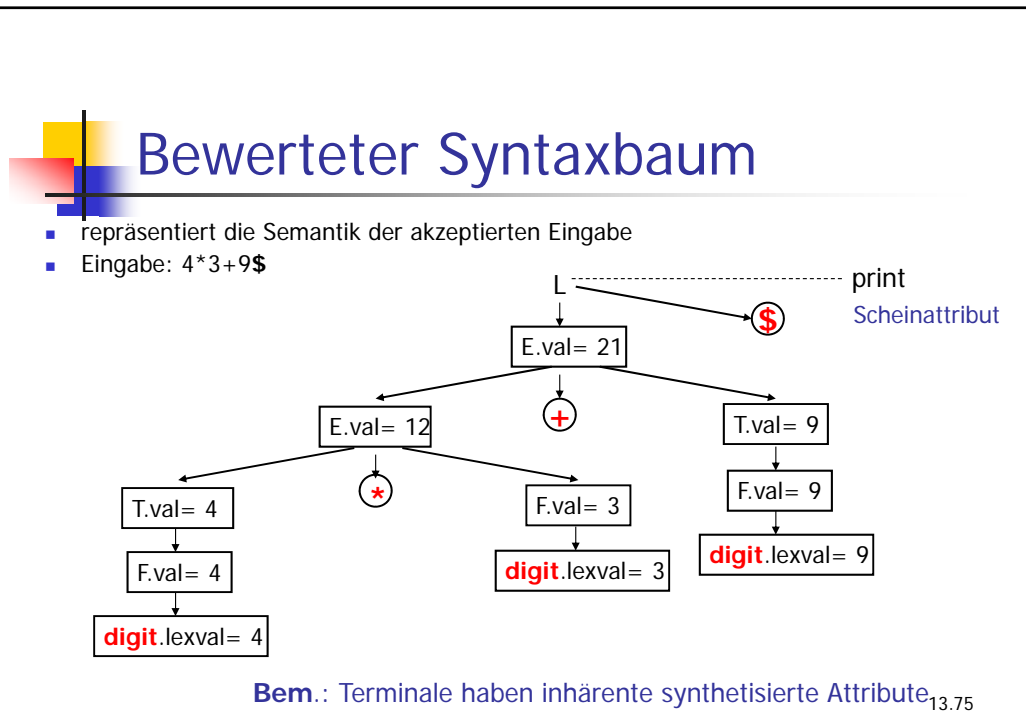
**nur mit synthetisierten Attributen** heißt **S-Attribut-Grammatik**

- Berechnung: Bottom-Up (leicht von LR-Parsern zu implementieren)

13.74

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung



### Beispiel für Attribut-Grammatik mit ererbten Attributen

ererbte Attribute erlauben die Abhängigkeit eines Programmkonstrukts vom Kontext auszudrücken

Achtung: T.type ist ein synthetisches Attribut

Produktion	Semantische Regel
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in$ $\text{addtype}(\text{id.entry}, L.in)$
$L \rightarrow \text{id}$	$\text{addtype}(\text{id.entry}, L.in)$

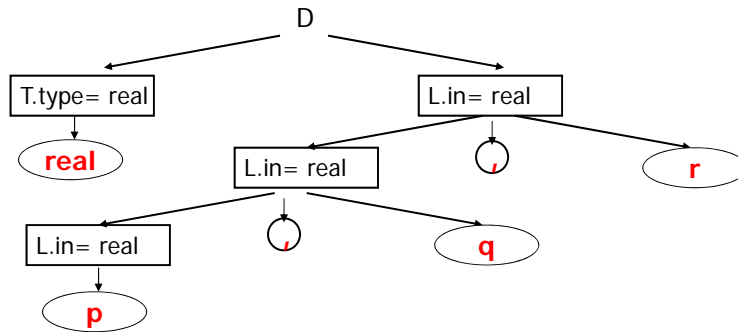
$L.in$ : ein ererbtes Attribut „verbreitet“ die Typ-information (Topdown)

Hinzufügen des Typs zum vorhandenen Bezeichner **id** in der Symboltabelle

13.76

## Bewerteter Syntaxbaum

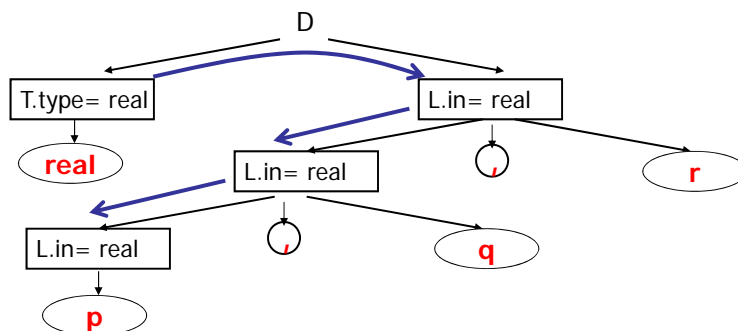
- repräsentiert die Semantik der akzeptierten Eingabe: **real** p, q, r



13.77

## Bewerteter Syntaxbaum

- Abhängigkeitsgraph für: **real** p, q, r



13.78

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Konstruktion des Abhängigkeitsgraphen

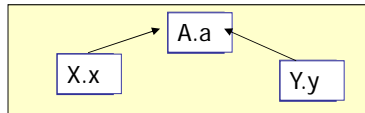
#### ■ Vorbereitung

- jede Semantikregel wird in folgende Form gebracht:  
 $b := f(c_1, c_2, \dots, c_k)$
- für jede Semantikregel, die nur aus einem Prozeduraufruf (i.e. kein Ergebnis hat) besteht, wird ein synthetisiertes **Scheinattribut** eingeführt
- der Graph hat für jedes Attribut einen Knoten sowie eine Kante von Knoten  $c$  zum Knoten  $b$ , wenn  $b$  von  $c$  abhängt

#### ■ Annahme:

Produktion	Semantische Regel
$A \rightarrow X Y$	$A.a := f(X.x, Y.y)$ <span style="color: blue;">synthetisiertes Attribut</span>
...	...

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



13.79

### Algorithmus zur Konstruktion des Abhängigkeitsgraphen

```
for each node  $n$  in the syntax tree do
  for each attribute  $a$  of a grammar symbol in  $n$  do
    add a node for  $a$  in the graph;

for each node  $n$  in the syntax tree do
  for each semantics rule  $b := f(c_1, c_2, \dots, c_k)$  associated with  $n$  do
    for  $i := 1$  to  $k$  do
      add an edge from node  $c_i$  to node  $b$ ;
```

13.80

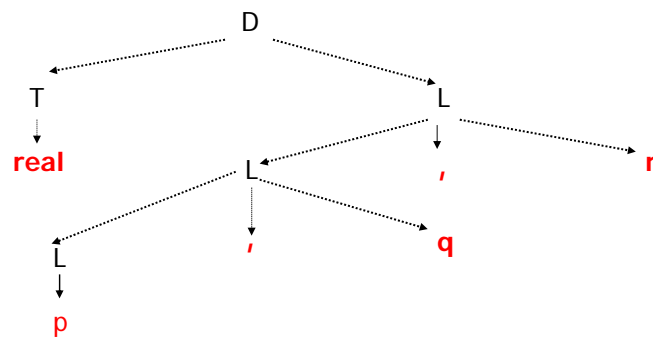


# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Konstruktion des Abhängigkeitsgraphen

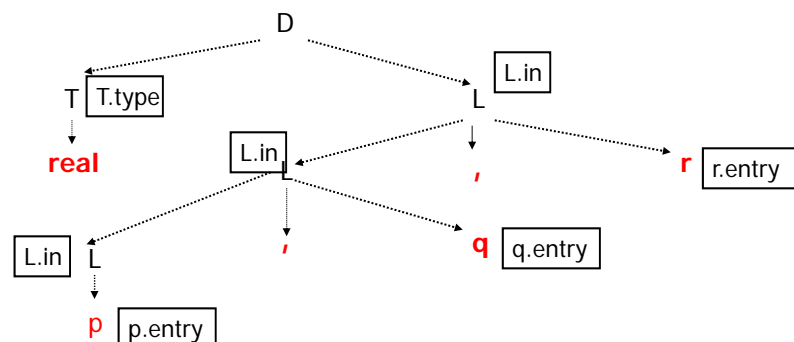
for each node **n** in the syntax tree do



13.81

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

for each node **n** in the syntax tree do  
for each attribute **a** of a grammar symbol in **n** do



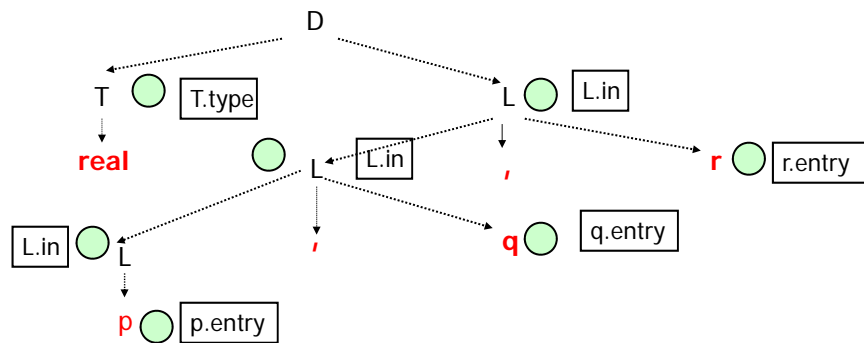
13.82

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

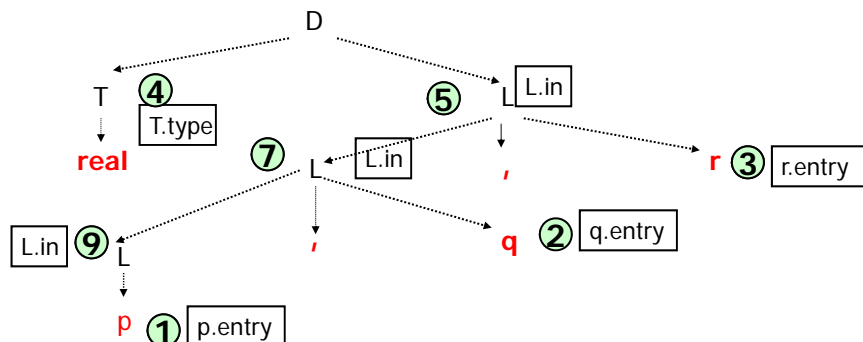
for each node **n** in the syntax tree do  
  for each attribute **a** of a grammar symbol in **n** do  
    add a node for **a** in the graph



13.83

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

for each node **n** in the syntax tree do  
  for each attribute **a** of a grammar symbol in **n** do  
    build a node for **a** in the graph



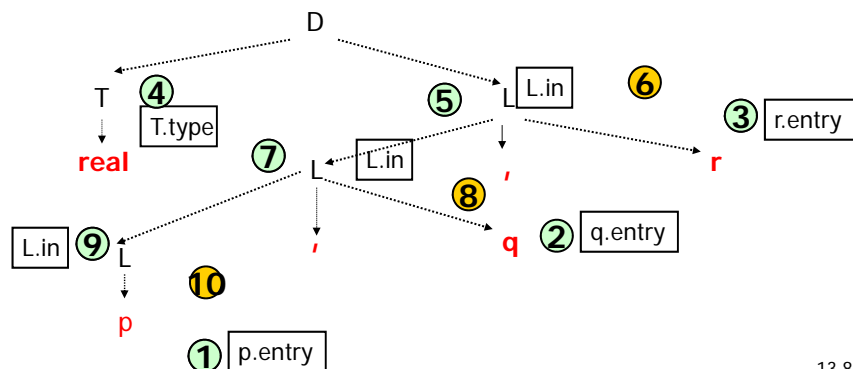
13.84

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

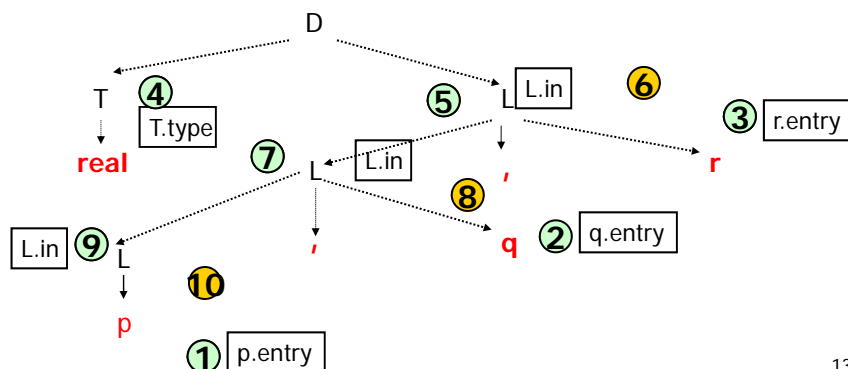
for each node **n** in the syntax tree do  
 for each "Scheinattribut" **a** of a grammar symbol in **n** do  
 build a node for **a** in the graph



13.85

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

for each node **n** in the syntax tree do  
 for each semantics rule  $b := f(c_1, c_2, \dots, c_k)$  associated with **n** do  
 for  $i := 1$  to  $k$  do add an edge from node **ci** to node **b**;



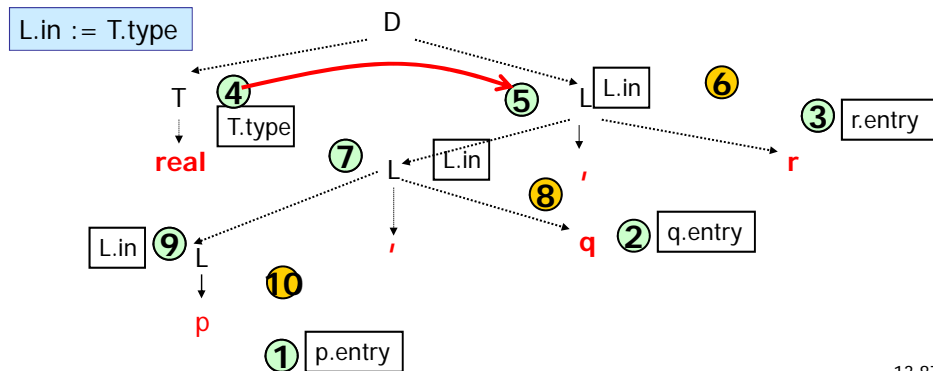
13.86

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

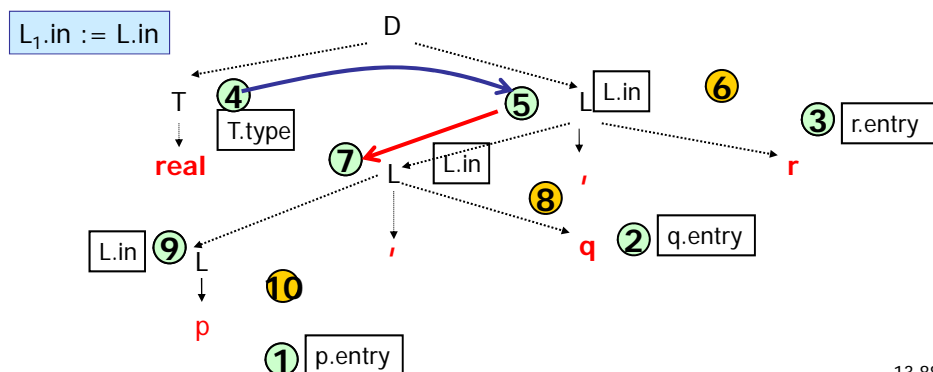
for each node **n** in the syntax tree do  
 for each semantics rule **b** :=  $f(c_1, c_2, \dots, c_k)$  associated with **n** do  
 for **i** := 1 to **k** do add an edge from node **ci** to node **b**;



13.87

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

for each node **n** in the syntax tree do  
 for each semantics rule **b** :=  $f(c_1, c_2, \dots, c_k)$  associated with **n** do  
 for **i** := 1 to **k** do add an edge from node **ci** to node **b**;



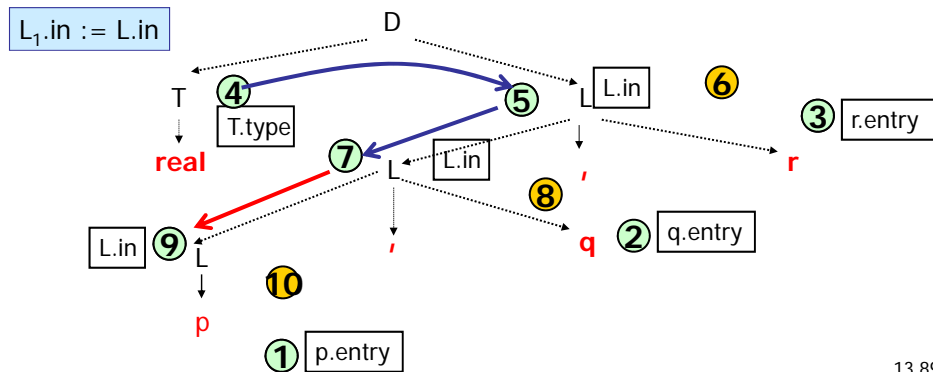
13.88

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

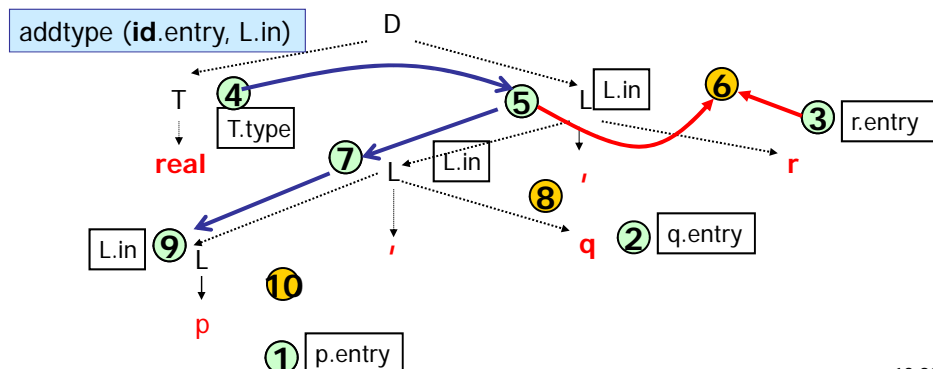
for each node **n** in the syntax tree do  
 for each semantics rule **b := f(c<sub>1</sub>, c<sub>2</sub>, ..., c<sub>k</sub>)** associated with **n** do  
 for **i := 1** to **k** do add an edge from node of **c<sub>i</sub>** to node of **b**;



13.89

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

for each node **n** in the syntax tree do  
 for each semantics rule **b := f(c<sub>1</sub>, c<sub>2</sub>, ..., c<sub>k</sub>)** associated with **n** do  
 for **i := 1** to **k** do add an edge from node of **c<sub>i</sub>** to node of **b**;



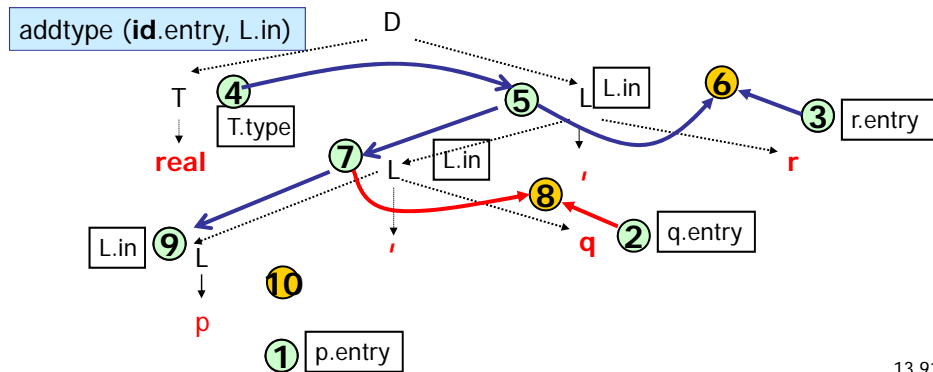
13.90

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

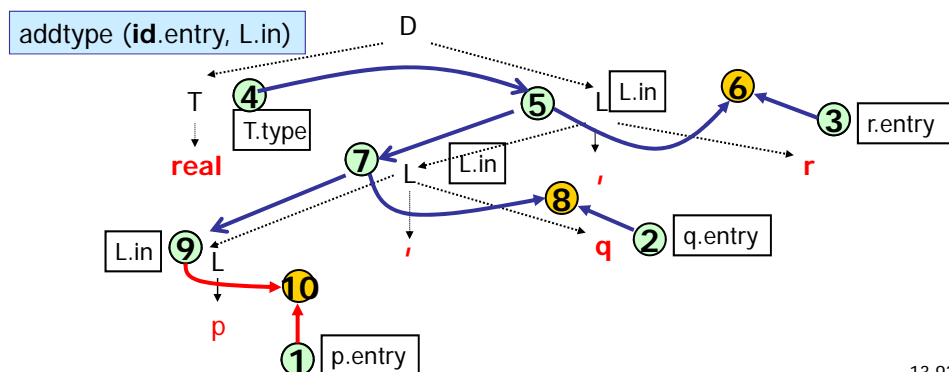
for each node **n** in the syntax tree do  
 for each semantics rule  $b := f(c_1, c_2, \dots, c_k)$  associated with **n** do  
 for  $i := 1$  to  $k$  do add an edge from node of **ci** to node of **b**;



13.91

### Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

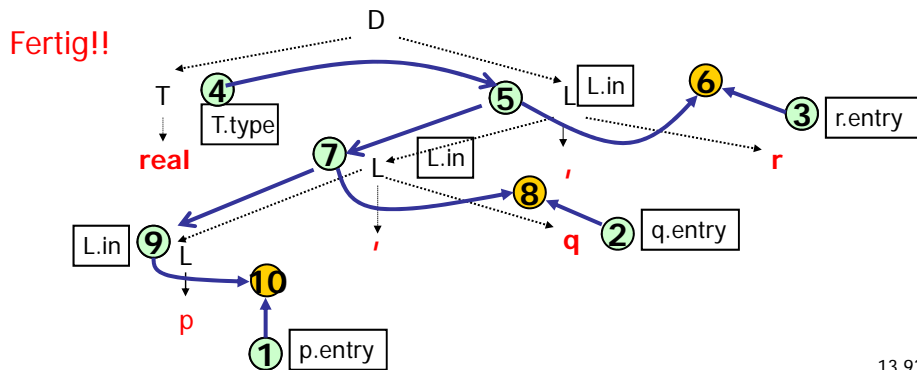
for each node **n** in the syntax tree do  
 for each semantics rule  $b := f(c_1, c_2, \dots, c_k)$  associated with **n** do  
 for  $i := 1$  to  $k$  do add an edge from node of **ci** to node of **b**;



13.92

## Beispiel: Konstruktion des Abhängigkeitsgraphen (Forts.)

for each node **n** in the syntax tree do  
 for each semantics rule  $b := f(c_1, c_2, \dots, c_k)$  associated with **n** do  
 for  $i := 1$  to  $k$  do build an arc from node of **ci** to node of **b**;



13.93

## Beispiel: Berechnungsreihenfolge

- (1)  $n_1 := \text{"p"}$
- (2)  $n_2 := \text{"q"}$
- (3)  $n_3 := \text{"r"}$
- (4)  $n_4 := \text{real}$
- (5)  $n_5 := n_4$
- (6)  $n_6 := \text{addtype}(n_3.\text{entry}, n_5)$
- (7)  $n_7 := n_5$
- (8)  $n_8 := \text{addtype}(n_2.\text{entry}, n_7)$
- (9)  $n_9 := n_7$
- (10)  $n_{10} := \text{addtype}(n_1.\text{entry}, n_9)$

13.94



### Strategien zur Attributberechnung

#### Parse-Baum-Methode (zum Zeitpunkt der Übersetzung)

- Erzeugen des Syntaxbaums und des Abhängigkeitsgraphen zur Übersetzungszeit
- Topologische Sortierung des Graphen
- Berechnung der Werte zur Übersetzungszeit
- **Nachteil:** nur bei zyklensfreien Graphen anwendbar

#### Regelbasierte Methode (zum Zeitpunkt der Compilererzeugung)

- Analyse der semantischen Regeln zum Zeitpunkt der Compilererzeugung (per Hand oder mit Werkzeug)
- Bestimme die **statische (!)** Reihenfolge für die Berechnung eines jeden Attributwertes der semantischen Bearbeitung (**unabhängig von der Eingabe**)
- Berechne diese Attribute in dieser Reihenfolge zur Übersetzungszeit

13.95



### Ideale Attributberechnung für LL(1)-Grammatiken

#### Vorraussetzungen

- Attribut-Grammatik ist **L-attributiert**  
(L steht für Links: die Attributinformation »fließt« von links nach rechts)
- Grammatik ist von einfacher Zuweisungsform

**Methode:** Tiefe-Zuerst-Suche (depth-first-Reihenfolge)  
angewendet auf die Wurzel des Syntaxbaums

13.96





## L-Attribut-Grammatiken

### Definition: L-Attribut-Grammatik

Sei  $X \rightarrow Y_1 Y_2 \dots Y_n$  eine Produktionsregel, dann hängen

- (1) die **ererbten** Attribute von  $Y_i$  ( $\text{Inh}(Y_i)$ ) nur von
  - ererbten Attributen des Nicht-Terminals  $X$  ( $\text{Inh}(X)$ ) und
  - beliebigen Attributen von  $Y_1 \dots Y_{i-1}$ ;
- (2) und die **synthetisierten** Attribute von  $X$  ( $\text{Syn}(X)$ ) nur von seinen ererbten und beliebigen Attributen der RHS

ab.

Impliziert folgende Berechnungsreihenfolge:

- $\text{Inh}(X), \text{Inh}(Y_1), \text{Syn}(Y_1), \dots, \text{Inh}(Y_n), \text{Syn}(Y_n), \text{Syn}(X)$
- dies stimmt mit der Auswertungsreihenfolge im LL-Parser (Top-Down-Parsing) überein !!

13.97



## Depth-First-Attributberechnung

```
procedure dfvisit (n: node)
Begin
  for each successor m of n from left to right do
    evaluate Inh(m);
    dfvisit (m);
  end
  evaluate Syn(n);
end
```

13.98

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

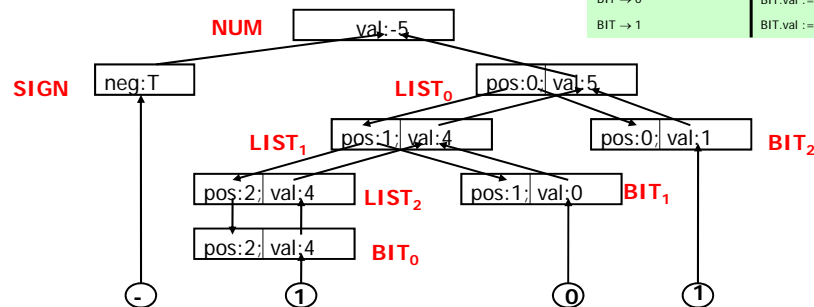
### Beispiel: Depth-First-Attri

```

procedure dfvisit (n: node)
begin
  for each successor m of n from left to right do
    evaluate Inh(m);
    dfvisit (m);
  end
  evaluate Syn(n);
end

```

Produktionsregel	Semantische Regel
NUM → SIGN LIST	LIST.pos := 0 if SIGN.neg then NUM.val := - LIST.val else NUM.val := LIST.val
SIGN → +	SIGN.neg := false
SIGN → -	SIGN.neg := true
LIST → BIT	BIT.pos := LIST.pos LIST.val := BIT.val
LIST → LIST <sub>1</sub> BIT	LIST <sub>1</sub> .pos := LIST.pos + 1 BIT.pos := LIST.pos LIST.val := LIST <sub>1</sub> .val + BIT.val
BIT → 0	BIT.val := 0
BIT → 1	BIT.val := 2 <sup>BIT.pos</sup>



### Beispiel: L-Attribut-Grammatik

Produktionsregel	Semantische Regel
NUM → SIGN LIST	LIST.pos := 0 if SIGN.neg then NUM.val := - LIST.val else NUM.val := LIST.val
SIGN → +	SIGN.neg := false
SIGN → -	SIGN.neg := true
LIST → BIT	BIT.pos := LIST.pos LIST.val := BIT.val
LIST → LIST <sub>1</sub> BIT	LIST <sub>1</sub> .pos := LIST.pos + 1 BIT.pos := LIST.pos LIST.val := LIST <sub>1</sub> .val + BIT.val
BIT → 0	BIT.val := 0
BIT → 1	BIT.val := 2 <sup>BIT.pos</sup>

val und neg sind synthetisierte Attribute  
pos ist ein abgeleitetes Attribut

13.100

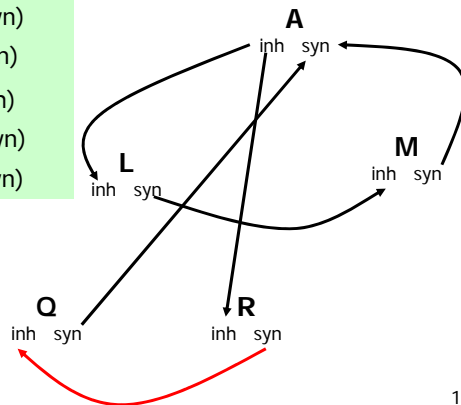
# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: keine L-Attribut-Grammatiken

Produktion	Semantische Regel
$A \rightarrow L M$	$L.inh := l(A.inh)$ $M.inh := m(L.syn)$ $A.syn := f(M.syn)$
$L \rightarrow Q R$	$R.inh := r(A.inh)$ $Q.inh := q(R.syn)$ $A.syn := f(Q.syn)$

*Q.inh ist abhängig von R.syn  
aber R steht nicht links von Q  
sondern rechts*



13.101

### S-Attribut-Grammatik

#### Definition: S-Attribut-Grammatik

Eine Grammatik ist **S-attributiert**, falls

- sie L-attributiert ist,
- Nicht-Terminale haben nur synthetisierte Attribute (d.h. Werte bestimmen sich aus den Werten von "Kinderknoten") (Bem.: Terminale haben immer nur inhärente Werte)
- semantische Aktionen treten erst ganz rechts am Ende der RHS auf

#### Eigenschaft:

S-Attribut-Grammatiken können in einem Bottom-Up-Durchlauf (LR-Parsing) erkannt und berechnet werden

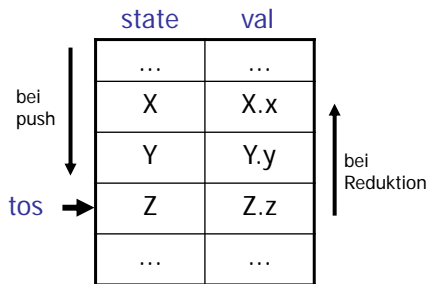
13.102

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Synthetisierte Attribute auf dem Parser-Kellerspeicher

Parser-Keller mit zusätzlichen Feldern für Attribute (hier: 1)



Zeiger/Index auf Syntaxanalyse-Tabelle des Parsers

- sei  $state[i]$  mit  $A$  assoziiert, dann ist  $val[i]$  der Wert von  $A.a$
- Annahme für synthetisiertes Attribut:  
 $A.a := f(X.x, Y.y, Z.z)$   
mit  $A \rightarrow XYZ$
- bevor  $XYZ$  auf  $A$  reduziert wird, ist  
 $Z.z = val[tos]$   
 $Y.y = val[tos-1]$   
 $X.x = val[tos-2]$
- hat ein Symbol kein Attribut, ist  $val$  undefiniert
- Reduktion  $XYZ \rightarrow A$ :  
 $tos := tos-2$ ;  $state[tos] := A$ ;  $val[tos] := A.a$

13.103

### Beispiel: S-Attribut-Grammatik

- NUM-Grammatik (Berechnung dezimaler Wert) ist **nicht** S-attribuiert
- Beispiel einer S-Attribut-Grammatik

"alter" Tischrechner  
mit modifizierter  
Notation der  
semantischen  
Regeln

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

13.104

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Ablauf bei Eingabe von: $3 * 5 + 4 \$$

Eingabe	state	val	benutzte Produktion, semantische Regel
3*5+4\$	-		
*5+4\$	3	3	
*5+4\$	F	3	$F \rightarrow \text{digit}$
*5+4\$	T	3	$T \rightarrow F$
5+4\$	T*	3_	
+4\$	T*5	3_5	
+4\$	T*F	3_5	$F \rightarrow \text{digit}$
+4\$	T	15	$T \rightarrow T * F, \quad \text{val}[\text{ntos}] := \text{val}[\text{tos}-2] * \text{val}[\text{tos}]$
+4\$	E	15	$E \rightarrow T$
4\$	E+	15_	
\$	E+4	15_4	
\$	E+F	15_4	$F \rightarrow \text{digit}$
\$	E+T	15_4	$T \rightarrow F$
\$	E	19	$E \rightarrow E + T, \quad \text{val}[\text{ntos}] := \text{val}[\text{tos}-2] + \text{val}[\text{tos}]$
	E\$	19_	
	L	19	$L \rightarrow E \$, \quad \text{print val}[\text{tos}]$

ntos := tos - r + 1  
bei  
 $B_1 B_2 \dots B_r \rightarrow A$

nach Ausführung  
tos := ntos

Aktionen werden  
an  
Reduktionen  
geknüpft

13.105

### Aber...

**Ererbte Attribute** sind auch für ein Bottom-Up-Parsing notwendig

- werden aus Konstanten, Werten des Eltern- und den (linken) Geschwisterknoten erzeugt;
- drücken Kontext aus (kontextsensitive Analyse)
- ererbte Attribute sind in vielen Fällen natürlich

**Ziel**

- beide Arten von Attributen sollen im Bottom-Up-Parsing möglich sein

**Schwierigkeit**

- Wo sind diese auf dem Keller zur Übersetzungszeit zu finden ?

13.106

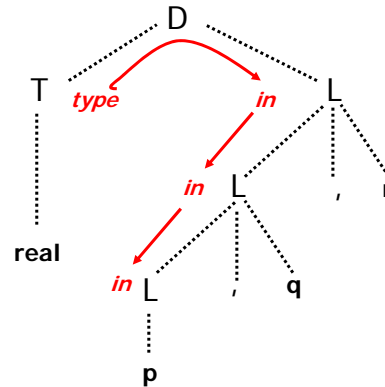
# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Wo ist Typinformation zu finden?

Eingabe: real p, q, r

Produktion	Semantische Regel
$D \rightarrow T L$	$L.in := T.type$
$T \rightarrow \text{int}$	$T.type := \text{integer}$
$T \rightarrow \text{real}$	$T.type := \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.in := L.in$ $\text{addtype}(\text{id.entry}, L.in)$
$L \rightarrow \text{id}$	$\text{addtype}(\text{id.entry}, L.in)$



13.107

### Beispiel: Bottom-Up-Parsing

Produktion	Semantische Regel - Codefragment
$D \rightarrow T L$	
$T \rightarrow \text{int}$	$\text{val}[\text{ntos}] = \text{integer}$
$T \rightarrow \text{real}$	$\text{val}[\text{ntos}] = \text{real}$
$L \rightarrow L_1, \text{id}$	$\text{addtype}(\text{val}[\text{tos}], \text{val}[\text{tos}-3])$
$L \rightarrow \text{id}$	$\text{addtype}(\text{val}[\text{tos}], \text{val}[\text{tos}-1])$

tos = momentaner »top-of-stack«  
ntos = »top-of-stack« nach der Reduktion

13.108

# Vorlesung Compilerbau (SoSe2018)

## Teil 13: Semantische Bearbeitung

### Beispiel: Bottom-Up-Parsing (Forts.)

Eingabe	Keller	Produktion
real p, q, r	-	
p, q, r	real	
p, q, r	T	$T \rightarrow \text{real}$
, q, r	T p	
, q, r	T L	$L \rightarrow \text{id}$
q, r	T L	
, r	T L, q	
, r	T L	$L \rightarrow L, \text{id}$
r	T L	
	T L, r	
	T L	$L \rightarrow L, \text{id}$
	D	$D \rightarrow T L$

#### Beobachtung

- T.type ist immer auf festem Platz im Keller (bezüglich tos) zu finden
  - Bei " $L \rightarrow L, \text{id}$ " ist dies tos - 3
  - Bei " $L \rightarrow \text{id}$ " ist dies tos - 1

#### Folgerung

- Deshalb braucht T.type nicht nach L.in kopiert zu werden, sondern kann **direkt** vom Keller benutzt werden

13.109

### Simulation ererbter Werte

Zugriff auf Keller nur dann möglich, wenn Position (unabhängig von anderen Regeln) immer vorher bekannt ist

#### Beispiel 1:

Produktionsregel	Semantikregel
$S \rightarrow aAC$	$C.inh := A.syn$
$S \rightarrow bABC$	$C.inh := A.syn$
$C \rightarrow c$	$C.syn := g(C.inh)$

#### Problem

- Für die Semantikregel der Produktion  $C \rightarrow c$  ist die Position nicht bekannt, da sie von der Anwendung der ersten beiden Regeln abhängt
- erste oder zweite Regel (mit NT "B") erzeugen unterschiedliche Positionen auf dem Keller, wo A.syn zu finden ist*

13.110

## Simulation ererbter Werte (Forts.)

- Umschreibung der Regeln in:

<u>Produktionsregel</u>	<u>Semantikregel</u>
$S \rightarrow aAC$	$C.inh := A.syn$
$S \rightarrow bABMC$	$M.inh := A.syn;$ $C.inh := M.syn$
$M \rightarrow \varepsilon$	$M.syn := M.inh$
$C \rightarrow c$	$C.syn := g(C.inh)$

- Mit dieser Umschreibung ist Position von A.syn im Keller »konstant« und unabhängig von der Nutzung der ersten beiden Regeln
  - M immer zurück mit 2
  - C immer zurück mit 1

13.111

## Simulation ererbter Werte (cont.)

- Beispiel 2:

<u>Produktionsregel</u>	<u>Semantikregel</u>
$S \rightarrow aAC$	$C.inh := f(A.syn)$

- C erbt  $f(A.syn)$ , aber nicht durch kopieren
- C möge C.inh weiter vererben
- Wert von  $f(A.syn)$  ist nicht auf dem Keller zu finden
- Umschreiben in:

<u>Produktionsregel</u>	<u>Semantikregel</u>
$S \rightarrow aAMC$	$M.inh := A.syn;$ $C.inh := M.syn$
$M \rightarrow \varepsilon$	$M.syn := f(M.inh)$

13.112





### Fazit

- Bottom-Up-Syntaxanalyse für L-Attribut-Grammatiken (mit Einschränkungen) möglich
- Attribut-Grammatiken
  - Vorteile
    - sauberer Formalismus
    - hochsprachliche Spezifikation semantischer Regeln (ohne Implementierungsaspekte)
  - Nachteil
    - Berechnungsstrategie beeinflusst Effizienz
    - erhöhter Speicherbedarf
    - Parse-Baum bedarf eines (zyklenfreien) Abhängigkeitsgraphen

13.113



### Typen und Typbeschreibungen

- Typangaben werden durch **Typausdrücke** formal beschrieben
- Ein Typausdruck ist eine textuelle Repräsentation für Typen
- Konstruktion von Typen
  - Basistypen: boolean, char, integer, real, ...
  - Typnamen
  - Konstruierte Typen (Konstruktor auf einen Typausdruck angewandt)
  - ...

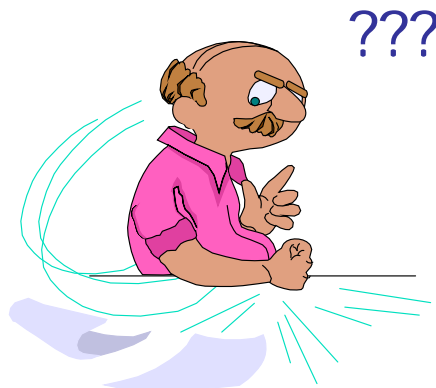
13.114

### Typen und Typbeschreibungen (Forts.)

- Konstruierte Typen (Konstruktor auf einen Typausdruck angewandt)
  - $\text{array}(I, T)$  beschreibt ein Feld des Basistyps  $T$  indiziert über  $I$ ;  
Beispiel:  $\text{array}(1 \dots 10, \text{integer})$
  - $T1 \times T2$  beschreibt ein kartesisches Produkt über die Typausdrücke  $T1$  und  $T2$
  - record: Felder haben Namen,  
d.h.  $\text{record}((a \times \text{integer}), (b \times \text{real}))$
  - Zeiger:  $\text{pointer}(T)$  beschreibt den Typ "Zeiger auf ein Objekt vom Typ  $T$ "
  - Funktion:  $D \rightarrow R$  beschreibt den Typ einer Funktion, die Werte des Typs  $D$  auf Werte des Typs  $R$  abbildet,  
d.h.  $\text{integer} \times \text{integer} \rightarrow \text{integer}$

13.115

### Fragen



13.116